Branch-and-Price Framework in jORLib

Joris Kinable

June 29, 2016

1 Introduction

When reading about Column Generation or Branch-and-Price in math books, the procedures are always very clear: define a Master and a Pricing Problem, and solve them iteratively until optimality is reached. If the solution is fractional: branch. The reality is however significantly more complex. The quality of any Column Generation procedure significantly depends on the many implementation choices one has to make. These design choices, as well a many other important details are frequently omitted when discussing Column Generation:

1. how to obtain an initial subset of columns?
2. in which order should the nodes in the Branch-and-Price tree be processed?
3. when should we check for violated inequalities in the Master Problem?
4. should we return one or more columns in the Pricing Problem?

Very often, there does not exist a clear-cut answer to these questions. The only way to find out, is simply by trial-and-error. Implementing a Branch-and-Price approach is a relatively complex and very time consuming task. Designing the approach on paper, implementing a basic implementation, testing and debugging the implementation takes up a considerable portion of the available development time, leaving little time to experiment with different settings or alternative implementation choices. A good framework could significantly reduce the required development time, and in addition, reduce the amount of debugging required.
The Column Generation framework in jORLib provides a flexible way to implement Column Generation and Branch-and-Price procedures. The framework does *not* provide any automated means to solve a problem through CG; it merely provides a template for your implementation. To implement a simple CG procedure through this framework, the user is required to provide a definition of the Master Problem, one or more Pricing Problems and corresponding algorithms to solve these Pricing Problems, and, of course, a definition of a Column. More advanced implementations may use features such as: the separation of valid inequalities for the Master Problem, or Branch-and-Price.
You do not need to be concerned about structural issues: the framework connects the Master Problem and Pricing Problems, it will solve the Pricing Problems in parallel for you, it provides means to calculate bounds on the Master Problem, Pricing Problem, etc. The Branch-and-Price extension takes care of the Branch-and-Price tree which is implemented as a fully reversible data structure. Consequently, nodes in the Branch-and-Price tree can be processed in arbitrary order. Nodes are pruned when bounds are exceeded.
The Column Generation and Branch-and-Price framework is designed in such a way that it offers you as much flexibility as possible (nearly every method can be overridden by a custom implementation), while still limiting the amount of code you would have to write to get a working solution. The framework does *not* impose restrictions on the the kind of solver used to solve the Master or Pricing Problems: the master problem may be solved through some Linear Programming solver such as CPLEX, or Gurobi, Ipopt, etc, while the Pricing Problems may be solved in any conceivable way.
The remainder of this manual discusses the functionality and structure of the framework, starting off with a number of examples.

2 Example 1 - Solving the Cutting Stock Problem through Column Generation

In the Cutting Stock Problem (CSP), a predefined number of so-called finals of a given size need to be produced. Finals can be cut from raws of fixed size. The goal is to cut all the finals, using the smallest number of raws. For a full description of the problem, including an example, refer to: `http://en.wikipedia.org/wiki/Cutting_stock_problem`. Here we will only state the Master Problem and Pricing Problem.
Techniques used in this example:

- Column generation

A full implementation can be found in the demo package.

2.1 Problem description

Parameter	Description
I	Set of cutting patterns. Each cutting pattern describes how to cut a single raw.
J	Set of finals
a_{ij}	Number of times that final $j \in J$ appears in cutting pattern $i \in I$
d_j	Demand for final $j \in J$
w_j	Width of final $j \in J$
L	Width of a raw

Table 1

The Master Problem (MP) for Cutting stock is defined as:

$$MP: min \sum_{i \in I} z_i \tag{1}$$

$$\text{s.t.} \sum_{i \in I} a_{ij} z_i \geq q_j \qquad \forall j \in J \tag{2}$$

$$z_i \geq 0 \qquad \forall i = 1, \ldots, n \tag{3}$$

Here, variable z_i indicates how often pattern i is used. The objective function (1) states that we want to use as little raws as possible. Constraints (2) state that sufficient finals of each width $j \in J$ are produced. The dual of the Master Problem (DMP) can be stated as:

$$DMP: max \sum_{j \in J} q_j u_j \tag{4}$$

$$\text{s.t.} \sum_{j \in J} a_{ij} u_j \leq 1 \qquad \forall i \in I \tag{5}$$

$$u_j \geq 0 \qquad \forall j \in J \tag{6}$$

Consequently, the Pricing Problem can be defined as: does there exists a pattern $i \in I$ such that $\sum_{j \in J} a_{ij} u_j > 1$? The Pricing Problem can be solved through the following knapsack problem:

$$PRICING: max \sum_{j \in J} a_j u_j \tag{7}$$

$$\text{s.t.} \sum_{j \in J} w_j a_j \leq L \tag{8}$$

$$a_j \text{ integer} \qquad \forall j \in J \tag{9}$$

2.2 Implementation

The following classes need to be extended to implement Column Generation for the Cutting Stock problem:

1. `AbstractColumn` - This class defines a column in the Column Generation process.
2. `AbstractMaster` - This problem defines the Master Problem.
3. `AbstractMasterData` - This class stores data from the Master Problem.
4. `AbstractPricingProblem` - This class defines a Pricing Problem and provides storages for shared data between the Master Problem and the Pricing Problem solvers.
5. `AbstractPricingProblemSolver` - A solver class for the Pricing Problem.

In addition, we need to define a data model class which defines the Cutting Stock problem.

In the remainder of this section we discuss each of these classes individually. A complete, working, implementation is provided in the demo package. The actual implementation may deviate from the examples shown below as many of the irrelevant details have been omitted.
Lets first start by defining a small class defining our problem. This class will be referred to as `dataModel` and will be accessible from all classes in the framework.

CuttingStock

```
1  public final class CuttingStock implements ModelInterface{
2    public final int nrFinals=4; //Number of different finals
3    public final int rollWidth=100; //Width of the raws
4    public final int[] finals={45, 36, 31, 14}; //Size of the finals
5    public final int[] demandForFinals={97, 610, 395, 211}; //Requested
         quantity of each final
6
7    @Override
8    public String getName() { return "CuttingStockExample";}
9  }
```

Next up, lets define a column in our Cutting Stock problem. Each column (`CuttingPattern`) maintains how many finals of a given width are produced. We will store this information in a '`yieldVector`':

CuttingPattern

```
1  public final class CuttingPattern extends AbstractColumn<CuttingStock,
       PricingProblem> {
2    /** Denotes the number of times each final is cut out of the raw. (vector
         a_j) **/
3    public final int[] yieldVector;
4
5    public CuttingPattern(PricingProblem pricingProblem, boolean isArtificial
       , String creator, int[] pattern) {
6      super(pricingProblem, isArtificial, creator);
7      this.yieldVector=pattern;
8    }
9
10   @Override
11   public boolean equals(Object o) {<Code Omitted>}
12
13   @Override
14   public int hashCode() {<Code Omitted>}
15
16   @Override
17   public String toString() {<Code Omitted>}
18 }
```

The constructor of AbstractColumn takes 3 arguments:

1. `String creator`: textual description denoting who created the column, i.e. some algorithm, initial solution etc. This is for debugging purposes to track the origin of the column.
2. **`boolean`** `isArtificial` artificial columns may be added to create an initial feasible solution to the Master Problem. artificial columns can never constitute a real solution.
3. `PricingProblem pricingProblem`: the Pricing Problem to which this column belongs.

Finally, make sure you implement the `equals` and `hashCode` methods; failing to implement these correctly may result in undefined behavior.

After defining what a Column is in our problem, we have to define a Master Problem and a Pricing Problem. The Master Problem consists of 7 important parts:

- Constructor - The constructor receives our data model which defines the problem, as well as a reference to our Pricing Problem.
- `buildModel` method - This method builds the Master Problem, which is usually a linear program defined in CPLEX or Gurobi
- `solveMasterProblem` method - This method solves the Master Problem
- `addColumn` method - This method is invoked when a Pricing Problem generates a new column; the column has to be added to the Master Problem
- `initializePricingProblem` method - This method is invoked after the Master Problem has been solved; The dual information required to solve the Pricing Problem has to be passed to the `PricingProblem` object.
- `getSolution` method - This method returns a list of columns constituting the solution, in our case a list of columns with non-zero z_i values.
- `close` method - This method is invoked at the end of the Column Generation procedure to close the Master Problem (analogous to destructor in C++).

For demonstration purposes, we will assume that the Master Problem is being solved by IBM Cplex, but we would like to emphasize that any kind of solver (Gurobi, Ipopt, CP Optimizer, ...) may be used.

