RMA User Guide

Ai Kagawa

Computational Science Initiative, Brookhaven National Laboratory

Last updated: 1/15/2018

RMA is implemented using PEBBL, and [1] explains how to use PEBBL. This chapter additionally explains specific parameters and some main procedures in the RMA solver. An example command to run the RMA solver with a debugging option is:

```
./rma --delta=10 datafile.txt
```

1 Parameters

`delta`

Layer:	Serial and Parallel
Datatype:	`double`
Default value:	`-1.0`
Constraints:	`[0, 1.0)` or `-1.0`

This parameter is δ, a relative tolerance to aggregate close consecutive values, as shown in Section ??. It is scaled by the 95% central confidence interval of the original data distribution.

`binSize`

Layer:	Serial and Parallel
Datatype:	`integer`
Default value:	`-1`
Constraints:	`a positive integer` or `-1`

If this value is set to $L > 0$, the original data are discretized into partitions of L equal intervals.

`limitInterval`

Layer:	Serial and Parallel
Datatype:	`double`
Default value:	`0.1`
Constraints:	`(0, 1.0)`

This parameter limits the maximum fraction of the entire data range in the original non-discretized data values, that can have the same discretized value. It is ρ in Section ??, and is scaled by the 95% central confidence interval of the original data distribution. If an interval, generated by the current δ, violates this limit, the violated range is recursively discretized by shrinking δ until this limit is no longer violated.

`shrinkDelta`

Layer:	Serial and Parallel
Datatype:	`double`
Default value:	`0.95`
Constraints:	`(0, 1.0)`

This parameter specifies the level of shrinking δ when `limitInterval` caught violated.

`limitDistVals`

Layer:	Serial and Parallel
Datatype:	`integer`
Default value:	`-1`
Constraints:	`a nonnegative integer` or `-1`

If this value is less than the distinct value ℓ_j in attribute j after the recursive discretization and `delta > 0`, then data in attribute j is furthermore discretized by partitioning into L equal intervals.

`removeDuplicateObs`

Layer:	Serial and Parallel
Datatype:	`bool`
Default value:	`true`

If `true`, RMA removes duplicated observations based on the discretized explanatory matrix X, and adds the weight of removed observation to the weight of merging observation.

`getInitialGuess`

Layer:	Serial and Parallel
Datatype:	`bool`
Default value:	`true`

If `true`, RMA implements the greedy heuristic in Section **??** to obtain an initial incumbent before the branch-and-bound procedure.

`perCachedCutPts`

Layer:	Serial and Parallel
Datatype:	`double`
Default value:	`0.000001`
Constraints:	`(0.0, 1.0]`

If this parameter value is less than `1.0`, RMA implements cutpoint cashing. The default value is `0.000001`. Thus, RMA generally considers only the cached cutpoints if there is at least one applicable cached cutpoint.

`binarySearchCutVal`

Layer:	Serial and Parallel
Datatype:	`integer`
Default value:	`-1`

If this value is greater than 3 and less than the number of distinct values ℓ_j in an attribute j, RMA implements binary cupoint search for attribute j.

`branchSelection`

Layer:	Serial
Datatype:	`integer`
Default value:	`0`
Constraints:	{`0,1,2`}

When there are optimal tied solutions, the RMA solver lets the option "0" to randomly chooses one, option "1" to select the first one, and option "2" to select the last one among them.

`countingSort`

Layer:	Serial and Parallel
Datatype:	`bool`
Default value:	`false`

The default sorting algorithm is bucket sort. If `true`, it is replaced by counting sort.

`testWt`
Layer: Serial and Parallel
Datatype: `bool`
Default value: `false`
RMA starts with that each observation has a weight, y_i divided by the number of training observations, in default setting. If this parameter is `true`, RMA can set different weights by using a weight data file, i.e. "testWt.txt", which contains different weights for a current dataset. It is specified after the current dataset in command line. The number of weights in the file has to be equal to the number of observations in the current data file. An example command to use this feature is:

```
./rma --testWt=true datafile.txt testW.txt
```

`bruteForceEC`
Layer: Serial and Parallel
Datatype: `bool`
Default value: `false`
If `true`, RMA constructs equivalence classes without using the rotation algorithm. This option was created to demonstrate the benefits of using the rotation algorithm.

