borg Documentation

Release 0.6

Bryan Silverthorn

November 17, 2013

Contents

Contents:

CHAPTER 1

Introducing borg

1.1 About

Modern heuristic solvers can tackle difficult computational problems, but each solver performs well only on certain tasks. An algorithm portfolio uses empirical knowledge—past experience of each solver's behavior—to run the best solvers on each task.

The borg project includes a practical algorithm portfolio for decision problems, a set of tools for algorithm portfolio development, and a research platform for the application of statistical models and decision-theoretic reasoning to the algorithm portfolio setting.

The project web site is:

http://nn.cs.utexas.edu/pages/research/borg/

The best description of the research surrounding the borg project can be found in the PhD dissertation of the primary author.

1.2 License

Borg is provided under the non-copyleft open-source "MIT" license. The complete legal notice can be found in the included LICENSE file.

CHAPTER 2

Installing borg

The installation process for borg is similar to that for other Python-based projects. Its installation involves three major steps, described in detail below:

1. installing any missing system-level dependencies;
2. installing virtualenv and creating an installation environment; and
3. installing borg and its required Python dependencies.

These instructions assume a `bash` shell on a Linux system.

2.1 Installing system dependencies

A reasonably complete development environment is required to compile borg and its dependencies. That includes at least:

- Python >= 2.6
- compilers: gcc, gfortran, and g++
- devel packages for
  - Python
  - linear algebra libraries: BLAS, LAPACK, and/or ATLAS

Most, if not all, of these packages are common dependencies, and should already be installed on a typical development machine.

2.1.1 Verifying Python >= 2.6

Make sure that you're using a recent Python version by running:

```
$ python --version
```

and checking that it reports at least "Python 2.6". Users on ancient platforms will likely need to install a local version of a more recent Python, as in the instructions for CentOS below.

Building Python on CentOS 5.4

CentOS 5.4, unfortunately, does not provide a modern version of Python. The recommended solution is to install one into a user-owned local directory, assumed to be `~/local` in the instructions that follow.

Download, unpack, build, and install Python 2.6:

```
$ wget http://www.python.org/ftp/python/2.6.6/Python-2.6.6.tar.bz2
$ tar jxvf Python-2.6.6.tar.bz2
$ cd Python-2.6.6
$ ./configure --prefix=$HOME/local/
$ make
$ make install
```

Add `~/local/bin` to the beginning of your path:

```
$ export PATH=~/local/bin:$PATH
```

Note that compiling a full Python system may require additional system dependencies, e.g., development packages for ncurses and zlib.

2.2 Creating an installation environment

The recommended approach to installing borg and its dependencies is to do so inside a "virtualenv", a self-contained local Python environment constructed with the virtualenv tool.

2.2.1 Obtaining virtualenv

The virtualenv tool may already be installed (try running "virtualenv" in your shell). If not, you may be able to install it using the system package manager. If you are using Ubuntu, for example, install it by performing:

```
$ sudo apt-get install python-virtualenv
```

If your system package manager does not include it, or you do not have system root access, you will need to download and use a local copy according to the instructions in the virtualenv documentation.

2.2.2 Creating an environment

Start by creating a virtual environment ("virtualenv") in some directory; we will assume `~/borg-venv`:

```
$ virtualenv --no-site-packages ~/borg-venv
```

The `--no-site-packages` flag isolates the virtualenv from Python packages installed globally.

2.2.3 Using the environment

Next, "activate" the virtualenv to use its Python installation in the current shell session:

```
$ source ~/borg-venv/bin/activate
```

Running `python` with this environment activated will use the local interpreter `~/borg-venv/bin/python`.

Note: The rest of the documentation assumes that you are operating with this environment activated.

2.2.4 Leaving the environment

The virtualenv can be later deactivated with:

```
$ deactivate
```

2.3 Installing borg

We can now install borg and its dependencies into this environment.

2.3.1 Installing the numpy dependency

Due to limitations in Python packaging, the numpy package must be installed first. Use

```
$ pip install numpy
```

to download, compile, and install numpy in the local environment. This may take a few minutes.

2.3.2 Installing borg and other dependencies

You should now be able to run

```
$ pip install borg
```

to download and install the latest release of borg from PyPI, as well its dependencies.

Note: Some of the borg dependencies, especially `cython`, `numpy`, `scipy`, and `scikit-learn`, are complex libraries that may take ten minutes or more to install from source using `pip`.

CHAPTER 3

Using borg

This chapter walks through the basic steps in using borg:

1. assembling solvers for some problem domain together into a "portfolio";
2. collecting performance data for each solver in that portfolio;
3. training a single portfolio solver to make solver execution decisions; and
4. running that portfolio solver on instances of the domain.

3.1 Assembling a portfolio of subsolvers

The first step in building a portfolio solver is to assemble its constituent solvers for the problem domain. We will often refer to these constituent solvers as "subsolvers", since they are wrapped by an outer "portfolio solver".

