pybnb

Release 0.7.dev0

unknown

Jul 08, 2021

CONTENTS

pybnb is a parallel branch-and-bound engine written in Python. It designed to run on distributed computing architectures, using mpi4py for fast inter-process communication.

This package is meant to serve as a back-end for problem-specific solution strategies that utilize a branch-and-bound algorithm. The following core functionality is included:

- work load distribution through a central dispatcher
- work task prioritization strategies (e.g., worst bound first, breadth first, custom)
- solver-like log output showing algorithm progress

To use this package, one must implement a branch-and-bound problem by subclassing the *Problem* interface and defining the methods shown in the example below.

```
>>> import pybnb
>>> # define a branch-and-bound problem
>>> class MyProblem(pybnb.Problem):
...     def sense(self): ...
...     def objective(self): ...
...     def bound(self): ...
...     def save_state(self, node): ...
...     def load_state(self, node): ...
...     def branch(self): ...
>>> # solve a problem instance
>>> result = pybnb.solve(MyProblem())
>>> print(result.solution_status)
'optimal'
```

CHAPTER

ONE

GETTING STARTED

1.1 Installation

You can install pybnb with pip:

```
$ pip install pybnb
```

pybnb requires mpi4py to solve problems in parallel. However, it will also solve problems in serial if this module is not available. Thus, mpi4py is not listed as a package requirement, and it may need to be installed in a separate step.

1.2 Complete Example

The code below shows a complete example script that (1) defines a problem, (2) creates a solver, and (3) solves the problem.

```
# simple.py


class Simple(pybnb.Problem):
    def __init__(self):
        self._xL, self._xU = 0, 1

    #
    # required methods
    #
    def sense(self):
        return pybnb.minimize

    def objective(self):
        return round(self._xU - self._xL, 3)

    def bound(self):
        return -((self._xU - self._xL) ** 2)

    def save_state(self, node):
        node.state = (self._xL, self._xU)

    def load_state(self, node):
        (self._xL, self._xU) = node.state
```

(continues on next page)

(continued from previous page)

```
    def branch(self):
        xL, xU = self._xL, self._xU
        xM = 0.5 * (xL + xU)
        child = pybnb.Node()
        child.state = (xL, xM)
        yield child
        child = pybnb.Node()
        child.state = (xM, xU)
        yield child

    #
    # optional methods
    #
    def notify_solve_begins(self, comm, worker_comm, convergence_checker):
        pass

    def notify_new_best_node(self, node, current):
        pass

    def notify_solve_finished(self, comm, worker_comm, results):
        pass


problem = Simple()
solver = pybnb.Solver()
results = solver.solve(problem, absolute_gap=1e-8)
```

To solve the problem in serial, the example script should be launched with the python interpretor:

```
$ python simple.py
```

To solve the problem in parallel, the example script should be launched using the same command as above, only wrapped with `mpiexec` (specifying the number processes):

```
$ mpiexec -n 4 python simple.py
```

Note that the parallel solve implementation used by `pybnb` always designates exactly one process as a dispatcher. If more than one process is involved in a solve, the dispatcher will only manage the global work queue, leaving the processing of all branch-and-bound nodes to the remaining processes. Thus, one should not expect any parallel speedup until at least three processes are used to solve a problem.

1.3 More Examples

`pybnb` is distributed with a number of example problem implementations. Each example can be run in serial or in parallel with the `mpiexec` command. Some examples require additional python packages or external binaries that are not listed as dependencies for `pybnb` (e.g., `pyomo`). See the comments at the top of each example file for a brief explanation.

The examples directory included with the source repository is organized into two top-level directories.

- command_line_problems: Includes basic problem implementations that expose all `pybnb` solver options as

command-line arguments. Simply execute one of the available examples with `--help` as an argument to see the list of available solver options.

  - binary_knapsack.py
  - lipschitz_1d.py (faster with `numba`, but it is optional)
  - bin_packing.py (requires: `pyomo` + `ipopt` binary)
  - rosenbrock_2d.py (requires: `pyomo` + `ipopt` binary)

- scripts: Includes problem implementations along with various usages of `pybnb` ranging from simple to advanced. Some of the examples accept a small set of command-line options, but most `pybnb` solver options are hard-coded and must be manually adjusted within each example file.
  - simple.py
  - range_reduction_pyomo.py (requires: `pyomo` + `ipopt` binary)
  - tsp/tsp_byvertex.py
  - tsp/tsp_byedge.py (requires: `numpy`)

1.4 Defining a Problem

To define a branch-and-bound problem with `pybnb`, one must define a class that implements the *Problem* interface, which includes defining at least the six required methods shown below.

```
import pybnb
class MyProblem(pybnb.Problem):
   def __init__(self): ...
   # required methods
   def sense(self): ...
   def objective(self): ...
   def bound(self): ...
   def save_state(self, node): ...
   def load_state(self, node): ...
   def branch(self): ...
   # optional methods
   def notify_solve_begins(self,
                           comm,
                           worker_comm,
                           convergence_checker):
       ...
   def notify_new_best_node(self,
                            node,
                            current):
       ...
   def notify_solve_finished(self,
                             comm,
                             worker_comm,
                             results):
       ...
```

Note: The *Problem* base class is a purely abstract interface that adds no additional data to a problem implementation.

It is not required to call `Problem.__init__` when defining the `__init__` method on a derived class.

The remainder of this section includes a detailed description of each of the required methods.

- *Problem.sense()*

  This is the easiest method to define for a branch-and-bound problem. It should return the objective sense of the problem, which should always be one of `minimize` or `maximize`, and should not change what it returns over the lifetime of a problem. For instance, to define a problem with an objective value that should be minimized, the implementation would look something like:

```
class MyProblem(pybnb.Problem):
   def sense(self):
       return pybnb.minimize
```

  The *Problem* base class defines two additional convenience methods *Problem.infeasible_objective()* and *Problem.unbounded_objective()* that return *+inf* or *-inf*, depending on the return value of *Problem.sense()*.

- *Problem.bound()*

  This method should return a valid bound for the objective function over the current problem domain (as defined by the current problem state), or it can return *self.unbounded_objective()* if a finite bound can not be determined.

- *Problem.objective()*

  This method should return a value for the objective function that is feasible for the current problem domain (as defined by the current problem state), or it can return *self.infeasible_objective()* if a feasible objective value can not be determined.

- *Problem.save_state(node)*

  This method should save any relevant state information about the problem onto the *state* attribute of node argument. If one wishes to utilize the MPI-based parallel solver, the only requirement for what goes into the node state is that it can be serialized using the `pickle` or `dill` modules. By default, `pybnb` is configured to use the `pickle` module for node serialization. See the section titled *Serialization Configuration* for details on how to adjust this and related settings.

- *Problem.load_state(node)*

  This method should load the problem state stored on the *state* attribute of the node argument. The code block below shows an example pair of *save_state* and *load_state* implementations.

```
class MyProblem(pybnb.Problem):
    def __init__(self):
        self._L = 0.0
        self._U = 1.0
    def save_state(self, node):
        node.state = (self._L, self._U)
    def load_state(self, node):
        (self._L, self._U) = node.state
```

- *Problem.branch()*

  This method should partition the problem domain defined by the current user state into zero or more child states and return them on new nodes. A child node can be created by directly instantiating a *pybnb.Node* object. Note that for the branching process to make sense, the bound computed from the child states should improve (or not be worse than) the bound for their parent node. Once the child bound is computed, the solver will issue a warning if

it is found to be worse than the bound from its parent node, as this is indicative of a programming error or other numerical issues.

Note that any child nodes returned from *Problem.branch()* will automatically be assigned the bound and objective from their parent for potential use in determining their prioritization in the global work queue. Users can override this by manually assigning a value to one or both of these node attributes before yielding them from the branch method.

Additionally, further control over the prioritization of a child node can be achieved by setting the *queue_strategy* solve option to "custom", and then directly assigning a value to the *queue_priority* attribute of the child node before it is yielded.

1.5 Solving a Problem

There are two approaches to solving a branch-and-bound problem with pybnb. The first is to simply call the *solve* convenience function. This will create a *Solver* object, call the *Solver.solve* method, and report the results as well as additional timing information about the solve.

```
import pybnb
problem = MyProblem()
results = pybnb.solve(problem,
                      relative_gap=1e-4)
```

The second approach is to manually create a *Solver* object and call the *Solver.solve* method directly.

Both approaches can solve a problem in serial or parallel. The difference is that the *solve* convenience function provides a few additional options that simplify the process of saving solver output and results to a file. Additionally, collecting the timing information reported by this function adds some additional communication overhead to the end of the solve; thus, the second approach of directly using a *Solver* can be more efficient.

1.5.1 Creating a Solver

The following example shows how to create a solver object.

```
import pybnb
solver = pybnb.Solver()
```

By default, the solver will automatically use `mpi4py.MPI.COMM_WORLD` as the communicator, and the rank 0 process will act as the dispatcher. If the `mpi4py` module is not available, this will result in an `ImportError`. The optional keywords *comm* and *dispatcher_rank* can be used to change the default behavior.

When a solver is created with `Solver(comm=None)`, this will disable any attempted import of `mpi4py`, allowing problems to be solved without the use of any parallel functionality. The *comm* keyword can also be assigned a communicator different from `mpi4py.MPI.COMM_WORLD`. If the solver communicator includes more than one process, the *dispatcher_rank* keyword can be assigned a process rank to control which process is designated as the dispatcher. However the solver is initialized, the following assertions hold true for the *is_dispatcher* and *is_worker* attributes of the solver object.

```
if (solver.comm is None) or \
   (solver.comm.size == 1):
    assert solver.is_dispatcher and \
        solver.is_worker
else:
```

(continues on next page)

(continued from previous page)

```
if solver.comm.rank == <dispatcher_rank>:
    assert solver.is_dispatcher and \
        (not solver.is_worker)
else:
    assert (not solver.is_dispatcher) and \
        solver.is_worker
```

1.6 How the Solver Calls the Problem Methods

The following block of pseudocode provides a high-level overview of how the solver calls the methods on a user-defined problem. Highlighted lines show where problem methods are called.

```
1  def solve(problem, ...):
2      #
3      # solve initialization
4      #
5      sense = problem.sense()
6      problem.notify_solve_begins(...)
7      root = Node()
8      problem.save_state(root)
9
10     #
11     # solve loop
12     #
13     while <solve_not_terminated>:
14         node, best_node = dispatcher.update(...)
15         if <conditional_1>:
16            problem.notify_new_best_node(node=best_node,
17                                         current=False)
18         problem.load_state(node)
19         bound = problem.bound()
20         if <conditional_2>:
21             objective = problem.objective()
22             if <conditional_3>:
23                 problem.notify_new_best_node(node=node,
24                                              current=True)
25             if <conditional_4>:
26                 children = problem.branch()
27
28     #
29     # solve finalization
30     #
31     problem.load_state(root)
32     problem.notify_solve_finished(...)
```

Note that during the main solve loop (starting on line 13), it is safe to assume that the six highlighted problem methods between line 13 and line 25 will be called in the relative order shown. The conditions under which these methods will be called are briefly discussed below:

- **<conditional_1>** (line 15): This condition is met when the *best_node* received from the dispatcher is not unbounded and improves upon the best node currently known to the worker process (i.e., has a better objective).