`bruteIncumbent`
Layer: Serial and Parallel
Datatype: `bool`
Default value: `false`
If `true`, RMA computes an incumbent by a brute force algorithm. It still utilizes Kadane's algorithm to find an incumbent, but it scans all observations for each distinct value of each attribute. This option is a validation method for the current incumbent computation. If the `bruteForceEC` parameter is `true`, then this parameter is automatically set to be `true` as well.

`perLimitAttrib`
Layer: Serial and Parallel
Datatype: `double`
Default value: `1.0`
Constraints: `(0.0, 1.0]`
This parameter limits the percentage of all attributes that RMA inspects. If the first x percentage of attributes are inspected already, RMA does not inspect the other attributes anymore. Therefore, an optimal solution may not be obtained.

`writeCutPts`
Layer: Serial
Datatype: `bool`
Default value: `false`
If `true`, RMA writes each cutpoint (j, v) chosen in order into a solution file. This information shows which cutpoints were chosen and in which order they are chosen in each branch of the tree.

2 Serial RMA procedures

The user-defined branching and sub-branching classes, `serRMA` and `serRMASub`, are created for the serial branch-and-bound procedures using PEBBL. They are respectively derived from the `branching` and `branchSub` classes of PEBBL.

2.1 Methods in `serRMA` class

Figure 1 shows procedures to read a dataset and to preprocess data in the `serRMA` class.

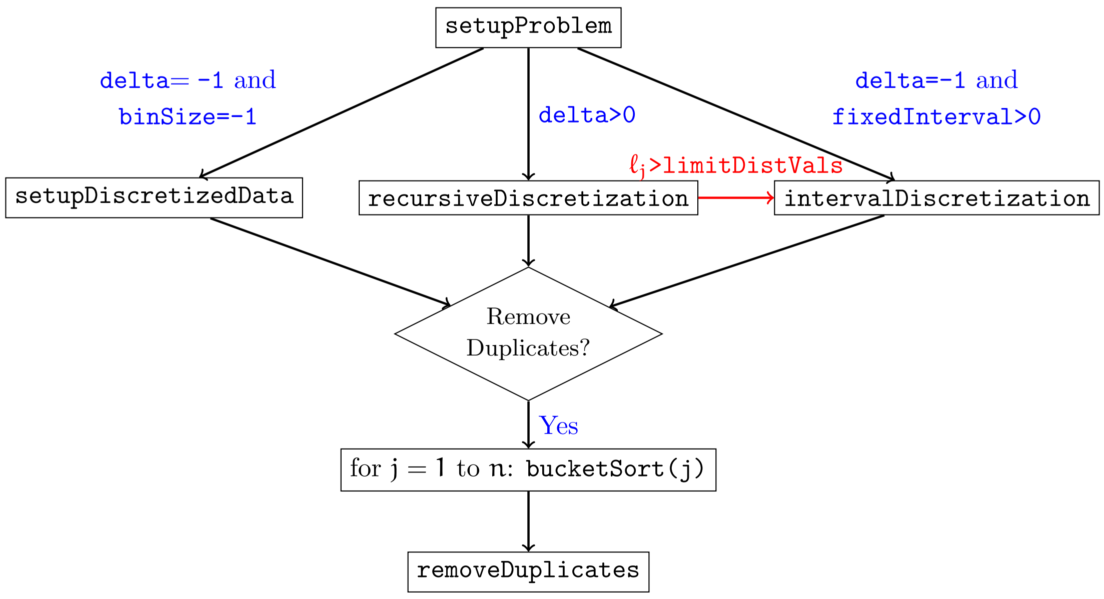

Figure 1: Procedures for reading a data file and preprocessing data

`bool serRMA::setupProblem(int& argc, char**& argv)`
This method reads a data file and stores the dataset information. If explanatory variables are not discretized yet, it can be discretized by the recursive discretization method or the discretization method by fixed intervals.

