BiRG

Jesper Brink and Per Høgfeldt

Apr 15, 2020

CONTENTS

BiRG is an automation tool for generating Conda recipes for the Bioconda channel. It is built on heuristics about how to find dependencies based on the error messages it gets from trying to build and run some given software.

BiRG achieves the same goal as conda skeleton, but instead of collecting the metadata from PyPI and building the recipe based on that, BiRG finds a package's dependencies by trying to build the package, and based on the error messages BiRG tries to detect the dependencies.

Therefore, BiRG is the choice if you don't have the software's metadata already available. However, in case you do have a package on PyPI, we recommend to simply use conda skeleton. A guide to this can be found here.

CHAPTER

ONE

USER'S GUIDE

1.1 User's Guide

1.1.1 Installation

BiRG is heavily dependent on Docker, therefore make sure that you have Docker installed and working. We use Docker to ensure that the environment, which we are building and testing in, is clean and reproducible.

If you don't have Docker installed, we recommend that you follow the Docker installation guide which can be found at the following link: Docker installation guide

When Docker is installed, we recommend following the post-installation step "Manage Docker as a non-root user", the linux guide can be found here . Setting up Docker to be used as non-root, will make it possible to run BiRG without escalated privileges.

From Docker Hub

We have prepared a docker image for running BiRG, the docker image can be downloaded from Docker Hub, by running the following command:

```
$ docker pull perhogfeldt/birg:latest
```

That's it you can now proceed to the tutorial, for instructions on how to use the docker image.

Tutorial

From Source

First clone the BiRG recipe from github with the following command:

```
$ git clone https://github.com/birgorg/birg.git
```

Next up is to create a conda environment from the environment.yml file in the BiRG repo folder:

```
$ cd birg
$ conda env create -f environment.yml
$ conda activate birg
```

Note: There might be problems with creating an environment from the environment.yml file, if you are on a mac. A solution is to run the following commands instead:

```
conda create -n birg python=3.6 bioconda-utils docker-py gitdb2=2.0.5
conda activate birg
```

Install it with the setup.py and check if BiRG is installed correctly by running -h:

```
$ pip install .
$ birg -h
```

1.1.2 Tutorial

To show how BiRG works, we will create a recipe for a software called kallisto. We will start with generating an initial recipe, with basic information about kallisto. This recipe will then be given as input to BiRG, which will find the necessary dependencies required to build, test and run the software.

We assume that you are using the docker image that we provide, but if you have installed BiRG from source, the tutorial still applies, just go directly to *Recipe Initilization*.

Using the Docker Container

The docker container should be called with the following settings, so that the container and the host system can communicate.

```
$ docker run -v /var/run/docker.sock:/var/run/docker.sock \
             -v $PWD:/home \
             --user $(id -u):$(id -g) \
             -it perhogfeldt/birg:latest \
             <birg-command>
```

In the rest of this tutorial you can just replace *birg* with the above command.

Here is a short explanation on what the settings are for, if you are not so familiar with docker.

The first volume we added, with *-v*, is the unix socket that the docker client uses to talk with the docker server. We need to add this socket, since the container will need to spin up more containers, when it tries to build the software. The second volume will bind the containers home direcetory to the host systems current directory, this will make sure that files can be shared. the *–user* option makes sure that files created on you system will be owned by your current user and *-it* just makes sure that you can communicate with the container directly in your current terminal.

If you prefer not to write all the settings every time you call birg, we recommend exporting the settings in an environment variable like this:

```
$ export BIRG="-v /var/run/docker.sock:/var/run/docker.sock \
                       -v $PWD:/home --user $(id -u):$(id -g) \
                       -it perhogfeldt/birg:latest"
```

Then you can just call birg with the environment variable:

```
$ docker run $BIRG <birg-command>
```

Recipe Initilization

First we will generate an initial recipe, this can be done by using the command *birg init*. BiRG will then ask for basic information about the software and create a directory named after the software, which contains two files called *meta.yaml* and *build.sh*. These files are the initial recipe, that we will transform into the final recipe.