By default, the check for objective improvement is exact, but it can be relaxed by assigning a nonzero value to the *comparison_tolerance* keyword of the `Solver.solve` method.

- **<conditional_2>** (line 20): This condition is met when the bound computed by the problem for the current node makes it eligible for the queue relative to the best objective known to the process. By default, this is true when the bound is better than the best objective by any nonzero amount, but this behavior can be influenced using the *queue_tolerance* keyword of the `Solver.solve` method.
- **<conditional_3>** (line 22): This condition is met when the objective computed by the problem for the current node is not unbounded and improves upon the objective of the best node currently known to the process. By default, the check for improvement is exact, but it can be relaxed by assigning a nonzero value to the *comparison_tolerance* keyword of the `Solver.solve` method.
- **<conditional_4>** (line 25): This condition is met when the objective computed by the problem for the current node is not unbounded, when **<conditional_2>** is still satisfied (based on a potentially new best objective), and when the difference between the node's updated bound and objective satisfies the branching tolerance. By default, the branching tolerance is zero, meaning that any nonzero distance between these two values will satisfy this check, but this can be adjusted using the *branching_tolerance* keyword of the `Solver.solve` method.

CHAPTER

TWO

ADVANCED USAGE

2.1 Setting the Queue Strategy and Solver Tolerances

pybnb uses a default queue strategy that prioritizes improving the global optimality bound over other solve metrics. The *queue_strategy* solve option controls this behavior. See the `QueueStrategy` enum for a complete list of available strategies.

The best queue strategy to use depends on characteristics of the problem being solved. Queue strategies such as "depth" and "lifo" tend to keep the queue size small and reduce the dispatcher overhead, which may be important for problems with relatively fast objective and bound evaluations. Setting the *track_bound* solve option to false will further reduce the dispatcher overhead of these queue strategies. On the other hand, using these strategies may result in a larger number of nodes being processed before reaching a given optimality gap.

The *absolute_gap* and *relative_gap* solve options can be adjusted to control when the solver considers a solution to be optimal. By default, optimality is defined as having an absolute gap of zero between the best objective and the global problem bound, and no relative gap is considered. (*absolute_gap=0*, *relative_gap=None*). To enable a check for relative optimality, simply assign a non-negative value to the *relative_gap* solver option (e.g., *relative_gap=1e-4*). Additionally, a function can be provided through the *scale_function* solver option for computing the scaling factor used to convert an absolute gap to a relative gap. This function should have the signature *f(bound, objective) -> float*. The default scale function is *max{1.0,|objective|}*.

Two additional solve options to be aware of are the *queue_tolerance* and *branch_tolerance*. The *queue_tolerance* setting controls when new child nodes are allowed into the queue. If left unset, it will be assigned the value of the *absolute_gap* setting. It is not affected by the *relative_gap* setting. See the section titled *Continuing a Solve After Stopping* for further discussion along with an example. Finally, the *branch_tolerance* setting controls when the *branch* method is called. The default setting of zero means that any non-zero gap between a node's local bound and objective will allow branching. Larger settings may be useful for avoiding tolerance issues in a problem implementation.

2.2 Terminating a Solve Early

A solve that is launched without the use of *mpiexec* can be terminated at any point by entering *Ctrl-C* (sending the process a *SIGINT* signal). If the signal is successfully received, the solver will attempt to gracefully stop the solve after it finishes processing the current node, and it will mark the `termination_condition` attribute of the solver results object with the `interrupted` status.

Solves launched through *mpiexec* typically can not be gracefully terminated using the *Ctrl-C* method. This is due to the way the MPI process manager handles the *SIGINT* signal. However, the solve can be gracefully terminated by sending a *SIGUSR1* signal to the dispatcher process (this also works for the case when the solve was launched without *mpiexec*). The pid and hostname of the dispatcher process are always output at the beginning of the solve.

```
$ mpiexec -n 4 python simple.py
Starting branch & bound solve:
 - dispatcher pid: <pid> (<hostname>)
...
```

Assuming one is logged in to the host where the dispatcher process is running, the solve can be terminated using a command such as:

```
$ kill -USR1 <pid>
```

2.3 Continuing a Solve After Stopping

It is possible to continue a solve with new termination criteria, starting with the candidate solution and remaining queued nodes from a previous solve. The following code block shows how this can be done.

```
solver = pybnb.Solver()
results = solver.solve(problem,
                       absolute_gap=1e-4,
                       queue_tolerance=1e-8,
                       time_limit=10)
queue = solver.save_dispatcher_queue()
results = solver.solve(problem,
                       best_objective=results.objective,
                       best_node=results.best_node,
                       initialize_queue=queue,
                       absolute_gap=1e-8)
```

For the dispatcher process, the *save_dispatcher_queue* method returns an object of type *DispatcherQueueData*, which can be assigned to the *initialize_queue* keyword of the *solve* method. For processes that are not the dispatcher, this function returns *None*, which is the default value of the *initialize_queue* keyword. The *best_node* attribute of the results object will be identical for all processes (possibly equal to None), and can be directly assigned to the *best_node* solver option.

Note the use of the *queue_tolerance* solve option in the first solve above. If left unused, this option will be set equal to the value of the *absolute_gap* setting (it is not affected by the *relative_gap* setting). The *queue_tolerance* setting determines when new child nodes are eligible to enter the queue. If the difference between a child node's bound estimate and the best objective is less than or equal to the *queue_tolerance* (or worse than the best objective by any amount), the child node will be discarded. Thus, in the example above, the first solve uses a *queue_tolerance* equal to the *absolute_gap* used in the second solve to avoid discarding child nodes in the first solve that may be required to achieve the tighter optimality settings used in the second solve.

Assigning the *objective* attribute of the results object to the *best_objective* solve option is only necessary if (1) the initial solve was given a *best_objective* and the solver did not obtain a best node with a matching objective, or (2) if the initial solve is unbounded. In the latter case, the *best_node* attribute of the results object will be None and the dispatcher queue will be empty, so the unboundedness of the problem can only be communicated to the next solve via the *best_objective* solve option. If one is careful about checking the status of the solution and no initial best objective is used (both recommended), then the *best_objective* solver option can be left unused, as shown below:

```
solver = pybnb.Solver()
results = solver.solve(problem,
                       asolute_gap=1e-4,
                       queue_tolerance=1e-8,
```

(continues on next page)

(continued from previous page)

```
                        time_limit=10)
if results.solution_status in ("optimal",
                               "feasible"):
    queue = solver.save_dispatcher_queue()
    results = solver.solve(problem,
                           best_node=results.best_node,
                           initialize_queue=queue,
                           absolute_gap=1e-8)
```

2.4 Serialization Configuration

The following configuration items are available for controlling how node state is transmitted during a parallel solve:

config item	type	default	meaning
SERIALIZER	str	"pickle"	The serializer used to transform the user-defined node state into a byte stream that can be transmitted with MPI. Allowed values are "pickle" and "dill".
SERIALIZER_PROTOCOL_VERSION	int	pickle.HIGHEST_PROTOCOL	The value assigned to the `protocol` keyword of the pickle or dill `dumps` function.
COMPRESSION	bool	False	Indicates if serialized node state should be compressed using zlib.
MARSHAL_PROTOCOL_VERSION	int	2	The value assigned to the `version` argument of the `marshal.dumps` function. The marshal module is used to serialize all other node attributes besides the user-defined state. It is unlikely that this setting would need to be adjusted.

These settings are available as attributes on the `pybnb.config` object. This object can be modified by the user to, for instance, change the serializer for the user-defined node state to the `dill` module. To do so, one would add the following to the beginning of their code:

```
pybnb.config.SERIALIZER = "dill"
```

Each of these settings can also be modified through the environment by exporting a variable with `PYBNB_` prepended to the attribute name on the config object:

```
export PYBNB_SERIALIZER=pickle
```

The environment is checked during the first import of `pybnb`, so when configurations are applied by directly modifying the `pybnb.config` object, this will override those applied through environment variables. The *pybnb.config.reset(...)* method can be called to restore all configuration options to their default setting (ignoring the environment

if specified).

CHAPTER

THREE

PYBNB.FUTURES

The *pybnb.futures* module stores utilities that are still in the early phase of development. They will typically be fairly well tested, but are subject to change or be removed without much notice from one release to the next.

3.1 Using a Nested Solve to Improve Parallel Performance

The `NestedSolver` object is a wrapper class for problems that provides an easy way to implement a custom two-layer, parallel branch-and-bound solve. That is, a branch-and-bound solve where, at the top layer, a single dispatcher serves nodes to worker processes over MPI, and those workers process each node by performing their own limited branch-and-bound solve in serial, rather than simply evaluating the node bound and objective and returning its immediate children to the dispatcher.

The above strategy can be implemented by simply wrapping the problem argument with this class before passing it to the solver, as shown below.

```
results = solver.solve(
    pybnb.futures.NestedSolver(problem,
                               queue_strategy=...,
                               track_bound=...,
                               time_limit=...,
                               node_limit=...),
    queue_strategy='bound',
    ...)
```

The *queue_strategy*, *track_bound*, *time_limit*, and *node_limit* solve options can be passed into the `NestedSolver` class when it is created to control these aspects of the sub-solves used by the workers when processing a node.

This kind of scheme can be useful for problems with relatively fast bound and objective computations, where the overhead of updates to the central dispatcher over MPI is a clear bottleneck. It is important to consider, however, that assigning large values to the *node_limit* or *time_limit* nested solve options may result in more work being performed to achieve the same result as the non-nested case. As such, the use of this solution scheme may not always result in a net benefit for the total solve time.

Next, we show how this class is used to maximize the parallel performance of the TSP example. Tests are run using CPython 3.7 and PyPy3 6.0 (Python 3.5.3) on a laptop with a single quad-core 2.6 GHz Intel Core i7 processor.

