Equibel Documentation

Release a1

Paul Vicol

May 30, 2016

Contents

Equibel is a Python package for working with consistency-based belief change in a graph-oriented setting.

CHAPTER 1

Currently Supported Platforms

- OS X with Python 2.7.x
- 64-bit Linux with Python 2.7.x

Note that while Equibel is distributed as a Python package, the core of the system is implemented using Answer Set Programming (ASP), and relies on a nunderlying ASP solver called `clingo`, which is part of the Potsdam Answer Set Solving Collection (Potassco).

In particular, Equibel has two ASP-related dependencies: the Python gringo module, which provides an interface to an ASP solver from within Python, and `asprin.parser`, which is a component of the `asprin` preference-handling framework. `asprin` is described in more detail here, and can be download from here.

The *Python* component of Equibel is highly portable across platforms; however, the `gringo` and `asprin.parser` dependencies must be compiled for specific system configurations, producing system-specific binaries. In order to simplify usage for some *common* system configurations, Equibel includes pre-compiled binaries of these dependencies for 64-bit Linux distributions and Mac OS. These are placed in the `equibel/includes/` directory, which is structured as follows:

```
equibel/includes/
-- __init__.py
-- linux_64
|   -- asprin.parser
|   -- gringo.so
|   -- __init__.py
-- mac
    -- asprin.parser
    -- gringo.so
    -- __init__.py
```

If Equibel does not function correctly once it is installed, this is likely due to the fact that the pre-compiled binaries are not compatible with your system. In this case, you must compile the dependencies manually, by downloading the required components directly from Potassco, and overwriting the resulting binaries in the folder that corresponds to your operating system.

CHAPTER 2

Installation

The following steps assume that you have the `pip` Python package manager installed. If you don't have `pip`, you can get it here.

1. The pre-compiled `gringo` modules included with Equibel for either 64-bit Linux or Mac OS require a dependency called Threading Building Blocks (`tbb`).
   - The easiest way to install the `tbb` library on **Mac OS** is to use [Homebrew](http://brew.sh):

```
$ brew install tbb
```

   - On **Ubuntu Linux**, the `tbb` library can be installed using the `apt` package manager:

```
$ sudo apt-get install libtbb-dev
```

2. Download the Equibel source code from Github:

```
$ git clone https://github.com/asteroidhouse/equibel.git
```

   This will create a folder called `equibel` in your current working directory.

3. Change directories to the `equibel` folder:

```
$ cd equibel
```

4. Install Equibel using `pip` from within the `equibel` folder as follows (note the `.` at the end):

```
$ sudo pip install .
```

5. Optionally, you can test whether everything works on your system by installing the `nose` testing tool and running the tests in the *tests* folder, as follows:

```
$ sudo pip install nose
```

```
$ nosetests tests/
```

   If all of the dependencies have installed correctly, the `nosetests tests/` should print a series of dots to the screen, one for each successfully completed test case.

   If the tests fail, this is most likely due to the *dependencies* of Equibel not being compatible with your platform. As noted above, Equibel includes pre-compiled binaries of the Python `gringo` module, as well as of `asprin.parser`, for 64-bit Linux distributions (tested on Ubuntu 14.04) and for Mac OS (tested on OSX 10.10). If you are not using one of these systems, you will need to manually compile the `gringo` and `asprin.parser` dependencies.

CHAPTER 3

Quickstart

To use Equibel within a Python program, you need to import the `equibel` module. Every form of belief change in Equibel takes as input an `EquibelGraph` object (that represents a graph and associated scenario) and outputs a new `EquibelGraph` object. The following Python script creates a path graph, assigns formulas to nodes, find the global completion, and prints the resulting formulas at each node:

```
import equibel as eb

if __name__ == '__main__':
    # Create an EquibelGraph object, which represents a graph and
    # associated scenario.
    G = eb.EquibelGraph()

    # Create nodes:
    G.add_nodes_from([1, 2, 3, 4])

    # Create edges:
    G.add_edges_from([(1,2), (2,3), (3,4)])

    # Add formulas to nodes:
    G.add_formula(1, "p & q & r")
    G.add_formula(4, "~p & ~q")

    # Find the global completion of the G-scenario:
    R = eb.global_completion(G)

    # Pretty-prints the resulting formulas at each node:
    eb.print_formulas(R)
```