`bool serRMA::setupDiscretizedData()`
If a dataset is already discretized, this method is invoked to setup the problem set.

`bool serRMA::recursiveDiscretization()`
If the `delta` parameter is greater than `0`, this function recursively discretizes explanatory variables as shown in Section **??**.

`bool serRMA::intervalDiscretization()`
If the `binSize` parameter is not `-1`, this function is invoked to discretize the original data into partitions of L equal intervals.

`solution* serRMA::initialGuess()`
This method is invoked after preprocessing data and before the branch-and-bound procedure. If the `getInitialGuess` parameter is `true` (default), it computes an initial greedy solution, as shown in Section **??**.

2.2 Methods in `serRMASub` class

Figure 2 graphically shows the order of methods invoked to compute the bounds and incumbents in the `serRMASub` class.

`void serRMASub::boundComputation()`
The method computes bounds for given applicable cutpoints by the branching option and selects the best one.

`void serRMASub::createInitialEquivClass()`
The method creates the initial equivalence classes based on given $(\underline{a}, \overline{a}, \underline{b}, \overline{b})$ in each subproblem.

`void serRMASub::computeIncumbent(const int& j)`
This method computes an incumbent using maximization and minimization Kadane's algorithms for attribute j.

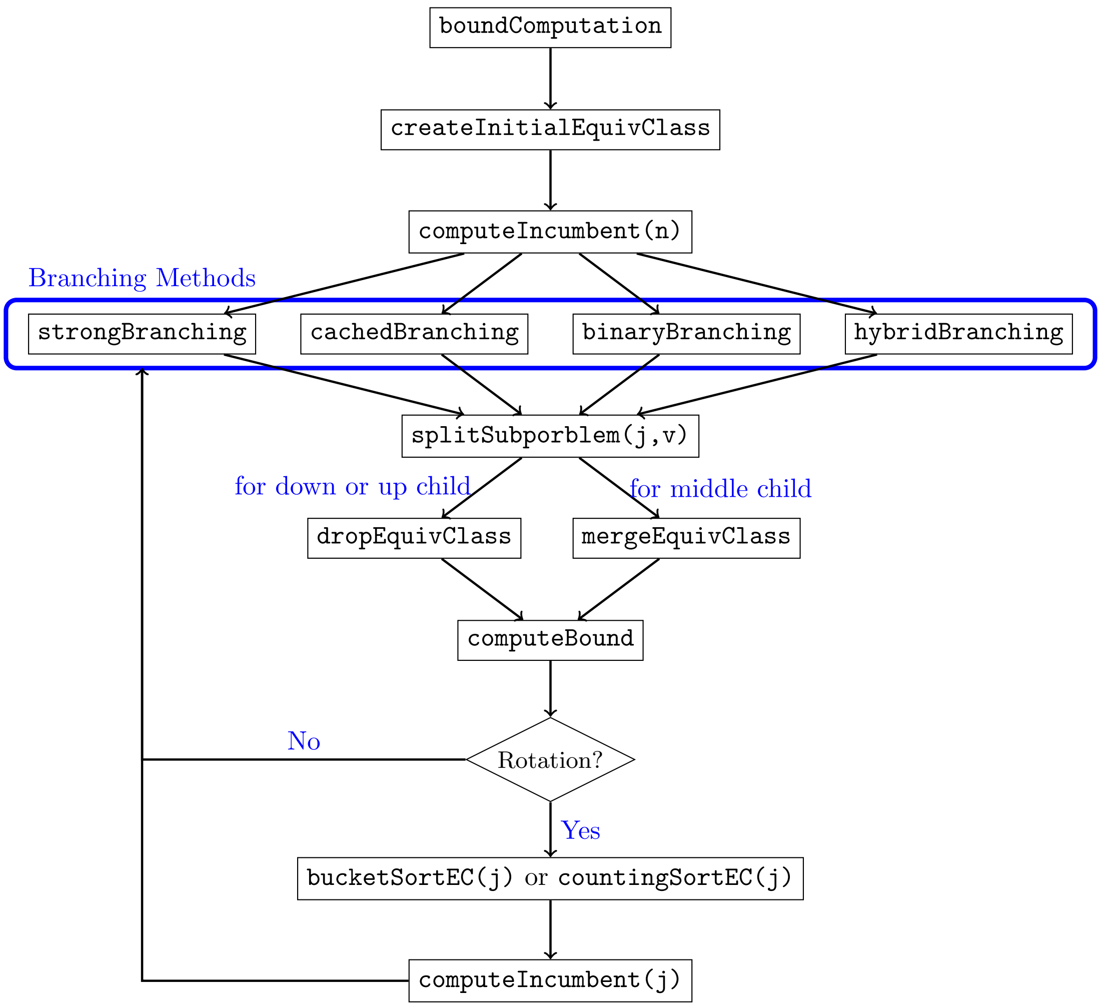

Figure 2: Procedures to compute bounds for child subproblems induced by the current subproblem: the selected branching method provides each cutpoint (j, v) from attribute $j = 1$ to n in this order. Once RMA finishes checking cutpoints in attribute j, it rotates the sorted observation list. Incumbent computations are embedded within the rotation algorithm.

`void serRMASub::strongBranching()`

If the parameters, `perCachedCutpts = 1` and `binarySearchCutVals = -1`, RMA invokes this method to perform strong branching.

`void serRMASub::cachedBranching()`

If the parameters, `perCachedCutpts < 1` and `binarySearchCutVals` = `-1`, as in the default settings, RMA invokes this method to implement cutpoint caching.

`void serRMASub::binaryBranching()`

If the parameters `perCachedCutpts = 1` and `binarySearchCutVals` > `0`, RMA invokes this method to perform binary cutpoint search for the attribute if its applicable cutvalues greater than the `binarySearchCutVals` parameter; otherwise, strong branching.