The code block below shows the main call to the solver used in the TSP example, except it has been modified so that the original problem is passed to the solver (no nested solve):

```
  results = solver.solve(
      problem,
      queue_strategy='depth',
```

(continues on next page)

(continued from previous page)

```
    initialize_queue=queue,
    best_node=best_node,
    objective_stop=objective_stop)
```

Running the serial case as follows,

```
$ python -O tsp_naive.py fri26_d.txt
```

on CPython 3.7 we achieve a peak performance of ~19k nodes processed per second, and on PyPy3 6.0 the performance peaks at ~150k nodes processed per second. Compare this with the parallel case (using three workers and one dispatcher),

```
$ mpirun -np 4 python -O tsp_naive.py fri26_d.txt
```

where with CPython 3.7 we achieve a peak performance of ~21k nodes per second, and with PyPy3 6.0 the performance actually drops to ~28k nodes per second (nowhere near the 3x increase one would hope for).

Now consider the TSP example in its original form, where the problem argument is wrapped with the *NestedSolver* object:

```
results = solver.solve(
    pybnb.futures.NestedSolver(problem,
                               queue_strategy='depth',
                               track_bound=False,
                               time_limit=1),
    queue_strategy='depth',
    initialize_queue=queue,
    best_node=best_node,
    objective_stop=objective_stop)
```

Running the parallel case, with CPython 3.7 we achieve a peak performance of ~60k nodes per second, and with PyPy3 6.0 we achieve ~450k nodes per second!

CHAPTER

FOUR

REFERENCE

4.1 Quick Links

- *pybnb.solve*
- *pybnb.Problem*
- *pybnb.Solver*
- *pybnb.SolverResults*
- *pybnb.SolutionStatus*
- *pybnb.TerminationCondition*
- *pybnb.QueueStrategy*

4.2 Modules

4.2.1 pybnb.configuration

Configuration settings for node serialization.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

class pybnb.configuration.**Configuration**

The main configuration object.

SERIALIZER

The name of serialization module used to transmit node state. (default: "pickle")

Type str, {'pickle', 'dill'}

SERIALIZER_PROTOCOL_VERSION

The protocol argument passed to the *dumps* function of the selected serialization module. (default: pickle.HIGHEST_PROTOCOL)

Type int

COMPRESSION

Indicates whether or not to compress the serialized node state using zlib. (default: False)

Type bool

MARSHAL_PROTOCOL_VERSION

The version argument passed to the `marshal.dumps()` function. (default: 2)

Type int

reset(*use_environment=True*)
Reset the configuration to default settings.

Parameters **use_environment** (*bool, optional*) – Controls whether or not to check for environment variables to overwrite the default settings. (default: True)

4.2.2 pybnb.common

Basic definitions and utilities.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

class pybnb.common.**ProblemSense**(*value*)
An enumeration.

minimize = 1
The objective sense defining a minimization problem.

maximize = -1
The objective sense defining a maximization problem.

pybnb.common.**inf = inf**
A floating point constant set to `float('inf')`.

pybnb.common.**nan = nan**
A floating point constant set to `float('nan')`.

class pybnb.common.**QueueStrategy**(*value*)
Strategies for prioritizing nodes in the central dispatcher queue. For all strategies, ties are broken by insertion order.

bound = 'bound'
The node with the worst bound is always selected next.

objective = 'objective'
The node with the best objective is always selected next.

breadth = 'breadth'
The node with the smallest tree depth is always selected next (i.e., breadth-first search).

depth = 'depth'
The node with the largest tree depth is always selected next (i.e., depth-first search).

local_gap = 'local_gap'
The node with the largest gap between its local objective and bound is always selected next.

fifo = 'fifo'
Nodes are served in first-in, first-out order.

lifo = 'lifo'
Nodes are served in last-in, first-out order.

random = 'random'
Nodes are assigned a random priority before entering the queue.

custom = 'custom'
The node with the largest value stored in the *queue_priority* attribute is always selected next. Users are expected to assign a priority to all nodes returned from the *branch* method on their problem.

class pybnb.common.SolutionStatus(*value*)
Possible values assigned to the solution_status attribute of a *SolverResults* object returned from a solve.

optimal = 'optimal'
Indicates that the best objective is finite and close enough to the global bound to satisfy the optimality tolerances used for the solve.

feasible = 'feasible'
Indicates that the best objective is finite but not close enough to the global bound to satisfy the optimality tolerances used for the solve.

infeasible = 'infeasible'
Indicates that both the best objective and global bound are equal to the infeasible objective value (+inf or -inf depending on the sense).

unbounded = 'unbounded'
Indicates that both the best objective and global bound are equal to the unbounded objective value (+inf or -inf depending on the sense).

invalid = 'invalid'
Indicates that the global bound is not a valid bound on the best objective found. This may be due to an ill-defined problem or other numerical issues.

unknown = 'unknown'
Indicates that the global bound is finite, but no feasible (finite) objective was found.

class pybnb.common.TerminationCondition(*value*)
Possible values assigned to the termination_condition attribute of a *SolverResults* object returned from a solve.

optimality = 'optimality'
The dispatcher terminated the solve based on optimality criteria.

objective_limit = 'objective_limit'
The dispatcher terminated the solve based on the user-supplied limit on the objective or bound being satisfied.

node_limit = 'node_limit'
The dispatcher terminated the solve due to the user-supplied explored node limit being surpassed.

time_limit = 'time_limit'
The dispatcher terminated the solve due to the user-supplied time limit being surpassed.

queue_empty = 'queue_empty'
The dispatcher terminated the solve due to the node queue becoming empty.

queue_limit = 'queue_limit'
The dispatcher terminated the solve due to the user-supplied queue size limit being exceeded.

interrupted = 'interrupted'
Solve termination was initiated by SIGINT or SIGUSR signal event.

4.2.3 pybnb.problem

Branch-and-bound problem definition.

class pybnb.problem.**Problem**

The abstract base class used for defining branch-and-bound problems.

abstractmethod sense()

Returns the objective sense for this problem.

> **Note:** This method is abstract and must be defined by the user.

abstractmethod bound()

Returns a value that is a bound on the objective of the current problem state or *self.unbounded_objective()* if no non-trivial bound is available.

> **Note:** This method is abstract and must be defined by the user.

abstractmethod objective()

Returns a feasible value for the objective of the current problem state or *self.infeasible_objective()* if the current state is not feasible.

> **Note:** This method is abstract and must be defined by the user.

abstractmethod save_state(*node*)

Saves the current problem state into the given *pybnb.node.Node* object.

This method is guaranteed to be called once at the start of the solve by all processes involved to collect the root node problem state, but it may be called additional times. When it is called for the root node, the *node.tree_depth* will be zero.

> **Note:** This method is abstract and must be defined by the user.

abstractmethod load_state(*node*)

Loads the problem state that is stored on the given *pybnb.node.Node* object.

> **Note:** This method is abstract and must be defined by the user.

abstractmethod branch()

Returns a list of *Node* objects that partition the node state into zero or more children. This method can also be defined as a generator.

> **Note:** This method is abstract and must be defined by the user.

infeasible_objective()

Returns the value that represents an infeasible objective (i.e., +inf or -inf depending on the sense). The *Problem* base class implements this method.

unbounded_objective()
Returns the value that represents an unbounded objective (i.e., +inf or -inf depending on the sense). The *Problem* base class implements this method.

notify_solve_begins(*comm*, *worker_comm*, *convergence_checker*)
Called when a branch-and-bound solver begins as solve. The *Problem* base class provides a default implementation for this method that does nothing.

Parameters

- **comm** (mpi4py.MPI.Comm or None) – The full MPI communicator that includes all processes. Will be None if MPI has been disabled.
- **worker_comm** (mpi4py.MPI.Comm or None) – The MPI communicator that includes only worker processes. Will be None if MPI has been disabled.
- **convergence_checker** (*ConvergenceChecker*:) – The class used for comparing the objective and bound values during the solve.

notify_new_best_node(*node*, *current*)
Called when a branch-and-bound solver receives a new best node from the dispatcher. The *Problem* base class provides a default implementation for this method that does nothing.

Parameters

- **node** (*Node*) – The new best node.
- **current** (*bool*) – Indicates whether or not the node argument is the currently loaded node (from the most recent load_state call).

notify_solve_finished(*comm*, *worker_comm*, *results*)
Called when a branch-and-bound solver finishes. The *Problem* base class provides a default implementation for this method that does nothing.

Parameters

- **comm** (mpi4py.MPI.Comm or None) – The full MPI communicator that includes all processes. Will be None if MPI has been disabled.
- **worker_comm** (mpi4py.MPI.Comm or None) – The MPI communicator that includes only worker processes. Will be None if MPI has been disabled.
- **results** (*SolverResults*) – The fully populated results container that will be returned from the solver.

4.2.4 pybnb.node

Branch-and-bound node implementation.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

pybnb.node.**dumps**(*obj*)
Return the serialized representation of the object as a bytes object, using the serialization module set in the current configuration.

pybnb.node.**loads**(*data*)
Read and return an object from the given serialized data, using the serialization module set in the current configuration.

class pybnb.node.**Node**
A branch-and-bound node that stores problem state.

objective
The objective value for the node.

Type float

bound
The bound value for the node.

Type float

tree_depth
The tree depth of the node (0-based).

Type int

queue_priority
The queue priority of the node.

Type float or tuple of floats

state
The user specified node state.

4.2.5 pybnb.solver_results

Branch-and-bound solver results object.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

class pybnb.solver_results.**SolverResults**
Stores the results of a branch-and-bound solve.

solution_status
The solution status will be set to one of the strings documented by the *SolutionStatus* enum.

Type string

termination_condition
The solve termination condition, as determined by the dispatcher, will be set to one of the strings documented by the *TerminationCondition* enum.

Type string

objective
The best objective found.

Type float

bound
The global optimality bound.

Type float

absolute_gap
The absolute gap between the objective and bound. This will only be set when the solution status sf "optimal" or "feasible"; otherwise, it will be None.

Type float or None

relative_gap
The relative gap between the objective and bound. This will only be set when the solution status sf "optimal" or "feasible"; otherwise, it will be None.

Type float or None

nodes
The total number of nodes processes by all workers.

Type int

wall_time
The process-local wall time (seconds). This is the only value on the results object that varies between processes.

Type float

best_node
The node with the best objective obtained during the solve. Note that if the best_objective solver option was used, the best_node on the results object may have an objective that is worse than the objective stored on the results (or may be None).