Master

```
1  public final class Master extends AbstractMaster<CuttingStock,
       CuttingPattern, PricingProblem, CuttingStockMasterData> {
2    public Master(CuttingStock dataModel, PricingProblem pricingProblem) {
3      super(dataModel, pricingProblem, OptimizationSense.MINIMIZE);
4    }
5
6    @Override
7    protected CuttingStockMasterData buildModel() {
8      //Define a container for the variables
9      Map<PricingProblem, OrderedBiMap<CuttingPattern, IloNumVar>>
           variableMap=...;
10
11     //Define objective
12     obj= cplex.addMinimize();
13
14     //Define constraint: \sum_{i \in I} a_{ij}z_i>= q_j for all j\in J
15     satisfyDemandConstr=new IloRange[dataModel.nrFinals];
16     for(int j=0; j< dataModel.nrFinals; j++)
17       satisfyDemandConstr[j]= cplex.addRange(dataModel.demandForFinals[j],
             Integer.MAXVALUE);
18
19     //Return a new data object which will hold data from the Master Problem
20     return new CuttingStockMasterData(variableMap);
21   }
22
23   @Override
24   protected boolean solveMasterProblem(long timeLimit) throws
         TimeLimitExceededException {
25     //Set time limit
26     double timeRemaining=timeLimit-System.currentTimeMillis())/1000.0;
27     cplex.setParam(IloCplex.DoubleParam.TiLim, timeRemaining);
28
29     //Solve the model
30     if(!cplex.solve() || cplex.getStatus()!=IloCplex.Status.Optimal){
31       if(cplex.getCplexStatus()==IloCplex.CplexStatus.AbortTimeLim)
32         throw new TimeLimitExceededException();
33       else
34         throw new RuntimeException("Master Problem solve failed!");
```

```
35      }else{
36        masterData.objectiveValue= cplex.getObjValue();
37      }
38      return true;
39    }
40
41    /**
42     * Store the dual information required by the Pricing Problems into the
          Pricing Problem object
43     */
44    @Override
45    public void initializePricingProblem(PricingProblem pricingProblem){
46      double[] duals= cplex.getDuals(satisfyDemandConstr);
47      pricingProblem.initPricingProblem(duals);
48    }
49
50    /**
51     * Function which adds a new column to the Master Problem
52     */
53    @Override
54    public void addColumn(CuttingPattern column) {
55      //Register column with objective
56      IloColumn iloColumn= cplex.column(obj,1);
57
58      //Register column with demand constraint
59      for(int i=0; i< dataModel.nrFinals; i++)
60        iloColumn=iloColumn.and(cplex.column(satisfyDemandConstr[i], column.
              yieldVector[i]));
61
62      //Create the variable and store it
63      IloNumVar var= cplex.numVar(iloColumn, 0, Double.MAX_VALUE, "z_"+","+
            masterData.getNrColumns());
64      cplex.add(var);
65      masterData.addColumn(column, var);
66    }
67
68    /**
69     * Return the solution, i.e columns with non-zero z_i values
70     */
71    @Override
72    public List<CuttingPattern> getSolution() {
73      List<CuttingPattern> solution=new ArrayList<>();
74      for(CuttingPattern cp : masterData.getColumns()){
75        cuttingPatterns[i].value=cplex.getValue(cp);
76        if(cuttingPatterns[i].value>=config.PRECISION){
77          solution.add(cuttingPatterns[i]);
78        }
79      }
80      return solution;
81    }
82
83    @Override
84    public void close() {<Code Omitted>}
85  }
```

Data originating from the Master Problem, such as its variables, constraint etc may be required by other classes. A shared object which extends `AbstractMasterData` is created for this purpose in the `buildModel` method of the Master Problem. This object is for example passed to a `CutHandler` to separate violated inequalities (cuts). For our CuttingStock problem, we only store the variables in this object:

CuttingStockMasterData

```
1  public final class CuttingStockMasterData extends MasterData<CuttingStock,
       CuttingPattern, PricingProblem, IloNumVar>{
2      public CuttingStockMasterData(Map<PricingProblem, OrderedBiMap<
           CuttingPattern, IloNumVar>> varMap) {
3          super(varMap);
4      }
5  }
```

Next up is the Pricing Problem. For Cutting Stock, there is only one Pricing Problem. The `AbstractPricingProblem` class mainly serves as a container to hold the dual information coming from the Master Problem. It *defines* a Pricing Problem which can be solved by some algorithm. In our case, the `PricingProblem` class maintains the u_j, $j \in J$ dual values.

PricingProblem

```
1  public final class PricingProblem extends AbstractPricingProblem<
       CuttingStock> {
2    public PricingProblem(CuttingStock modelData, String name) {
3      super(modelData, name);
4    }
5  }
```

After defining the data object which stores the information required to solve the Pricing Problem, one or more solvers for the Pricing Problem have to be defined. The Pricing Problem of the Cutting Stock problem is a knapsack problem. Each time the Pricing Problem is solved, only the objective changes ($max \sum_{j \in J} a_j u_j$); the constraint remain the same.

ExactPricingProblemSolver

```
1  public final class ExactPricingProblemSolver extends
       AbstractPricingProblemSolver<CuttingStock, CuttingPattern,
       PricingProblem> {
2    public ExactPricingProblemSolver(CuttingStock dataModel, PricingProblem
         pricingProblem) {
3      super(dataModel, pricingProblem);
4      this.name="ExactSolver"; //Set a name for the solver
5      knapsackSolver=new KnapsackSolver();
6    }
7
8    @Override
9    protected List<CuttingPattern> generateNewColumns() throws
         TimeLimitExceededException {
10     List<CuttingPattern> newPatterns=new ArrayList<>();
11     knapsackSolver.solve();
12     if(knapsackSolver.getObjective() >= 1+config.PRECISION){ //Generate new
           column
13       int[] pattern=knapsackSolver.getItemCoefficients();
14       CuttingPattern column=new CuttingPattern("exactPricing", false,
             pattern, pricingProblem);
15       newPatterns.add(column);
16     }
17     return newPatterns;
18   }
19
20   /**
21    * Set the objective value
22    */
```

```
23    @Override
24    protected void setObjective() {
25      //Update the objective function with the new dual values, by using the
            fields in the pricingProblem, e.g. pricingProblem.dualCosts
26      knapsackSolver.setObjective(pricingProblem.dualCosts);
27    }
28
29    @Override
30    public void close() {<Code Omitted>}
31  }
```

Each time the Pricing Problem is solved, the `setObjective` function gets invoked first. During the execution of this function, the solver can prepare its data structures and get the new dual information. Next the `generateNewColumns` method is invoked which produces zero or more columns. Finally, when the Column Generation procedure terminates, the close() function is invoked which allows for cleanup. It serves the same purpose as a destructor in C++.
Finally, we need to bring everything together in a main class.

CuttingStockSolver

```
1  public final class CuttingStockSolver {
2    public CuttingStockSolver(CuttingStock dataModel){
3
4      //Create the Pricing Problem
5      PricingProblem pricingProblem=new PricingProblem(dataModel, "
           cuttingStockPricing");
6
7      //Create the Master Problem
8      Master master=new Master(dataModel, pricingProblem);
9
10     //Define which solvers to use
11     List<Class<? extends AbstractPricingProblemSolver<CuttingStock,
           CuttingPattern, PricingProblem>>> solvers= Collections.
           singletonList(ExactPricingProblemSolver.class);
12
13     //Define an upper bound (stronger is better). In this case we simply
           sum the demands, i.e. cut each final from its own raw (Rather poor
           initial solution).
14     int upperBound=IntStream.of(dataModel.demandForFinals).sum();
15
16     //Create a set of initial Columns.
17     List<CuttingPattern> initSolution=this.getInitialSolution(
           pricingProblem);
18
19     //Lower bound on column generation solution (stronger is better):
           calculate least amount of finals needed to fulfil the order (ceil(\
           sum_j d_j*w_j /L)
20     double lowerBound= Math.ceil(1.0* IntStream.range(0, dataModel.nrFinals
           ).mapToObj(i -> dataModel.demandForFinals[i] * dataModel.finals[i])
           .mapToInt(i -> i).sum() / dataModel.rollWidth);
21
22     //Create a column generation instance
23     ColGen<CuttingStock, CuttingPattern, PricingProblem> cg=new ColGen<>(
           dataModel, master, pricingProblem, solvers, initSolution,
           upperBound, lowerBound);
24
25     //OPTIONAL: add a debugger
26     SimpleDebugger debugger=new SimpleDebugger(cg);
27
28     //OPTIONAL: add a logger
```

```
29      SimpleCGLogger logger=new SimpleCGLogger(cg, new File("./cuttingStock.
            log"));
30
31      //Solve the problem through column generation
32      cg.solve(System.currentTimeMillis()+1000L);
33
34      //Print solution:
35      <Code Omitted>
36
37      //Clean up:
38      cg.close(); //This closes both the master and Pricing Problems
39    }
40
41    /**
42     * Create an initial solution for the Cutting Stock Problem.
43     */
44    private List<CuttingPattern> getInitialSolution(PricingProblem
          pricingProblem){<Code Omitted>}
45  }
```

In the code snippet above, the only new class we have not introduced before is the `ColGen` class. This class manages the entire solve procedure by invoking the master and Pricing Problems iteratively. This class is provided as part of the framework; its basic implementation suffices for most use cases. Nevertheless, when desired each of its methods may be overridden by a custom implementation.

