SANA Software
Experiment Module

Nil Mamano

April 10, 2016

1 Introduction

This report describes SANA's built-in experiment module. It is a powerful and easily customizable module that allows to submit multiple runs to the cluster, evaluate all the obtained alignments, and synthesize the data in useful output files. It requires minimal or no manipulation of the code.

The module works as follows: there is a experiment file (3.1) detailing the parameters of the experiments: number of runs, methods used, datasets used, ... When the experiment is executed, SANA creates one bash script for each run, and then submits all the scripts to the cluster.

Each script consists of a call to SANA with a specific set of arguments to achieve the required setting. Thus, each individual run part of the experiment is equivalent to a "normal" run executed through a terminal. This gives the user precise control over what is going on.

2 File system overview

All the associated files are in the `experiments` folder.

Some parameters are common to all the experiments. These are factorized out in two configuration files, `datasets.cnf` and `methods.cnf` (3.2).

Each experiment should have one subfolder in `experiments`. In it there has to be the experiment file describing the parameters of the experiment. The experiment file must have the same name as the folder with the `.exp` extension, e.g., `biogridBeta05/biogridBeta05.exp`. In this documents it is denoted by `experiment`.

Within an experiment, each run is identified by 3 items:

- The method, e.g., HubAlign.
- The network pair $G1$ and $G2$, e.g., Yeast and Human.
- The *submission number*. For each method and network pair combination, the experiment performs one or more runs. These runs are identical in all

the arguments. This number helps distinguish cases where there is more than one run per each method and network pair. The submission number ranges from 0 to $n-1$ for each method and network pair, where n is the number of runs for each method and network pair.

Each run has a *textual identifier*: "*method_G1_G2_runnumber*". For example, "hubalign_Yeast_Human_0". All the files associated to a given individual run use the textual identifier as part of the file name. In this document it is denoted by `run_id`.

Once the experiment is run, a set of files and folders are generated inside the experiment folder:

Intermediary files

- The `scripts/` folder contains all the bash scripts. Each run has a script `run_id.sh`. This is useful to double check exactly with which arguments SANA is being called.
- The `outs/` folder contains all the standard outputs. Each run has a file `run_id.out` with its standard output. This is useful for debug purposes. For instance, if some run crashed, this can point to what was happening when it crashed. It is updated in real time as the scripts run in the cluster, so another use is to check the progress of the runs. If a run is not finishing, you can see at which point it is at.
- The `errs/` folder contains all the standard errors. Each run has a file `run_id.err` with its standard error. This is useful for debug purposes and updated in real time.
- The `results/` folder contains all the reports (and the alignments, which are in the first line of the reports) generated by SANA. Each run has a file named `run_id` containing the report of the obtained alignment. It is useful to check additional results related to the run which are left out in the synthesized output files, as well as extracting really good alignments for future use. The alignments in the reports are also used by the experiment itself during the data-synthesis phase to evaluate the alignments and find all the required scores.

Output files

- The `experimentPlainResults.txt` output file contains all the data that is used to generate the other output files of the experiment, without any preprocessing. It contains one row for each result. Each result consists of the measure being evaluated, the network pair, the method used, the submission number, and the score. For instance:

  ```
  s3 RNorvegicus SPombe lgraal 0 0.184569
  ```

- The `experimentHumanReadableResults.txt` output file contains the results arranged in tidy tables that are easy to read. It contains two sets of tables. The first set consists of score tables. There is one table for each measure being evaluated, with one row per network pair, and one column per method. Scores are averaged among all runs with the same method and network pair. It includes averages for each method.

  The second set of tables is analogous, but instead of scores, it contains the ranking of the average score of the methods, with 1 being the best average score, 2 the second best, and so on. Tied methods share ranking: for instance, a set of scores $0.5, 0.5, 0.5, 0.2, 0.2, 0.2$ are listed as having ranking $1, 1, 1, 4, 4, 4$.

- The `experimentForPlotResults.csv` output file contains the same data as the score tables in `experimentHumanReadableResults.txt`, but in CSV format. It is intended to be used as input for a plotting tool such as *Veusz* without any preprocessing required[1].

