lab Documentation

Jendrik Seipp

Jun 07, 2019

Contents

The **Downward Lab** Python package facilitates running experiments for the Fast Downward planning system. It uses the generic experimentation package **Lab**.

Lab is a Python package for running code on a large set of benchmarks. Experiments can be conducted on a single machine or on a cluster. The package also contains code for parsing results and creating reports. Currently, Lab comes bundled with Downward Lab.

Code: https://bitbucket.org/jendrikseipp/lab

Documentation: https://lab.readthedocs.io

Cite: please cite Downward Lab by using

```
@Misc{seipp-et-al-misc2017,
  author =       "Jendrik Seipp and Florian Pommerening and
                  Silvan Sievers and Malte Helmert",
  title =        "{Downward} {Lab}",
  year =         "2017",
  doi =          "10.5281/zenodo.790461",
  url =          "https://doi.org/10.5281/zenodo.790461",
  howpublished = "\url{https://doi.org/10.5281/zenodo.790461}"
}
```

CHAPTER 1

Documentation for Lab

1.1 Lab tutorial

1.1.1 Install Lab

```
# Choose destination for Lab.
LAB=/path/to/lab
# Install dependencies.
sudo apt-get install mercurial python2.7 python-matplotlib python-simplejson
# Clone Lab repository.
hg clone https://bitbucket.org/jendrikseipp/lab ${LAB}
```

Notes about requirements:

- Lab needs Python 2.7 or Python >= 3.5.
- python-matplotlib is only needed for reports that generate graphs.
- python-simplejson is optional, but makes generating reports much faster.

Add to your `.bashrc` to make Lab available on the `PYTHONPATH`:

```
export PYTHONPATH=/path/to/lab
```

Update shell configuration:

```
source ~/.bashrc
```

1.1.2 Run tutorial experiment

The following script shows a simple experiment that runs a naive vertex cover solver on a set of benchmarks. You can find the whole experiment under `examples/vertex-cover/`.

```
#! /usr/bin/env python

"""
Example experiment using a simple vertex cover solver.
"""

import glob
import os
import platform

from lab.environments import LocalEnvironment, BaselSlurmEnvironment
from lab.experiment import Experiment
from lab.reports import Attribute

from downward.reports.absolute import AbsoluteReport


# Create custom report class with suitable info and error attributes.
class BaseReport(AbsoluteReport):
    INFO_ATTRIBUTES = ['time_limit', 'memory_limit', 'seed']
    ERROR_ATTRIBUTES = [
        'domain', 'problem', 'algorithm', 'unexplained_errors', 'error', 'node']


NODE = platform.node()
REMOTE = NODE.endswith(".scicore.unibas.ch") or NODE.endswith(".cluster.bc2.ch")
SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
BENCHMARKS_DIR = os.path.join(SCRIPT_DIR, "benchmarks")
BHOSLIB_GRAPHS = sorted(glob.glob(os.path.join(BENCHMARKS_DIR, 'bhoslib', '*.mis')))
RANDOM_GRAPHS = sorted(glob.glob(os.path.join(BENCHMARKS_DIR, 'random', '*.txt')))
ALGORITHMS = ["2approx", "greedy"]
SEED = 2018
TIME_LIMIT = 1800
MEMORY_LIMIT = 2048

if REMOTE:
    ENV = BaselSlurmEnvironment(email="my.name@unibas.ch")
    SUITE = BHOSLIB_GRAPHS + RANDOM_GRAPHS
else:
    ENV = LocalEnvironment(processes=4)
    # Use smaller suite for local tests.
    SUITE = BHOSLIB_GRAPHS[:1] + RANDOM_GRAPHS[:1]
ATTRIBUTES = [
    'cover', 'cover_size', 'error', 'solve_time', 'solver_exit_code',
    Attribute('solved', absolute=True)]

# Create a new experiment.
exp = Experiment(environment=ENV)
# Add solver to experiment and make it available to all runs.
exp.add_resource('solver', os.path.join(SCRIPT_DIR, 'solver.py'))
# Add custom parser.
exp.add_parser('parser.py')

for algo in ALGORITHMS:
    for task in SUITE:
        run = exp.add_run()
        # Create a symbolic link and an alias. This is optional. We
```

(continues on next page)

(continued from previous page)

```
        # could also use absolute paths in add_command().
        run.add_resource('task', task, symlink=True)
        run.add_command(
            'solve',
            ['{solver}', '--seed', str(SEED), '{task}', algo],
            time_limit=TIME_LIMIT,
            memory_limit=MEMORY_LIMIT)
        # AbsoluteReport needs the following attributes:
        # 'domain', 'problem' and 'algorithm'.
        domain = os.path.basename(os.path.dirname(task))
        task_name = os.path.basename(task)
        run.set_property('domain', domain)
        run.set_property('problem', task_name)
        run.set_property('algorithm', algo)
        # BaseReport needs the following properties:
        # 'time_limit', 'memory_limit', 'seed'.
        run.set_property('time_limit', TIME_LIMIT)
        run.set_property('memory_limit', MEMORY_LIMIT)
        run.set_property('seed', SEED)
        # Every run has to have a unique id in the form of a list.
        run.set_property('id', [algo, domain, task_name])

# Add step that writes experiment files to disk.
exp.add_step('build', exp.build)

# Add step that executes all runs.
exp.add_step('start', exp.start_runs)

# Add step that collects properties from run directories and
# writes them to *-eval/properties.
exp.add_fetcher(name='fetch')

# Make a report.
exp.add_report(
    BaseReport(attributes=ATTRIBUTES),
    outfile='report.html')

# Parse the commandline and run the given steps.
exp.run_steps()
```

You can see the available steps with

```
./exp.py
```

Select steps by name or index:

```
./exp.py build
./exp.py 2
./exp.py 3 4
```

Here is the parser that the experiment uses:

```
#! /usr/bin/env python

"""
Solver example output:

```

(continues on next page)

(continued from previous page)

```
Algorithm: 2approx
Cover: set([1, 3, 5, 6, 7, 8, 9])
Cover size: 7
Solve time: 0.000771s
"""

from lab.parser import Parser


def solved(content, props):
    props['solved'] = int('cover' in props)


def error(content, props):
    if props['solved']:
        props['error'] = 'cover-found'
    else:
        props['error'] = 'unsolved'


if __name__ == '__main__':
    parser = Parser()
    parser.add_pattern(
        'node', r'node: (.+)\n', type=str, file='driver.log', required=True)
    parser.add_pattern(
        'solver_exit_code', r'solve exit code: (.+)\n', type=int, file='driver.log')
    parser.add_pattern('cover', r'Cover: set\(\[(.*)\]\)', type=str)
    parser.add_pattern('cover_size', r'Cover size: (\d+)\n', type=int)
    parser.add_pattern('solve_time', r'Solve time: (.+)s', type=float)
    parser.add_function(solved)
    parser.add_function(error)
    parser.parse()
```

Have a look at other example experiments under `examples/` or go directly to the Lab API.

1.2 `lab.experiment` — Create experiments

1.2.1 Experiment

class lab.experiment.**Experiment**(*path=None*, *environment=None*)

Base class for Lab experiments.

An **experiment** consists of multiple **steps**. Most experiments will have steps for building and executing the experiment:

```
>>> exp = Experiment()
>>> exp.add_step('build', exp.build)
>>> exp.add_step('start', exp.start_runs)
```

Moreover, there are usually steps for fetching the results and making reports:

```
>>> from lab.reports import Report
>>> exp.add_fetcher(name='fetch')
>>> exp.add_report(Report(attributes=["error"]))
```

When calling *start_runs()*, all **runs** part of the experiment are executed. You can add runs with the *add_run()* method. Each run needs a unique ID and at least one **command**:

```
>>> for algo in ["algo1", "algo2"]:
...     for value in range(10):
...         run = exp.add_run()
...         run.set_property('id', [algo, str(value)])
...         run.add_command('solve', [algo, str(value)])
```

You can pass the names of selected steps to your experiment script or use `--all` to execute all steps. At the end of your script, call `exp.run_steps()` to parse the commandline and execute the selected steps.

The experiment will be built at *path*. It defaults to `<scriptdir>/data/<scriptname>/`. E.g., for the script `experiments/myexp.py`, the default *path* will be `experiments/data/myexp/`.

environment must be an *Environment* instance. You can use *LocalEnvironment* to run your experiment on a single computer (default). If you have access to the computer grid in Basel you can use the predefined grid environment *BaselSlurmEnvironment*. Alternatively, you can derive your own class from *Environment*.

add_command(*name*, *command*, *time_limit=None*, *memory_limit=None*, *soft_stdout_limit=1024*, *hard_stdout_limit=10240*, *soft_stderr_limit=64*, *hard_stderr_limit=10240*, ***kwargs*)

Call an executable.

If invoked on a *run*, this method adds the command to the **specific** run. If invoked on the experiment, the command is appended to the list of commands of **all** runs.

name is a string describing the command. It must start with a letter and consist exclusively of letters, numbers, underscores and hyphens.

command has to be a list of strings where the first item is the executable.

After *time_limit* seconds the signal SIGXCPU is sent to the command. The process can catch this signal and exit gracefully. If it doesn't catch the SIGXCPU signal, the command is aborted with SIGKILL after five additional seconds.

The command is aborted with SIGKILL when it uses more than *memory_limit* MiB.

You can limit the log size (in KiB) with a soft and hard limit for both stdout and stderr. When the soft limit is hit, an unexplained error is registered for this run, but the command is allowed to continue running. When the hard limit is hit, the command is killed with SIGTERM. This signal can be caught and handled by the process.

By default, there are limits for the log and error output, but time and memory are not restricted.

All *kwargs* (except `stdin`) are passed to subprocess.Popen. Instead of file handles you can also pass filenames for the `stdout` and `stderr` keyword arguments. Specifying the `stdin` kwarg is not supported.

```
>>> exp = Experiment()
>>> run = exp.add_run()
>>> # Add commands to a *specific* run.
>>> run.add_command('list-directory', ['ls', '-al'])
>>> run.add_command(
...     'solver', ['mysolver', 'input-file'], time_limit=60)
>>> # Add a command to *all* runs.
>>> exp.add_command('cleanup', ['rm', 'my-temp-file'])
```

add_fetcher(*src=None*, *dest=None*, *merge=None*, *name=None*, *filter=None*, ***kwargs*)

Add a step that fetches results from experiment or evaluation directories into a new or existing evaluation directory.

You can use this method to combine results from multiple experiments.

src can be an experiment or evaluation directory. It defaults to `exp.path`.

dest must be a new or existing evaluation directory. It defaults to `exp.eval_dir`. If *dest* already contains data and *merge* is set to None, the user will be prompted whether to override the existing data or to merge the old and new data. Setting *merge* to True or to False has the effect that the old data is merged or replaced (and the user will not be prompted).

If no *name* is given, call this step "fetch-`basename(src)`".