3 Example 2 - Solving Graph Coloring through Branch-and-Price

In this example, we will solve the well-known Graph Coloring problem (GCP) through Branch-and-Price. In GCP, one wants to assign a color to each vertex in an undirected graph, such that no two neighboring vertices have the same color, while minimizing the total number of different colors needed to color the entire graph. Two vertices are neighbors whenever there is an edge between them. The minimum number of colors needed to color a graph in this way is commonly referred to as the *Chromatic number* of a graph. An example is provided in Figure 1. The Branch-and-Price implementation provided in the next subsections is based on the paper by Mehrotra and Trick (1995).
An *Independent Set* in a graph is defined as a set of vertices such that no two vertices are connected by an edge in the graph. In the context of Graph Coloring, all vertices belonging to the same Independent Set may be colored by the same color. Consequently, a feasible coloring of a graph can be interpreted as a partitioning of the graph into disjoint independent sets, such that each vertex belongs to exactly one independent set. For example, in Figure 1, the independent sets $\{\{2,4\},\{1\},\{3\}\}$ represent a feasible coloring.
In the following Sections we will show how this problem can be solved through our Branch-and-Price framework. Techniques used in this example:

- Column generation
- Branch-and-Price

A full implementation can be found in the demo package.

3.1 Problem description

The Master Problem (MP) for GCP is defined as:

$$MP: min \sum_{s \in S} x_s \tag{10}$$

$$\text{s.t.} \sum_{s \in S: i \in s} x_s \geq 1 \qquad \forall i \in V \tag{11}$$

$$x_s \in \{0,1\} \qquad \forall s \in S \tag{12}$$

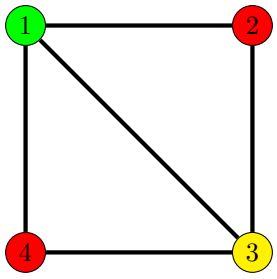


Figure 1: Example of an optimal GCP solution having 3 colors.

Parameter	Description
V	Set of vertices in the graph.
E	Set of undirected edges in the graph.
S	Set of all possible independent sets

Table 2

Binary variables x_s denote whether a particular independent set s is part of the solution or not. Constraint (11) states that each vertex must be covered by an independent set. The Reduced Master Problem is obtained from MP by relaxing integrality constraints on the x_s variables, and by replacing S by $\overline{S}$, where $\overline{S}$ is a subset of S. Associate dual variables π_i with constraints (11). The pricing problem becomes:

$$PRICING : max \sum_{i \in V} \pi_i \tag{13}$$

$$\text{s.t. } z_i + z_j \leq 1 \qquad \forall (i,j) \in E \tag{14}$$

$$z_i \in \{0,1\} \qquad \forall i \in V \tag{15}$$

This pricing problem is also known as a *weighted independent set* problem, where the weight of each vertex is given by the dual values π_i. Binary variables z_i indicate whether a particular vertex $i \in V$ is part of a maximum weight independent set. If the optimal solution to the pricing problem has a value larger than 1, then the z_i with value 1 correspond to an independent set that should be added to $\overline{S}$ in the Reduced Master Problem.
After solving the Reduced Master Problem to optimality, the solution may be fractional, and branching is required. Define the following operations on a Graph Coloring Problem:

- SAME(i,j) requires that two vertices i, j have the same color.
- DIFFER(i,j) requires that two vertices i, j have different colors.

Mehrotra and Trick (1995) show that in any fractional solution there must exist two independent sets $S_1, S_2 \in S$ and two vertices i and j such that $i \in S_1 \cap S_2$ and $j \in S_1 \setminus S_2$. For a given pair of such vertices i, j, we can create two branches: in one branch SAME(i,j) holds, in the other branch DIFFER(i,j) holds. Implementing these branches is simple. For SAME(i,j), we can add the constraint $z_i = z_j$ to the PRICING problem. For DIFFER(i,j), we add $z_i + z_j \leq 1$.

3.2 Implementation

4 Example 3 - Solving the Traveling Salesman Problem through Branch-and-Price

In this example, we will solve the well-known Traveling Salesman Problem (TSP) through Branch-and-Price. In the TSP problem, we are looking for a least cost Hamiltonian cycle in a graph. In this particular example we will assume that the number of vertices in the graph is even, and that the graph is undirected. A *matching* in a graph is defined as a set of edges such that no two edges in the set have an endpoint in common. A matching in a graph is said to be *perfect* if every vertex in the graph is incident to exactly one edge in the matching. By definition, a perfect matching only exists in graphs with an even number of

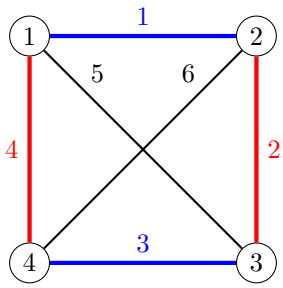


Figure 2: Example of a TSP solution. The blue and red perfect matchings together form a Hamiltonian cycle. The blue perfect matching has a weight of $1 + 3 = 4$, whereas the red perfect matching has a weight of $2 + 4 = 6$. The objective of the solution equals $4 + 6 = 10$

vertices. The weight of a matching is defined as the sum of the weights of the edges in the matching, where the weight of an edge is specified in the underlying graph. We can now interpret the TSP problem as the problem of finding exactly 2 perfect matching (say a red matching and a blue matching) of minimum total weight which together form a Hamiltonian cycle. Figure 2 gives an example. The red edges and the blue edges each form a perfect matching, and together they create a Hamiltonian cycle. The cost of the cycle is equal to the sum of the costs of these two matchings.

In the following Sections we will show how this problem can be solved through our Branch-and-Price framework. Techniques used in this example:

- Column generation
- Branch-and-Price
- Cuts
- Multiple Pricing Problems

A full implementation can be found in the demo package.

4.1 Problem description

Parameter	**Description**
M^r	Set of red matchings.
M^b	Set of blue matchings.
$M = M^r \cup M^b$	Set of matchings, both red and blue.
V	Set of vertices in the graph.
E	Set of undirected edges in the graph.
c_m	Cost of matching $m \in M$, i.e. the sum of its edge weights.
c_e	Cost of edge $e \in E$.
$\delta(S)$	Set of edges with exactly one endpoint in S, $S \subseteq V$.

Table 3

The Master Problem (MP) for TSP is defined as:

$$MP: min \sum_{m \in M} c_m z_m \tag{16}$$

$$\text{s.t.} \sum_{m \in M^r} z_m = 1 \tag{17}$$

$$\sum_{m \in M^b} z_m = 1 \tag{18}$$

$$\sum_{\substack{m \in M: \\ e \in m}} z_m \leq 1 \qquad \forall e \in E \tag{19}$$

$$\sum_{e \in \delta(S)} \sum_{\substack{m \in M: \\ e \in m}} z_m \geq 2 \qquad \forall S \subset V, S \neq \emptyset \tag{20}$$

$$z_m \in \{0, 1\} \qquad \forall m \in M \tag{21}$$

Binary variables z_m, $m \in M$ indicate whether a matching is selected or not. Constraint (17) states that exactly one red matching must be selected. Similarly, (18) states that exactly 1 blue matching must be selected. Constraints (19) ensure that each edge can show up in at most one matching. Finally, subtour elimination constraints (Constraints (20)) ensure that the selected matchings together form a Hamiltonian cycle. For the sake of this example, it is not necessary to fully understand Constraints (20). Since there may exist a very large number of matchings for each color, we will solve the above model through Branch-and-Price. Furthermore, observe that there are potentially a large number of subtour elimination constraints (Constraints (20)): a single constraint for each possible subset of vertices. Instead of generating all these constraints at once, we will generate them on the fly using a separation routine. For details, look up separation of Danzig Fulkerson Johnson subtour elimination constraints for TSP.
When we relax the integrality requirement of Constraints (21), i.e. we replace Constraints (21) by $z_m \geq 0$ for all $m \in M$ we obtain the Relaxed Master Problem:

$$RMP: min \sum_{m \in M} c_m z_m \tag{22}$$

$$\text{s.t.} \sum_{m \in M^r} z_m = 1 \tag{23}$$

$$\sum_{m \in M^b} z_m = 1 \tag{24}$$

$$\sum_{\substack{m \in M: \\ e \in m}} z_m \leq 1 \qquad \forall e \in E \tag{25}$$

$$\sum_{e \in \delta(S)} \sum_{\substack{m \in M: \\ e \in m}} z_m \geq 2 \qquad \forall S \subset V, S \neq \emptyset \tag{26}$$

$$z_m \geq 0 \qquad \forall m \in M \tag{27}$$

Associate dual variables α, β, γ_e, ζ_S with Constraints (22)-(27). The dual of the Relaxed Master Problem, which we will denote as DMP, can then be stated as:

$$DMP: max\ \alpha + \beta + \sum_{e \in E} \gamma_e + 2 \sum_{\substack{S \subset V: \\ S \neq \emptyset}} \zeta_S \tag{28}$$

$$\text{s.t.} \sum_{e \in E} \left(\gamma_e - c_e + \sum_{\substack{S \subset V: \\ S \neq \emptyset; \\ e \in \delta(S)}} \zeta_S \right) \leq -\alpha \qquad \forall m \in M^r \tag{29}$$

$$\sum_{e \in E} \left(\gamma_e - c_e + \sum_{\substack{S \subset V: \\ S \neq \emptyset; \\ e \in \delta(S)}} \zeta_S \right) \leq -\beta \qquad \forall m \in M^b \tag{30}$$

$$\alpha, \beta \in \mathbb{R} \tag{31}$$

$$\gamma_e \leq 0 \qquad \forall e \in E \tag{32}$$

$$\zeta_S \geq 0 \qquad \forall S \subset V, S \neq \emptyset \tag{33}$$

We now obtain two Pricing Problems, one for the red matchings and one for the blue matchings:

