API Documentation

API Documentation

February 2, 2015

Contents

1 Package coinor.grumpy

1.1 Modules

1.2 Variables

Name	**Description**
__package__	**Value:** `'coinor.grumpy'`

2 Module coinor.grumpy.BB

Author: Brady Hunsaker, Osman Ozaltin, Ted Ralphs, Aykut Bulut

2.1 Functions

CreatePerlStyleBooleanFlag(*parser*, *flag_text*, *variable_name*, *help_text*)

```
Add two options to an optparse.OptionParser, one with a 'no' prefix.
Two options are created.  One has the flag_text and one has 'no' prepended
to the flag_text.  For example, --foo and --nofoo.  This is similar to a
common style in Perl.
Args:
  parser: optparse.OptionParser object.
  flag_text: String text for the flag.
  variable_name: String name of the variable to store the flag results.
  help_text: String that describes the flag.
```

parse_options()

Parse arguments and flags

2.2 Variables

Name	Description
__maintainer__	This package is for visualizing branch-and-bound. It also contains a basic branch-and-bound implementation primarily for classroom and educational use. Communication with solvers is through a grammar described in separate documentation. Solvers can interface to this class in a number of different ways and a number of different types of images may be created. Images at intervals that can be specified on the command line as well as after new incumbent solutions are found. Note that the generation of tree images takes significantly longer than other images because every node appears in the image. **Value:** `'Aykut Bulut (aykut@lehigh.edu)'`
BRANCH_STRATEGY	**Value:** `None`
SEARCH_STRATEGY	**Value:** `None`
MOST_FRACTIONAL	**Value:** `'Most Fraction'`
FIXED_BRANCHING	**Value:** `'Fixed Branching'`
PSEUDOCOST_BRANCHING	**Value:** `'Pseudocost Branching'`
DEPTH_FIRST	**Value:** `'Depth First'`
BEST_FIRST	**Value:** `'Best First'`
BEST_ESTIMATE	**Value:** `'Best Estimate'`

continued on next page

Name	**Description**
INFINITY	**Value:** `2147483647`
DOT2TEX_TEMPLATE	**Value:** `'\n\\documentclass[landscape]{article}\n\\usepackage[x11n...`
__package__	**Value:** `'coinor.grumpy'`

2.3 Class BBTree

object

src.gimpy.graph.Graph

src.gimpy.tree.Tree

src.gimpy.tree.BinaryTree

coinor.grumpy.BB.BBTree

Methods to process and visualize information about a b&b tree. It can process an output file (in a specific format, see BAK project in COIN-OR) of a solver that has three information. See run.py in examples directory fot this use. Moreover it implements a branch and bound method that can solve binary programs (0-1 variables only) using PuLP as an LP solver. It provides different branching and searching strategies. See test_strategies.py in test directory.

This is the main class of GrUMPy. It inherits BinaryTree from GIMPy and keeps the entire branch-and-bound tree in self.

2.3.1 Methods

__init__(*self*, ***attrs*)

```
API: __init__(self, **attrs)
Description:
    Class constructor.
Input:
    attrs: Tree attributes in keyword arguments format. See Graph and
    Tree class for details.
```

Overrides: object.__init__ extit(inherited documentation)

process_file(*self*, *file_name*)

write_as_dynamic_gexf(*self*, *filename*, *mode*=`'Dot'`)

set_display_mode(*self*, *mode*)

```
API:
    set_display_mode(self, value)
Description:
    Sets display mode to value.
Input:
    value: New display mode.
Post:
    Display mode attribute of graph is updated.
```

Overrides: src.gimpy.graph.Graph.set_display_mode extit(inherited documentation)

display(*self*, *item*=`'all'`, *basename*=`'graph'`, *format*=`'png'`, *count*=`None`)

Displays/Saves images requested. BranchAndBound method calls this method to visualize the branch and bound tree.