You can fetch only a subset of runs (e.g., runs for specific domains or algorithms) by passing *filters* with the *filter* argument.

Example setup:

```
>>> exp = Experiment('/tmp/exp')
```

Fetch all results and write a single combined properties file to the default evaluation directory (this step is added by default):

```
>>> exp.add_fetcher(name='fetch')
```

Merge the results from "other-exp" into this experiment's results:

```
>>> exp.add_fetcher(src='/path/to/other-exp-eval')
```

Fetch only the runs for certain algorithms:

```
>>> exp.add_fetcher(filter_algorithm=['algo_1', 'algo_5'])
```

add_new_file(*name*, *dest*, *content*, *permissions=420*)

Write *content* to /path/to/exp-or-run/*dest* and make the new file available to the commands as *name*.

name is an alias for the resource in commands. It must start with a letter and consist exclusively of letters, numbers and underscores.

```
>>> exp = Experiment()
>>> run = exp.add_run()
>>> run.add_new_file('learn', 'learn.txt', 'a = 5; b = 2; c = 5')
>>> run.add_command('print-trainingset', ['cat', '{learn}'])
```

add_parse_again_step()

Add a step that copies the parsers from their originally specified locations to the experiment directory and runs all of them again. This step overwrites the existing properties file in each run dir.

Do not forget to run the default fetch step again to overwrite existing data in the -eval dir of the experiment.

add_parser(*path_to_parser*)

Add a parser to each run of the experiment.

Add the parser as a resource to the experiment and add a command that executes the parser to each run. Since commands are executed in the order they are added, parsers should be added after all other commands. If you need to change your parsers and execute them again you can use the *add_parse_again_step()* method.

path_to_parser must be the path to an executable file. The parser is executed in the run directory and manipulates the run's "properties" file. The last part of the filename (without the extension) is used as a resource name. Therefore, it must be unique among all parsers and other resources. Also, it must start with a letter and contain only letters, numbers, underscores and dashes (which are converted to underscores automatically).

For information about how to write parsers see *lab.parser — Parse logs*.

add_report (*report*, *name=''*, *eval_dir=''*, *outfile=''*)
Add *report* to the list of experiment steps.

This method is a shortcut for `add_step(name, report, eval_dir, outfile)` and uses sensible defaults for omitted arguments.

If no *name* is given, use *outfile* or the *report*'s class name.

By default, use the experiment's standard *eval_dir*.

If *outfile* is omitted, compose a filename from *name* and the *report*'s format. If *outfile* is a relative path, put it under *eval_dir*.

```
>>> from downward.reports.absolute import AbsoluteReport
>>> exp = Experiment("/tmp/exp")
>>> exp.add_report(AbsoluteReport(attributes=["coverage"]))
```

add_resource (*name*, *source*, *dest=''*, *symlink=False*)
Include the file or directory *source* in the experiment or run.

name is an alias for the resource in commands. It must start with a letter and consist exclusively of letters, numbers and underscores. If you don't need an alias for the resource, set name=''.

source is copied to /path/to/exp-or-run/*dest*. If *dest* is omitted, the last part of the path to *source* will be taken as the destination filename. If you only want an alias for your resource, but don't want to copy or link it, set *dest* to None.

Example:

```
>>> exp = Experiment()
>>> exp.add_resource('planner', 'path/to/planner')
```

includes my-planner in the experiment directory. You can use `{planner}` to reference my-planner in a run's commands:

```
>>> run = exp.add_run()
>>> run.add_resource('domain', 'path-to/gripper/domain.pddl')
>>> run.add_resource('task', 'path-to/gripper/prob01.pddl')
>>> run.add_command('plan', ['{planner}', '{domain}', '{task}'])
```

add_run (*run=None*)
Schedule *run* to be part of the experiment.

If *run* is None, create a new run, add it to the experiment and return it.

add_step (*name*, *function*, **args*, ***kwargs*)
Add a step to the list of experiment steps.

Use this method to add experiment steps like writing the experiment file to disk, removing directories and publishing results. To add fetch and report steps, use the convenience methods *add_fetcher()* and *add_report()*.

name is a descriptive name for the step. When selecting steps on the command line, you may either use step names or their indices.

function must be a callable Python object, e.g., a function or a class implementing *__call__*.

args and *kwargs* will be passed to *function* when the step is executed.

```
>>> import shutil
>>> import subprocess
```

(continues on next page)

(continued from previous page)

```
>>> from lab.experiment import Experiment
>>> exp = Experiment('/tmp/myexp')
>>> exp.add_step('build', exp.build)
>>> exp.add_step('start', exp.start_runs)
>>> exp.add_step('rm-eval-dir', shutil.rmtree, exp.eval_dir)
>>> exp.add_step('greet', subprocess.call, ['echo', 'Hello'])
```

build(*write_to_disk=True*)
Finalize the internal data structures, then write all files needed for the experiment to disk.

If *write_to_disk* is False, only compute the internal data structures. This is only needed on grids for FastDownwardExperiments.build() which turns the added algorithms and benchmarks into Runs.

eval_dir
Return the name of the default evaluation directory.

This is the directory where the fetched and parsed results will land by default.

name
Return the directory name of the experiment's `path`.

run_steps()
Parse the commandline and run selected steps.

set_property(*name*, *value*)
Add a key-value property.

These can be used later, for example, in reports.

```
>>> exp = Experiment()
>>> exp.set_property('suite', ['gripper', 'grid'])
>>> run = exp.add_run()
>>> run.set_property('domain', 'gripper')
>>> run.set_property('problem', 'prob01.pddl')
```

Each run must have the property *id* which must be a *unique* list of strings. They determine where the results for this run will land in the combined properties file.

```
>>> run.set_property('id', ["algo1", "task1"])
>>> run.set_property('id', ["algo2", "domain1", "problem1"])
```

start_runs()
Execute all runs that were added to the experiment.

Depending on the selected environment this method will start the runs locally or on a computer grid.

Custom command line arguments

lab.experiment.**ARGPARSER**
ArgumentParser instance that can be used to add custom command line arguments. You can import it, add your arguments and call its `parse_args()` method to retrieve the argument values. To avoid confusion with step names you shouldn't use positional arguments.

Note: Custom command line arguments are only passed to locally executed steps.

```
from lab.experiment import ARGPARSER

ARGPARSER.add_argument(
    "--test",
    choices=["yes", "no"],
    required=True,
    dest="test_run",
    help="run experiment on small suite locally")

args = ARGPARSER.parse_args()
if args.test_run:
    print "perform test run"
else:
    print "run real experiment"
```

1.2.2 Run

class lab.experiment.**Run**(*experiment*)

An experiment consists of multiple runs. There should be one run for each (algorithm, benchmark) pair.

A run consists of one or more commands.

experiment must be an *Experiment* instance.

add_command(*name*, *command*, *time_limit=None*, *memory_limit=None*, *soft_stdout_limit=1024*, *hard_stdout_limit=10240*, *soft_stderr_limit=64*, *hard_stderr_limit=10240*, ***kwargs*)

Call an executable.

If invoked on a *run*, this method adds the command to the **specific** run. If invoked on the experiment, the command is appended to the list of commands of **all** runs.

name is a string describing the command. It must start with a letter and consist exclusively of letters, numbers, underscores and hyphens.

command has to be a list of strings where the first item is the executable.

After *time_limit* seconds the signal SIGXCPU is sent to the command. The process can catch this signal and exit gracefully. If it doesn't catch the SIGXCPU signal, the command is aborted with SIGKILL after five additional seconds.

The command is aborted with SIGKILL when it uses more than *memory_limit* MiB.

You can limit the log size (in KiB) with a soft and hard limit for both stdout and stderr. When the soft limit is hit, an unexplained error is registered for this run, but the command is allowed to continue running. When the hard limit is hit, the command is killed with SIGTERM. This signal can be caught and handled by the process.

By default, there are limits for the log and error output, but time and memory are not restricted.

All *kwargs* (except `stdin`) are passed to subprocess.Popen. Instead of file handles you can also pass filenames for the `stdout` and `stderr` keyword arguments. Specifying the `stdin` kwarg is not supported.

```
>>> exp = Experiment()
>>> run = exp.add_run()
>>> # Add commands to a *specific* run.
>>> run.add_command('list-directory', ['ls', '-al'])
>>> run.add_command(
...     'solver', ['mysolver', 'input-file'], time_limit=60)
```

(continues on next page)

(continued from previous page)

```
>>> # Add a command to *all* runs.
>>> exp.add_command('cleanup', ['rm', 'my-temp-file'])
```

add_new_file(*name*, *dest*, *content*, *permissions=420*)

Write *content* to /path/to/exp-or-run/*dest* and make the new file available to the commands as *name*.

name is an alias for the resource in commands. It must start with a letter and consist exclusively of letters, numbers and underscores.

```
>>> exp = Experiment()
>>> run = exp.add_run()
>>> run.add_new_file('learn', 'learn.txt', 'a = 5; b = 2; c = 5')
>>> run.add_command('print-trainingset', ['cat', '{learn}'])
```

add_resource(*name*, *source*, *dest=''*, *symlink=False*)

Include the file or directory *source* in the experiment or run.

name is an alias for the resource in commands. It must start with a letter and consist exclusively of letters, numbers and underscores. If you don't need an alias for the resource, set name=''.

source is copied to /path/to/exp-or-run/*dest*. If *dest* is omitted, the last part of the path to *source* will be taken as the destination filename. If you only want an alias for your resource, but don't want to copy or link it, set *dest* to None.

Example:

```
>>> exp = Experiment()
>>> exp.add_resource('planner', 'path/to/planner')
```

includes my-planner in the experiment directory. You can use `{planner}` to reference my-planner in a run's commands:

```
>>> run = exp.add_run()
>>> run.add_resource('domain', 'path-to/gripper/domain.pddl')
>>> run.add_resource('task', 'path-to/gripper/prob01.pddl')
>>> run.add_command('plan', ['{planner}', '{domain}', '{task}'])
```

set_property(*name*, *value*)

Add a key-value property.

These can be used later, for example, in reports.

```
>>> exp = Experiment()
>>> exp.set_property('suite', ['gripper', 'grid'])
>>> run = exp.add_run()
>>> run.set_property('domain', 'gripper')
>>> run.set_property('problem', 'prob01.pddl')
```

Each run must have the property *id* which must be a *unique* list of strings. They determine where the results for this run will land in the combined properties file.

```
>>> run.set_property('id', ["algo1", "task1"])
>>> run.set_property('id', ["algo2", "domain1", "problem1"])
```

1.2.3 Environment

class lab.environments.**Environment**(*randomize_task_order=True*)
Abstract base class for all environments.

If *randomize_task_order* is True (default), tasks for runs are started in a random order. This is useful to avoid systematic noise due to, e.g., one of the algorithms being run on a machine with heavy load. Note that due to the randomization, run directories may be pristine while the experiment is running even though the logs say the runs are finished.

class lab.environments.**LocalEnvironment**(*processes=None*, ***kwargs*)
Environment for running experiments locally on a single machine.