1. Does there exist a perfect matching $m \in M^r$ such that $\sum_{e \in E} \left(\gamma_e - c_e + \sum_{\substack{S \subset V: \\ S \neq \emptyset; \\ e \in \delta(S)}} \zeta_S \right) > -\alpha$
2. Does there exist a perfect matching $m \in M^b$ such that $\sum_{e \in E} \left(\gamma_e - c_e + \sum_{\substack{S \subset V: \\ S \neq \emptyset; \\ e \in \delta(S)}} \zeta_S \right) > -\beta$

For each edge $e \in E$, we can define the modified edge costs $\overline{c}_e$ as:

$$\overline{c}_e = \gamma_e - c_e + \sum_{\substack{S \subset V: \\ S \neq \emptyset; \\ e \in \delta(S)}} \zeta_S \tag{34}$$

The Pricing Problems can be modeled through the following MIP formulation which calculates a maximum weight perfect matching:

$$PRICING: max \sum_{e \in E} \overline{c}_e \tag{35}$$

$$\text{s.t.} \sum_{e \in \delta(\{i\})} x_e = 1 \qquad \forall i \in V \tag{36}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E \tag{37}$$

A new matching for the red Pricing Problem is generated if the above algorithm yields a solution with an objective value larger than $-\alpha$. Similarly, a new matching for the blue Pricing Problem is generated if the above algorithm yields a solution with an objective value larger than $-\beta$.
After solving the Relaxed Master Problem (Constraints (22)-(27)) to optimality, we may end up with a fractional solution, i.e. some of the z_m variables may have a fractional value. Consequently, branching is required. If the solution is fractional, then there must exist an edge $e \in E$ such that $0 < \sum_{\substack{m \in M^r: \\ e \in m}} z_m < 1$ or $0 < \sum_{\substack{m \in M^b: \\ e \in m}} z_m < 1$. Indeed, the integrality of the z_m variables implies that an edge $e \in E$ is either part of a red matching, part of a blue matching, or not used at all. Consequently, when there exists an edge $e \in E$ such that $0 < \sum_{\substack{m \in M^r: \\ e \in m}} z_m < 1$, we can create two new problems: one where edge e must be part of any red matching, and one where edge e is *not* used by any red matchings. Similarly, when $0 < \sum_{\substack{m \in M^b: \\ e \in m}} z_m < 1$ holds for an edge $e \in E$, we can again create two new subproblems: one where edge e must be part of any blue matching, and one where edge e is *not* used by any blue matchings.

4.2 Implementation

This section is split into three parts: in the first part we solve the Relaxed Master Problem through Column Generation. In the second part we show how constraints can be added to the master problem

during the solve process (cut generation). In the third part we solve the original Master Problem through Branch-and-Price.

4.2.1 Part 1: Column Generation

Similarly to the Cutting Stock example (Section 2), extensions to the following classes need to be provided to implement the Column Generation model:

1. `AbstractColumn` - This class defines a column in the Column Generation process.
2. `AbstractMaster` - This problem defines the Master Problem.
3. `MasterData` - This class stores data from the Master Problem.
4. `AbstractPricingProblem` - This class defines a Pricing Problem and provides storages for shared data between the Master Problem and the Pricing Problem solvers.
5. `AbstractPricingProblemSolver` - A solver class for the Pricing Problem.

In addition, we assume that there is a class `TSP` which implements the graph as our data model. Finally, instead of generating all the subtour elimination constraints (Constraints (26)) a-priori, we will separate them when they are violated. To this extend, we also need to extend the following two classes:

1. `AbstractInequality` - This class defines an inequality, in our case we will use this class to represent subtour elimination constraints.
2. `AbstractCutGenerator` - This class separates violated inequalities and adds them to the Master Problem.

Lets start by defining the Pricing Problems. We have two Pricing Problems, one for the red and one for the blue matchings:

PricingProblemByColor

```
1  public final class PricingProblemByColor extends AbstractPricingProblem<TSP
       > {
2    /** Color of the Pricing Problem. Can be either Red or Blue **/
3    public final MatchingColor color;
4
5    public PricingProblemByColor(TSP modelData, String name, MatchingColor
         color) {
6      super(modelData, name);
7      this.color=color;
8    }
9  }
```

Next we define a column: a perfect matching.

Matching

```
1  public final class Matching {
2    /** Edges in the matching **/
3    public final Set<Edge> edges;
4    /** Weighted cost of the matching **/
5    public final int cost;
6
7    public Matching(String creator, boolean isArtificial,
         PricingProblemByColor associatedPricingProblem,
8        Set<Edge> edges,
9        int cost) {
10     super(creator, isArtificial, associatedPricingProblem);
11     this.edges=edges;
12     this.cost=cost;
13   }
14
15   @Override
16   public boolean equals(Object o) {<Code Omitted>}
17   @Override
18   public int hashCode() {<Code Omitted>}
```

```
19    @Override
20    public String toString() {<Code Omitted}
21  }
```

Note that it is not necessary to specify the color of the `Matching` since each `Matching` is associated with a specific `PricingProblem`. The color of a `Matching m` can be queried through `m.associatedPricingProblem.color`.
We can now define a solver for our Pricing Problems:

ExactPricingProblemSolver

```
1  public final class ExactPricingProblemSolver extends
       AbstractPricingProblemSolver<TSP, Matching, PricingProblemByColor> {
2    public ExactPricingProblemSolver(TSP dataModel, PricingProblemByColor
         pricingProblem) {
3      super(dataModel, pricingProblem);
4      this.name="ExactMatchingCalculator";
5      matchingSolver=new MatchingSolver();
6    }
7
8    @Override
9    protected List<Matching> generateNewColumns()throws
         TimeLimitExceededException {
10     List<Matching> newPatterns=new ArrayList<>();
11     matchingSolver.solve();
12     if(matchingSolver.getObjective() >= -pricingProblem.dualCost+config.
           PRECISION){ //Generate new column
13       List<Edge> edges=matchingSolver.getEdgesInMatching();
14       int cost=matchingSolver.getCostOfMatching();
15       Matching column=new Matching("exactPricing", false, pricingProblem,
             matching, cost);
16       newPatterns.add(column);
17     }
18     return newPatterns;
19   }
20
21   @Override
22   protected void setObjective() {
23     matchingSolver.setObjective(pricingProblem.dualCosts);
24   }
25   @Override
26   public void close() {<Code Omitted>}
27 }
```

Each instance of the `ExactPricingProblemSolver` is associated with a `PricingProblemByColor` instance (see the constructor of `ExactPricingProblemSolver`). For the red Pricing Problem, the variable `pricingProblem.dualCost` contains the value of the dual variable α, whereas the same field contains the value of the dual variable β for the blue Pricing Problem. Finally, the vector `pricingProblem.dualCosts` contains the modified edge costs (Equation (34)).
The Master Problem implementation is similar to the Master Problem in the Cutting Stock example (Section 2.2). We therefore omit most of the implementation and focus on what's different.

Master

```
1  public final class Master extends AbstractMaster<TSP, Matching,
       PricingProblemByColor, TSPMasterData> {
2
3    public Master(TSP modelData, List<PricingProblemByColor> pricingProblems,
         CutHandler<TSP, TSPMasterData> cutHandler) {
```

```
4       super(modelData, pricingProblems, cutHandler, OptimizationSense.
            MINIMIZE);
5     }
6
7     @Override
8     protected TSPMasterData buildModel() {<Code Omitted>}
9     @Override
10    protected boolean solveMasterProblem(long timeLimit)  throws
          TimeLimitExceededException {<Code Omitted>}
11    @Override
12    public void addColumn(Matching column) {<Code Omitted>}
13    @Override
14    public void initializePricingProblem(PricingProblemByColor pp) {
15      //Store modified edge costs and values of dual variables alpha, beta in
            PricingProblemByColor object
16      double[] modifiedEdgeCosts=...;
17      double alpha=...; double beta=... ;
18      double dualCost = (pp.color==MatchingColor.RED ? alpha : beta);
19      pp.initPricingProblem(modifiedCosts, dualCost);
20    }
21
22    @Override
23    public List<Matching> getSolution() {<Code Omitted>}
24    @Override
25    public void close() {<Code Omitted>}
26  }
```

We have now implemented the Master Problem, Pricing Problem and a solver for this Pricing Problem. In the next Subsection we will discuss how to add a `CutHandler` to deal with the subtour inequalities (Equation 26).

4.2.2 Part 2: Separating valid inequalities

Very often, cuts can be generated to strengthen the Master Problem. Similarly, it could be the case that you do not want to add all inequalities at once to the model because there may be a very large amount of them. Instead, you only want to add them whenever they are violated. The latter is the case for our TSP model: there exist 2^n different subtour elimination constraints (Equation 26)), where n is the number of vertices in the graph. We certainly do not want to add all of these constraints to the initial Master Problem. Instead, we want to generate them in a lazy manner, thereby only generating them whenever such a constraint is violated. To facilitate this kind of functionality, a `CutHandler` is added to the Master Problem. The `CutHandler` maintains a set of Cut Generators. Each Cut Generator checks whether there are violated inequalities of a specific type. In case a Cut Generator finds any violated inequalities, it will add them to the Master Problem. Through the `CutHandler`, each Cut Generator has access to the `AbstractMasterData` object. This is a shared object which is created (`buildModel` method) and maintained by the `Master` class.
To implement our subtour elimination constraint, we need to extend the following two classes:

1. `AbstractInequality` - This class defines an inequality, in our case we will use this class to represent subtour elimination constraints.
2. `AbstractCutGenerator` - This class separates violated inequalities and adds them to the Master Problem.