Overrides: src.gimpy.graph.Graph.display

display_all(*self*)

Assumes all the images have the same size.

display_image(*self*, *gnuplot*)

set_label(*self*, *label*)

set_logscaley(*self*, *boolean*)

set_fathom(*self*, *boolean*)

set_edge_limit(*self*, *limit*)

set_sample_tree(*self*, *number*)

AddOrUpdateNode(*self*, *id*, *parent_id*, *branch_direction*, *status*, *lp_bound*, *integer_infeasibility_count*, *integer_infeasibility_sum*, ***attrs*)

This method designed to update nodes (in BAK) but we use it for updating/adding arcs. This is because of the tree data structure the authors adopted in BAK. We can divide these attributes such that some will belong to the edge parent_id->id and the others belong to the id node. The following shows whether the attribute belongs to edge or node. branch direction -> edge status -> node lp_bound -> node integer_infeasibility_count -> node integer_infeasibility_sum -> node parent_id -> node

IsBetterThan(*self*, *value1*, *value2*)

```
Returns True if value1 is better than value2 as an objective value.
This depends on the optimization sense of the instance.
Args:
  value1: Float.
  value2: Float.
Returns:
  True if value1 is better than value2 as an objective value.
```

IsBetterThanIncumbent(*self*, *value*)

```
Returns True if the passed value is better than current incumbent.
Args:
  value: Float to use for comparison.
Returns:
  True if the passed value is better than the current incumbent.
  'Better' is determined by the sense of optimization.
```

UpdateObjectiveValueLimits(*self*, *value*)

```
Updates the min and max objective values if appropriate.
Args:
  value: Float objective value.
```

AddProgressMeasures(*self*)

GetImageCounterString(*self*)

Returns a string with the image counter.

WriteHistogramScript(*self*, *num_bins*, *bin_width*, *max_bin_count*, *lp_bound*, *data_filename*, *output_file*)

```
Write a Gnuplot script file to generate a histogram image.
Args:
  num_bins: Integer number of bins for the histogram.
  bin_width: Float width of the bins in terms of objective values.
  max_bin_count: Integer number of the highest bin count.
  lp_bound: Float value of the current LP bound.
  data_filename: String name of the file; used for display purposes.
```

AdjustHistogramEndBins(*self*, *objective_list*, *num_bins*, *bin_width*, *bin_counts*, *bin_centers*, *bin_widths*)

```
Adjusts the two end bins if necessary to make them narrower.
The two end bins may need to be narrower than the other bins so that
they do not go past the current incumbent value on one end and the
current lp bound on the other.  So that the histogram is still correct
in areas, the height of these bins needs to be adjusted so that the
area does not change.

Note that there is likely to be some bias toward taller bins on
the ends since they always have a point at one end of their width.  It
may be more accurate visually to ignore or discount that one point when
determining the bin height, but that is not currently done.

Args:
  objective_list: List of float objective values.
  num_bins: Integer number of bins.
  bin_width: Float standard width of bins in terms of objective values.
  bin_counts: List of integer counts for each bin.
  bin_centers: List of float coordinates for the center of each bin.
  bin_widths: List of float widths for bins, allowing for individualized
    widths.
```

GenerateHistogram(*self*, *output_file*=False)

```
Generate files necessary for a histogram image.
Two files are necessary: a data file and a Gnuplot script file (which
references the data file).
Args:
  time: Float number of seconds since the start of optimization.
```

GetImageObjectiveBounds(*self*, *min_value*, *max_value*)

```
Return min and max bounds to be used for images.
Images should use bounds that are slightly wider than observed
objective values.  Also, the special case of a single value must be
handled.
Args:
  min_value: Float minimum objective value.
  max_value: Float maximum objective value.
Returns:
  A tuple of two float values (lower_bound, upper_bound).
```

WriteScatterplotScript(*self*, *data_filename*, *output_file*)

```
Write a Gnuplot script file to generate a scatterplot image.
Args:
  data_filename: String name of the file; used for display purposes.
```