If given, *processes* must be between 1 and #CPUs. If omitted, it will be set to #CPUs.

See *Environment* for inherited parameters.

class lab.environments.**GridEnvironment**(*email=None*, *extra_options=None*, ***kwargs*)
Abstract base class for grid environments.

If the main experiment step is part of the selected steps, the selected steps are submitted to the grid engine. Otherwise, the selected steps are run locally.

Note: If the steps are run by the grid engine, this class writes job files to the directory `<exppath>-grid-steps` and makes them depend on one another. Please inspect the *.log and *.err files in this directory if something goes wrong. Since the job files call the experiment script during execution, it mustn't be changed during the experiment.

If *email* is provided and the steps run on the grid, a message will be sent when the last experiment step finishes.

Use *extra_options* to pass additional options. The *extra_options* string may contain newlines. Slurm example that reserves two cores per run:

```
extra_options='#SBATCH --cpus-per-task=2'
```

See *Environment* for inherited parameters.

class lab.environments.**BaselSlurmEnvironment**(*partition=None*, *qos=None*, *memory_per_cpu=None*, *export=None*, *setup=None*, ***kwargs*)
Environment for Basel's AI group.

partition must be a valid Slurm partition name. In Basel you can choose from

- "infai_1": 24 nodes with 16 cores, 64GB memory, 500GB Sata (default)
- "infai_2": 24 nodes with 20 cores, 128GB memory, 240GB SSD

qos must be a valid Slurm QOS name. In Basel this must be "normal".

memory_per_cpu must be a string specifying the memory allocated for each core. The string must end with one of the letters K, M or G. The default is "3872M". The value for *memory_per_cpu* should not surpass the amount of memory that is available per core, which is "3872M" for infai_1 and "6354M" for infai_2. Processes that surpass the *memory_per_cpu* limit are terminated with SIGKILL. To impose a soft limit that can be caught from within your programs, you can use the `memory_limit` kwarg of *add_command()*. Fast Downward users should set memory limits via the `driver_options`.

Slurm limits the memory with cgroups. Unfortunately, this often fails on our nodes, so we set our own soft memory limit for all Slurm jobs. We derive the soft memory limit by multiplying the value denoted by the

memory_per_cpu parameter with 0.98 (the Slurm config file contains "AllowedRAMSpace=99" and we add some slack). We use a soft instead of a hard limit so that child processes can raise the limit.

Examples that reserve the maximum amount of memory available per core:

```
>>> env1 = BaselSlurmEnvironment(partition="infai_1", memory_per_cpu="3872M")
>>> env2 = BaselSlurmEnvironment(partition="infai_2", memory_per_cpu="6354M")
```

Example that reserves 12 GiB of memory on infai_1:

```
>>> # 12 * 1024 / 3872 = 3.17 -> round to next int -> 4 cores per task
>>> # 12G / 4 = 3G per core
>>> env = BaselSlurmEnvironment(
...     partition="infai_1",
...     memory_per_cpu="3G",
...     extra_options='#SBATCH --cpus-per-task=4')
```

Example that reserves 12 GiB of memory on infai_2:

```
>>> # 12 * 1024 / 6354 = 1.93 -> round to next int -> 2 cores per task
>>> # 12G / 2 = 6G per core
>>> env = BaselSlurmEnvironment(
...     partition="infai_2",
...     memory_per_cpu="6G",
...     extra_options='#SBATCH --cpus-per-task=2')
```

Use *export* to specify a list of environment variables that should be exported from the login node to the compute nodes (default: ["PATH"]).

You can alter the environment in which the experiment runs with the **setup** argument. If given, it must be a string of Bash commands. If omitted, *BaselSlurmEnvironment* adds Lab to the PYTHONPATH.

See *GridEnvironment* for inherited parameters.

1.2.4 Various

`lab.__version__`
Lab version number. A "+" is appended to all non-tagged revisions.

1.3 `lab.parser` — Parse logs

Parse logs and output files.

A parser can be any program that analyzes files in the run's directory (e.g. `run.log`) and manipulates the `properties` file in the same directory.

To make parsing easier, however, you can use the `Parser` class. The parser `examples/ff/ff-parser.py` serves as an example:

```
#! /usr/bin/env python

"""
FF example output:

[...]
```

(continues on next page)

(continued from previous page)

```
ff: found legal plan as follows

step    0: UP F0 F1
        1: BOARD F1 P0
        2: DOWN F1 F0
        3: DEPART F0 P0


time spent:    0.00 seconds instantiating 4 easy, 0 hard action templates
               0.00 seconds reachability analysis, yielding 4 facts and 4 actions
               0.00 seconds creating final representation with 4 relevant facts
               0.00 seconds building connectivity graph
               0.00 seconds searching, evaluating 5 states, to a max depth of 2
               0.00 seconds total time
"""

import re

from lab.parser import Parser


def error(content, props):
    if props['planner_exit_code'] == 0:
        props['error'] = 'plan-found'
    else:
        props['error'] = 'unsolvable-or-error'


def coverage(content, props):
    props['coverage'] = int(props['planner_exit_code'] == 0)


def get_plan(content, props):
    # All patterns are parsed before functions are called.
    if props.get('evaluations') is not None:
        props['plan'] = re.findall(r'^(?:step)?\s*\d+: (.+)$', content, re.M)


def get_times(content, props):
    props['times'] = re.findall(r'(\d+\.\d+) seconds', content)


def trivially_unsolvable(content, props):
    props['trivially_unsolvable'] = int(
        'ff: goal can be simplified to FALSE. No plan will solve it' in content)


parser = Parser()
parser.add_pattern(
    'node', r'node: (.+)\n', type=str, file='driver.log', required=True)
parser.add_pattern(
    'planner_exit_code', r'run-planner exit code: (.+)\n', type=int, file='driver.log
↪')
parser.add_pattern('evaluations', r'evaluating (\d+) states')
parser.add_function(error)
parser.add_function(coverage)
parser.add_function(get_plan)
```

(continues on next page)

(continued from previous page)

```
parser.add_function(get_times)
parser.add_function(trivially_unsolvable)
parser.parse()
```

You can add this parser to alls runs by using *add_parser()*:

```
>>> import os.path
>>> from lab import experiment
>>> exp = experiment.Experiment()
>>> # The path can be absolute or relative to the working directory
>>> # at build time.
>>> parser = os.path.abspath(
...     os.path.join(__file__, '../../examples/ff/ff-parser.py'))
>>> exp.add_parser(parser)
```

All added parsers will be run in the order in which they were added after executing the run's commands.

If you need to change your parsers and execute them again, use the *add_parse_again_step()* method to re-parse your results.

1.3.1 Parser API

class lab.parser.**Parser**

Parse files in the current directory and write results into the run's `properties` file.

add_function(*function*, *file='run.log'*)

Call `function(open(file).read(), properties)` during parsing.

Functions are applied **after** all patterns have been evaluated.

The function is passed the file contents and the properties dictionary. It must manipulate the passed properties dictionary. The return value is ignored.

Example:

```
>>> import re
>>> from lab.parser import Parser
>>> # Example content: f=14, f=12, f=10
>>> def find_f_values(content, props):
...     props['f_values'] = re.findall(r'f=(\d+)', content)
...
>>> parser = Parser()
>>> parser.add_function(find_f_values)
```

You can use `props.add_unexplained_error("message")` when your parsing function detects that something went wrong during the run.

add_pattern(*attribute*, *regex*, *file='run.log'*, *type=<type 'int'>*, *flags=''*, *required=False*)

Look for *regex* in *file*, cast what is found in brackets to *type* and store it in the properties dictionary under *attribute*. During parsing roughly the following code will be executed:

```
contents = open(file).read()
match = re.compile(regex).search(contents)
properties[attribute] = type(match.group(1))
```

flags must be a string of Python regular expression flags (see https://docs.python.org/2/library/re.html). E.g., `flags='M'` lets "^" and "$" match at the beginning and end of each line, respectively.

If *required* is True and the pattern is not found in *file*, an error message is printed to stderr.

```
>>> parser = Parser()
>>> parser.add_pattern('facts', r'Facts: (\d+)', type=int)
```

parse()
Search all patterns and apply all functions.

The found values are written to the run's `properties` file.

1.4 `lab.reports` – Make reports

lab.reports.**arithmetic_mean**(*values*)
Compute the arithmetic mean of a sequence of numbers.

```
>>> arithmetic_mean([20, 30, 70])
40.0
```

lab.reports.**geometric_mean**(*values*)
Compute the geometric mean of a sequence of numbers.

```
>>> round(geometric_mean([2, 8]), 2)
4.0
```

class lab.reports.**Attribute**(*name*, *absolute=False*, *min_wins=True*, *functions=<built-in function sum>*, *scale=None*, *digits=2*)
A string subclass for attributes in reports.

Use this class if your **own** attribute needs a non-default value for:

- *absolute*: If False, only include tasks for which all task runs have values in a domain-wise table (e.g. `coverage` is absolute, whereas `expansions` is not, because we can't compare algorithms A and B for task X if B has no value for `expansions`).
- *min_wins*: Set to True if a smaller value for this attribute is better, to False if a higher value is better and to None if values can't be compared. (E.g., *min_wins* is False for `coverage`, but it is True for `expansions`).
- *functions*: Set the function or functions used to group values of multiple runs for this attribute. The first entry is used to aggregate values for domain-wise reports (e.g. for `coverage` this is `sum()`, whereas `expansions` uses *`geometric_mean()`*). This can be a single function or a list of functions and defaults to `sum()`.
- *scale*: Default scaling. Can be one of "linear", "log" and "symlog". If *scale* is None (default), the reports will choose the scaling.
- *digits*: Number of digits after the decimal point.

The `downward` package automatically uses appropriate settings for most attributes.

```
>>> avg_h = Attribute('avg_h', min_wins=False)
>>> abstraction_done = Attribute(
...     'abstraction_done', absolute=True, min_wins=False)
```

class lab.reports.**Report**(*attributes=None*, *format='html'*, *filter=None*, ***kwargs*)
Base class for all reports.

Inherit from this or a child class to implement a custom report.

Depending on the type of output you want to make, you will have to overwrite the *write()*, *get_text()* or *get_markup()* method.

attributes is the list of attributes you want to include in your report. If omitted, use all numerical attributes. Globbing characters * and ? are allowed. Example:

```
>>> report = Report(attributes=['coverage', 'translator_*'])
```

When a report is made, both the available and the selected attributes are printed on the commandline.

format can be one of e.g. html, tex, wiki (MediaWiki), doku (DokuWiki), pmw (PmWiki), moin (MoinMoin) and txt (Plain text). Subclasses may allow additional formats.

If given, *filter* must be a function or a list of functions that are passed a dictionary of a run's attribute keys and values. Filters must return True, False or a new dictionary. Depending on the returned value, the run is included or excluded from the report, or replaced by the new dictionary, respectively.