Type *Node*

pprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Prints a nicely formatted representation of the results.

Parameters **stream** (*file-like object or string, optional*) – A file-like object or a filename where results should be written to. (default: `sys.stdout`)

write(*stream*, *prefix=''*, *pretty=False*)
Writes results in YAML format to a stream or file. Changing the parameter values from their defaults may result in the output becoming non-compatible with the YAML format.

Parameters

- **stream** (*file-like object or string*) – A file-like object or a filename where results should be written to.
- **prefix** (*string, optional*) – A string to use as a prefix for each line that is written. (default: '')
- **pretty** (*bool, optional*) – Indicates whether or not certain recognized attributes should be formatted for more human-readable output. (default: False)

Example

```
>>> import six
>>> import pybnb
>>> results = pybnb.SolverResults()
>>> results.best_node = pybnb.Node()
>>> results.best_node.objective = 123
>>> out = six.StringIO()
>>> # the best_node is serialized
>>> results.write(out)
>>> del results
>>> import yaml
>>> results_dict = yaml.safe_load(out.getvalue())
>>> # de-serialize the best_node
>>> best_node = pybnb.node.loads(results_dict['best_node'])
>>> assert best_node.objective == 123
```

4.2.6 pybnb.solver

Branch-and-bound solver implementation.

class pybnb.solver.**Solver**(*comm=<class 'pybnb.solver._NoArgumentGiven'>*, *dispatcher_rank=0*)
A branch-and-bound solver.

Parameters

- **comm** (mpi4py.MPI.Comm, optional) – The MPI communicator to use. If unset, the mpi4py.MPI.COMM_WORLD communicator will be used. Setting this keyword to None will disable the use of MPI and avoid an attempted import of mpi4py.MPI (which avoids triggering a call to *MPI_Init()*).
- **dispatcher_rank** (*int, optional*) – The process with this rank will be designated as the dispatcher process. If MPI functionality is disabled (by setting comm=None), this keyword must be 0. (default: 0)

property is_worker
Indicates if this process has been designated as a worker.

property is_dispatcher
Indicates if this process has been designated as the dispatcher.

property comm
The full MPI communicator that includes the dispatcher and all workers. Will be None if MPI functionality has been disabled.

property worker_comm
The worker MPI communicator. Will be None on any processes for which *Solver.is_worker* is False, or if MPI functionality has been disabled.

property worker_count
The number of worker processes associated with this solver.

collect_worker_statistics()
Collect individual worker statistics about the most recent solve.

Returns A dictionary whose keys are the different statistics collected, where each entry is a list storing a value for each worker.

Return type dict

save_dispatcher_queue()
Saves the dispatcher queue.

Returns **queue** – If this process is the dispatcher, this method will return an object storing any nodes currently in the dispatcher queue. If this process is not the dispatcher, this method will return None. The returned object can be used to reinitialize a solve (e.g., with different algorithms settings) by assigning it to the initialize_queue keyword of the *Solver.solve()* method.

Return type *pybnb.dispatcher.DispatcherQueueData* or None

solve(*problem*, *best_objective=None*, *best_node=None*, *disable_objective_call=False*, *absolute_gap=0*, *relative_gap=None*, *scale_function=<function _default_scale>*, *queue_tolerance=<class 'pybnb.convergence_checker._auto_queue_tolerance'>*, *branch_tolerance=0*, *comparison_tolerance=0*, *objective_stop=None*, *bound_stop=None*, *node_limit=None*, *time_limit=None*, *queue_limit=None*, *track_bound=True*, *initialize_queue=None*, *queue_strategy='bound'*, *log_interval_seconds=1.0*, *log_new_incumbent=True*, *log=<class 'pybnb.solver._NoArgumentGiven'>*, *disable_signal_handlers=False*)

Solve a problem using branch-and-bound.

Note: Parameters for this function are treated differently depending on whether the process is a worker or dispatcher. For the serial case (no MPI), the single process is both a worker and a dispatcher. For the parallel case, exactly one process is a dispatcher and all other processes are workers. A **(W)** in the parameter description indicates that it is only used by worker processes (ignored otherwise). A **(D)** in the parameter description indicates that it is only used by the dispatcher process (ignored otherwise). An **(A)** indicates that it is used by all processes, and it is assumed the same value is provided for each process; otherwise, the behavior is undefined.

Parameters

- **problem** (*pybnb.Problem*) – An object defining a branch-and-bound problem.
- **best_objective** (*float, optional*) – Initializes the solve with an assumed best objective. Both this and the best_node option can be set to different values on all processes. The dispatcher will collect all values and use the best. Note that setting this option at, or too close to, the true optimal objective value may prevent the solver from collecting a node that stores the optimal user state information, so use this option with care. The recommended way to re-continue a solve from a known candidate solution is to assign the best_node attribute of a results object to the best_node solve option. Also note that the best node will be tracked separately from the given initial best objective until a node is found that improves upon the best objective. If this never happens, the best_node attribute on the solver results may be None or may have an objective that is worse than the objective attribute of the solver results. (default: None)
- **best_node** (*Node*, optional) – Initializes the solve with an assumed best node. This option can (and should) be used in place of the best_objective option when a best node from a previous solve has been collected. It can also be assigned a node object that was created manually by the user. The objective attribute is the only property of the node that will affect the solve. It must be set to a numeric value. (default: None)
- **disable_objective_call** (*bool, optional*) – **(W)** Disables requests for an objective value from subproblems. (default: False)
- **absolute_gap** (*float, optional*) – **(A)** The maximum absolute difference between the global bound and best objective for the problem to be considered solved to optimality. Setting to *None* will disable this optimality check. By default, this option also controls eligibility for the queue. See the "queue_tolerance" setting for more information. (default: 0)
- **relative_gap** (*float, optional*) – **(A)** The maximum relative difference (absolute difference scaled by *max{1.0,|objective|}*) between the global bound and best objective for the problem to be considered solved to optimality. The default setting of *None* means this optimality check is not used. (default: None)
- **scale_function** (*function, optional*) – **(A)** A function with signature *f(bound, objective) -> float* that returns a positive scale factor used to convert the absolute difference

between the bound and objective into a relative difference. The relative difference is compared with the *relative_gap* convergence tolerance to determine if the solver should terminate. The default is equivalent to *max{1.0,|objective|}*. Other examples one could use are *max{|bound|,|objective|}*, *(|bound|+|objective|)/2*, etc.

- **queue_tolerance** (*float, optional*) – **(A)** The absolute tolerance used when deciding if a node is eligible to enter the queue. The difference between the node bound and the incumbent objective must be greater than this value. Leaving this argument at its default value indicates that this tolerance should be set equal to the "absolute_gap" setting. Setting this to zero means that nodes whose bound is equal to the incumbent objective are not eligible to enter the queue. Setting this to larger values can be used to limit the queue size, but it should be kept small enough to allow absolute and relative optimality tolerances to be met. This option can also be set to *None* to allow nodes with a bound equal to (but not greater than) the incumbent objective to enter the queue.
- **branch_tolerance** (*float, optional*) – **(A)** The absolute tolerance used when deciding if the computed objective and bound for a node are sufficiently different to branch into the node. The default value of zero means that branching will occur if the bound is not exactly equal to the objective. This option can be set to *None* to enable branching for nodes with a bound and objective that are exactly equal. (default: 0)
- **comparison_tolerance** (*float, optional*) – **(A)** The absolute tolerance used when deciding if two objective or bound values are sufficiently different to be considered improved or worsened. This tolerance controls when the solver considers a new incumbent objective to be found. It also controls when warnings are output about bounds becoming worse on child nodes. Setting this to larger values can be used to avoid the above solver actions due to insignificant numerical differences, but it is better to deal with these numerical issues by rounding numbers to a reliable precision before returning them from the problem methods. (default: 0)
- **objective_stop** (*float, optional*) – **(A)** If provided, the solve will terminate when a feasible objective is found that is at least as good as the specified value, and the termination_condition flag on the results object will be set to 'objective_limit'. If this value is infinite, the solve will terminate as soon as a finite objective is found. (default: None)
- **bound_stop** (*float, optional*) – **(A)** If provided, the solve will terminate when the global bound on the objective is at least as good as the specified value, and the termination_condition flag on the results object will be set to 'objective_limit'. If this value is infinite, the solve will terminate as soon as a finite bound is found. (default: None)
- **node_limit** (*int, optional*) – **(D)** If provided, the solve will begin to terminate once this many nodes have been served from the dispatcher queue, and the termination_condition flag on the results object will be set to 'node_limit'. (default: None)
- **time_limit** (*float, optional*) – **(D)** If provided, the solve will begin to terminate once this amount of time has passed, and the termination_condition flag on the results object will be set to 'time_limit'. Note that the solve may run for an arbitrarily longer amount of time, depending how long worker processes spend completing their final task. (default: None)
- **queue_limit** (*int, optional*) – **(D)** If provided, the solve will begin to terminate once the size of the dispatcher queue exceeds this amount, and the termination_condition flag on the results object will be set to 'queue_limit'. Note that the queue may become arbitrarily larger than this limit, depending how many child nodes are returned from worker processes on their final update. (default: None)
- **track_bound** (*bool, optional*) – **(D)** Indicates whether the dispatcher should track the global queue bound while running. Setting this to false can reduce the overhead of dispatcher updates for some priority queue strategies. (default: True)

- **initialize_queue** (*pybnb.dispatcher.DispatcherQueueData*, optional) – **(D)** Initializes the dispatcher queue with that remaining from a previous solve (obtained by calling *Solver.save_dispatcher_queue()* after the solve). If left as None, the queue will be initialized with a single root node created by calling *problem.save_state*. (default: None)
- **queue_strategy** (*QueueStrategy* or tuple) – **(D)** Sets the strategy for prioritizing nodes in the central dispatcher queue. See the *QueueStrategy* enum for the list of acceptable values. This keyword can be assigned one of the enumeration attributes or an equivalent string name. This keyword can also be assigned a tuple of choices to define a lexicographic sorting strategy. (default: 'bound')
- **log_interval_seconds** (*float, optional*) – **(D)** The approximate time (in seconds) between solver log updates. More time may pass between log updates if no updates have been received from worker processes, and less time may pass if a new incumbent objective is found. (default: 1.0)
- **log_new_incumbent** (*bool, optional*) – **(D)** Controls whether updates to the best objective are logged immediately (overriding the log interval). Setting this to false can be useful when frequent updates to the incumbent are expected and the additional logging slows down the dispatcher. (default: True)
- **log** (*logging.Logger, optional*) – **(D)** A log object where solver output should be sent. The default value causes all output to be streamed to the console. Setting to None disables all output.
- **disable_signal_handlers** (*bool, optional*) – **(D)** Setting to true disables the registering of signal handlers that allow gracefully terminating a solve early. (default: False)

Returns **results** – An object storing information about the solve.