Recall the subtour inequality (Equation 26)):

$$\sum_{e \in \delta(S)} \sum_{\substack{m \in M: \\ e \in m}} z_m \geq 2 \qquad \forall S \subset V, S \neq \emptyset \tag{38}$$

Each of these inequalities is defined for a subset of vertices $S \subset V$. We can model these inequalities as follows:

SubtourInequality

```
1  public final class SubtourInequality extends AbstractInequality {
2    /** Vertices in the cut set **/
3    public final Set<Vertex> subSet;
4
5    public SubtourInequality(AbstractCutGenerator maintainingGenerator, Set<
         Vertex> subSet) {
6      super(maintainingGenerator);
7      this.subSet=subSet;
8    }
9    @Override
10   public boolean equals(Object o) {<Code Omitted}
11   @Override
12   public int hashCode() {<Code Omitted>}
13 }
```

Each inequality belongs to a Cut Generator. Furthermore, each inequality implements the `equals` and `hashCode` methods. This is to ensure that no duplicate inequalities are generated as this should never happen in practice, unless there is some bug in the implementation. Finally, each inequality has some unique fields, in our example the inequality maintains a subset of vertices S, $S \subset V, S \neq \emptyset$.
The subtour inequalities are generated by the class `SubtourInequalityGenerator` which extends the `AbstractCutGenerator` class:

SubtourInequalityGenerator

```
1  public final class SubtourInequalityGenerator extends AbstractCutGenerator<
       TSP, TSPMasterData> {
2    public SubtourInequalityGenerator(TSP dataModel) {
3      super(dataModel, "subtourIneqGenerator");
4      separator=new SubtourSeparator<>(dataModel);
5    }
6    @Override
7    public List<AbstractInequality> generateInqualities() {
8      //Check for violated subtours. When found, generate an inequality
9      separator.separateSubtour(masterData.edgeValueMap);
10     if(separator.hasViolatedInequality()){
11       SubtourInequality inequality=new SubtourInequality(this, separator.
             getVertexSubSet());
12       this.addCut(inequality);
13       return Collections.singletonList(inequality);
14     }
15     return Collections.emptyList();
16   }
17   @Override
18   private void addCut(SubtourInequality subtourInequality){
19     if(masterData.subtourInequalities.containsKey(subtourInequality))
20       throw new RuntimeException("Duplicate inquality");
21     //Create and add a new constraint to the Master Problem.
22     <Code Omitted>
23     masterData.subtourInequalities.put(subtourInequality,
           subtourConstraintForMaster);
24   }
25   @Override
26   public void addCut(AbstractInequality cut) {
27     if(!(cut instanceof SubtourInequality))
28       throw new IllegalArgumentException("This AbstractCutGenerator can
             ONLY add SubtourInequalities");
29     this.addCut((SubtourInequality) subtourInequality);
30   }
```

```
31    @Override
32    public List<AbstractInequality> getCuts() {
33      return new ArrayList<>(masterData.subtourInequalities.keySet());
34    }
35    @Override
36    public void close() {} //Nothing to do here
37  }
```

The two main methods in this class are:

1. `generateInqualities` - This method computes violated inequalities by using data originating from the Master Problem. In practice this method often relies on some separation routines. When violated inequalities are found, they are added to the Master Problem through the `addCut` method
2. `addCut` - This method adds a violated inequality to the Master Problem. Note that this method does not (and should not) have direct access to the `Master` object; instead it can only access the shared resources in the `MasterData` object. If the Master Problem is implemented in Gurobi or Cplex, it would make sense to store a handle to the corresponding modeling objects in the `MasterData` object such that the Cut Generator can directly add a constraint to the Cplex/Gurobi model.

The CutHandler is already implemented. At any time in the Column Generation or Branch-and-Price procedure, Cut Generators may be registered with or removed from the CutHandler at any time before, during or after the solve procedure. Finally we can bring the aforementioned classes from Part 1 and Part 2 together in a single column generation procedure:

TSPCGSolver

```
1  public final class TSPCGSolver {
2    public TSPCGSolver(TSP tsp){
3      //Create a cutHandler, then create a SubtourInequalityGenerator and add
           it to the handler
4      CutHandler<TSP, TSPMasterData> cutHandler=new CutHandler<>();
5      SubtourInequalityGenerator cutGen=new SubtourInequalityGenerator(tsp);
6      cutHandler.addCutGenerator(cutGen);
7
8      //Create the two Pricing Problems
9      List<PricingProblemByColor> pricingProblems=new ArrayList<>();
10     pricingProblems.add(new PricingProblemByColor(tsp, "redPricing",
           MatchingColor.RED));
11     pricingProblems.add(new PricingProblemByColor(tsp, "bluePricing",
           MatchingColor.BLUE));
12
13     //Create the Master Problem
14     Master master=new Master(tsp, pricingProblems, cutHandler);
15
16     //Define which solvers to use
17     List<Class<? extends AbstractPricingProblemSolver<TSP, Matching,
           PricingProblemByColor>>> solvers= Collections.singletonList(
           ExactPricingProblemSolver.class);
18
19     //Create an initial solution and use it as an upper bound
20     List<Matching> initSolution=... //Create a set of initial
           initialColumns.
21
22     //Lower bound on solution (stronger is better), e.g. sum of cheapest
           edge out of every node.
23     double lowerBound=0;
24
25     //Create a column generation instance
26     ColGen<TSP, Matching, PricingProblemByColor> cg=new ColGen<>(tsp,
           master, pricingProblems, solvers, initSolution,
           tourLengthInitSolution, lowerBound);
```

```
27
28      //OPTIONAL: add a debugger
29      SimpleDebugger debugger=new SimpleDebugger(cg, cutHandler);
30
31      //OPTIONAL: add a logger
32      SimpleCGLogger logger=new SimpleCGLogger(cg, new File("./tspCG.log"));
33
34      //Solve the problem through column generation
35      cg.solve(System.currentTimeMillis()+1000L);
36
37      //Print solution:
38      <Code Omitted>
39
40      //Clean up:
41      cg.close(); //This closes both the master and Pricing Problems
42    }
43  }
```

The `TSPCGSolver` code solves the Relaxed Master Problem (Equations (22)-(27)). The solution to this problem may however be fractional; a Branch-and-Price framework is required to solve this problem. The next Section discusses how a Branch-and-Price solution can be implemented.

4.2.3 Part 3: Branch-and-Price

A basic implementation of the Branch-and-Price framework requires the user to extend the following classes:

1. `AbstractBranchAndPrice` - This class defines how the Branch-and-Price problem is solved.
2. `AbstractBranchCreator` - Given a fractional node in the Branch-and-Price tree, this class performs the branching, thereby creating two ore more child nodes.
3. `BranchingDecision` - An instance of this class describes a single branching decision; it represents an edge in the Branch-and-Price tree.

In addition, each branching decision enforces a number of changes to the Pricing Problems, the Master Problem or both. Lets first look at the two branching decisions we have for our problem:

1. a particular edge should be used by a matching of a particular color
2. a particular edge should *not* be used by a matching of a particular color.

These branching decisions are implemented as follows:

Branching decision 1: FixEdge

```
1  public final class FixEdge implements BranchingDecision<TSP,Matching> {
2      /** Pricing Problem **/
3      public final PricingProblemByColor pricingProblem;
4      /** Edge on which we branch **/
5      public final Edge edge;
6
7      public FixEdge(PricingProblemByColor pricingProblem, Edge edge){
8          this.pricingProblem=pricingProblem;
9          this.edge=edge;
10     }
11
12     @Override
13     public boolean inEqualityIsCompatibleWithBranchingDecision(
           AbstractInequality inequality) {
14         return true;  //In this example we only have subtourInequalities.
               They remain valid, independent of whether we fix an edge.
15     }
16
17     @Override
```

```
18      public boolean columnIsCompatibleWithBranchingDecision(Matching column)
            {
19          return column.associatedPricingProblem != this.pricingProblem ||
                column.edges.contains(edge);
20      }
21
22      @Override
23      public String toString(){<Code Omitted>}
24  }
```

Branching decision 2: RemoveEdge

```
1   public final class RemoveEdge implements BranchingDecision<TSP,Matching> {
2       /** Pricing Problem **/
3       public final PricingProblemByColor pricingProblem;
4       /** Edge on which we branch **/
5       public final Edge edge;
6
7       public RemoveEdge(PricingProblemByColor pricingProblem, Edge edge){
8           this.pricingProblem=pricingProblem;
9           this.edge=edge;
10      }
11
12      @Override
13      public boolean inEqualityIsCompatibleWithBranchingDecision(
            AbstractInequality inequality) {
14          return true;  //In this example we only have subtourInequalities.
                They remain valid, independent of whether we remove an edge.
15      }
16
17      @Override
18      public boolean columnIsCompatibleWithBranchingDecision(Matching column)
            {
19          return column.associatedPricingProblem != this.pricingProblem || !
                column.edges.contains(edge);
20      }
21
22      @Override
23      public String toString(){<Code Omitted>}
24  }
```