Filters for properties can be given in shorter form without defining a function. To include only runs where attribute `foo` has value v, use `filter_foo=v`. To include only runs where attribute `foo` has value v1, v2 or v3, use `filter_foo=[v1, v2, v3]`.

Filters are applied sequentially, i.e., the first filter is applied to all runs before the second filter is executed. Filters given as `filter_*` kwargs are applied *after* all filters passed via the `filter` kwarg.

Examples:

Include only the "cost" attribute in a LaTeX report:

```
>>> report = Report(attributes=['cost'], format='tex')
```

Only include successful runs in the report:

```
>>> report = Report(filter_coverage=1)
```

Only include runs in the report where the initial h value is at most 100:

```
>>> def low_init_h(run):
...     return run['initial_h_value'] <= 100
>>> report = Report(filter=low_init_h)
```

Only include runs from "blocks" and "barman" with a timeout:

```
>>> report = Report(
...     filter_domain=['blocks', 'barman'],
...     filter_search_timeout=1)
```

Add a new attribute:

```
>>> def add_expansions_per_time(run):
...     expansions = run.get('expansions')
...     time = run.get('search_time')
...     if expansions is not None and time:
...         run['expansions_per_time'] = expansions / time
...     return run
>>> report = Report(
...     attributes=['expansions_per_time'],
...     filter=[add_expansions_per_time])
```

Rename, filter and sort algorithms:

```
>>> def rename_algorithms(run):
...     name = run['algorithm']
...     paper_names = {
...         'lama11': 'LAMA 2011', 'fdss_sat1': 'FDSS 1'}
...     run['algorithm'] = paper_names[name]
...     return run
```

```
>>> # We want LAMA 2011 to be the leftmost column.
>>> # filter_* filters are evaluated last, so we use the updated
>>> # algorithm names here.
>>> algorithms = ['LAMA 2011', 'FDSS 1']
>>> report = Report(
...     filter=rename_algorithms, filter_algorithm=algorithms)
```

Compute a new attribute from multiple runs (from *examples/showcase-options.py*):

```
class QualityFilters(object):
    """Compute the IPC quality score.

    The IPC score is computed over the list of runs for each task. Since
    filters only work on individual runs, we can't compute the score
    with a single filter, but it is possible by using two filters:
    *store_costs* saves the list of costs per task in a dictionary
    whereas *add_quality* uses the stored costs to compute IPC quality
    scores and adds them to the runs.

    The *add_quality* filter can only be executed after *store_costs*
    has been executed. Also, both filters require the "cost" attribute
    to be parsed.

    >>> from downward.reports.absolute import AbsoluteReport
    >>> quality_filters = QualityFilters()
    >>> report = AbsoluteReport(filter=[quality_filters.store_costs,
    ...                                 quality_filters.add_quality])

    """
    def __init__(self):
        self.tasks_to_costs = defaultdict(list)

    def _get_task(self, run):
        return (run['domain'], run['problem'])

    def _compute_quality(self, cost, all_costs):
        if cost is None:
            return 0.0
        assert all_costs
        min_cost = min(all_costs)
        if cost == 0:
            assert min_cost == 0
            return 1.0
        return min_cost / cost

    def store_costs(self, run):
        cost = run.get('cost')
        if cost is not None:
            assert run['coverage']
            self.tasks_to_costs[self._get_task(run)].append(cost)
```

(continues on next page)

(continued from previous page)

```
        return True

    def add_quality(self, run):
        run['quality'] = self._compute_quality(
            run.get('cost'), self.tasks_to_costs[self._get_task(run)])
        return run
```

__call__(*eval_dir*, *outfile*)
Make the report.

This method is called automatically when the report step is executed. It loads the data and calls *write()*.

eval_dir must be a path to an evaluation directory containing a `properties` file.

The report will be written to *outfile*.

get_markup()
Return txt2tags markup for the report.

get_text()
Return text (e.g., HTML, LaTeX, etc.) for the report.

By default this method calls *get_markup()* and converts the markup to the desired output *format*.

write()
Write the report files.

By default this method calls *get_text()* and writes the obtained text to *outfile*.

Overwrite this method if you want to write the report file(s) directly. You should write them to *self.outfile*.

class lab.reports.filter.**FilterReport**(***kwargs*)
Filter properties files.

This report only applies the given filter and writes a new properties file to the output destination.

```
>>> def remove_openstacks(run):
...     return 'openstacks' not in run['domain']
```

```
>>> from lab.experiment import Experiment
>>> report = FilterReport(filter=remove_openstacks)
>>> exp = Experiment()
>>> exp.add_report(report, outfile='path/to/new/properties')
```

CHAPTER 2

Documentation for Downward Lab

2.1 Downward Lab tutorial

2.1.1 Install Lab and Downward Lab

```
# Choose destination for Lab.
LAB=/path/to/lab
# Install dependencies.
sudo apt-get install mercurial python2.7 python-matplotlib python-simplejson
# Clone Lab repository.
hg clone https://bitbucket.org/jendrikseipp/lab ${LAB}
```

Notes about requirements:

- Lab needs Python 2.7 or Python >= 3.5.
- python-matplotlib is only needed for reports that generate graphs.
- python-simplejson is optional, but makes generating reports much faster.

Add to your `.bashrc` to make Lab available on the `PYTHONPATH`:

```
export PYTHONPATH=/path/to/lab
```

Update shell configuration:

```
source ~/.bashrc
```

2.1.2 Download benchmarks

```
DOWNWARD_BENCHMARKS=/path/to/downward-benchmarks
hg clone https://bitbucket.org/aibasel/downward-benchmarks \
    ${DOWNWARD_BENCHMARKS}
```

Some example experiments need the `DOWNWARD_BENCHMARKS` environment variable so we recommend adding it to your `~/.bashrc` file.

2.1.3 Install Fast Downward

(See also http://www.fast-downward.org/ObtainingAndRunningFastDownward and http://www.fast-downward.org/LPBuildInstructions)

```
DOWNWARD_REPO=/path/to/fast-downward-repo
sudo apt-get install mercurial g++ cmake make python
hg clone http://hg.fast-downward.org ${DOWNWARD_REPO}
# Optionally check that Fast Downward works:
cd ${DOWNWARD_REPO}
./build.py
./fast-downward.py ${DOWNWARD_BENCHMARKS}/grid/prob01.pddl \
    --search "astar(lmcut())"
```

2.1.4 Install VAL

```
sudo apt-get install g++ make flex bison
git clone https://github.com/KCL-Planning/VAL.git
cd VAL
make clean  # Remove old object files and binaries.
sed -i 's/-Werror //g' Makefile  # Ignore warnings.
make
sudo cp validate /usr/local/bin  # Add binary to a directory on PATH.
```

MacOS: clone the repo, add `VAL/bin/MacOSExecutables/validate` to your `PATH` and make it executable (`chmod + x`).

2.1.5 Run tutorial experiment

The script below is an example Fast Downward experiment. It is located at `${LAB}/examples/lmcut.py`. After setting `REPO` to `FAST_DOWNWARD` and `BENCHMARKS_DIR` to `BENCHMARKS`, you can see the available steps with

```
./lmcut.py
```

Run all steps with

```
./lmcut.py --all
```

Run individual steps with

```
./lmcut.py build
./lmcut.py 2
./lmcut.py 3 4
```

You can use this file as a basis for your own experiments.

```
#! /usr/bin/env python

"""Solve some tasks with A* and the LM-Cut heuristic."""
```

(continues on next page)

(continued from previous page)

```
import os
import os.path
import platform

from lab.environments import LocalEnvironment, BaselSlurmEnvironment

from downward.experiment import FastDownwardExperiment
from downward.reports.absolute import AbsoluteReport
from downward.reports.scatter import ScatterPlotReport


ATTRIBUTES = ['coverage', 'error', 'expansions', 'total_time']

NODE = platform.node()
if NODE.endswith(".scicore.unibas.ch") or NODE.endswith(".cluster.bc2.ch"):
    # Create bigger suites with suites.py from the downward-benchmarks repo.
    SUITE = ['depot', 'freecell', 'gripper', 'zenotravel']
    ENV = BaselSlurmEnvironment(email="my.name@unibas.ch")
else:
    SUITE = ['depot:p01.pddl', 'gripper:prob01.pddl', 'mystery:prob07.pddl']
    ENV = LocalEnvironment(processes=2)
# Use path to your Fast Downward repository.
REPO = os.environ["DOWNWARD_REPO"]
BENCHMARKS_DIR = os.environ["DOWNWARD_BENCHMARKS"]
REVISION_CACHE = os.path.expanduser('~/lab/revision-cache')

exp = FastDownwardExperiment(environment=ENV, revision_cache=REVISION_CACHE)

# Add built-in parsers to the experiment.
exp.add_parser(exp.EXITCODE_PARSER)
exp.add_parser(exp.TRANSLATOR_PARSER)
exp.add_parser(exp.SINGLE_SEARCH_PARSER)
exp.add_parser(exp.PLANNER_PARSER)

exp.add_suite(BENCHMARKS_DIR, SUITE)
exp.add_algorithm(
    'blind', REPO, 'default', ['--search', 'astar(blind())'])
exp.add_algorithm(
    'lmcut', REPO, 'default', ['--search', 'astar(lmcut())'])

# Add step that writes experiment files to disk.
exp.add_step('build', exp.build)

# Add step that executes all runs.
exp.add_step('start', exp.start_runs)

# Add step that collects properties from run directories and
# writes them to *-eval/properties.
exp.add_fetcher(name='fetch')

# Add report step (AbsoluteReport is the standard report).
exp.add_report(
    AbsoluteReport(attributes=ATTRIBUTES), outfile='report.html')

# Add scatter plot report step.
exp.add_report(
    ScatterPlotReport(
```

(continues on next page)

(continued from previous page)

```
        attributes=["expansions"], filter_algorithm=["blind", "lmcut"]),
    outfile='scatterplot.png')

# Parse the commandline and show or run experiment steps.
exp.run_steps()
```

Have a look at other Fast Downward experiments in the `examples` directory and the downward API.

2.2 `downward.experiment` — Fast Downward experiment

class downward.experiment.**FastDownwardExperiment**(*path=None*, *environment=None*, *revision_cache=None*)

Conduct a Fast Downward experiment.

The most important methods for customizing an experiment are *add_algorithm()*, *add_suite()*, *add_parser()*, *add_step()* and *add_report()*.

Note: To build the experiment, execute its runs and fetch the results, add the following steps (previous Lab versions added these steps automatically):

```
>>> exp = FastDownwardExperiment()
>>> exp.add_step('build', exp.build)
>>> exp.add_step('start', exp.start_runs)
>>> exp.add_fetcher(name='fetch')
```

Note: By default, "output.sas" translator output files are deleted after the driver exits. To keep these files use `del exp.commands['remove-output-sas']` in your experiment script.