The init command will ask for the following informations:

- Name
- Version
- Source url
- A build strategy to use

Here is the initialization of kallisto:

```
$ birg init
Package name: kallisto
Version: 0.46.2
Url to download the code: https://github.com/pachterlab/kallisto/archive/v0.46.2.tar.
↪gz
Choose the strategy that you use to run your code
['cmake', 'python2', 'python3']: cmake
```

The Recipe

The basic recipe created by *init*, can be found in the newly created directory called *kallisto* and should look like this:

Listing 1: kallisto/meta.yaml

```
package:
   name: kallisto
   version: 0.46.2
source:
   url: https://github.com/pachterlab/kallisto/archive/v0.46.2.tar.gz
   md5: a6257231c6b16cac7fb8ccff1b7cb334
build:
   number: 0
```

Listing 2: kallisto/build.sh

```
 #!/bin/bash
 mkdir -p build
 cd build
 cmake ..
 make
 make install
```

This is the minimal initial recipe, that you can give as input to BiRG. To make it easier for BiRG to find run-time dependencies it is important to add tests to the *meta.yaml* file. If you have a patch or would like to add some additional meta data, feel free to do so. For information on what data and configuration you can add to a recipe, see the official Conda documentation here

Before using the recipe for kallisto as input to BiRG, we will add some tests to the *meta.yaml* file. By adding tests, we makes sure that BiRG will try and find run-time dependencies as well as build-time dependencies.

Listing 3: kallisto/meta.yaml

```
package:
   name: kallisto
   version: 0.46.2
source:
   url: https://github.com/pachterlab/kallisto/archive/v0.46.2.tar.gz
   md5: a6257231c6b16cac7fb8ccff1b7cb334
build:
   number: 0
test:
   commands:
      - kallisto cite
```

We will also edit the the *build.sh*, as kallisto requires us to run autoreconf and to set some flags for cmake:

Listing 4: kallisto/build.sh

```
#!/bin/bash

cd ext/htslib
autoreconf
cd ../..

mkdir -p $PREFIX/bin
mkdir -p build
cd build
cmake -DCMAKE_INSTALL_PREFIX:PATH=$PREFIX .. -DUSE_HDF5=ON
make
make install
```

Note: The recipe for kallisto can be found in our github repo here

Recipe Build

We are now ready to give our inital recipe as input to BiRG. The build command takes two required arguments as shown below:

```
$ birg build --help
usage: birg build [-h] [-d] recipe_path {cmake,python2,python3}

positional arguments:
  recipe_path           Path to folder with meta.yaml and build.sh templates
  {cmake,python2,python3}
                        The ? that you used when creating the template with
                        'init'

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Set this flag if you want to activate the debug mode.
                        This creates an debug.log file that contains all debug
                        prints
```

recipe_path: Is the path to the recipe directory which was created by running *birg init*.

strategy: Here you must tell BiRG which building strategy to use. BiRG currently supports three strategies: cmake, python2 and python3.

Here is an example on how BiRG is called for building kallisto:

```
$ birg build kallisto/ cmake
```

When BiRG is running it will print out a lot of text, this is the output from it's building process. BiRG will also, sometimes, ask for your help, to determine which version of a dependency it should use.

When BiRG is done running it will tell you if it was able to build and run your software, and the output recipe can be found in the directory which was created by the *init* command.

Here is the final recipe for kallisto:

Listing 5: kallisto/meta.yaml

```
package:
  name: kallisto
  version: 0.46.2
source:
  url: https://github.com/pachterlab/kallisto/archive/v0.46.2.tar.gz
  md5: a6257231c6b16cac7fb8ccff1b7cb334
build:
  number: 2
test:
  commands:
  - kallisto cite
requirements:
  build:
  - cmake
  - make
  - automake
  - {{ compiler('cxx') }}
  host:
  - hdf5
  run:
  - hdf5
  - zlib
```

Listing 6: kallisto/build.sh

```
#!/bin/bash

cd ext/htslib
autoreconf
cd ../..

mkdir -p $PREFIX/bin
mkdir -p build
cd build
cmake -DCMAKE_INSTALL_PREFIX:PATH=$PREFIX .. -DUSE_HDF5=ON
make
make install
```

Congratulation you can now add your recipe to Bioconda and share your software.

CHAPTER

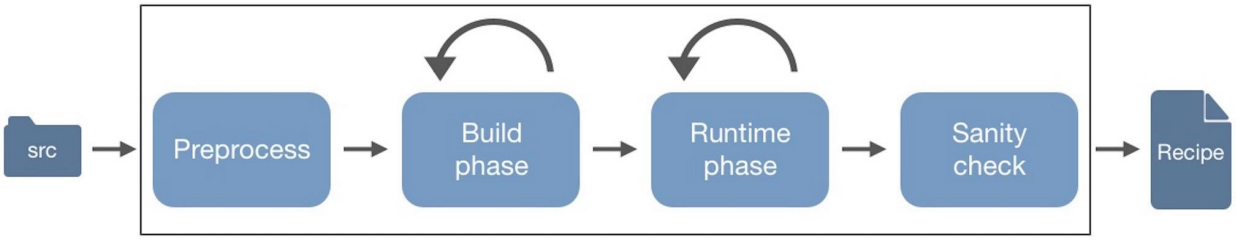
TWO

DEVELOPMENT

2.1 Development

2.1.1 How BiRG Works

BiRG is split into four phases: preprocess, build phase, runtime phase and sanity check.