Return type *SolverResults*

pybnb.solver.**summarize_worker_statistics**(*stats*, *stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Writes a summary of workers statistics to an output stream.

Parameters

- **stats** (*dict*) – A dictionary of worker statistics returned from a call to `collect_worker_statistics()`.
- **stream** (*file-like object, or string, optional*) – A file-like object or a filename where results should be written to. (default: `sys.stdout`)

pybnb.solver.**solve**(*problem*, *comm=<class 'pybnb.solver._NoArgumentGiven'>*, *dispatcher_rank=0*, *log_filename=None*, *results_filename=None*, ***kwds*)

Solves a branch-and-bound problem and returns the solution.

Note: This function also collects and summarizes runtime workload statistics, which may introduce additional overhead. This overhead can be avoided by directly instantiating a *Solver* object and calling the *Solver.solve()* method.

Parameters

- **problem** (*pybnb.Problem*) – An object that defines a branch-and-bound problem
- **comm** (`mpi4py.MPI.Comm`, optional) – The MPI communicator to use. If unset, the mpi4py.MPI.COMM_WORLD communicator will be used. Setting this keyword to None

will disable the use of MPI and avoid an attempted import of mpi4py.MPI (which avoids triggering a call to *MPI_Init()*).

- **dispatcher_rank** (*int, optional*) – The process with this rank will be designated the dispatcher process. If MPI functionality is disabled (by setting comm=None, or when comm.size==1), this keyword must be left at 0. (default: 0)
- **log_filename** (*string, optional*) – A filename where solver output should be sent in addition to console. This keyword will be ignored if the *log* keyword is set. (default: None)
- **results_filename** (*string, optional*) – Saves the solver results into a YAML-formatted file with the given name. (default: None)
- ****kwds** – Additional keywords to be passed to *Solver.solve()*. See that method for additional keyword documentation.

Returns **results** – An object storing information about the solve.

Return type *SolverResults*

4.2.7 pybnb.convergence_checker

Convergence checking implementation.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

pybnb.convergence_checker.**compute_absolute_gap**(*sense*, *bound*, *objective*)

Returns the absolute gap between the bound and the objective, respecting the sign relative to the objective sense of this problem.

pybnb.convergence_checker.**compute_relative_gap**(*sense*, *bound*, *objective*, *scale=<function _default_scale>*)

Returns the relative gap between the bound and the objective, respecting the sign relative to the objective sense of this problem.

class pybnb.convergence_checker.**ConvergenceChecker**(*sense*, *absolute_gap=0*, *relative_gap=None*, *scale_function=<function _default_scale>*, *queue_tolerance=<class 'pybnb.convergence_checker._auto_queue_tolerance'>*, *branch_tolerance=0*, *comparison_tolerance=0*, *objective_stop=None*, *bound_stop=None*)

A class used to check convergence.

Parameters

- **sense** ({minimize, maximize}) – The objective sense for the problem.
- **absolute_gap** (*float, optional*) – The absolute difference between the objective and bound that determines optimality. By default, this option also controls eligibility for the queue. See the "queue_tolerance" setting for more information. (default: 0)
- **relative_gap** (*float, optional*) – The relative difference between the objective and bound that determines optimality. (default: None)
- **scale_function** (*function, optional*) – A function with signature *f(bound, objective) -> float* that returns a positive scale factor used to convert the absolute difference between the bound and objective into a relative difference. The relative difference is compared with the *relative_gap* convergence tolerance to determine if the solver should terminate. The default is equivalent to *max{1.0,|objective|}*. Other examples one could use are *max{|bound|,|objective|}*, *(|bound|+|objective|)/2*, etc.

- **queue_tolerance** (*float, optional*) – The absolute tolerance used when deciding if a node is eligible to enter the queue. The difference between the node bound and the incumbent objective must be greater than this value. Leaving this argument at its default value indicates that this tolerance should be set equal to the "absolute_gap" setting. Setting this to zero means that nodes whose bound is equal to the incumbent objective are not eligible to enter the queue. Setting this to larger values can be used to limit the queue size, but it should be kept small enough to allow absolute and relative optimality tolerances to be met. This option can also be set to *None* to allow nodes with a bound equal to (but not greater than) the incumbent objective to enter the queue.
- **branch_tolerance** (*float, optional*) – The absolute tolerance used when deciding if the computed objective and bound for a node are sufficiently different to branch into the node. The default value of zero means that branching will occur if the bound is not exactly equal to the objective. This option can be set to *None* to enable branching for nodes with a bound and objective that are exactly equal. (default: 0)
- **comparison_tolerance** (*float, optional*) – The absolute tolerance used when deciding if two objective or bound values are sufficiently different to be considered improved or worsened. This tolerance controls when the solver considers a new incumbent objective to be found. It also controls when warnings are output about bounds becoming worse on child nodes. Setting this to larger values can be used to avoid the above solver actions due to insignificant numerical differences, but it is better to deal with these numerical issues by rounding numbers to a reliable precision before returning them from the problem methods. (default: 0)
- **objective_stop** (*float, optional*) – If provided, the "objective_limit" termination criteria is met when a feasible objective is found that is at least as good as the specified value. If this value is infinite, the termination criteria is met as soon as a finite objective is found. (default: None)
- **bound_stop** (*float, optional*) – If provided, the "objective_limit" termination criteria is met when the best bound on the objective is at least as good as the specified value. If this value is infinite, the termination criteria is met as soon as a finite objective is found. (default: None)

check_termination_criteria(*global_bound*, *best_objective*)
Checks if any termination criteria are met and returns the corresponding *TerminationCondition* enum value; otherwise, *None* is returned.

objective_is_optimal(*objective*, *bound*)
Determines if the objective is optimal by checking if the optimality gap is small enough relative to the absolute gap or relative gap settings.

compute_absolute_gap(*bound*, *objective*)
Returns the absolute gap between the bound and the objective, respecting the sign relative to the objective sense of this problem.

compute_relative_gap(*bound*, *objective*)
Returns the relative gap between the bound and the objective, respecting the sign relative to the objective sense of this problem.

eligible_for_queue(*bound*, *objective*)
Returns True when the queue object with the given bound is eligible for the queue relative to the given objective.

eligible_to_branch(*bound*, *objective*)
Returns True when the bound and objective are sufficiently far apart to allow branching.

bound_worsened(*new*, *old*)
Returns True when the new bound is worse than the old bound by greater than the comparison tolerance.

objective_improved(*new*, *old*)
Returns True when the new objective is better than the old objective by greater than the comparison tolerance.

worst_bound(**args*, ***kwds*)
Returns the worst bound, as defined by the objective sense, from a given iterable of bound values. This function passes all keywords and arguments directly to the built-ins *min* and *max*.

best_bound(**args*, ***kwds*)
Returns the best bound, as defined by the objective sense, from a given iterable of bound values. This function passes all keywords and arguments directly to the built-ins *min* and *max*.

worst_objective(**args*, ***kwds*)
Returns the worst objective, as defined by the objective sense, from a given iterable of objective values. This function passes all keywords and arguments directly to the built-ins *min* and *max*.

best_objective(**args*, ***kwds*)
Returns the best objective, as defined by the objective sense, from a given iterable of objective values. This function passes all keywords and arguments directly to the built-ins *min* and *max*.

4.2.8 pybnb.priority_queue

A collection of priority queue implementations that can be used by the dispatcher.

class pybnb.priority_queue.**IPriorityQueue**(**args*, ***kwds*)
Bases: `object`

The abstract interface for priority queues that store node data for the dispatcher.

size()
Returns the size of the queue.

put(*node*)
Puts an node in the queue, possibly updating the value of *queue_priority*, depending on the queue implementation. This method returns a unique counter associated with each put.

get()
Returns the next node in the queue. If the queue is empty, returns None.

bound()
Returns the weakest bound of all nodes in the queue. If the queue is empty, returns None.

filter(*func*)
Removes nodes from the queue for which *func(node)* returns False. The list of nodes removed is returned. If the queue is empty or no nodes are removed, the returned list will be empty.

items()
Iterates over the queued nodes in arbitrary order without modifying the queue.

class pybnb.priority_queue.**WorstBoundFirstPriorityQueue**(*sense*, *track_bound*)
Bases: *pybnb.priority_queue.IPriorityQueue*

A priority queue implementation that serves nodes with the worst bound first.

Parameters

- **sense** ({`minimize`, `maximize`}) – The objective sense for the problem.

- **track_bound** (*bool*) – Indicates whether or not to track the global queue bound. Note that this particular queue implementation always tracks the global bound. This argument is ignored.

size()
Returns the size of the queue.

put(*node*)
Puts an node in the queue, possibly updating the value of *queue_priority*, depending on the queue implementation. This method returns a unique counter associated with each put.

get()
Returns the next node in the queue. If the queue is empty, returns None.

bound()
Returns the weakest bound of all nodes in the queue. If the queue is empty, returns None.

filter(*func*)
Removes nodes from the queue for which *func(node)* returns False. The list of nodes removed is returned. If the queue is empty or no nodes are removed, the returned list will be empty.

items()
Iterates over the queued nodes in arbitrary order without modifying the queue.

class pybnb.priority_queue.**CustomPriorityQueue**(*sense*, *track_bound*, *_queue_type_=pybnb.priority_queue._NoThreadingMaxPriorityFirstQueu*

Bases: *pybnb.priority_queue.IPriorityQueue*

A priority queue implementation that can handle custom node priorities. It uses an additional data structure to reduce the amount of time it takes to compute a queue bound.