See *lab.experiment.Experiment* for an explanation of the *path* and *environment* parameters.

revision_cache is the directory for caching Fast Downward revisions. It defaults to `<scriptdir>/data/revision-cache`. This directory can become very large since each revision uses about 30 MB.

```
>>> from lab.environments import BaselSlurmEnvironment
>>> env = BaselSlurmEnvironment(email="my.name@unibas.ch")
>>> exp = FastDownwardExperiment(environment=env)
```

You can add parsers with *add_parser()*. See *lab.parser — Parse logs* for how to write custom parsers and *Built-in parsers* for the list of built-in parsers. Which parsers you should use depends on the algorithms you're running. For single-search experiments, we recommend adding the following parsers in this order:

```
>>> exp.add_parser(exp.EXITCODE_PARSER)
>>> exp.add_parser(exp.TRANSLATOR_PARSER)
>>> exp.add_parser(exp.SINGLE_SEARCH_PARSER)
>>> exp.add_parser(exp.PLANNER_PARSER)
```

add_algorithm(*name*, *repo*, *rev*, *component_options*, *build_options=None*, *driver_options=None*)

Add a Fast Downward algorithm to the experiment, i.e., a planner configuration in a given repository at a given revision.

name is a string describing the algorithm (e.g. `"issue123-lmcut"`).

repo must be a path to a Fast Downward repository.

rev must be a valid revision in the given repository (e.g., `"default"`, `"tip"`, `"issue123"`).

component_options must be a list of strings. By default these options are passed to the search component. Use `"--translate-options"`, `"--preprocess-options"` or `"--search-options"` within the component options to override the default for the following options, until overridden again.

If given, *build_options* must be a list of strings. They will be passed to the `build.py` script. Options can be build names (e.g., `"release32"`, `"debug64"`), `build.py` options (e.g., `"--debug"`) or options for Make. If *build_options* is omitted, the `"release32"` version is built.

If given, *driver_options* must be a list of strings. They will be passed to the `fast-downward.py` script. See `fast-downward.py --help` for available options. The list is always prepended with `["--validate", "--overall-time-limit", "30m", "--overall-memory-limit', "3584M"]`. Specifying custom limits overrides the default limits.

Example experiment setup:

```
>>> import os.path
>>> exp = FastDownwardExperiment()
>>> repo = os.environ["DOWNWARD_REPO"]
```

Test iPDB in the latest revision on the default branch:

```
>>> exp.add_algorithm(
...     "ipdb", repo, "default",
...     ["--search", "astar(ipdb())"])
```

Test LM-Cut in an issue experiment:

```
>>> exp.add_algorithm(
...     "issue123-v1-lmcut", repo, "issue123-v1",
...     ["--search", "astar(lmcut())"])
```

Run blind search in debug mode:

```
>>> exp.add_algorithm(
...     "blind", repo, "default",
...     ["--search", "astar(blind())"],
...     build_options=["--debug"],
...     driver_options=["--debug"])
```

Run FF in 64-bit mode:

```
>>> exp.add_algorithm(
...     "ff", repo, "default",
...     ["--search", "lazy_greedy([ff()])"],
...     build_options=["release64"],
...     driver_options=["--build", "release64"])
```

Run LAMA-2011 with custom planner time limit:

```
>>> exp.add_algorithm(
...     "lama", repo, "default",
...     [],
...     driver_options=[
```

(continues on next page)

(continued from previous page)

```
...         "--alias", "seq-saq-lama-2011",
...         "--overall-time-limit", "5m"])
```

add_suite(*benchmarks_dir*, *suite*)

Add benchmarks to the experiment.

benchmarks_dir must be a path to a benchmark directory. It must contain domain directories, which in turn hold PDDL files.

suite must be a list of domain or domain:task names.

```
exp.add_suite(benchmarks_dir, ["depot", "gripper"])
exp.add_suite(benchmarks_dir, ["gripper:prob01.pddl"])
```

One source for benchmarks is http://bitbucket.org/aibasel/downward-benchmarks. After cloning the repo, you can generate suites with the `suites.py` script. We recommend using the suite `optimal_strips` for optimal planning and `satisficing` for satisficing planning:

```
# Create standard optimal planning suite.
$ path/to/downward-benchmarks/suites.py optimal_strips
['airport', ..., 'zenotravel']
```

You can copy the generated list into your experiment script:

```
>>> benchmarks_dir = REPO = os.environ["DOWNWARD_BENCHMARKS"]
>>> exp = FastDownwardExperiment()
>>> exp.add_suite(benchmarks_dir, ['airport', 'zenotravel'])
```

2.2.1 Built-in parsers

The following constants are paths to built-in parsers that can be passed to *exp.add_parser()*. The "Required attributes" and "Parsed attributes" lists describe the dependencies between the parsers.

FastDownwardExperiment.**EXITCODE_PARSER**

Parsed attributes: "error", "planner_exit_code", "unsolvable".

FastDownwardExperiment.**TRANSLATOR_PARSER**

Parsed attributes: "translator_peak_memory", "translator_time_done", etc.

FastDownwardExperiment.**SINGLE_SEARCH_PARSER**

Parsed attributes: "coverage", "memory", "total_time", etc.

FastDownwardExperiment.**ANYTIME_SEARCH_PARSER**

Parsed attributes: "cost", "cost:all", "coverage".

FastDownwardExperiment.**PLANNER_PARSER**

Used attributes: "memory", "total_time", "translator_peak_memory", "translator_time_done".

Parsed attributes: "node", "planner_memory", "planner_time", "planner_wall_clock_time".

2.3 downward.reports — Fast Downward reports

2.3.1 Tables

class downward.reports.**PlanningReport**(***kwargs*)

This is the base class for planner reports.

The *INFO_ATTRIBUTES* and *ERROR_ATTRIBUTES* class members hold attributes for Fast Downward experiments by default. You may want to adjust the two lists in derived classes.

See *Report* for inherited parameters.

You can filter and modify runs for a report with *filters*. For example, you can include only a subset of algorithms or compute new attributes. If you provide a list for *filter_algorithm*, it will be used to determine the order of algorithms in the report.

```
>>> # Use a filter function to select algorithms.
>>> def only_blind_and_lmcut(run):
...     return run['algorithm'] in ['blind', 'lmcut']
>>> report = PlanningReport(filter=only_blind_and_lmcut)
```

```
>>> # Use "filter_algorithm" to select and *order* algorithms.
>>> r = PlanningReport(filter_algorithm=['lmcut', 'blind'])
```

Filters can be very helpful so we recommend reading up on them to use their full potential.

ERROR_ATTRIBUTES = ['domain', 'problem', 'algorithm', 'unexplained_errors', 'error', '

Attributes shown in the unexplained-errors table. Can be overriden in subclasses.

INFO_ATTRIBUTES = ['local_revision', 'global_revision', 'revision_summary', 'build_opt

Attributes shown in the algorithm info table. Can be overriden in subclasses.

PREDEFINED_ATTRIBUTES = ['cost', 'coverage', 'dead_ends', 'evaluations', 'expansions',

List of predefined `Attribute` instances. If PlanningReport receives `attributes=['coverage']`, it converts the plain string `'coverage'` to the attribute instance `Attribute('coverage', absolute=True, min_wins=False, scale='linear')`. The list can be overriden in subclasses.

class downward.reports.absolute.**AbsoluteReport**(***kwargs*)

Report absolute values for the selected attributes.

This report should be part of all your Fast Downward experiments as it includes a table of unexplained errors, e.g. invalid solutions, segmentation faults, etc.

```
>>> from downward.experiment import FastDownwardExperiment
>>> exp = FastDownwardExperiment()
>>> exp.add_report(
...     AbsoluteReport(attributes=["expansions"]),
...     outfile='report.html')
```

Example output:

expansions	hFF	hCEA
gripper	118	72
zenotravel	21	17

class downward.reports.taskwise.**TaskwiseReport**(***kwargs*)

For each task report all selected attributes in a single row.

If the experiment contains more than one algorithm, use `filter_algorithm='my_algorithm'` to select exactly one algorithm for the report.

```
>>> from downward.experiment import FastDownwardExperiment
>>> exp = FastDownwardExperiment()
>>> exp.add_report(TaskwiseReport(
...     attributes=["expansions", "search_time"],
...     filter_algorithm=["lmcut"]))
```

Example output:

	expansions	search_time
grid:prob01.pddl	118234	20.02
gripper:prob01.pddl	21938	17.58

class downward.reports.compare.**ComparativeReport**(*algorithm_pairs*, ***kwargs*)

Compare pairs of algorithms.

See *AbsoluteReport* for inherited parameters.

algorithm_pairs is the list of algorithm pairs you want to compare.

All columns in the report will be arranged such that the compared algorithms appear next to each other. After the two columns containing absolute values for the compared algorithms, a third column ("Diff") is added showing the difference between the two values.

Algorithms may appear in multiple comparisons. Algorithms not mentioned in *algorithm_pairs* are not included in the report.

If you want to compare algorithms A and B, instead of a pair `('A', 'B')` you may pass a triple `('A', 'B', 'A vs. B')`. The third entry of the triple will be used as the name of the corresponding "Diff" column.

For example, if the properties file contains algorithms A, B, C and D and *algorithm_pairs* is `[('A', 'B', 'Diff BA'), ('A', 'C')]` the resulting columns will be A, B, Diff BA (contains B - A), A, C , Diff (contains C - A).

Example:

```
>>> from downward.experiment import FastDownwardExperiment
>>> exp = FastDownwardExperiment()
>>> algorithm_pairs = [
...     ('default-lmcut', 'issue123-lmcut', 'Diff lmcut')]
>>> exp.add_report(ComparativeReport(
...     algorithm_pairs, attributes=['coverage']))
```

Example output:

coverage	default-lmcut	issue123-lmcut	Diff lmcut
depot	15	17	2
gripper	7	6	-1

2.3.2 Plots

class downward.reports.plot.**PlotReport**(*title=None*, *xscale=None*, *yscale=None*, *xlabel=''*, *ylabel=''*, *matplotlib_options=None*, ***kwargs*)

Abstract base class for Plot classes.

The inherited *format* parameter can be set to 'png' (default), 'eps', 'pdf', 'pgf' (needs matplotlib 1.2) or 'tex'. For the latter a pgfplots plot is created.

If *title* is given it will be used for the name of the plot. Otherwise, the only given attribute will be the title. If none is given, there will be no title.

xscale and *yscale* can have the values 'linear', 'log' or 'symlog'. If omitted sensible defaults will be used.

xlabel and *ylabel* are the axis labels.

matplotlib_options may be a dictionary of matplotlib rc parameters (see http://matplotlib.org/users/customizing.html):

```
>>> from downward.reports.scatter import ScatterPlotReport
>>> matplotlib_options = {
...     'font.family': 'serif',
...     'font.weight': 'normal',
...     # Used if more specific sizes not set.
...     'font.size': 20,
...     'axes.labelsize': 20,
...     'axes.titlesize': 30,
...     'legend.fontsize': 22,
...     'xtick.labelsize': 10,
...     'ytick.labelsize': 10,
...     'lines.markersize': 10,
...     'lines.markeredgewidth': 0.25,
...     'lines.linewidth': 1,
...     # Width and height in inches.
...     'figure.figsize': [8, 8],
...     'savefig.dpi': 100,
... }
>>> report = ScatterPlotReport(
...     attributes=['initial_h_value'],
...     matplotlib_options=matplotlib_options)
```

You can see the full list of matplotlib options and their defaults by executing

```
import matplotlib
print matplotlib.rcParamsDefault
```

class downward.reports.scatter.**ScatterPlotReport**(*show_missing=True*, *get_category=None*, ***kwargs*)

Generate a scatter plot for a specific attribute.