Preprocess

The preprocess step only applies to the python build strategy at the moment. What BiRG does for python packages is to look for a setup.py file and if it finds it BiRG runs *python setup.py sdist*. This outputs a requires.txt file, which hopefully contains most of the packages needed to build the given software. These packages are then added to the recipe before going to the next phase (and therefore reducing the overall running time).

Build Phase

The build phase is an iterative process where BiRG takes the current recipe and the given software, and runs the *conda build* command. In this phase, the *–build-only* flag is set, to make sure it stops before running the tests in the recipe.

This gives a way to find all the build and host dependencies for the recipe before moving on to the runtime phase.

Runtime Phase

The runtime phase works like the build phase, but does also execute the tests given in the recipe. This is therefore where BiRG finds all dependencies used at runtime.

Sanity Check

The sanity check comes as the last thing, after the runtime phase has added all the dependencies it could detect. Here BiRG runs the final recipe with bioconda-utils to see whether or not the final recipe can build the package successfully. If successful, then the recipe is ready to build the software, and if not, the user has to see if it is possible to manually detect the missing package(s) or version(s).

2.1.2 Test Setup

The way we have been developing BiRG is by taking a set of packages from bioconda-recipes and then use our test pipeline which can be found in the birg_ci repo. This pipeline runs BiRG on all the chosen packages and outputs for each package:

- a file with everything from stdout that was generated doing the build
- the generated meta.yaml
- the generated build.sh

Furthermore, it outputs a single overview file with each package name and whether or not it did build.

Exactly how to use the pipeline (or part of it) is explained in the the README of the birg_ci repo.

2.1.3 Experiments

To see how well BiRG performs, we found all python (exclusively python, no other code like for example c) and cmake (packages with a CMakeList.txt in the root folder) packages on bioconda-recipes.

We then tried to build the recipes for these packages with BiRG. We considered the recipe created successfully if running *bioconda-utils build* with the recipe resulted in a successful build, as this is how a recipe should be used to make a conda package.

Note: Before running the packages on BiRG, we filtered out all packages that didn't get a successful built with *bioconda-utils build* given its own recipe (E.g. blacklisted packages)

Results

- **Python packages (both python2 and python3):**
  - Date: 15/3/20
  - Total amount: 397
  - Successful builds: 256
  - Percentage of packages successfully built: 64%
- **Cmake packages:**
  - Date: 13/11/19
  - Total amount: 49

- Successful builds: 28
- Percentage of packages successfully built: 57%

2.1.4 Suggetions for future work

This page is a compilation of suggestions for improvements that we haven't had the time to make.

Generalizing how to find binaries/libraries in packages

Motivation: When a package fails to build because of a missing binary file or library header, it can be very hard to figure out which package should be included as a dependency. Our current solution is to map known 'failing signatures' to known packages, and then add the package as a dependency, to observe if adding the package solves the problem. This way of mapping seems to work, for a large amount of packages, since many of the packages on Bioconda, seems to use the same dependecies. But it doesn't scale well, since we have to observe and add new signatures manually.

Suggestion: It would be really helpful to index all files in all Bioconda packages, so that we could search for a specific executable or library header file in all Bioconda packages. By doing this we could generalize finding binaries/library files by extracting the file name with a regular expression and search for the packages containing this file. If more than one result comes up, it would be easy to implement some kind of heuristic for choosing the right package or simply ask the user for help. As a part of the priliminary study for this project we downloaded, unpacked and made a primitive index of all files in all packages from Bioconda, so we was able to answer questions like "How many packages on Bioconda have a Makefile in it's project root dir?" or "How many packages have a setup.py file?". This was quite fast and could probably be automated a lot with a good CI setup. Creating such a search engine for Bioconda would not only benefit this tool, but would also benefit developers trying to create a recipe for their software by hand. We therefore suggest that such a tool could be developed as a separate piece of software, which we then could integrate in this tool.