GenerateScatterplot(*self*, *output_file*=`False`)

```
Generate files necessary for a scatterplot image.
Two files are necessary: a data file and a Gnuplot script file (which
references the data file).
Args:
    output_file: if not given the gnuplot image will not be written
to disk but returned (to be displayed in pygame window)
```

WriteIncumbentPathScript(*self*, *data_filename*)

```
Write a Gnuplot script file to generate an incumbent path image.
Args:
  data_filename: String name of the file; used for display purposes.
```

WriteAllIncumbentPathsScript(*self*)

```
Return a Gnuplot script string to generate an incumbent path image.
Args:
  data_filenames: List of string names of files.
```

GenerateIncumbentPath(*self*)

Generate files necessary for an incumbent scatterplot path image. Two files are necessary: a data file and a Gnuplot script file (which references the data file).

GenerateAllIncumbentPaths(*self*)

Generate file for a path image with all incumbent paths. Data files were previously generated for each incumbent. This re-uses those files.

WriteTreeScript(*self*, *additional_lines*=`None`)

```
Write a Gnuplot script file to generate a tree image.
Args:
  additional_lines: String with additional lines to be added to the
    script file.
```

GetTreeFixedHorizontalPositions(*self*)

```
Returns horizontal positions for all nodes based on fixed positions.
Returns:
  Dictionary of float horizontal positions, keyed by node id.
```

GetTreeHorizontalPositions(*self*)

```
Returns horizontal positions for all nodes.
Each node is given equal horizontal space.
Returns:
  Dictionary of float horizontal positions, keyed by node id.
```

WriteDataFileFromList(*self*, *filename*, *data_list*)

```
Write a list of string data to a file with one entry per line.
Args:
  filename: String filename to open.
  data_list: List of string values to write.
```

GenerateTreeImage(*self*, *fixed_horizontal_positions*=`False`)

Generate files necessary for a tree image. Two files are necessary: a data file and a Gnuplot script file (which references the data file).

ProcessLine(*self*, *line*)

```
Process a line of the input file, generating images if appropriate.
Parses the line, updates internal data structures, and creates images
if appropriate.
Args:
  line: String input line to process.
```

ProcessHeuristicLine(*self*, *remaining_tokens*)

```
Core processing for a line of type 'heuristic'.
Args:
  remaining_tokens: List of string tokens. These are those that remain
    after any common tokens are processed.
```

ProcessIntegerLine(*self*, *node_id*, *parent_id*, *branch_direction*, *remaining_tokens*)

```
Core processing for a line of type 'integer'.
Args:
  node_id: String node id.
  parent_id: String node id of parent.
  branch_direction: String of 'L' or 'R' indicating whether this node
  is the left or right child of its parent.
  remaining_tokens: List of string tokens. These are those that remain
    after any common tokens are processed.
```

ProcessFathomedLine(*self*, *node_id*, *parent_id*, *branch_direction*, *remaining_tokens*)

```
Core processing for a line of type 'fathomed'.
Args:
  node_id: String node id.
  parent_id: String node id of parent.
  branch_direction: String of 'L' or 'R' indicating whether this node is
    the left or right child of its parent.
  remaining_tokens: List of string tokens. These are those that remain
    after any common tokens are processed.
```

ProcessPregnantLine(*self*, *node_id*, *parent_id*, *branch_direction*, *remaining_tokens*)

```
Core processing for a line of type 'pregnant'.
Args:
  node_id: String node id.
  parent_id: String node id of parent.
  branch_direction: String of 'L' or 'R' indicating whether this node is
    the left or right child of its parent.
  remaining_tokens: List of string tokens. These are those that remain
    after any common tokens are processed.
```

ProcessBranchedLine(*self*, *node_id*, *parent_id*, *branch_direction*, *remaining_tokens*)

```
Core processing for a line of type 'branched'.
Args:
  node_id: String node id.
  parent_id: String node id of parent.
  branch_direction: String of 'L' or 'R' indicating whether this node
  is the left or right child of its parent.
  remaining_tokens: List of string tokens. These are those that remain
    after any common tokens are processed.
```