See *PlotReport* for inherited arguments.

The keyword argument *attributes* must contain exactly one attribute.

Use the *filter_algorithm* keyword argument to select exactly two algorithms.

If only one of the two algorithms has a value for a run, only add a coordinate if *show_missing* is True.

get_category can be a function that takes **two** runs (dictionaries of properties) and returns a category name. This name is used to group the points in the plot. If there is more than one group, a legend is automatically added. Runs for which this function returns None are shown in a default category and are not contained in the legend. For example, to group by domain:

```
>>> def domain_as_category(run1, run2):
...     # run2['domain'] has the same value, because we always
...     # compare two runs of the same problem.
...     return run1['domain']
```

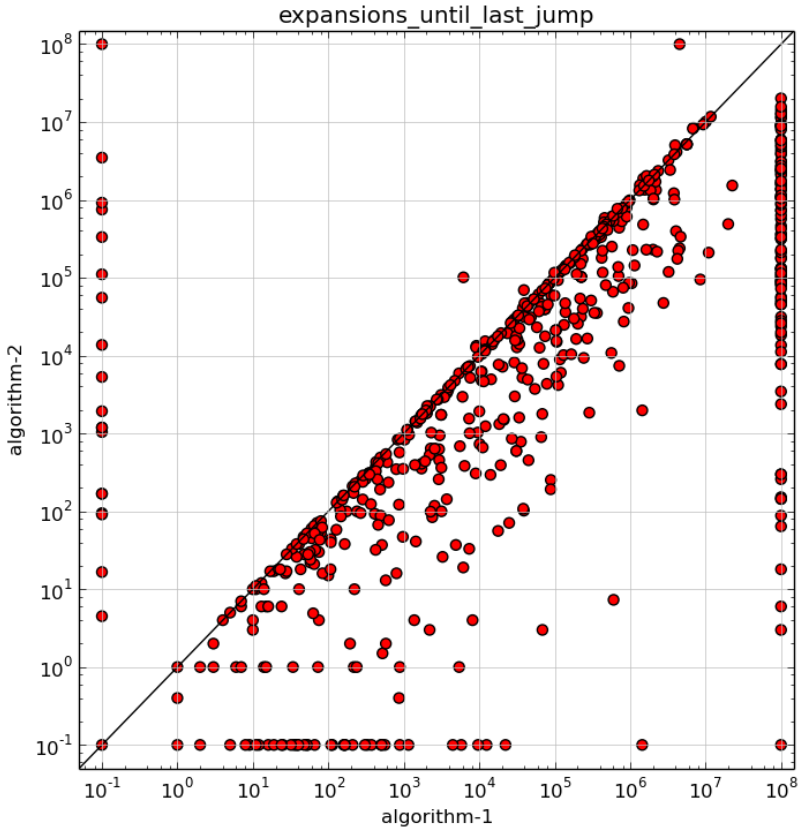
Example grouping by difficulty:

```
>>> def improvement(run1, run2):
...     time1 = run1.get('search_time', 1800)
...     time2 = run2.get('search_time', 1800)
...     if time1 > time2:
...         return 'better'
...     if time1 == time2:
...         return 'equal'
...     return 'worse'
```

```
>>> from downward.experiment import FastDownwardExperiment
>>> exp = FastDownwardExperiment()
>>> exp.add_report(ScatterPlotReport(
...     attributes=['search_time'],
...     get_category=improvement))
```

Example comparing the number of expanded states for two algorithms:

```
>>> exp.add_report(ScatterPlotReport(
...         attributes=["expansions_until_last_jump"],
...         filter_algorithm=["algorithm-1", "algorithm-2"],
...         get_category=domain_as_category,
...         format="png",  # Use "tex" for pgfplots output.
...         ),
...     name="scatterplot-expansions")
```


expansions_until_last_jump
algorithm-2
algorithm-1
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
10^{8}

CHAPTER 3

General documentation

3.1 News

3.1.1 v4.1 (2019-06-03)

- Add support for Python 3. Lab now supports Python 2.7 and Python >= 3.5.

3.1.2 v4.0 (2019-02-19)

Lab

- Parser: don't try to parse missing files. Print message to stdout instead.
- Add soft memory limit of "memory_per_cpu * 0.98" for Slurm runs to safeguard against cgroup failures.
- Abort if report contains duplicate attribute names.
- Make reports even if fetcher detects unexplained errors.
- Use `flags=''` for *lab.parser.Parser.add_pattern()* by default again.
- Include node names in standard reports and warn if report mixes runs from different partitions.
- Add new example experiment using a simple vertex cover solver.
- `BaselSlurmEnvironment`: don't load Python 2.7.11 since it might conflict with an already loaded module.
- Raise default `nice` value to 5000.

Downward Lab

- Support new Fast Downward exitcodes (Silvan).
- Parse "planner_wall_clock_time" attribute in planner parser.

- Include "planner_wall_clock_time" and "raw_memory" attributes in unexplained errors table.
- Make PlanningReport more generic by letting derived classes override the new `PREDEFINED_ATTRIBUTES`, `INFO_ATTRIBUTES` and `ERROR_ATTRIBUTES` class members (Augusto).
- Don't compute the "quality" attribute automatically. The docs and `showcase-options.py` show how to add the two filters that together add the IPC quality score to each run.

3.1.3 v3.0 (2018-07-10)

Lab

- Add *exp.add_parser()* method. See also *lab.parser — Parse logs* (Silvan).
- Add *exp.add_parse_again_step()* method for running parsers again (Silvan).
- Require that the `build`, `start_runs` and `fetch` steps are added explicitly (see *Experiment*).
- Remove *required* argument from `add_resource()`. All resources are now required.
- Use stricter naming rules for commands and resources. See respective `add_*` methods for details.
- Use `required=False` and `flags='M'` by default for *lab.parser.Parser.add_pattern()*.
- Only support custom command line arguments for locally executed steps.
- Log errors to stderr.
- Log exit codes and wall-clock times of commands to driver.log.
- Add unexplained error if driver.log is empty.
- Let fetcher fetch `properties` and `static-properties` files.
- Remove deprecated possibility of passing Step objects to `add_step()`.
- Remove deprecated `exp.__call__()` method.

Downward Lab

- Add "planner_timer" and "planner_memory" attributes.
- Reorganize parsers and don't add any parser implicitly. See *Built-in parsers*.
- Add anytime-search parser that parses only "cost", "cost:all" and "coverage".
- Revise and simplify single-search parser.
- Parse new Fast Downward exit codes (http://issues.fast-downward.org/issue739).
- Don't exclude (obsolete) "benchmarks" directory when caching revisions.
- Only copy "raw_memory" value to "memory" when "total_time" is present.
- Rename "fast-downward" command to "planner".
- Make "error" attribute optional for reports.

3.1.4 v2.3 (2018-04-12)

Lab

- BaselSlurmEnvironment: Use `infai_1` and `normal` as default Slurm partition and QOS.
- Remove `OracleGridEngineEnvironment`.

Downward Lab

- Use `--overall-time-limit=30m` and `--overall-memory-limit=3584M` for all Fast Downward runs by default.
- Don't add `-j` option to build options (`build.py` now uses all CPUs automatically).

3.1.5 v2.2 (2018-03-16)

Lab

- Print run and task IDs during local experiments.
- Make warnings and error messages more informative.
- Abort after fetch step if fetcher finds unexplained errors.
- Improve examples and docs.

Downward Lab

- Don't parse preprocessor logs anymore.
- Make regular expressions stricter in parsers.
- Don't complain if SAS file is missing.

3.1.6 v2.1 (2017-11-27)

Lab

- Add BaselSlurmEnvironment (Florian).
- Support running experiments in virtualenv (Shuwa).
- Redirect output to `driver.log` and `driver.err` as soon as possible.
- Store all observed unexplained errors instead of a single one (Silvan).
- Report unexplained error if `run.err` or `driver.err` contain output.
- Report unexplained error if "error" attribute is missing.
- Add configurable soft and hard limits for output to `run.log` and `run.err`.
- Record grid node for each run and add it to warnings table.
- Omit toprule and bottomrule in LaTeX tables.
- Add `lab.reports.Table.set_row_order()` method.

- Only escape text in table cells if it doesn't contain LaTeX or HTML markup.
- Allow run filters to change a run's ID (needed for renaming algorithms).
- Add `merge` kwarg to `add_fetcher()` (Silvan).
- Exit with returncode 1 if fetcher finds unexplained errors.
- Let fetcher show warning if `slurm.err` is not empty.
- Include content of `slurm.err` in reports if it contains text.
- Add continuous integration testing.
- Add `--skip-experiments` option for `tests/run-tests` script.
- Clean up code.
- Polish documentation.

Downward Lab

- For each error outcome show number of runs with that outcome in summary table and dedicated tables.
- Add standalone exit code parser. Allow removing translate and search parsers (Silvan).
- Allow passing `Problem` instances to `FastDownwardExperiment.add_suite()` (Florian).
- Don't filter duplicate coordinates in scatter plots.
- Don't round scatter plot coordinates.
- Remove output.sas instead of compressing it.
- Fix scatter plots for multiple categories **and** the default `None` category (Silvan).

3.1.7 v2.0 (2017-01-09)

Lab

- Show warning and ask for action when evaluation dir already exists.
- Add `scale` parameter to Attribute. It is used by the plot reports.
- Add `digits` parameter to Attribute for specifying the number of digits after the decimal point.
- Pass name, function, args and kwargs to `exp.add_step()`. Deprecate passing Step objects.
- After calling `add_resource("mynick", ...)`, use resource in commands with "{mynick}".
- Call: make `name` parameter mandatory, rename `mem_limit` kwarg to `memory_limit`.
- Store grid job files in `<exp-dir>-grid-steps`.
- Use common `run-dispatcher` script for local and remote experiments.
- LocalEnvironment: support randomizing task order (enabled by default).
- Make `path` parameter optional for all experiments.
- Warn if steps are listed explicitly and `--all` is used.
- Change main experiment step name from "start" to "run".
- Deprecate `exp()`. Use `exp.run_steps()` instead.