Each of these branching decisions implement two important methods:

- `inEqualityIsCompatibleWithBranchingDecision`
- `columnIsCompatibleWithBranchingDecision`

Each node has an associated solution (set of columns) and a set of inequalities which have been separated at that node. Once the node is solved and turns out to yield a fractional solution, the branching process creates a number of child nodes. The solution of the parent node, as well as the inequalities separated at the parent node are transferred to the child nodes to serve as an initial starting point. However, not all columns (inequalities) are compliant with each branching decision. The method `columnIsCompatibleWithBranchingDecision` (`inEqualityIsCompatibleWithBranchingDecision`) determine whether a given column (inequality) is complient with the given branching decision. For example, if we fix a particular edge $e \in E$ for the red Pricing Problem, than all columns for the red Pricing Problem must contain this particular edge. Consequently, any column not containing edge e cannot be part of the initial columns for the red Pricing Problem and hence must be filtered out.
Each Branch-and-Price implementation must implement at least one `AbstractBranchCreator`. A branch creator takes a fractional node in the Branch-and-Price tree and creates two or more child nodes. Two methods have to be implemented:

1. `canPerformBranching` determines whether the particular branch creator can create the child nodes. In the example code below, the `canPerformBranching` determines whether there is a fractional edge to branch on.

2. `getBranches` creates the actual branches

BranchOnEdge

```
1  public final class BranchOnEdge extends AbstractBranchCreator<TSP, Matching
       , PricingProblemByColor>{
2      private Edge edgeForBranching=null; //Edge to branch on
3      private PricingProblemByColor pricingProblemForMatching=null; //Edge is
           fractional in red or blue matching
4
5      public BranchOnEdge(TSP modelData, List<PricingProblemByColor>
           pricingProblems){
6          super(modelData, pricingProblems);
7      }
8
9      @Override
10     protected boolean canPerformBranching(List<Matching> solution) {
11         if(this.blueMatchingsHasFractionalEdge(){
12             //Fix edgeForBranching, pricingProblemForMatching: Code
                   Omitted
13         }else if(this.redMatchingsHasFractionalEdge()){
14     //Fix edgeForBranching, pricingProblemForMatching: Code Omitted
15         }else{
16                 return false;
17         }
18         return true;
19     }
20
21     @Override
22     protected List<BAPNode<TSP,Matching>> getBranches(BAPNode<TSP,Matching>
           parentNode) {
23         //Branch 1: remove the edge:
24         RemoveEdge branchingDecision1=new RemoveEdge(
               pricingProblemForMatching, edgeForBranching);
25         BAPNode<TSP,Matching> node2=this.createBranch(parentNode,
               branchingDecision1, parentNode.getSolution(), parentNode.
               getInequalities());
26
27         //Branch 2: fix the edge:
28         FixEdge branchingDecision2=new FixEdge(pricingProblemForMatching,
               edgeForBranching);
29         BAPNode<TSP,Matching> node1=this.createBranch(parentNode,
               branchingDecision2, parentNode.getSolution(), parentNode.
               getInequalities());
30
31         return Arrays.asList(node1,node2);
32     }
33 }
```

The Branch-and-Price framework will first invoke the `canPerformBranching` method. If this method returns **`true`**, the `getBranches` method is invoked. Multiple branch creators may be supplied to the Branch-and-Price framework. The framework will invoke them one-by-one until either of them produces a set of child nodes. A common example where multiple Branching rules are desirable can be found in certain vehicle routing applications. As a first branching rule, you may want to branch on the number of vehicles used. If for a given solution the number of vehicles in use is fractional, two branches are created where the number of vehicles in use is rounded up/down. In a second branching decision, one may want to branch on a particular edge, e.g. should customer i be visited immediately after customer j or not.

Now we have defined our Branching Decisions we can have a look at how they are used within the framework. A small Branch-and-Price tree is depicted in Figure 3: black nodes have been processed, the green node is the current active node, and the purple nodes are waiting to be processed. The labels on the arcs are Branching Decisions. The Branch-and-Price framework utilizes reversible data structures. Currently, node 3 is being processed. To get to node 3 from the root node, first Branching Decision b_{01} is executed, and then b_{13}. Lets assume that node 3 yields an integer solution, and that node 4 is the next node to be processed. That means that we have to backtrack one level in the Branch-and-Price tree, thereby reverting Branching Decision b_{13}, and that we have to perform Branching Decision b_{14} after that. All of this is already implemented in the Branch-and-Price framework. Any class which needs to update its data structures after a Branching Decision has been executed/reverted, must implement the `BranchingDecisionListener` interface:

BranchingDecisionListener

```
1  public interface BranchingDecisionListener {
2      void branchingDecisionPerformed(BranchingDecision bd);
3      void branchingDecisionReverted(BranchingDecision bd);
4  }
```

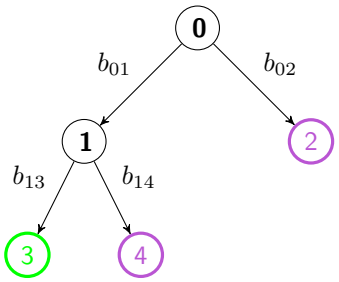


Figure 3: Example of a Branch-and-Price tree: Black nodes have been processed, the green node is the current active node, and purple are nodes which are waiting to be processed.

By default, the *AbstractMaster* and *AbstractPricingProblemSolver* classes already implement this interface, and as a result, they are informed when a Branching Decision has been executed/reverted. Returning to our TSP example, where we have two Branching Decisions, `RemoveEdge` and `FixEdge`, we are now ready to define what happens when either of them gets executed/reverted. When we execute a `RemoveEdge` Branching Decision, then the Pricing Problem defined in this Branching Decision can no longer produce matchings with this edge. Similarly, if we execute a `FixEdge` Branching Decision, then the Pricing Problem defined in this Branching Decision can only produce matchings which contain this edge. To implement this behavior, we need to modify our `ExactPricingProblemSolver` class (see 4.2.1), thereby adding the following two methods:

ExactPricingProblemSolver continued

```
1  @Override
2  public void branchingDecisionPerformed(BranchingDecision bd) {
3    if(bd instanceof FixEdge){
4      FixEdge fixEdgeDecision = (FixEdge) bd;
5      if(fixEdgeDecision.pricingProblem == this.pricingProblem)
6        matchingSolver.fixEdge(fixEdgeDecision.edge); //Columns returned must
             contains this edge.
7    }else if(bd instanceof RemoveEdge){
8      RemoveEdge removeEdgeDecision= (RemoveEdge) bd;
9      if(removeEdgeDecision.pricingProblem == this.pricingProblem)
10       matchingSolver.removeEdge(removeEdgeDecision.edge); //Columns
             returned CANNOT contain this edge.
11   }
12 }
13
```

```
@Override
public void branchingDecisionReverted(BranchingDecision bd) {
  if(bd instanceof FixEdge){
    FixEdge fixEdgeDecision = (FixEdge) bd;
    if(fixEdgeDecision.pricingProblem == this.pricingProblem)
      matchingSolver.undoFixEdge(fixEdgeDecision.edge);
  }else if(bd instanceof RemoveEdge){
    RemoveEdge removeEdgeDecision= (RemoveEdge) bd;
    if(removeEdgeDecision.pricingProblem == this.pricingProblem)
      matchingSolver.undoRemoveEdge(removeEdgeDecision.edge);
  }
}
```

Also in the `Master` class, a minor modification has to be made. The Master Problem currently maintains a set of columns. Some of these columns may not longer be valid when we start processing a new node. Two straightforward solutions exist:

1. if a Branching Decision is executed, we iterate over all columns currently active in the Master model. For each column we test whether it is compliant with the Branching Decision. If not, we remove the column from the Master Problem. Remember that this compliance test can be performed using the `columnIsCompatibleWithBranchingDecision` method in the `BranchingDecision` class.
2. simply destroy the current Master Problem and rebuild it.

For the sake of simplicity we opt to go for the second approach, thereby adding the following method to the `Master` class:

Master continued

```
@Override
public void branchingDecisionPerformed(BranchingDecision bd) {
  this.close(); //Invoke the destructor
  masterData=this.buildModel(); //Create a new model without any
      initialColumns
  cutHandler.setMasterData(masterData); //Inform the cutHandler about the
      new master model
}

@Override
public void branchingDecisionReverted(BranchingDecision bd) { } //No action
     required
```

The last class to extend is the `AbstractBranchAndPrice` class. This requires you to implement the following two methods:

- `generateArtificialSolution` Details on this method are provided in Section 5.5.
- `isIntegralSolution` this method determines whether a particular solution is an integer solution. In our case, an integer solution consist of 2 columns: one red matching and one blue matching, so we can safely conclude that if the master returns only 2 columns with a non-zero z_m value, that the solution must be an integer solution.

The following code block gives the `BranchAndPrice` implementation.