ProcessInfeasibleLine(*self*, *node_id*, *parent_id*, *branch_direction*, *remaining_tokens*)

```
Core processing for a line of type 'infeasible'.
Args:
  node_id: String node id.
  parent_id: String node id of parent.
  branch_direction: String of 'L' or 'R' indicating whether this node is
    the left or right child of its parent.
  remaining_tokens: List of string tokens. These are those that remain
    after any common tokens are processed.
```

ProcessCandidateLine(*self*, *node_id*, *parent_id*, *branch_direction*, *remaining_tokens*)

```
Core processing for a line of type 'candidate'.
Args:
  node_id: String node id.
  parent_id: String node id of parent.
  branch_direction: String of 'L' or 'R' indicating whether this node
  is the left or right child of its parent.
  remaining_tokens: List of string tokens. These are those that remain
    after any common tokens are processed.
```

RunGnuplotOnAllFiles(*self*)

Runs Gnuplot on all files in self._gnuplot_files.

CreateAnimatedImages(*self*)

Create animated images based on the static images.

GeneratePredictionImages(*self*)

GenerateForecastImages(*self*)

GenerateRandomMIP(*self*, *numVars*=`40`, *numCons*=`20`, *density*=`0.2`, *maxObjCoeff*=`10`, *maxConsCoeff*=`10`, *tightness*=`2`, *rand_seed*=`2`)

BranchAndBound(*self*, *CONSTRAINTS*, *VARIABLES*, *OBJ*, *MAT*, *RHS*, *branch_strategy*=`'Most Fraction'`, *search_strategy*=`'Depth First'`, *complete_enumeration*=`False`, *display_interval*=`None`, *binary_vars*=`True`)

Inherited from src.gimpy.tree.BinaryTree

add_left_child(), add_right_child(), add_root(), bfs(), del_node(), dfs(), get_left_child(), get_right_child(), postordereval(), print_nodes(), printexp(), traverse()

Inherited from src.gimpy.tree.Tree

add_child(), get_children(), get_parent()

Inherited from src.gimpy.graph.Graph

__contains__(), __repr__(), add_edge(), add_node(), augment_cycle(), check_edge(), create(), create_cluster(), create_residual_graph(), cycle_canceling(), del_edge(), edge_to_string(), fifo_label_correcting(), find_cycle_capacity(), find_feasible_flow(), floyd_warshall(), floyd_warshall_get_cycle(), floyd_warshall_get_path(), get_degree(), get_diameter(), get_edge_attr(), get_edge_cost(), get_edge_list(), get_edge_num(), get_in_neighbors(), get_layout(), get_negative_cycle(), get_neighbors(), get_node(), get_node_attr(), get_node_list(), get_node_num(), get_out_neighbors(), get_simplex_solution_graph(), label_components(), label_correcting_check_cycle(), label_correcting_get_cycle(), label_strong_component(), max_flow(), max_flow_preflowpush(), min_cost_flow(), minimum_spanning_tree_kruskal(), minimum_spanning_tree_prim(), network_simplex(), page_rank(), print_flow(), process_edge_dijkstra(), process_edge_flow(), process_edge_prim(), process_edge_search(), process_node_search(), random(), relabel(), search(), set_edge_attr(), set_layout(), set_node_attr(), show_flow(), simplex_augment_cycle(), simplex_compute_potentials(), simplex_connect(), simplex_determine_leaving_arc(), simplex_find_cycle(), simplex_find_tree(), simplex_identify_cycle(), simplex_mark_entering_arc(), simplex_mark_leaving_arc(), simplex_mark_st_arcs(), simplex_optimal(), simplex_redraw(), simplex_remove_arc(), simplex_search(), simplex_select_entering_arc(), strong_connect(), tarjan(), to_string(), write()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