- Don't filter `None` values in `lab.reports` helper functions.
- Make logging clearer.
- Add example FF experiment.
- Remove deprecated code (e.g. predefined Step objects, `tools.sendmail()`).
- Remove `Run.require_resource()`. All resources have always been available for all runs.
- Fetcher: remove `write_combined_props` parameter.
- Remove `Sequence` class.
- Parser: remove `key_value_patterns` parameter. A better solution is in the works.
- Remove `tools.overwrite_dir()` and `tools.get_command_output()`.
- Remove `lab.reports.minimum()`, `lab.reports.maximum()`, `lab.reports.stddev()`.
- Move `lab.reports.prod()` to `lab.tools.product()`.
- Rename `lab.reports.gm()` to `lab.reports.geometric_mean()` and `lab.reports.avg()` to `lab.reports.arithmetic_mean()`.
- Many speed improvements and better error messages.
- Rewrite docs.

Downward Lab

- Always validate plans. Previous Lab versions don't add `--validate` since older Fast Downward versions don't support it.
- HTML reports: hide tables by default, add buttons for toggling visibility.
- Unify "score_*", "quality" and "coverage" attributes: assign values in range [0, 1] and compute only sum and no average.
- Don't print tables on commandline.
- Remove DownwardExperiment and other deprecated code.
- Move `FastDownwardExperiment` into `downward/experiment.py`.
- Rename `config` attribute to `algorithm`. Remove `config_nick` attribute.
- Change call name from "search" to "fast-downward".
- Remove "memory_capped", and "id_string" attributes.
- Report raw memory in "unexplained errors" table.
- Parser: remove `group` argument from `add_pattern()`, and always use group 1.
- Remove `cache_dir` parameter. Add `revision_cache` parameter to `FastDownwardExperiment`.
- Fetcher: remove `copy_all` option.
- Remove predefined benchmark suites.
- Remove IpcReport, ProblemPlotReport, RelativeReport, SuiteReport and TimeoutReport.
- Rename CompareConfigsReport to ComparativeReport.
- Remove possibility to add `_relative` to an attribute to obtain relative results.
- Apply filters sequentially instead of interleaved.

- PlanningReport: remove `derived_properties` parameter. Use two filters instead: one for caching results, the other for adding new properties (see `QualityFilters` in `downward/reports/__init__.py`).
- PlotReport: use fixed legend location, remove `category_styles` option.
- AbsoluteReport: remove `colored` parameter and always color HTML reports.
- Don't use domain links in Latex reports.
- AbsoluteReport: Remove `resolution` parameter and always use `combined` resolution.
- Rewrite docs.

3.1.8 v1.12 (2017-01-09)

Downward Lab

- Only compress "output" file if it exists.
- Preprocess parser: make legacy preprocessor output optional.

3.1.9 v1.11 (2016-12-15)

Lab

- Add bitbucket-pipelines.yml for continuous integration testing.

Downward Lab

- Add IPC 2014 benchmark suites (Silvan).
- Set `min_wins=False` for `dead_ends` attribute.
- Fit coordinates better into plots.
- Add finite_sum() function and use it for `initial_h_value` (Silvan).
- Update example scripts for repos without benchmarks.
- Update docs.

3.1.10 v1.10 (2015-12-11)

Lab

- Add `permissions` parameter to *`lab.experiment.Experiment.add_new_file()`*.
- Add default parser which checks that log files are not bigger than 100 MB. Maybe we'll make this configurable in the future.
- Ensure that resource names are not shared between runs and experiment.
- Show error message if resource names are not unique.
- Table: don't format list items. This allows us to keep the quotes for configuration lists.

Downward Lab

- Cleanup `downward.suites`: update suite names, add STRIPS and ADL versions of all IPCs. We recommend selecting a subset of domains manually to only run your code on "interesting" benchmarks. As a starting point you can use the suites `suite_optimal_strips` or `suite_satisficing`.

3.1.11 v1.9.1 (2015-11-12)

Downward Lab

- Always prepend build options with `-j<num_cpus>`.
- Fix: Use correct revisions in `FastDownwardExperiment`.
- Don't abort parser if resource limits can't be found (support old planner versions).

3.1.12 v1.9 (2015-11-07)

Lab

- Add *lab.experiment.Experiment.add_command()* method.
- Add *lab.__version__* string.
- Explicitly remove support for Python 2.6.

Downward Lab

- Add *downward.experiment.FastDownwardExperiment* class for whole-planner experiments.
- Deprecate `downward.experiments.DownwardExperiment` class.
- Repeat headers between domains in *downward.reports.taskwise.TaskwiseReport*.

3.1.13 v1.8 (2015-10-02)

Lab

- Deprecate predefined experiment steps (`remove_exp_dir`, `zip_exp_dir`, `unzip_exp_dir`).
- Docs: add FAQs, update docs.
- Add more regression and style tests.

Downward Lab

- Parse both evaluated states (evaluated) and evaluations (evaluations).
- Add example experiment showing how to make reports for data obtained without Lab.
- Add suite_sat_strips().
- Parse negative initial h values.
- Support CMake builds.

3.1.14 v1.7 (2015-08-19)

Lab

- Automatically determine whether to queue steps sequentially on the grid.
- Reports: right-align headers (except the left-most one).
- Reports: let `lab.reports.gm()` return 0 if any of the numbers is 0.
- Add test that checks for dead code with vulture.
- Remove Step.remove_exp_dir step.
- Remove default time and memory limits for commands. You can now pass `mem_limit=None` and `time_limit=None` to disable limits for a command.
- Pass `extra_options` kwarg to `lab.environments.OracleGridEngineEnvironment` to set additional options like parallel environments.
- Sort `properties` files by keys.

Downward Lab

- Add support for new python driver script `fast-downward.py`.
- Use booktabs package for latex tables.
- Remove vertical lines from Latex tables (recommended by booktabs docs).
- Capitalize attribute names and remove underscores for Latex reports.
- Allow fractional plan costs.
- Set search_time and total_time to 0.01 instead of 0.1 if they are 0.0 in the log.
- Parse initial h-value for aborted searches (Florian).
- Use EXIT_UNSOLVABLE instead of logs to determine unsolvability. Currently, this exit code is only returned by EHC.
- Exit with warning if search parser is not executable.
- Deprecate `downward/configs.py` module.
- Deprecate `examples/standard_exp.py` module.
- Remove `preprocess-all.py` script.
- By default, use all CPUs for compiling Fast Downward.

3.1.15 v1.6

Lab

- Restore earlier default behavior for grid jobs by passing all environment variables (e.g. `PYTHONPATH`) to the job environments.

Downward Lab

- Use write-once revision cache: instead of *cloning* the full FD repo into the revision cache only *copy* the `src` directory. This greatly reduces the time and space needed to cache revisions. As a consequence you cannot specify the destination for the clone anymore (the `dest` keyword argument is removed from the `Translator`, `Preprocessor` and `Planner` classes) and only local FD repositories are supported (see `downward.checkouts.HgCheckout`). After the files have been copied into the cache and FD has been compiled, a special file (`build_successful`) is written in the cache directory. When the cached revision is requested later an error is shown if this file is missing.
- Only use exit codes to reason about error reasons. Merge from FD master if your FD version does not produce meaningful exit codes.
- Preprocess parser: only parse logs and never output files.
- Never copy `all.groups` and `test.groups` files. Old Fast Downward branches need to merge from master.
- Always compress `output.sas` (also for `compact=False`). Use `xz` for compressing.

3.1.16 v1.5

Lab

- Add `Experiment.add_fetcher()` method.
- If all columns have the same value in an uncolored table row, make all values bold, not grey.
- In `Experiment.add_resource()` and `Run.add_resource()` set `dest=None` if you don't want to copy or link the resource, but only need an alias to reference it in a command.
- Write and keep all logfiles only if they actually have content.
- Don't log time and memory consumption of process groups. It is still an unexplained error if too much wall-clock time is used.
- Randomize task order for grid experiments by default. Use `randomize_task_order=False` to disable this.
- Save wall-clock times in properties file.
- Do not replace underscores by dashes in table headers. Instead allow browsers to break lines after underscores.
- Left-justify string and list values in tables.

Downward Lab

- Add optional *nick* parameter to Translator, Preprocessor and Planner classes. It defaults to the revision name *rev*.
- Save `hg id` output for each checkout and include it in reports.
- Add *timeout* parameter to `DownwardExperiment.add_config()`.
- Count malformed-logs as unexplained errors.
- Pass `legend_location=None` if you don't need a legend in your plot.
- Pass custom benchmark directories in `DownwardExperiment.add_suite()` by using the *benchmarks_dir* keyword argument.
- Do not copy logs from preprocess runs into search runs.

- Reference preprocessed files in run scripts instead of creating links if `compact=True` is given in the experiment constructor (default).
- Remove `unexplained_error` attribute. Errors are unexplained if `run['error']` starts with 'unexplained'.
- Remove `*_error` attributes. It is preferrable to inspect `*_returncode` attributes instead (e.g. `search_returncode`).
- Make report generation faster (10-fold speedup for big reports).
- Add `DownwardExperiment.add_search_parser()` method.
- Run `make clean` in revision-cache after compiling preprocessor and search code.
- Strip executables after compilation in revision-cache.
- Do not copy Lab into experiment directories and grid-steps. Use the global Lab version instead.

3.1.17 v1.4

Lab

- Add *exp.add_report()* method to simplify adding reports.
- Use simplejson when available to make loading properties more than twice as fast.
- Raise default check-interval in Calls to 5s. This should reduce Lab's overhead.
- Send mail when grid experiment finishes. Usage: `MaiaEnvironment(email='mymail@example.com')`.
- Remove `steps.Step.publish_reports()` method.
- Allow creating nested new files in experiment directory (e.g. `exp.add_new_file('path/to/file.txt')`).
- Remove duplicate attributes from reports.
- Make commandline parser available globally as *lab.experiment.ARGPARSER* so users can add custom arguments.
- Add `cache_dir` parameter in *Experiment* for specifying where Lab stores temporary data.

Downward Lab

- Move `downward.experiment.DownwardExperiment` to `downward.experiments.DownwardExperiment`, but keep both import locations indefinitely.
- Flag invalid plans in absolute reports.
- PlanningReport: When you append '_relative' to an attribute, you will get a table containing the attribute's values of each configuration relative to the leftmost column.
- Use bzip2 for compressing output.sas files instead of tar+gzip to save space and make opening the files easier.
- Use bzip2 instead of gzip for compressing experiment directories to save space.
- Color absolute reports by default.
- Use log-scale instead of symlog-scale for plots. This produces equidistant grid lines.
- By default place legend right of scatter plots.

- Remove `--dereference` option from tar command.
- Copy (instead of linking) PDDL files into preprocessed-tasks dir.
- Add table with Fast Downward commandline strings and revisions to AbsoluteReport.

3.1.18 v1.3

Lab

- For Latex tables only keep the first two and last two hlines.