Parameters

- **sense** ({minimize, maximize}) – The objective sense for the problem.
- **track_bound** (*bool*) – Indicates whether or not to track the global queue bound.

size()
Returns the size of the queue.

put(*node*)
Puts an node in the queue, possibly updating the value of *queue_priority*, depending on the queue implementation. This method returns a unique counter associated with each put.

get()
Returns the next node in the queue. If the queue is empty, returns None.

bound()
Returns the weakest bound of all nodes in the queue. If the queue is empty, returns None.

filter(*func*)
Removes nodes from the queue for which *func(node)* returns False. The list of nodes removed is returned. If the queue is empty or no nodes are removed, the returned list will be empty.

items()
Iterates over the queued nodes in arbitrary order without modifying the queue.

class pybnb.priority_queue.**BestObjectiveFirstPriorityQueue**(*sense*, *track_bound*, *_queue_type_=pybnb.priority_queue._NoThreadingMaxP*

Bases: *pybnb.priority_queue.CustomPriorityQueue*

A priority queue implementation that serves nodes with the best objective first.

sense [{minimize, maximize}] The objective sense for the problem.

put(*node*)
Puts an node in the queue, possibly updating the value of *queue_priority*, depending on the queue implementation. This method returns a unique counter associated with each put.

bound()
Returns the weakest bound of all nodes in the queue. If the queue is empty, returns None.

filter(*func*)
Removes nodes from the queue for which *func(node)* returns False. The list of nodes removed is returned. If the queue is empty or no nodes are removed, the returned list will be empty.

get()
Returns the next node in the queue. If the queue is empty, returns None.

items()
Iterates over the queued nodes in arbitrary order without modifying the queue.

size()
Returns the size of the queue.

class pybnb.priority_queue.**BreadthFirstPriorityQueue**(*sense*, *track_bound*, *_queue_type_=pybnb.priority_queue._NoThreadingMaxPriorityF*

Bases: *pybnb.priority_queue.CustomPriorityQueue*

A priority queue implementation that serves nodes in breadth-first order.

sense [{minimize, maximize}] The objective sense for the problem.

put(*node*)
Puts an node in the queue, possibly updating the value of *queue_priority*, depending on the queue implementation. This method returns a unique counter associated with each put.

bound()
Returns the weakest bound of all nodes in the queue. If the queue is empty, returns None.

filter(*func*)
Removes nodes from the queue for which *func(node)* returns False. The list of nodes removed is returned. If the queue is empty or no nodes are removed, the returned list will be empty.

get()
Returns the next node in the queue. If the queue is empty, returns None.

items()
Iterates over the queued nodes in arbitrary order without modifying the queue.

size()
Returns the size of the queue.

class pybnb.priority_queue.**DepthFirstPriorityQueue**(*sense*, *track_bound*, *_queue_type_=pybnb.priority_queue._NoThreadingMaxPriorityFirs*

Bases: *pybnb.priority_queue.CustomPriorityQueue*

A priority queue implementation that serves nodes in depth-first order.

sense [{minimize, maximize}] The objective sense for the problem.

put(*node*)
Puts an node in the queue, possibly updating the value of *queue_priority*, depending on the queue implementation. This method returns a unique counter associated with each put.

bound()
Returns the weakest bound of all nodes in the queue. If the queue is empty, returns None.

filter(*func*)
Removes nodes from the queue for which *func(node)* returns False. The list of nodes removed is returned. If the queue is empty or no nodes are removed, the returned list will be empty.

get()
Returns the next node in the queue. If the queue is empty, returns None.

items()
Iterates over the queued nodes in arbitrary order without modifying the queue.

size()
Returns the size of the queue.

class pybnb.priority_queue.**FIFOQueue**(*sense*, *track_bound*)
Bases: *pybnb.priority_queue.CustomPriorityQueue*

A priority queue implementation that serves nodes in first-in, first-out order.

sense [{minimize, maximize}] The objective sense for the problem.

put(*node*)
Puts an node in the queue, possibly updating the value of *queue_priority*, depending on the queue implementation. This method returns a unique counter associated with each put.

bound()
Returns the weakest bound of all nodes in the queue. If the queue is empty, returns None.

filter(*func*)
Removes nodes from the queue for which *func(node)* returns False. The list of nodes removed is returned. If the queue is empty or no nodes are removed, the returned list will be empty.

get()
Returns the next node in the queue. If the queue is empty, returns None.

items()
Iterates over the queued nodes in arbitrary order without modifying the queue.

size()
Returns the size of the queue.

class pybnb.priority_queue.**LIFOQueue**(*sense*, *track_bound*)
Bases: *pybnb.priority_queue.CustomPriorityQueue*

A priority queue implementation that serves nodes in last-in, first-out order.

sense [{minimize, maximize}] The objective sense for the problem.

put(*node*)
Puts an node in the queue, possibly updating the value of *queue_priority*, depending on the queue implementation. This method returns a unique counter associated with each put.

bound()
Returns the weakest bound of all nodes in the queue. If the queue is empty, returns None.

filter(*func*)
Removes nodes from the queue for which *func(node)* returns False. The list of nodes removed is returned. If the queue is empty or no nodes are removed, the returned list will be empty.

get()
Returns the next node in the queue. If the queue is empty, returns None.

items()
Iterates over the queued nodes in arbitrary order without modifying the queue.

size()
Returns the size of the queue.

class pybnb.priority_queue.**RandomPriorityQueue**(*sense*, *track_bound*, *_queue_type_=pybnb.priority_queue._NoThreadingMaxPriorityFirstQueu*
Bases: *pybnb.priority_queue.CustomPriorityQueue*

A priority queue implementation that assigns a random priority to each incoming node.

sense [{minimize, maximize}] The objective sense for the problem.

put(*node*)
Puts an node in the queue, possibly updating the value of *queue_priority*, depending on the queue implementation. This method returns a unique counter associated with each put.

bound()
Returns the weakest bound of all nodes in the queue. If the queue is empty, returns None.

filter(*func*)
Removes nodes from the queue for which *func(node)* returns False. The list of nodes removed is returned. If the queue is empty or no nodes are removed, the returned list will be empty.

get()
Returns the next node in the queue. If the queue is empty, returns None.

items()
Iterates over the queued nodes in arbitrary order without modifying the queue.

size()
Returns the size of the queue.

class pybnb.priority_queue.**LocalGapPriorityQueue**(*sense*, *track_bound*, *_queue_type_=pybnb.priority_queue._NoThreadingMaxPriorityFirstQ*
Bases: *pybnb.priority_queue.CustomPriorityQueue*

A priority queue implementation that serves nodes with the largest gap between the local objective and bound first.

sense [{minimize, maximize}] The objective sense for the problem.

put(*node*)
Puts an node in the queue, possibly updating the value of *queue_priority*, depending on the queue implementation. This method returns a unique counter associated with each put.

bound()
Returns the weakest bound of all nodes in the queue. If the queue is empty, returns None.

filter(*func*)
Removes nodes from the queue for which *func(node)* returns False. The list of nodes removed is returned. If the queue is empty or no nodes are removed, the returned list will be empty.

get()
Returns the next node in the queue. If the queue is empty, returns None.

items()
Iterates over the queued nodes in arbitrary order without modifying the queue.

size()
Returns the size of the queue.

class pybnb.priority_queue.**LexicographicPriorityQueue**(*queue_types*, *sense*, *track_bound*)
Bases: *pybnb.priority_queue.CustomPriorityQueue*

A priority queue implementation that serves nodes with the largest gap between the local objective and bound first.

sense [{minimize, maximize}] The objective sense for the problem.

put(*node*)
Puts an node in the queue, possibly updating the value of *queue_priority*, depending on the queue implementation. This method returns a unique counter associated with each put.

bound()
Returns the weakest bound of all nodes in the queue. If the queue is empty, returns None.

filter(*func*)
Removes nodes from the queue for which *func(node)* returns False. The list of nodes removed is returned. If the queue is empty or no nodes are removed, the returned list will be empty.

get()
Returns the next node in the queue. If the queue is empty, returns None.

items()
Iterates over the queued nodes in arbitrary order without modifying the queue.

size()
Returns the size of the queue.

pybnb.priority_queue.**PriorityQueueFactory**(*name*, **args*, ***kwds*)
Returns a new instance of the priority queue type registered under the given name.

pybnb.priority_queue.**register_queue_type**(*name*, *cls*)
Registers a new priority queue class with the PriorityQueueFactory.

4.2.9 pybnb.dispatcher

Branch-and-bound dispatcher implementation.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

class pybnb.dispatcher.**DispatcherQueueData**(*nodes*, *worst_terminal_bound*, *sense*)
A namedtuple storing data that can be used to initialize a dispatcher queue.

nodes
A list of *Node* objects.

Type list

worst_terminal_bound
The worst bound of any node where branching did not continue.

Type float or None

sense
The objective sense for the problem that produced this queue.

Type {minimize, maximize}

bound()
Returns the global bound defined by this queue data.

class pybnb.dispatcher.**StatusPrinter**(*dispatcher*, *log*, *log_interval_seconds=1.0*)
Logs status information about the branch-and-bound solve.

Parameters

- **dispatcher** (pybnb.dispatcher.Dispatcher) – The central dispatcher that will be monitored.
- **log** (logging.Logger) – A log object where solver output should be sent.
- **log_interval_seconds** (*float*) – The approximate maximum time (in seconds) between solver log updates. More time may pass between log updates if no updates have been received from any workers, and less time may pass if a new incumbent is found. (default: 1.0)

log_info(*msg*)
Pass a message to log.info

log_warning(*msg*)
Pass a message to log.warning

log_debug(*msg*)
Pass a message to log.debug

log_error(*msg*)
Pass a message to log.error

log_critical(*msg*)
Pass a message to log.critical

new_objective(*report=True*)
Indicate that a new objective has been found

Parameters **report** (*bool, optional*) – Indicate whether or not to force the next *tic* log output. (default: False)

tic(*force=False*)
Provide an opportunity to log output if certain criteria are met.

Parameters **force** (*bool, optional*) – Indicate whether or not to force logging of output, even if logging criteria are not met. (default: False)

class pybnb.dispatcher.**DispatcherBase**
The base dispatcher implementation with some core functionality shared by the distributed and local implementations.

initialize(*best_objective*, *best_node*, *initialize_queue*, *queue_strategy*, *converger*, *node_limit*, *time_limit*, *queue_limit*, *track_bound*, *log*, *log_interval_seconds*, *log_new_incumbent*)
Initialize the dispatcher for a new solve.

Parameters

- **best_objective** (*float*) – The assumed best objective to start with.
- **best_node** (*Node*) – A node storing the assumed best objective.
- **initialize_queue** (*pybnb.dispatcher.DispatcherQueueData*) – The initial queue.
- **queue_strategy** (*QueueStrategy*) – Sets the strategy for prioritizing nodes in the central dispatcher queue. See the *QueueStrategy* enum for the list of acceptable values.
- **converger** (*pybnb.convergence_checker.ConvergenceChecker*) – The branch-and-bound convergence checker object.
- **node_limit** (*int or None*) – An integer representing the maximum number of nodes to processes before beginning to terminate the solve. If None, no node limit will be enforced.
- **time_limit** (*float or None*) – The maximum amount of time to spend processing nodes before beginning to terminate the solve. If None, no time limit will be enforced.