If the above code is saved in a file called `completion.py`, then it can be run by typing `python completion.py` at the command line, as follows:

```
$ python completion.py
Node 1:
p  q  r
Node 2:
r
Node 3:
r
Node 4:
r  ¬p  ¬q
```

CHAPTER 4

Implemented Approaches

Equibel allows for experimentation with several different approaches to consistency-based belief change in a graph-oriented setting, namely:

1. Global completion,
2. Simple iteration,
3. Expanding iteration,
4. Augmenting iteration, and
5. The ring method.

The global completion opteration is performed on an `EquibelGraph G` by `eb.global_completion(G)`; this performs a "one-shot" procedure to update the information at every node in the graph, and thus is not an iterative approach. All of the other approaches—*simple*, *expanding*, *augmenting*, and *ring*—can be performed iteratively, and each one iterates to a *fixpoint*. The table below summarizes the Equibel functions to perform single iterations of each approach, as well as to find the fixpoints reached by each approach:

Method	Single Iteration	Iterate to Fixpoint
Simple Iteration	`eb.iterate_simple(G)`	`eb.iterate_simple_fixpoint(G)`
Expanding Iteration	`eb.iterate_expanding(G)`	`eb.iterate_expanding_fixpoint(G)`
Augmenting Iteration	`eb.iterate_augmenting(G)`	`eb.iterate_augmenting_fixpoint(G)`
Ring Iteration	`eb.iterate_ring(G)`	`eb.iterate_ring_fixpoint(G)`

Each of the approaches has two separate implementations, corresponding to its equivalent *semantic* and *syntactic* characterizations. In addition, there are two ways of performing the core optimization procedure over equivalences, involving either *inclusion-based* or *cardinality-based* maximization.

Each function listed above can take three optional arguments:

1. `method`, which is a string that is either "semantic" or "syntactic", representing the method to use when performing the approach; e.g. based on either the syntactic or semantic characterizations
   - The default `method` is *semantic*
   - To avoid typos when entering strings, Equibel has constants `eb.SEMANTIC` and `eb.SYNTACTIC` which equal the strings "semantic" and "syntactic", respectively.
2. `opt_type`, which is a string that is either "inclusion" or "cardinality", representing the type of maximization to be performed over equivalences
   - The default `opt_type` is *inclusion*
   - To avoid typos when entering strings, Equibel has constants `eb.INCLUSION` and `eb.CARDINALITY` which equal the strings "inclusion" and "cardinality", respectively.

3. `simplify`, which is a Boolean flag specifying whether to simplify the final formulas at each node.
    - The default value for `simplify` is `False`

By definition, the semantic and syntactic characterizations of an approach yield *equivalent results*; however, depending on the input scenario and type of approach, the performance of the characterizations may differ significantly. A good example of this is in the case of expanding iteration, where we have an *early-stoppping condition* over the radius of the expanding neighbourhood when using the semantic characterization, but not when using the syntactic characterization (causing the semantic characterization to be significantly faster for large graphs in practice).

4.1 Some Examples

To show how the `method` and `opt_type` arguments can be combined, we consider the following (by no means exhausitive) examples.