BranchAndPrice

```
public final class BranchAndPrice extends AbstractBranchAndPrice<TSP,
    Matching, PricingProblemByColor> {
    public BranchAndPrice(TSP modelData,
                          Master master,
                          List<PricingProblemByColor> pricingProblems,
```

```
5                           List<Class<? extends AbstractPricingProblemSolver
                                <TSP, Matching, PricingProblemByColor>>>
                                solvers,
6                           List<? extends AbstractBranchCreator<TSP,
                                Matching, PricingProblemByColor>>
                                branchCreators,
7                           int objectiveInitialSolution,
8                           List<Matching> initialSolution){
9           super(modelData, master, pricingProblems, solvers, branchCreators,
                0, objectiveInitialSolution);
10          this.warmStart(objectiveInitialSolution, initialSolution);
11      }
12
13      @Override
14      protected List<Matching> generateArtificialSolution() {<Code Omitted>}
15
16      @Override
17      protected boolean isIntegerNode(BAPNode<TSP,Matching> node) {return
            node.getSolution().size()==2;}
18  }
```

Finally, we can bring all the parts together in a solver class:

TSPSolver

```
1   public final class TSPSolver {
2     public TSPSolver(TSP tsp){
3       //Create a cutHandler, then create a Subtour AbstractInequality
            Generator and add it to the handler
4       CutHandler<TSP, TSPMasterData> cutHandler=new CutHandler<>();
5       SubtourInequalityGenerator cutGen=new SubtourInequalityGenerator(tsp);
6       cutHandler.addCutGenerator(cutGen);
7
8       //Create the two Pricing Problems
9       List<PricingProblemByColor> pricingProblems=new ArrayList<>();
10      pricingProblems.add(new PricingProblemByColor(tsp, "redPricing",
            MatchingColor.RED));
11      pricingProblems.add(new PricingProblemByColor(tsp, "bluePricing",
            MatchingColor.BLUE));
12
13      //Create the Master Problem
14      Master master=new Master(tsp, pricingProblems, cutHandler);
15
16      //Define which solvers to use
17      List<Class<? extends AbstractPricingProblemSolver<TSP, Matching,
            PricingProblemByColor>>> solvers= Collections.singletonList(
            ExactPricingProblemSolver.class);
18
19      //OPTIONAL: Get an initial solution and use it as an upper bound
20      List<Matching> initSolution=<Code Omitted>
21
22      //Define one or more Branch creators
23      List<? extends AbstractBranchCreator<TSP, Matching,
            PricingProblemByColor>> branchCreators= Collections.singletonList(
            new BranchOnEdge(tsp, pricingProblems));
24
25      //Create a Branch-and-Price instance
26      BranchAndPrice bap=new BranchAndPrice(tsp, master, pricingProblems,
            solvers, branchCreators, initObjective, initSolution);
27
28      //OPTIONAL: Attach a debugger
```

```
29      SimpleDebugger debugger=new SimpleDebugger(bap, cutHandler, true);
30
31      //OPTIONAL: Attach a logger to the Branch-and-Price procedure.
32      SimpleBAPLogger logger=new SimpleBAPLogger(bap, new File("./tsp.log"));
33
34      //Solve the TSP problem through Branch-and-Price
35      bap.runBranchAndPrice(System.currentTimeMillis()+8000000L);
36
37      //Print solution:
38      <Code Omitted>
39
40      //Clean up:
41      bap.close(); //Close master and Pricing Problems
42      cutHandler.close(); //Close the cut handler. The close() call is
            propagated to all registered AbstractCutGenerator classes
43    }
44  }
```

5 Implementation details

5.1 Column Generation solve method

Figure 4 gives a possible flow of the `solve` method in the `ColGen` class. First the Master Problem is solved. Then a check is performed whether the solve procedure may be terminated based on any of the bounds (see Section 5.3). After that the Pricing Problem is solved. Two algorithms are provided for this purpose: if the first algorithm fails to identify columns with negative reduced cost, the second algorithm gets invoked. After one or more columns are returned, a bound on the optimal solution of the Master Problem may be calculate. When no columns are returned, the Master Problem is per definition solved to optimality. The `CutHandler` is invoked to check whether the current solution of the Master Problem violates any valid inequalities. If this is the case, these inequalities are added to the Master Problem and a new Column Generation iteration is started. If no inequalities are violated, the Column Generation procedure terminates.

5.2 Branch-and-Price solve method

The `AbstractBranchAndPrice` class implements a solve method which processes Branch-and-Price nodes from a queue until the queue is empty, or a time limit is reached. After solving a node, the following 4 conditions may hold:

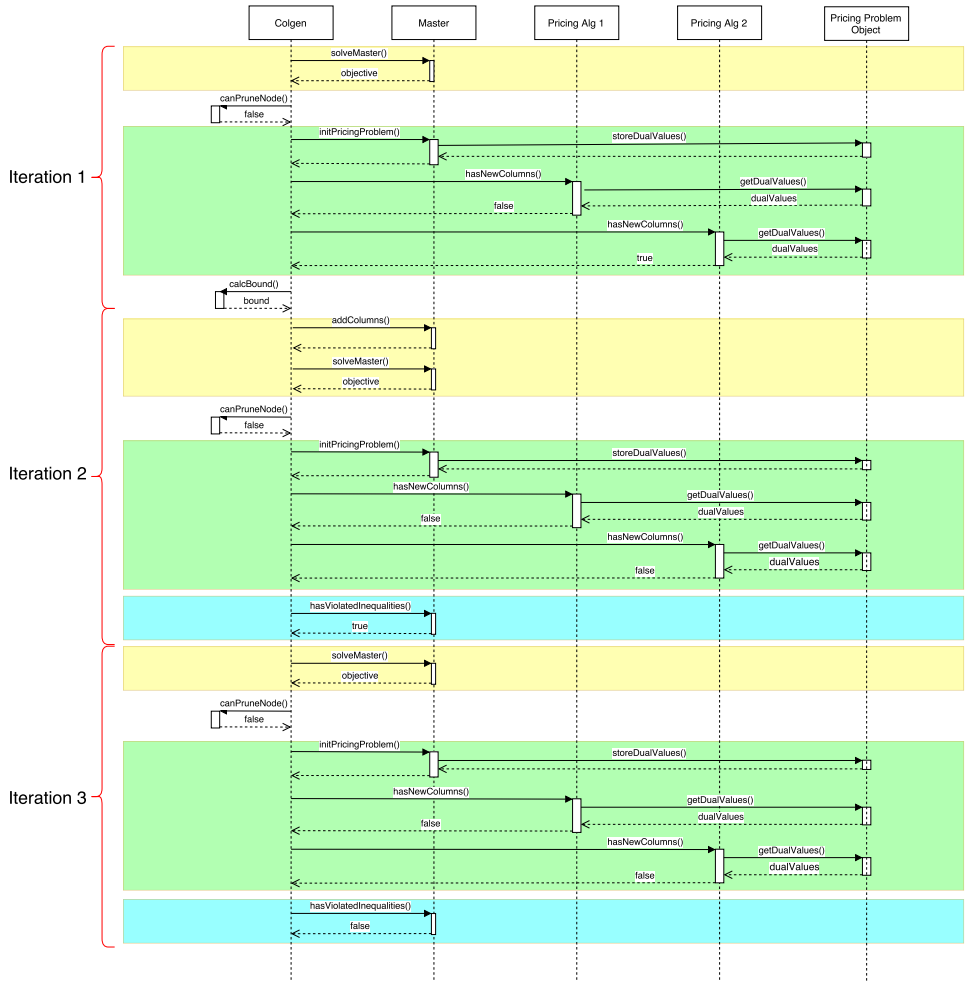
- the node is infeasible (no solution to its Master Problem exists)
- the node is integer. The best integer solution is recorded.
- the node is fractional. The bound on the node is compared with the best integer solution; if the bound exceeds the best integer solution, the node is pruned. Otherwise, branching is performed, thereby creating two or more child nodes which are added to the queue.
- the status of the node is unknown: this may for example occur when the node is being solved while the time limit is reached. The node is placed back into the queue.

5.3 Bounds

To speed up the CG procedure, bounds can be computed on the optimal solution. These bounds are often problem dependent; hence no standard implementation is provided. However, the framework supports the calculation of bounds in various ways. A valid bound on the optimal CG objective is frequently calculated by combining information from the Master Problem, and information from the the Pricing Problems (the optimal solution to the Pricing Problem, or a bound thereof).

- Information from the Master Problem can be queried through the function: `getBoundComponent`.

Figure 4: Flow chart of Column Generation procedure.



- Each `AbstractPricingProblemSolver` has a method `getBound`. If a particular `AbstractPricingProblemSolver` solves the Pricing Problem to optimality, the `getBound` function would typically return the optimal solution. Alternatively, the objective value of a relaxation of the Pricing Problem may be returned, e.g. the LP relaxation when the Pricing Problem is implemented as a MIP, or the value of a Lagrangian Relaxation.
- When there are multiple Pricing Problems, multiple bounds on these Pricing Problems may be queried in parallel through the `getBoundsOnPricingProblems` method in the `PricingProblemManager`. The latter is accessible through the `ColGen` class (or an extension thereof). The number of parallel threads used to calculate these bounds are specified in the `Configuration` class.
- To calculate a bound on the optimal value of the Master Problem, the method `calculateBoundOnMasterObjective` is provided in the `ColGen` class. This method has access to the aforementioned methods.