- **queue_limit** (*int or None*) – The maximum allowed queue size. If exceeded, the solve will terminate. If None, no size limit on the queue will be enforced.
- **log** (logging.Logger) – A log object where solver output should be sent.
- **log_interval_seconds** (*float*) – The approximate maximum time (in seconds) between solver log updates. More time may pass between log updates if no updates have been received from any workers, and less time may pass if a new incumbent is found.
- **log_new_incumbent** (*bool*) – Controls whether updates to the best objective are logged immediately (overriding the log interval). Setting this to false can be useful when frequent updates to the incumbent are expected and the additional logging slows down the dispatcher.

log_info(*msg*)
Pass a message to log.info

log_warning(*msg*)
Pass a message to log.warning

log_debug(*msg*)
Pass a message to log.debug

log_error(*msg*)
Pass a message to log.error

log_critical(*msg*)
Pass a message to log.critical

save_dispatcher_queue()
Saves the current dispatcher queue. The result can be used to re-initialize a solve.

Returns **queue_data** – An object storing information that can be used to re-initialize the dispatcher queue to its current state.

Return type *pybnb.dispatcher.DispatcherQueueData*

class pybnb.dispatcher.**DispatcherLocal**
The central dispatcher for a serial branch-and-bound algorithm.

initialize(*best_objective*, *best_node*, *initialize_queue*, *queue_strategy*, *converger*, *node_limit*, *time_limit*, *queue_limit*, *track_bound*, *log*, *log_interval_seconds*, *log_new_incumbent*)
Initialize the dispatcher. See the *pybnb.dispatcher.DispatcherBase.initialize()* method for argument descriptions.

update(*best_objective*, *best_node*, *terminal_bound*, *solve_info*, *node_list*)
Update local worker information.

Parameters

- **best_objective** (*float or None*) – A new potential best objective found by the worker.
- **best_node** (*Node* or None) – A new potential best node found by the worker.
- **terminal_bound** (*float or None*) – The worst bound of any terminal nodes that were processed by the worker since the last update.
- **solve_info** (_SolveInfo) – The most up-to-date worker solve information.
- **node_list** (*list*) – A list of nodes to add to the queue.

Returns

- **solve_finished** (*bool*) – Indicates if the dispatcher has terminated the solve.
- **new_objective** (*float*) – The best objective known to the dispatcher.

- **best_node** (*Node* or None) – The best node known to the dispatcher.
- **data** (*Node* or None) – If solve_finished is false, a new node for the worker to process. Otherwise, a tuple containing the global bound, the termination condition string, and the number of explored nodes.

class pybnb.dispatcher.**DispatcherDistributed**(*comm*)

The central dispatcher for a distributed branch-and-bound algorithm.

Parameters **comm** (mpi4py.MPI.Comm, optional) – The MPI communicator to use. If set to None, this will disable the use of MPI and avoid an attempted import of *mpi4py.MPI* (which avoids triggering a call to *MPI_Init()*).

initialize(*best_objective*, *best_node*, *initialize_queue*, *queue_strategy*, *converger*, *node_limit*, *time_limit*, *queue_limit*, *track_bound*, *log*, *log_interval_seconds*, *log_new_incumbent*)

Initialize the dispatcher. See the *pybnb.dispatcher.DispatcherBase.initialize()* method for argument descriptions.

update(*best_objective*, *best_node*, *terminal_bound*, *solve_info*, *node_list*, *source*)

Update local worker information.

Parameters

- **best_objective** (*float or None*) – A new potential best objective found by the worker.
- **best_node** (*Node* or None) – A new potential best node found by the worker.
- **terminal_bound** (*float or None*) – The worst bound of any terminal nodes that were processed by the worker since the last update.
- **solve_info** (_SolveInfo) – The most up-to-date worker solve information.
- **node_list** (*list*) – A list of nodes to add to the queue.
- **source** (*int*) – The worker process rank that the update came from.

Returns

- **solve_finished** (*bool*) – Indicates if the dispatcher has terminated the solve.
- **new_objective** (*float*) – The best objective value known to the dispatcher.
- **best_node** (*Node* or None) – The best node known to the dispatcher.
- **data** (array.array or None) – If solve_finished is false, a data array representing a new node for the worker to process. Otherwise, a tuple containing the global bound, the termination condition string, and the number of explored nodes.

serve()

Start listening for distributed branch-and-bound commands and map them to commands in the local dispatcher interface.

save_dispatcher_queue()

Saves the current dispatcher queue. The result can be used to re-initialize a solve.

Returns **queue_data** – An object storing information that can be used to re-initialize the dispatcher queue to its current state.

Return type *pybnb.dispatcher.DispatcherQueueData*

4.2.10 pybnb.dispatcher_proxy

A proxy interface to the central dispatcher that is used by branch-and-bound workers.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

pybnb.dispatcher_proxy.**ProcessType = _ProcessType(worker=0, dispatcher=1)**
A namespace of typecodes that are used to categorize processes during dispatcher startup.

pybnb.dispatcher_proxy.**DispatcherAction = _DispatcherAction(update=111, finalize=211, log_info=311, log_warning=411, log_debug=511, log_error=611, log_critical=711, stop_listen=811)**
A namespace of typecodes that are used to categorize messages received by the dispatcher from workers.

pybnb.dispatcher_proxy.**DispatcherResponse = _DispatcherResponse(work=1111, nowork=2111)**
A namespace of typecodes that are used to categorize responses received by workers from the dispatcher.

class pybnb.dispatcher_proxy.**DispatcherProxy**(*comm*)
A proxy class for interacting with the central dispatcher via message passing.

update(*best_objective*, *best_node*, *previous_bound*, *solve_info*, *node_list_*)
A proxy to pybnb.dispatcher.Dispatcher.update().

log_info(*msg*)
A proxy to pybnb.dispatcher.Dispatcher.log_info().

log_warning(*msg*)
A proxy to pybnb.dispatcher.Dispatcher.log_warning().

log_debug(*msg*)
A proxy to pybnb.dispatcher.Dispatcher.log_debug().

log_error(*msg*)
A proxy to pybnb.dispatcher.Dispatcher.log_error().

log_critical(*msg*)
A proxy to pybnb.dispatcher.Dispatcher.log_critical().

stop_listen()
Tell the dispatcher to abruptly stop the listen loop.

4.2.11 pybnb.mpi_utils

Various utility function for MPI.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

class pybnb.mpi_utils.**Message**(*comm*)
A helper class for probing for and receiving messages. A single instance of this class is meant to be reused.

Parameters **comm** (mpi4py.MPI.Comm) – The MPI communicator to use.

probe(***kwds*)
Perform a blocking test for a message

recv(*datatype=None*, *data=None*)
Complete the receive for the most recent message probe and return the data as a numeric array or a string, depending on the datatype keyword.

Parameters

- **datatype** ({mpi4py.MPI.DOUBLE, mpi4py.MPI.CHAR}, optional) – An MPI datatype used to interpret the received data. If None, mpi4py.MPI.DOUBLE will be used. (default: None)
- **data** (*array.array or None, optional*) – An existing data array to store data into. If None, one will be created. (default: None)

pybnb.mpi_utils.**recv_nothing**(*comm*, *status*)

A helper function for receiving an empty message. This function is not thread safe.

Parameters

- **comm** (mpi4py.MPI.Comm) – An MPI communicator.
- **status** (mpi4py.MPI.Status) – An MPI status object that has been populated with information about the message to be received via a probe. If None, a new status object will be created and an empty message will be expected from any source with any tag. (default: None)

Returns **status** – If the original status argument was not None, it will be returned after being updated by the receive. Otherwise, the status object that was created will be returned.

Return type mpi4py.MPI.Status

pybnb.mpi_utils.**send_nothing**(*comm*, *dest*, *tag=0*)

A helper function for sending an empty message with a given tag. This function is not thread safe.

Parameters

- **comm** (mpi4py.MPI.Comm) – An MPI communicator.
- **dest** (*int*) – The process rank to send the message to.
- **tag** (*int, optional*) – A valid MPI tag to use for the message. (default: 0)

pybnb.mpi_utils.**recv_data**(*comm*, *status*, *datatype*, *out=None*)

A helper function for receiving numeric or string data sent using the lower-level buffer-based mpi4py routines.

Parameters

- **comm** (mpi4py.MPI.Comm) – An MPI communicator.
- **status** (mpi4py.MPI.Status) – An MPI status object that has been populated with information about the message to be received via a probe.
- **datatype** (mpi4py.MPI.Datatype) – An MPI datatype used to interpret the received data. If the datatype is mpi4py.MPI.CHAR, the received data will be converted to a string.
- **out** (*buffer-like object, optional*) – A buffer-like object that is compatible with the datatype argument and can be passed to comm.Recv. Can only be left as None when the datatype is mpi4py.MPI.CHAR.

Returns

Return type string or user-provided data buffer

for ... in pybnb.mpi_utils.**dispatched_partition**(*comm*, *items*, *root=0*)

A generator that partitions the list of items across processes in the communicator. If the communicator size is greater than 1, the root process will be yielded no items and instead will serve them dynamically by sending list indices to workers as work requests are received.

Parameters

- **comm** (mpi4py.MPI.Comm or None) – An MPI communicator or None in the serial processing case.

- **items** (*list*) – The list of items to partition. This This function assumes each process has an identical copy of the items list. Therefore, items in the list are not transferred (only indices).
- **root** (*integer, optional*) – An integer indicating which process rank should be designated as the dispatcher. (default: 0)

Returns

Return type string or user-provided data buffer

4.2.12 pybnb.misc

Miscellaneous utilities used for development.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

class pybnb.misc.**MPI_InterruptHandler**(*handler*, *disable=False*)
A context manager for temporarily assigning a handler to SIGINT and SIGUSR1, depending on the availability of these signals in the current OS.

pybnb.misc.**metric_format**(*num*, *unit='s'*, *digits=1*, *align_unit=False*)
Format and scale output with metric prefixes.

Example

```
>>> metric_format(0)
'0.0 s'
>>> metric_format(0, align_unit=True)
'0.0 s '
>>> metric_format(0.002, unit='B')
'2.0 mB'
>>> metric_format(2001, unit='B')
'2.0 KB'
>>> metric_format(2001, unit='B', digits=3)
'2.001 KB'
```

pybnb.misc.**time_format**(*num*, *digits=1*, *align_unit=False*)
Format and scale output according to standard time units.