Downward Lab

- Plots: Make category_styles a dictionary mapping from names to dictionaries of matplotlib plotting parameters to allow for more and simpler customization. This means e.g. that you can now change the line style in plots.
- Produce a combined domain- and problem-wise AbsoluteReport if `resolution=combined`.
- Include info in AbsoluteReport if a table has no entries.
- Plots: Add `params` argument for specifying matplotlib parameters like font-family, label sizes, line width, etc.
- AbsoluteReport: If a non-numerical attribute is included in a domain-wise report, include some info in the table instead of aborting.
- Add *`Attribute`* class for wrapping custom attributes that need non-default report options and aggregation functions.
- Parse `expansions_until_last_jump` attribute.
- Tex reports: Add number of tasks per domain with new `\numtasks{x}` command that can be cutomized in the exported texts.
- Add pgfplots backend for plots.

3.1.19 v1.2

Lab

- Fetcher: Only copy the link not the content for symbolic links.
- Make properties files more compact by using an indent of 2 instead of 4.
- Nicer format for commandline help for experiments.
- Reports: Only print available attributes if none have been set.
- Fetcher: Pass custom parsers to fetcher to parse values from a finished experiment.
- For geometric mean calculation substitute 0.1 for values <= 0.
- Only show warning if not all attributes for the report are found in the evaluation dir, don't abort if at least one attribute is found.
- If an attribute is None for all runs, do not conclude it is not numeric.
- Abort if experiment path contains a colon.
- Abort with warning if all runs have been filtered for a report.

- Reports: Allow specifying a *single* attribute as a string instead of a list of one string (e.g. attributes='coverage').

Downward Lab

- If compact=True for a DownwardExperiment, link to the benchmarks instead of copying them.
- Do not call ./build-all script, but build components only if needed.
- Fetch and compile sources only when needed: Only prepare translator and preprocessor for preprocessing experiments and only prepare planners for search experiments. Do it in a grid job if possible.
- Save space by deleting the benchmarks directories and omitting the search directory and validator for preprocess experiments.
- Only support using 'src' directory, not the old 'downward' dir.
- Use `downward` script regardless of other binaries found in the search directory.
- Do not try to set parent-revision property. It cannot be determined without fetching the code first.
- Make ProblemPlotReport class more general by allowing the get_points() method to return an arbitrary number of points and categories.
- Specify xscale and yscale (linear, log, symlog) in PlotReports.
- Fix removing downward.tmp.* files (use bash for globbing). This greatly reduces the needed space for an experiment.
- Label axes in ProblemPlots with `xlabel` and `ylabel`.
- If a grid environment is selected, use all CPUs for compiling Fast Downward.
- Do not use the same plot style again if it has already been assigned by the user.
- Only write plot if valid points have been added.
- DownwardExperiment: Add member `include_preprocess_results_in_search_runs`.
- Colored reports: If all configs have the same value in a row and some are None, highlight the values in green instead of making them grey.
- Never set 'error' to 'none' if 'search_error' is true.
- PlotReport: Add `legend_location` parameter.
- Plots: Sort coordinates by x-value for correct connections between points.
- Plots: Filter duplicate coordinates for nicer drawing.
- Use less padding for linear scatterplots.
- Scatterplots: Add `show_missing` parameter.
- Absolute reports: For absolute attributes (e.g. coverage) print a list of numbers of problems behind the domain name if not all configs have a value for the same number of problems.
- Make 'unsolvable' an absolute attribute, i.e. one where we consider problem runs for which not all configs have a value.
- If a non-numeric attribute is present in a domain-wise report, state its type in the error message.
- Let plots use the `format` parameter given in constructor.
- Allow generation of pgf plot files (only available in matplotlib 1.2).
- Allow generation of pdf and eps plots.

- DownwardReport: Allow passing a single function for `derived_properties`.
- Plots: Remove code that sets parameters explicitly, sort items in legend.
- Add parameters to PlotReport that set the axes' limits.
- Add more items to Downward Lab FAQ.

3.1.20 v1.1

Lab

- Add filter shortcuts: `filter_config_nick=['lama', 'hcea'], filter_domain=['depot']` (see *Report*) (Florian)
- Ability to use more than one filter function (Florian)
- Pass an optional filter to `Fetcher` to fetch only a subset of results (Florian)
- Better handling of timeouts and memory-outs (Florian)
- Try to guess error reason when run was killed because of resource limits (Florian)
- Do not abort after failed commands by default
- Grid: When –all is passed only run all steps if none are supplied
- Environments: Support Uni Basel maia cluster (Malte)
- Add "pi" example
- Add example showing how to parse custom attributes
- Do not add resources and files again if they are already added to the experiment
- Abort if no runs have been added to the experiment
- Round all float values for the tables
- Add function `lab.tools.sendmail()` for sending e-mails
- Many bugfixes
- Added more tests
- Improved documentation

Downward Lab

- Make the files output.sas, domain.pddl and problem.pddl optional for search experiments
- Use more compact table of contents for AbsoluteReports
- Use named anchors in AbsoluteReport (`report.html#expansions`, `report.html#expansions-gripper`)
- Add colored absolute tables (see *AbsoluteReport*)
- Do not add summary functions in problem-wise reports
- New report class `ProblemPlotReport`
- Save more properties about experiments in the experiments's properties file for easy lookup (suite, configs, portfolios, etc.)

- Use separate table for each domain in problem-wise reports
- Sort table columns based on given config filters if given (Florian)
- Do not add VAL source files to experiment
- Parse number of reopened states
- Remove temporary Fast Downward files even if planner was killed
- Divide scatter-plot points into categories and lable them (see *ScatterPlotReport*) (Florian)
- Only add a highlighting and summary functions for numeric attributes in AbsoluteReports
- Compile validator if it isn't compiled already
- Downward suites: Allow writing SUITE_NAME_FIRST to run the first instance of all domains in SUITE_NAME
- LocalEnvironment: If `processes` is given, use as many jobs to compile the planner in parallel
- Check python version before creating preprocess experiment
- Add avg, min, max and stddev rows to relative reports
- Add RelativeReport
- Add `DownwardExperiment.set_path_to_python()`
- Many bugfixes
- Improved documentation

3.2 Frequently asked questions

3.2.1 How can I parse and report my own attributes?

See the parsing documentation.

3.2.2 How can I combine the results from multiple experiments?

```
exp = Experiment('/new/path/to/combined-results')
exp.add_fetcher('path/to/first/eval/dir')
exp.add_fetcher('path/to/second/eval/dir')
exp.add_fetcher('path/to/experiment/dir')
exp.add_report(AbsoluteReport())
```

3.2.3 I forgot to parse something. How can I run only the parsers again?

See the parsing documentation for how to write parsers. Once you have fixed your existing parsers or added new parsers, add `exp.add_parse_again_step()` to your experiment script `my-exp.py` and then call

```
./my-exp.py parse-again
```

3.2.4 How can I make reports and plots for results obtained without Lab?

See `examples/report-external-results.py` for an example.

3.2.5 Which experiment class should I use for which Fast Downward revision?

- Before CMake: use DownwardExperiment in Lab 1.x
- With CMake and optional validation: use FastDownwardExperiment in Lab 1.x
- With CMake and mandatory validation: use FastDownwardExperiment in Lab 2.x
- New translator exit codes (issue739): use FastDownwardExperiment in Lab 3.x

3.3 Using Downward Lab for other planners

The script below shows how to run the FF planner on a number of classical planning benchmarks. It is located at `examples/ff/ff.py`. You can see the available steps with

```
./ff.py
```

Select steps by name or index:

```
./ff.py build
./ff.py 2
./ff.py 3 4
```

You can use this file as a basis for your own experiments. For Fast Downward experiments, we recommend taking a look at the downward.tutorial.

```
#! /usr/bin/env python

"""
Example experiment for the FF planner
(http://fai.cs.uni-saarland.de/hoffmann/ff.html).
"""

import os
import platform

from lab.environments import LocalEnvironment, BaselSlurmEnvironment
from lab.experiment import Experiment

from downward import suites
from downward.reports.absolute import AbsoluteReport


# Create custom report class with suitable info and error attributes.
class BaseReport(AbsoluteReport):
    INFO_ATTRIBUTES = ['time_limit', 'memory_limit']
    ERROR_ATTRIBUTES = [
        'domain', 'problem', 'algorithm', 'unexplained_errors', 'error', 'node']


NODE = platform.node()
REMOTE = NODE.endswith(".scicore.unibas.ch") or NODE.endswith(".cluster.bc2.ch")
BENCHMARKS_DIR = os.environ["DOWNWARD_BENCHMARKS"]
if REMOTE:
    ENV = BaselSlurmEnvironment(email="my.name@unibas.ch")
else:
```

(continues on next page)

(continued from previous page)

```
    ENV = LocalEnvironment(processes=4)
SUITE = [
    'grid', 'gripper:prob01.pddl',
    'miconic:s1-0.pddl', 'mystery:prob07.pddl']
ATTRIBUTES = [
    'coverage', 'error', 'evaluations', 'plan', 'times',
    'trivially_unsolvable']
TIME_LIMIT = 1800
MEMORY_LIMIT = 2048


# Create a new experiment.
exp = Experiment(environment=ENV)
# Add custom parser for FF.
exp.add_parser('ff-parser.py')

for task in suites.build_suite(BENCHMARKS_DIR, SUITE):
    run = exp.add_run()
    # Create symbolic links and aliases. This is optional. We
    # could also use absolute paths in add_command().
    run.add_resource('domain', task.domain_file, symlink=True)
    run.add_resource('problem', task.problem_file, symlink=True)
    # 'ff' binary has to be on the PATH.
    # We could also use exp.add_resource().
    run.add_command(
        'run-planner',
        ['ff', '-o', '{domain}', '-f', '{problem}'],
        time_limit=TIME_LIMIT,
        memory_limit=MEMORY_LIMIT)
    # AbsoluteReport needs the following properties:
    # 'domain', 'problem', 'algorithm', 'coverage'.
    run.set_property('domain', task.domain)
    run.set_property('problem', task.problem)
    run.set_property('algorithm', 'ff')
    # BaseReport needs the following properties:
    # 'time_limit', 'memory_limit'.
    run.set_property('time_limit', TIME_LIMIT)
    run.set_property('memory_limit', MEMORY_LIMIT)
    # Every run has to have a unique id in the form of a list.
    # The algorithm name is only really needed when there are
    # multiple algorithms.
    run.set_property('id', ['ff', task.domain, task.problem])

# Add step that writes experiment files to disk.
exp.add_step('build', exp.build)

# Add step that executes all runs.
exp.add_step('start', exp.start_runs)

# Add step that collects properties from run directories and
# writes them to *-eval/properties.
exp.add_fetcher(name='fetch')

# Make a report.
exp.add_report(
    BaseReport(attributes=ATTRIBUTES),
    outfile='report.html')
```

(continues on next page)

(continued from previous page)

```
# Parse the commandline and run the specified steps.
exp.run_steps()
```

Python Module Index

l

Index

Symbols

A

B

C

E

F

G

I

L

N

P

R

S

T

W