  The first row contains the headers. Then, there is one row for each network pair and the last one contains the averages. The headers are named `experiment_method_measure`, e.g., `biogridBeta05_lgraal_s3`. The experiment name is added to the column name so that the outputs of multiple experiments can be inputted to a plotting tool at the same time without collisions.

3 Experiment setting

3.1 Experiment file

This is an example of valid experiment file:

```
60
2
-wecnodesim graphletlgraal -objfuntype beta -beta 0.5
lgraal hubalign sanalg sanaec sanas3 random*
biogrid
s3 lccs ec wecgraphletlgraal importance* sequence gocov* go1* go2*
```

The top 6 lines have a specific meaning and format, while the rest of the document is ignored. The first 6 lines are:

1. Execution time per run, in minutes. The line should consist of a single floating point number.

2. Number of submissions per method/network pair combination. It should consist of a single natural number.

[1] This is one of the situations where you might want to actually modify the code, to produce exactly the output that you need.

3. Common arguments to all experiments. In this line you can add arguments that will be appended "as is" to the command line command of all the runs. In general, here are included parameters that affect multiple methods, such as `-objfuntype`, or parameters that affect a certain measure, such as `-wecnodesim`. The arguments must work fine with all the methods used in the experiment.

4. Space separated list of method identifiers. Each identifier must appear in `experiments/methods.cnf` (3.2).

   Optionally, each identifier can be appended with "`*`". The only difference is that the given method is omitted in the `experimentForPlotResults.csv` file. This is useful for when you want to compare a method to the others but it should not be included in the plots.

5. Dataset identifier. It must appear in `experiments/datasets.cnf` (3.2).

6. Measures to be evaluated. Similarly to the methods, measures appended with "`*`" are omitted from the CSV file.

3.2 Configuration files

The additional configuration files that affect the experiment are:

- `experiments/methods.cnf`. Contains one line per method. The first word is the method identifier used in the experiment files. The rest of the line consists of arguments common to all runs with this method. They will be appended "as is" to the command line command of the corresponding runs. For instance

  ```
  random -method random
  sanas3 -method sana -objfuntype generic -topmeasure s3 -s3 1 -ec 0
  ```

  One of the arguments should be `-method`, but there is no direct correspondence between the implemented methods and the method identifiers. This gives the possibility to define multiple identifiers for variants of a given method, such as `sanaec` and `sanas3`.

  Note that in the example, for the method `sanas3` we have `-topmeasure s3`, even though this argument is only used when `-objfuntype` is not `generic`. This is because the common arguments in the experiment file are appended *after* the method-specific ones, and thus could override the value of `-objfuntype`. In that case, the method `sanas3` would still optimize $S3$.

- `experiments/datasets.cnf`. Contains one or more datasets, one after the other. A dataset consists of one line with the dataset identifier followed by a list of network pairs, each pair in an individual line. The identifier can be any string, while the network names must match names of networks in

the `networks` folder. A network can appear in more than one pair within a given dataset and in more than one dataset. The network names become the values of the `-g1` and `-g2` arguments.

4 How to use the experiment module

4.1 Preliminaries and Setup

1. Make sure you can submit jobs to the cluster with `-qsub` command.
2. Make sure SANA works properly in *Cluster Mode*. **You need to customize the variable** `projectFolder` at the top of `ClusterMode.cpp` to the absolute address of the root of your project (the root git directory of SANA)[2]. The current value is set after my own folder:

   ```
   string projectFolder = "/extra/wayne0/preserve/nmamano/sana/";
   ```

   To test that the Cluster Mode works properly, try

   ```
   $ ./sana -mode cluster -qmode normal
   ```

   The program should run fine. You can further check with the `qstat` command that now there is a job running in the cluster.
3. Create a new subfolder in `experiments/` and create the experiment file with a matching name and `.exp` extension (tip: create it by copying an already existing experiment file and modifying it).
4. Fill the 6 first lines of the experiment file with the expected format.
   - Choose a dataset from the `datasets.cnf` file or add a new one.
   - Choose methods from the `methods.cnf` file or add new ones. Make sure that the common arguments and the method-specific arguments are consistent. **The common arguments override the method-specific arguments.**

4.2 Cluster submission phase

You are now ready to run the experiment. It is run with

```
$ ./sana -mode exp -experiment experiment
```

[2] If someone knows how to find out this folder from within the code, so that this variable can be initialized automatically, please change it!