2.3.2 Properties

Name	Description
Inherited from object	
__class__	

3 Module coinor.grumpy.forecasting

Author: Brady Hunsaker, Osman Ozaltin

3.1 Variables

Name	Description
__maintainer__	Double exponential smoothing forecasting in chained sequences. A single sequence is stored in a DoubleExponentialSmoothingForecaster, which also provides forecasts for time to completion based on the stored measures, which should be monotonically decreasing. Sequences are chained together in a ForecastingChainedSequences object. Such an object includes scale factors for each sequence. The scale factors will be applied to the measurements, usually for the purpose of making the chained sequence monotonically decreasing. **Value:** `'Brady Hunsaker (bhunsaker@google.com)'`
__package__	**Value:** `'coinor.grumpy'`

3.2 Class ProgressMeasurement

```
object ─┐
        coinor.grumpy.forecasting.ProgressMeasurement
```

Key data recording progress. Data members are public.

3.2.1 Methods

__init__(*self*, *time*, *value*, *active_node_count*, *node_count*)

x.__init__(...) initializes x; see help(type(x)) for signature

Overrides: object.__init__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

3.2.2 Properties

Name	Description
Inherited from object	
__class__	

3.3 Class TimeForecast

object ⌐
coinor.grumpy.forecasting.TimeForecast

Time-stamped forecast of time remaining.

3.3.1 Methods

__init__(*self*, *time*, *forecast*)

x.__init__(...) initializes x; see help(type(x)) for signature

Overrides: object.__init__ extit(inherited documentation)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

3.3.2 Properties

Name	Description
Inherited from object	
__class__	

3.4 Class DoubleExponentialSmoothingForecaster

```
object ─┐
        │
        coinor.grumpy.forecasting.DoubleExponentialSmoothingForecaster
```

Uses double exponential smoothing to forecast values.

3.4.1 Methods

__init__(*self*, *scale_factor*, *first_value*, *first_time*)

x.__init__(...) initializes x; see help(type(x)) for signature

Overrides: object.__init__ extit(inherited documentation)

AddMeasure(*self*, *measurement*)

ComputeForecast(*self*)

GetForecasts(*self*)

GetMeasures(*self*)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

3.4.2 Properties

Name	Description
Inherited from object	
__class__	

3.5 Class ForecastingChainedSequences

```
object ─┐
        │
        coinor.grumpy.forecasting.ForecastingChainedSequences
```

3.5.1 Methods

__init__(*self*)

x.__init__(...) initializes x; see help(type(x)) for signature

Overrides: object.__init__ extit(inherited documentation)

AddMeasure(*self*, *time*, *value*, *active_node_count*, *node_count*)

StartNewSequence(*self*, *scale_factor*)

Starts a new sequence of measures.

The scale factor should compare only to the previous sequence.

GetAllForecasts(*self*)

GetAllMeasures(*self*)

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

3.5.2 Properties

Name	Description
Inherited from object	
__class__	

4 Module coinor.grumpy.polyhedron2D

4.1 Variables

Name	Description
__package__	**Value:** `'coinor.grumpy'`

4.2 Class Polyhedron2D

4.2.1 Methods

__init__(*self*, *points*=`None`, *rays*=`None`, *A*=`None`, *b*=`None`)

make_integer_hull(*self*)

determine_hull_size(*self*)

determine_plot_size(*self*, *padding*=`[-1, 2]`)

4.3 Class Figure

4.3.1 Methods

__init__(*self*)

initialize(*self*)

add_polyhedron(*self*, *p*, *color*=`'blue'`, *linestyle*=`'solid'`, *label*=`None`, *show_int_points*=`False`)

add_line_segment(*self*, *point1*, *point2*, *color*=`'blue'`, *linestyle*=`'solid'`, *label*=`None`)

add_line(*self*, *coeffs*, *level*, *xlim*=`None`, *ylim*=`None`, *color*=`'blue'`, *linestyle*=`'solid'`, *label*=`None`)

add_point(*self*, *center*, *radius*=`0.02`, *color*=`'red'`)

add_text(*self*, *loc*, *text*)

set_xlim(*self*, *xlim*)

set_ylim(*self*, *ylim*)

show(*self*)

Index