`void serRMASub::hybrindBranching()`

If the parameters, `perCachedCutpts < 1` and `binarySearchCutVals > 0` , RMA invokes this method to implement hybrid branching. RMA performs binary cutpoint search for the attribute if its applicable cutvalues greater than the `binarySearchCutVals` parameter; otherwise, cutpoint caching.

`void serRMASub::splitSubproblem(const int& j, const int& v)`

This method splits the current subproblem into two or three children by branching at cutpoint (j, v).

`void serRMASub::dropEquivClass()`

For a up or down child, RMA invokes this method to drop equivalence classes no longer covered.

`void serRMASub::mergeEquivClass()`

For a middle child, RMA invokes this method to merge some equivalence classes.

`void serRMASub::computeBound()`

This method computes the bounds of two or three children.

3 Parallel RMA procedures

The `parRMA` and `parRMASub` classes contains methods required for the parallel implementation using PEBBL.

3.1 Methods in `parRMASub` class

`void parRMA::pack()` `void parRMA::unpack()`

These methods in the `parRMA` class pack or unpack, respectively, the problem instance information such as the number of observation, the number of attributes, explanatory variables, response values, weights, the number of distinct features in each attribute, the number of total cutpoints, and applicable parameters for the parallel implementation.

3.2 Methods in `parRMASub` class

`void parRMASub::pack()` `void parRMASub::unpack()`

These methods in the `parRMASub` class pack or unpack, respectively, the four vectors of $(\underline{a}, \overline{a}, \underline{b}, \overline{b})$ for each subproblem and the best local cutpoint (j^*, v^*) to share with the other processors in asynchronous search.

`void parRMASub::boundComputation()`

This method in the `parRMASub` class only computes bounds for assigned cutpoints for each processor in the ramp-up process. In this process, RMA selects the best cutpoint, and broadcasts it to the other processor. Once the ramp-up process ends and PEBBL enters its asynchrous phase, each processor is assigned one subproblem and computes bounds using the `boundComputation` method in the serial class of `RMASub`.

References

[1] Jonathan Eckstein, William E. Hart, and Cynthia A. Phillips. PEBBL: an object-oriented framework for scalable parallel branch and bound. *Math. Program. Comput.*, 7(4):429–469, 2015.