Example

```
>>> time_format(0)
'0.0 s'
>>> time_format(0, align_unit=True)
'0.0 s '
>>> time_format(0.002)
'2.0 ms'
>>> time_format(2001)
'33.4 m'
>>> time_format(2001, digits=3)
'33.350 m'
```

pybnb.misc.**get_gap_labels**(*gap*, *key='gap'*, *format='f'*)
Get format strings with enough size and precision to print a given gap tolerance.

pybnb.misc.**as_stream**(*stream*, *mode='w'*, ***kwds*)

A utility for handling function arguments that can be a filename or a file object. This function is meant to be used in the context of a with statement.

Parameters

- **stream** (*file-like object or string*) – An existing file-like object or the name of a file to open.
- **mode** (*string*) – Assigned to the mode keyword of the built-in function `open` when the *stream* argument is a filename. (default: "w")
- ****kwds** – Additional keywords passed to the built-in function `open` when the *stream* argument is a filename.

Returns A file-like object that can be written to. If the input argument was originally an open file, a dummy context will wrap the file object so that it will not be closed upon exit of the with block.

Return type file-like object

Example

```
>>> import tempfile
>>> with tempfile.NamedTemporaryFile() as f:
...     # pass a file
...     with as_stream(f) as g:
...         assert g is f
...     assert not f.closed
...     f.close()
...     # pass a filename
...     with as_stream(f.name) as g:
...         assert not g.closed
...     assert g.closed
```

pybnb.misc.**get_default_args**(*func*)

Get the default arguments for a function as a dictionary mapping argument name to default value.

Example

```
>>> def f(a, b=None):
...     pass
>>> get_default_args(f)
{'b': None}
```

pybnb.misc.**get_keyword_docs**(*doc*)

Parses a numpy-style docstring to summarize information in the 'Parameters' section into a dictionary.

pybnb.misc.**get_simple_logger**(*name='<local>'*, *filename=None*, *stream=None*, *console=True*, *level=20*, *formatter=None*)

Creates a logging object configured to write to any combination of a file, a stream, and the console, or hide all output.

Parameters

- **name** (*string*) – The name assigned to the `logging.Logger` instance. (default: "<local>")
- **filename** (*string, optional*) – The name of a file to write to. (default: None)

- **stream** (*file-like object, optional*) – A file-like object to write to. (default: None)
- **console** (*bool, optional*) – If True, the logger will be configured to print output to the console through stdout and stderr. (default: True)
- **level** (*int, optional*) – The logging level to use. (default: `logging.INFO`)
- **formatter** (`logging.Formatter`, optional) – The logging formatter to use. (default: None)

Returns A logging object

Return type `logging.Logger`

pybnb.misc.**create_command_line_solver**(*problem*, *parser=None*)
Convert a given problem implementation to a command-line example by exposing the *pybnb.solver.solve()* function arguments using argparse.

4.2.13 pybnb.pyomo

pybnb.pyomo.misc

Miscellaneous utilities used for development.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

pybnb.pyomo.misc.**hash_joblist**(*jobs*)
Create a hash of a Python list by casting each entry to a string.

pybnb.pyomo.misc.**add_tmp_component**(*model*, *name*, *obj*)
Add a temporary component to a model, adjusting the name as needed to make sure it is unique.

pybnb.pyomo.misc.**correct_integer_lb**(*lb*, *integer_tolerance*)
Converts a lower bound for an integer optimization variable to an integer equal to *ceil(ub)*, taking care not to move a non-integer bound away from an integer point already within a given tolerance.

pybnb.pyomo.misc.**correct_integer_ub**(*ub*, *integer_tolerance*)
Converts an upper bound for an integer optimization variable to an integer equal to *floor(ub)*, taking care not to move a non-integer bound away from an integer point already within a given tolerance.

pybnb.pyomo.misc.**create_optimality_bound**(*problem*, *pyomo_objective*, *best_objective_value*)
Returns a constraint that bounds an objective function with a known best value. That is, the constraint will require the objective function to be better than the given value.

pybnb.pyomo.misc.**generate_cids**(*model*, *prefix=()*, ***kwds*)
Generate forward and reverse mappings between model components and deterministic, unique identifiers that are safe to serialize or use as dictionary keys.

pybnb.pyomo.problem

A Base class for defining a branch-and-bound problem based on a pyomo.kernel model.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

class pybnb.pyomo.problem.**PyomoProblem**(**args*, ***kwds*)
An extension of the *pybnb.Problem* base class for defining problems with a core Pyomo model.

property pyomo_object_to_cid
The map from pyomo model object to component id.

property cid_to_pyomo_object
The map from component id to pyomo model object.

property pyomo_model
Returns the pyomo model for this problem.

Note: This method is abstract and must be defined by the user.

property pyomo_model_objective
Returns the pyomo model objective for this problem.

Note: This method is abstract and must be defined by the user.

sense()
Returns the objective sense for this problem.

Note: This method is abstract and must be defined by the user.

pybnb.pyomo.range_reduction

A Problem interface for implementing parallel range reduction on a PyomoProblem during a branch-and-bound solve.

Copyright by Gabriel A. Hackebeil (gabe.hackebeil@gmail.com).

class pybnb.pyomo.range_reduction.**RangeReductionProblem**(*problem*, *best_objective=None*, *comm=None*)
A specialized implementation of the *pybnb.Problem* interface that can be used to perform optimality-based range reduction on a fully implemented PyomoProblem by defining additional abstract methods.

listen(*root=0*)
Listen for requests to run range reduction. All processes within the communicator, except for the root process, should call this method.

Parameters **root** (*int*) – The rank of the process acting as the root. The root process should not call this function.

sense()
Returns the objective sense for this problem.

Note: This method is abstract and must be defined by the user.

objective()
Returns a feasible value for the objective of the current problem state or *self.infeasible_objective()* if the current state is not feasible.

Note: This method is abstract and must be defined by the user.

bound()
Returns a value that is a bound on the objective of the current problem state or *self.unbounded_objective()* if no non-trivial bound is available.

Note: This method is abstract and must be defined by the user.

save_state(*node*)
Saves the current problem state into the given *pybnb.node.Node* object.

This method is guaranteed to be called once at the start of the solve by all processes involved to collect the root node problem state, but it may be called additional times. When it is called for the root node, the *node.tree_depth* will be zero.

Note: This method is abstract and must be defined by the user.

load_state(*node*)
Loads the problem state that is stored on the given *pybnb.node.Node* object.

Note: This method is abstract and must be defined by the user.

branch()
Returns a list of *Node* objects that partition the node state into zero or more children. This method can also be defined as a generator.

Note: This method is abstract and must be defined by the user.

notify_new_best_node(*node*, *current*)
Called when a branch-and-bound solver receives a new best node from the dispatcher. The *Problem* base class provides a default implementation for this method that does nothing.

Parameters

- **node** (*Node*) – The new best node.
- **current** (*bool*) – Indicates whether or not the node argument is the currently loaded node (from the most recent `load_state` call).

notify_solve_finished(*comm*, *worker_comm*, *results*)
Called when a branch-and-bound solver finishes. The *Problem* base class provides a default implementation for this method that does nothing.

Parameters

- **comm** (`mpi4py.MPI.Comm` or `None`) – The full MPI communicator that includes all processes. Will be None if MPI has been disabled.
- **worker_comm** (`mpi4py.MPI.Comm` or `None`) – The MPI communicator that includes only worker processes. Will be None if MPI has been disabled.
- **results** (*SolverResults*) – The fully populated results container that will be returned from the solver.

range_reduction_model_setup()
Called prior to starting range reduction solves to set up the Pyomo model

range_reduction_objective_changed(*objective*)
Called to notify that the range reduction routine has changed the objective

range_reduction_constraint_added(*constraint*)
Called to notify that the range reduction routine has added a constraint

range_reduction_constraint_removed(*constraint*)
Called to notify that the range reduction routine has removed a constraint

range_reduction_get_objects()
Called to collect the set of objects over which to perform range reduction solves

range_reduction_solve_for_object_bound(*x*)
Called to perform a range reduction solve for a Pyomo model object

range_reduction_model_cleanup()
Called after range reduction has finished to allow the user to execute any cleanup to the Pyomo model.

range_reduction_process_bounds(*objects*, *lower_bounds*, *upper_bounds*)
Called to process the bounds obtained by the range reduction solves

4.2.14 pybnb.futures

class pybnb.futures.**NestedSolver**(*problem*, *node_limit=None*, *time_limit=5*, *queue_limit=None*, *track_bound=True*, *queue_strategy='depth'*)

A class for creating a nested branch-and-bound solve strategy. An instance of this class (wrapped around a standard problem) can be passed to the solver as the problem argument.

Parameters

- **problem** (*pybnb.Problem*) – An object defining a branch-and-bound problem.
- **node_limit** (*int, optional*) – The same as the standard solver option, but applied to the nested solver to limit the number of nodes to explore when processing a work item. (default: None)
- **time_limit** (*float, optional*) – The same as the standard solver option, but applied to the nested solver to limit the amount of time spent processing a work item. (default: 5)
- **queue_limit** (*int, optional*) – The same as the standard solver option, but applied to the nested solver to limit the size of the queue. (default: None)
- **track_bound** (*bool, optional*) – The same as the standard solver option, but applied to the nested solver control bound tracking. (default: True)
- **queue_strategy** (*QueueStrategy* or tuple) – The same as the standard solver option, but applied to the nested solver to control the queue strategy used when processing a work item. (default: 'depth')

objective()
The solver does not call this method when it sees the problem implements a nested solve.

bound()
The solver does not call this method when it sees the problem implements a nested solve.

branch()
The solver does not call this method when it sees the problem implements a nested solve.

sense()
Calls the sense() method on the user-provided problem.

save_state(*node*)
Calls the save_state() method on the user-provided problem.

load_state(*node*)
Calls the load_state() method on the user-provided problem and prepares for a nested solve.

notify_solve_begins(*comm*, *worker_comm*, *convergence_checker*)
Calls the notify_solve_begins() method on the user-provided problem and prepares for a solve.

notify_new_best_node(*node*, *current*)
Calls the notify_new_best_node() method on the user-provided problem and stores the best node for use in the next nested solve.

notify_solve_finished(*comm*, *worker_comm*, *results*)
Calls the notify_solve_finished() method on the user-provided problem and does some final cleanup.

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

H

I

L

M

N

O

P

T

U

W