In the following example, we can see the difference between using inclusion-based optimization and cardinality-based optimization in the global completion:

```
import equibel as eb

if __name__ == '__main__':

    # Creates a star graph with nodes [0, 1, 2, 3] and undirected edges [(0,1), (0,2), (0,3)]
    G = eb.star_graph(3)
    G.add_formula(1, 'p')
    G.add_formula(2, 'p')
    G.add_formula(3, '~p')

    # Using inclusion-based maximization over equivalences
    R_inclusion = eb.global_completion(G, method=eb.SEMANTIC, opt_type=eb.INCLUSION, simplify=False)
    eb.print_formulas(R_inclusion)

    # Using cardinality-based maximization over equivalences
    R_cardinality = eb.global_completion(G, method=eb.SEMANTIC, opt_type=eb.CARDINALITY, simplify=Fal
    eb.print_formulas(R_cardinality)
```

Saving this code in a file `inclusion_vs_cardinality.py` and running it yields:

```
$ python inclusion_vs_cardinality.py
Node 0:
p  ¬p
Node 1:
p
Node 2:
p
Node 3:
¬p


Node 0:
p
Node 1:
p
Node 2:
p
Node 3:
¬p
```

The following example function calls show the flexible way in which options can be combined and used with any approach in Equibel:

- **`R_semantic = eb.global_completion(G)`**
  - This function call computes the global completion of `G`. With no options explicitly specified, the defaults are used; thus, this call involves the *semantic characterization* with *inclusion-based* optimization, and does not simplify the resultant formulas.
  - With all options explicitly specified, the above function call is equivalent to `R_semantic = eb.global_completion(G, method=eb.SEMANTIC, opt_type=eb.INCLUSION, simplify=False)`
- **`R_semantic = eb.global_completion(G, method=eb.SYNTACTIC)`**
  - This call finds the global completion of `G`, using the *syntactic characterization*, the default *inclusion-based* optimization, and no simplification of formulas.
- **`R_semantic = eb.global_completion(G, method=eb.SYNTACTIC, opt_type=CARDINALITY)`**
  - This call finds the global completion of `G`, using the *syntactic characterization*, *cardinality-based* optimization, and no simplification of formulas.
- **`R_semantic = eb.iterate_simple(G, method=eb.SEMANTIC, simplify=True)`**
  - This function call computes the graph and scenario that result from performing a single *simple iteration* over `G`, using the *semantic characterization* with default *inclusion-based* optimization. With the `simplify=True` option, the resulting scenario will have simplified formulas for each node in the graph.
- **`R_syntactic = eb.iterate_simple(G, method=eb.SYNTACTIC, simplify=True)`**
  - This call is similar to the previous call, except that it uses the *syntactic characterization* of simple iteration, rather than the semantic characterization.
- **`R_semantic_fixpoint = eb.iterate_simple_fixpoint(G, method=eb.SEMANTIC, opt_type=eb.CA`**
  - This computes the fixpoint reached by a sequence of *simple iterations* starting from the graph and scenario represented by `G`, using the *semantic characterization* and *cardinality-based* optimization.
- **`R_semantic = eb.iterate_expanding(G, simplify=True)`**
  - This function call computes the graph and scenario that result from performing a single *expanding iteration* over `G`, using the default *semantic characterization* with default *inclusion-based* optimization. Since `simplify=True`, the resulting scenario will have simplified formulas for each node in the graph.
- **`R_semantic = eb.iterate_augmenting_fixpoint(G, simplify=True)`**
  - This computes the fixpoint reached by a sequence of *augmenting iterations* starting from the graph and scenario represented by `G`, using the default *semantic characterization* and *inclusion-based* optimization. Since `simplify=True`, the resulting scenario will have simplified formulas for each node in the graph.

CHAPTER 5

Contents

5.1 equibel

5.1.1 equibel package

Subpackages

equibel.asp package

Module contents

equibel.formatters package

Submodules

equibel.formatters.aspformatter module

equibel.formatters.bcfformatter module

Module contents

equibel.includes package

Subpackages

Module contents

equibel.parsers package

Submodules

equibel.parsers.bcfparser module

equibel.parsers.formulaparser module

equibel.parsers.parsetab module

Module contents

Submodules

equibel.asprin module

equibel.draw module

equibel.formulagen module

equibel.graph module

equibel.graphgen module

equibel.solver module

Module contents

5.2 Licence

CHAPTER 6

Indices and tables

- genindex
- modindex
- search