The default implementation of the class `ColGen` provided in the framework terminates the `solve` procedure under the following conditions:

1. No more columns are returned by the Pricing Problem(s) (and no inequalities are violated): Solved to optimality
2. Time limit exceeded (throws exception)
3. If the optimality gap is closed, i.e. the objective of the Master Problem equals the bound calculated on the optimal solution of the Master Problem (see `getBound` method in ColGen class).

4. If the bound on the Master Problem objective exceeds the cutoff value which was provided during the construction of the `ColGen` instance. In the context of Branch-and-Price, one often has a incumbent solution to the original problem. A node in the Branch-and-Price tree may be pruned if its bound exceeds the value of the incumbent solution. Therefore, an integer solution (cutoff value) may be provided to the Column Generation procedure.

Obviously, any of the above conditions may be modified by overriding the `solve` method with a custom implementation.
Note: bounds computed at various iterations of the column generation procedure are usually not monotonically increasing in strength. That is, the bound calculated at iteration i may be stronger than the bound calculated at iteration j, $j > i$. The framework keeps track of the strongest bound.

5.4 Solving the Pricing Problem(s)

The *Pricing Problem* defines under which conditions a new column has to be added to the *Master Problem*. Each Pricing Problem has to be solved by some algorithm which is defined as a `PricingProblemSolver`. Multiple algorithms may be specified in a hierarchical fashion. One can for example specify a fast heuristic Pricing Problem solver. If the heuristic solver fails, one can fall back on an exact solver.
Very often, a column generation procedure consist of one Master Problem, and multiple independent Pricing Problems. The Pricing Problems are automatically solved in parallel; no interference of the user is required (the number of threads for parallel execution can be specified through the `Configuration` class). An example of a problem with multiple, independent Pricing Problems is given in Section 4.

5.5 Initial solution

Any node in the Branch-and-Price tree must be initialized with an initial feasible solution, i.e. a subset of columns which provide an initial feasible solution to the restricted Master Problem. Alternatively, when such a set of columns does not exist, one needs to prove infeasibility of the Master Problem. The root node can be straightforwardly initialized by any feasible solution to the optimization problem, but finding an initial feasible solution for any of its siblings tends to be difficult. Moreover, proving that such a solution does not exist is a cumbersome task. In fact, finding such a solution is about as hard as solving the actual optimization problem you are trying to solve in the first place. Hence it would be beneficial when the Master Problem itself could be used to prove nonexistence of a feasible solution. With this goal in mind, *artificial columns*, which together meet all the constraints of the Master Problem, are introduced. Each artificial column has a cost strictly larger than the cost of any non-artificial column; hence, the Master Problem favors cheaper non-artificial columns over the artificial ones. When the column generation procedure terminates, and the resulting solution still contains artificial columns, one can conclude that no feasible solution exists as they would have been generated by the Pricing Problem. Artificial columns can be added to the Branch-and-Price implementation by setting the *isArtificial* field to true (see the `AbstractColumn` class for details). If a solution to any of the nodes in the Branch-and-Price tree contains an artificial column, the node is automatically pruned and marked as "infeasible".

5.6 Branch-and-Price node ordering

The nodes in a Branch-and-Price tree may be processed in an arbitrary order. The default implementation uses a Depth-First-Search processing order. Nevertheless, a user may provide a custom ordering. To do so, a `Comparator<BAPNode>` must be implemented. Two examples are provided in the classes `BFSbapNodeComparator` and `DFSbapNodeComparator` respectively. The comparator must be provided to the Branch-and-Price implementation through the `setNodeOrdering` method in the class `AbstractBranchAndPrice`.

6 Miscellaneous topics

6.1 Logging and Debugging

Implementing Column Generation applications is hard and prone to errors. Having a good set of debugging tools, preferably without adding print statements everywhere, is a necessity. To simplify the debugging process, a number of tools are available within the framework.

Each class in the framework is equipped with a logger. For details refer to the documentation of Logback (log) and SLF4j (slf). The logger can be used to write debugging output directly to the screen, or redirect the output to a file. Debugging can be selectively enabled and disabled through a Logback configuration file.
Next to a logger, the classes `ColGen`, `AbstractBranchAndPrice` and `CutHandler` are equipped with a notifier which fires events during various stages of the Branch-and-Price and Column Generation procedures. These events can be captured by resp. a `CGListener`, `BAPListener` and `CHListener`. Examples of a logger and a debugger are provided by the `SimpleBAPLogger`, `SimpleCGLogger` and `SimpleDebugger` classes. Information about a specific event fired by a notifier can be extracted from the event that has been fired. Furthermore, the listeners and loggers have access to the `ColGen`, `AbstractBranchAndPrice` and `CutHandler` classes, allowing them to query specific information whenever events are received. The events fired by the notifiers could also be used to develop for example a graphical application which depicts a live representation of the solve process.
All classes providing logging or debugging can be easily extended. Here is an example. Imagine that you want to know exactly which columns are provided to initialize a node somewhere in the middle of the Branch-and-Price tree. Currently, none of the debuggers does this because the number of columns could be large, thereby creating vast amounts of debugging output. One cumbersome implementation would be to overwrite the `solveBAPNode` method in the `AbstractBranchAndPrice` class, thereby adding some print statements to the beginning of the method. These kind of modifications are obviously dangerous because they may potentially modify the flow of the algorithm and could introduce errors. A much more elegant way is to implement a new debugger (see `SimpleDebugger` for an example). Each debugger is essentially a listener which listens for events fired by the notifiers. There is no restriction on the number of listeners. Alternatively to implementing a new debugger, one could simply extend `SimpleDebugger` as follows:

Extending `SimpleDebugger`

```
1  public class InitColumnsDebugger extends SimpleDebugger{
2      public InitColumnsDebugger(AbstractBranchAndPrice bap, boolean
           captureColumnGenerationEventsBAP) {
3          super(bap, true);
4      }
5
6      @Override
7      public void processNextNode(ProcessingNextNodeEvent event) {
8          super.processNextNode(event);
9          for(Column c: (List<Column>)event.node.getInitialColumns())
10             System.out.println(c);
11     }
12 }
```

In the above code snippet, we override a method which handles `ProcessingNextNodeEvent` events. This particular event is fired whenever a new node in the Branch-and-Price tree is about to be processed. Through this event we have access to the node that will be processed, and we can print the columns that will be used to initialize the node.
Finally, the class `SimpleBAPLogger` may be used to write a concise summary of the Branch-and-Price algorithm execution to a file, thereby giving information about the nodes that have been solved, their bound, objective, branching decisions etc. This summary is particularly helpful to check whether Branching rules are implemented correctly, to verify whether bounds are being processed correctly, to check the progress of the algorithm, and to check convergence of the algorithm.

6.2 Configuration

A number of parameters may be configured through the `Configuration` class, including the number of threads used to solve the Pricing Problems, and whether valid inequalities should be separated or not. The configuration files can be accessed through the `AbstractMaster`, `AbstractPricingProblemSolver`, `CutHandler` classes. The user may also specify custom configurations, as well as additional parameters. A list of all parameters and their default values can be found in Table 4. Note that all parameters are defined as `Final` by default. Changes to the parameters can only be made *prior* to the construction of

Parameter	Default	Description ($)
MAXTHREADS	4	Number of threads used by the column generation procedure. This parameter may be passed to the LP solver used to solve the Master problem. The number of threads also controls how many pricing problems are solved in parallel. When there are multiple pricing problems, as for example in the TSP example (Section 4), using multiple threads would enable parallel solving of the pricing problems. Currently, the Branch-and-Price tree is *not* solved in parallel, so increasing the number of threads won't have any effect on the processing speed of the Branch-and-Price tree.
PRECISION	0.000001	Precision is used for rounding purposes. Be very careful when you adjust this parameter! Note that the default precision is the same value used by several popular LP solvers. The precision also controls whether a column will be added to the Master problem, i.e. whether a column has negative reduced cost.
CUTSENABLED	true	Boolean parameter which enables/disables the separation of valid inequalities. Obviously, to separate valid inequalities, one or more separation routines need to be specified (see Section 4.2.2).
QUICK_RETURN_AFTER_CUTS_FOUND	true	The CutHandler invokes the CutGenerator(s) one by one to generate inequalities. When a particular cutGenerator does not yield any inequalities, the cutHandler will move on to the next registered cutGenerator. When quickReturnAfterCutsFound is set to true, the cutHandler will return as soon as any inequality have been found. When set to false, all cutGenerators will be invoked, independent of the number of cuts returned.
EXPORT_MODEL	false	Convenience parameter used for debugging. Often you want to be able to export your Master problem for debugging purposes. This parameter controls whether the Master is exported or not. Refer to the example implementations of the Master problem.
EXPORT_MASTER_DIR	"./output/masterLP/"	Directory where master problems are exported. See EXPORT_MODEL for details.

Table 4: Parameter description

any of the Column Generation or Branch-and-Price classes! To change the parameters, use the following code:

Example on how to change parameter values

```
1  Properties properties=new Properties();
2  properties.setProperty("MAXTHREADS", "5"); //Use 5 threads
3  Configuration.readFromFile(properties);
```

References

"Logback," http://logback.qos.ch/.

"SLF4j," http://www.slf4j.org/.

A. Mehrotra and M. A. Trick, "A column generation approach for graph coloring," *INFORMS Journal on Computing*, vol. 8, pp. 344–354, 1995.