These subsolvers must be selected and prepared for execution, and then borg must be configured to execute them and to interpret their output. Here, we will build a portfolio solver for the SAT problem.

3.1.1 Fetching SAT solver binaries

In this example, we will build a simple portfolio consisting of several solvers from the 2011 SAT competition. Much of the data from these competitions can be accessed at

http://www.satcompetition.org/

including static binaries of the solvers entered into the competition. Let's download these and unpack them:

```
$ wget http://www.cril.univ-artois.fr/SAT11/solvers/SAT2011-static-binaries.tar.gz
$ tar zxvf SAT2011-static-binaries.tar.gz
```

> **Warning:** This tarball is 125MiB compressed, and may take some time to download.

After unpacking, we find the solvers under `SAT2011/bin/static` and information about how to execute them inside `SAT2011/bin/static/CommandLines.txt`.

3.1.2 Selecting solvers to use

While a portfolio solver can be a useful tool, it still requires significant domain knowledge to select the solvers to include in the portfolio. In this example, we will simply select three of the best-performing solvers from the 2009 SAT competition and the 2010 SAT race:

- `cryptominisat` (Mate Soos),
- `clasp` (Martin Gebser, Benjamin Kaufmann, and Torsten Schaub), and
- `TNM` (Wanxia Wei and Chu Min Li).

This selection follows the lead of the `ppfolio` tool, a basic parallel portfolio that performed well in the 2011 competition. More information is available in its documentation.

Since `ppfolio` has helpfully packaged these solvers for the 2011 competition, we can access them inside the `SAT2011/bin/static/main/sat11-11-roussel/bin` directory. Let's make a symlink to that directory with

```
$ ln -s SAT2011/bin/static/main/sat11-11-roussel/bin ppfolio-bin
```

so that we can access it more easily later.

3.1.3 Constructing a solver suite

We will now write a configuration file that allows borg to execute these SAT solvers by name. In borg, a collection of named solvers is referred to as a "suite".

A suite configuration file is simply a Python module that establishes how to execute each solver. Here is a configuration file for the suite of solvers selected above:

```
import borg

domain = borg.get_domain("sat")
commands = {
    "cryptominisat": ["{root}/ppfolio-bin/cryptominisat", "--randomize={seed}", "{task}"],
    "clasp": ["{root}/ppfolio-bin/clasp", "--seed={seed}", "{task}"],
    "TNM": ["{root}/ppfolio-bin/TNM", "{task}", "{seed}"],
    }
solvers = borg.make_solvers(borg.domains.sat.solvers.SAT_SolverFactory, __file__, commands)
```

We will call this file `suite_sat_ppfolio.py`.

A solver suite is required to provide two top-level variables:

- `domain`, which must be an instance of a class such as `borg.domains.sat.Satisfiability` that allows `borg` to parse instances of the problem domain, compute features on those instances, and determine basic properties of "answers" to instances of the domain; and
- `solvers`, a dictionary that maps arbitrary solver names (e.g., "minisat") to instances of a solver factory class, such as `borg.domains.sat.solvers.SAT_SolverFactory`, that allow `borg` to initiate solver runs on problem instances and to understand their output.

`borg` includes support for the output formats of various common solver types. In this case, the class `borg.domains.sat.solvers.SAT_SolverFactory` supports the typical output format of SAT competition entries.

3.2 Collecting solver performance data

The second step is to collect subsolver performance data for use in training. Each subsolver in the portfolio is run on each problem instance in the training set, often multiple times.

3.2.1 Gathering training instances

In this example, we will collect such data for the ppfolio suite on a small set of instances from the SAT2011 competition. Unfortunately, doing so requires downloading the entire set of instances from the competition:

```
$ mkdir benchmarks
$ cd benchmarks
$ wget http://www.cril.univ-artois.fr/SAT11/bench/SAT11-Competition-SelectedBenchmarks.tar
$ tar xvf SAT11-Competition-SelectedBenchmarks.tar
```

Warning: This tarball is 1.7GiB compressed, and may take some time to download.

The archive contains a huge number of individually compressed instances. For now, we will train our portfolio on a small subset of those instances. The easiest way to create such a subset is simply to symlink or copy the relevant instances into a common location—here, into a new directory named "selected":

```
$ mkdir selected
$ cd selected
$ cp ../SAT11/random/large/unif-k3-r4.2-v10000* .
$ cp ../SAT11/application/fuhs/AProVE11/* .
$ bunzip2 \*.bz2
```

We have now pulled together an arbitrarily selected set of 20 instances to use as our training set.

3.2.2 Executing solvers repeatedly

Borg can use an HTCondor or IPython.parallel cluster to execute solvers repeatedly and collect training data. For this experiment, set up a local IPython cluster by running:

```
$ ipcluster start -n 2
```

In this invocation, the ipcluster script will launch two engines for parallel processing. The value of the "-n" argument can be bumped up if you have more cores.