The first time the experiment is run, it will submit all the jobs to the cluster and finish. If the special argument `-dbg` is present, instead of submitting the jobs to the cluster, it will print all the combinations of method, network pair, and submission number that would be submitted to the cluster. It is recommended to test this before submitting to the cluster, to make sure that the experiment setup is correct.

```
$ ./sana -mode exp -experiment experiment -dbg
```

It is common that some runs get stuck or interrupted. A run is considered not finished if its corresponding file in the `results/` folder is missing. If the experiment is run again, **only** the submissions that haven't finished are submitted again. Again, the `-dbg` option will show which runs will be submitted and which ones will be omitted. It is good practice to use this option to control what is going on before submitting jobs to the cluster.

4.3 Collection phase

Once all the runs have finished, i.e. all the runs have a file in the `results/` folder, running the experiment again will **have a completely different behavior**. It will enter in the **collect and synthesize phase**. This phase can also be accessed even if not all runs have finished with the special argument `-collect`.

```
$ ./sana -mode exp -experiment experiment -collect
```

If the `-collect` argument is used and not all runs have fininshed, runs that have not finished will appear with a score of -1 in the `experimentPlainResults.txt` file. In the other output files, `experimentHumanReadableResults.txt` and `experimentForPlotResults.csv`, scores of -1 will be ignored when computing the average score for each method and network pair. Thus, if there is at least one score for each method and network pair combination, the program will finish successfully and all the output files will be correct. However, if none of the runs for a given method and network pair combination finished, the program will crash.

5 Conclusions

Practical tips and things to watch out for:

- When creating new experiments, existing experiments are very useful as reference.
- Always use the `-dbg` option before submitting jobs to the cluster.
- Use the `errs/`, `outs/`, `scripts/`, and `results/` folders to verify the behavior of the experiment.

- To repeat a run, delete its corresponding file from the `results/` folder and rerun the experiment. To restart the whole experiment, delete the `results/` folder.
- Remember that the common arguments in the experiment file overwrite the method-specific arguments. The `-o` argument is always overriden. Do not add incoherent arguments such as `-g1` as part of the common arguments.
- Even if only one run for each method and network pair combination is required, it is sensible to do two or three (if the computational resources are available) in case the first run is stuck or interrupted. As long as one of the runs finishes, you will be able to carry on with the collection phase. If none of the runs finishes, it might be a problem with the method (some methods run out of RAM with big networks).
- Remember that to run experiments using β as objective function, all the corresponding entries in the `topologySequenceScoreTable.cnf` file must exist. This is not something particular to the Experiment Module, it is also required for normal runs.
- The collection phase loads and evaluates all the alignments from zero. Thus, if it is wished to evaluate a newly added measure, or if one of the measures being evaluated changes somehow, it is not necessary to rerun the methods, it suffices to add it to the experiment file and rerun the collection phase.
- When measures are evaluated, they are constructed directly from the code in the `loadMeasure` function in the `Experiment` class, **ignoring all the common and method-specific arguments**. For example, the Node Density measure is created as follows:

  ```
  new NodeDensity(G1, G2, {0.1, 0.25, 0.5, 0.15});
  ```

  Thus, if for example there was a method in `methods.cnf` that optimizes Node Density with a different set of weights, the score obtained by the method and the score evaluated during the collect phase would be different. This can lead to bugs and I wished to fix it, however it is not simple to change. Hence, for now, a warning message is printed each time some argument could have affected a measure being evaluated.

For questions and doubts, contact Nil Mamano at `nil.mamano@gmail.com`.