Note: If you do not have access to a cluster, StarCluster and other projects let you easily run one on EC2—but you will pay for it.

The borg run_solvers tool collects run duration data. By default, it uses the local IPython cluster. To invoke it, specify the portfolio configuration above (in this case, "suite_sat_ppfolio.py"), the directory containing the set of benchmarks on which to execute the solver suite (in this case, "benchmarks/selected"), and the run duration cutoff in seconds (in this case, 300 seconds).

```
$ python -m borg.tools.run_solvers suite_sat_ppfolio.py benchmarks/selected/ 300
```

Warning: Solver run data collection is extremely expensive in general. For example, even though this suite of solvers and collection of benchmark instances are both quite restricted, this set of runs will take a substantial amount of time—12 hours or more—to complete using one or two cores.

3.2.3 File format: subsolver run records

Run records are typically stored in CSV files with the suffix `.runs.csv`. The full suffix can be modified through the "-suffix" flag to run_solvers, among other borg tools.

The following columns are expected in the following order:

`solver` Unique name of the solver.

`budget` Budget allotted to the run, in CPU seconds.

`cost` Computational cost of the run, in CPU seconds.

`succeeded` Did the solver succeed on this run?

`answer` Base64-encoded gzipped pickled answer returned by the solver on this run, if any.

3.3 Generating instance feature information

Portfolios typically use domain-specific information about a given problem instance to make better solver execution decisions.

Borg's "get_features" tool collects such information for the domains that it supports. The set of features collected is built into borg.

We can use this tool to collect feature information from the set of selected benchmarks:

```
python -m borg.tools.get_features sat benchmarks/selected/
```

Like the run_solvers tool, get_features uses the local IPython cluster by default.

3.3.1 File format: instance features

Instance features are stored in CSV files with the suffix `.features.csv`. The first column must be `cost`, the computational cost of feature computation, in CPU seconds. The remaining columns are domain-specific, one per feature.

3.4 Training a portfolio solver

This section will walk you through the process of training a borg portfolio.

At this point we have both a suite of subsolvers and a set of training data. The third and penultimate step in constructing a borg portfolio is fitting a predictive model to these data. Use the "train" tool to fit a model:

```
$ python -m borg.tools.train borg-sat-ppofolio.model.pickle borg-mix+class solvers/pb/portfolio.py ta
```

This process can take ten minutes or more, depending on the amount of training data and the nubmer of subsolvers. It will write a portfolio model (of type "borg-mix+class") to the file "pb-model.pickle".

Finally, we need to calibrate the solver to the local execution environment, since the machines used to collect training data may be faster or slower than the machine on which you're running the portfolio. Every collection of training data includes a "calibration" directory, which contains a problem instance, a SOLVER_NAME file, and a runs file with the ".runs.train.csv" suffix. This directory stores runs made by a single solver (that named in SOLVER_NAME), on a single instance, on the machine(s) used for training data.

To collect runs using the same solver on the local machine, run

```
$ python -m borg.tools.run_solvers solvers/pb/portfolio.py tasks/pb/calibration/ 120 -r 9 -suffix .lo
```

which will make 9 runs and store them in the corresponding "<instance>.local.runs.csv" file.

Then compute the local machine calibration factor with:

```
$ python -m borg.tools.get_calibration tasks/pb/calibration/normalized-cache-ibm-q-unbounded.Icl2arit
```

The ratio that it prints can be used as a value for the "–speed" parameter to the "solve" tool discussed below.

3.5 Running the trained portfolio solver

Now let's solve the same calibration using the full portfolio, with

```
$ python -m borg.tools.solve --speed 1.0 borg-pb.model.pickle solvers/pb/portfolio.py tasks/pb/calibr
```

changing the speed parameter given the output of the "get_calibration" tool above.

Borg will parse the instance, compute instance features, condition its internal model, and run a sequence of solvers—replanning as necessary. In this case, it should quickly solve the instance with its first solver run.

CHAPTER 4

Contributions and development

If you are interested in modifying, contributing to, or basing your own project on top of `borg`, this chapter provides a bit of documentation on doing so.

4.1 Obtaining the project source code

The suggested way to download the project source is to use git, which will simplify acquiring future updates and making local changes. Assuming that git is installed, clone the two relevant repositories from github:

```
$ git clone git@github.com:borg-project/cargo.git
$ git clone git@github.com:borg-project/borg.git
```

If git is not installed and cannot be installed, tarball snapshots of the source trees can be downloaded from github's web interface; see the borg and cargo repositories under the borg project github page.

4.2 Building documentation

The borg documentation can be generated in various formats from its reStructuredText source. The *sphinx* tool is used to drive this process. It can be installed as usual via *pip*, with:

```
$ pip install sphinx
```

```
$ cd borg/docs
$ make html
```

CHAPTER 5

API reference

5.1 `borg` module

5.1.1 Submodules

`borg.expenses` module

Members

`borg.solver_io` module

Members

5.1.2 Members

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*