Design Description of the ICN Evaluator (ICE)

Mays AL-Naday

2018-03-13

Overview

ICE is an open source R package that enables modelling ICN, *-over-ICN and IP/DNS based networks, particularly in terms of DNS-based nearest replica assignment. ICE is devided into atomic functions and wrapper functions, where the latter connects a set of functions to produces a larger outcome. ICE store input data and modelling results, and reuse them whenever needed. This offers some advantages as: it minimizes the data generation and allows for shorter processing time in subsequent runs, eaiser reproducability of previously run scenarios and reusability of the same data in different functions/runs. Each object in ICE corresponds to a single permutation of multiple configuration arguments (in config.yml), and within it contains number of randomised generations/results equal to `tests` or `sim.tests` arguments, respectively.

Here, we will go through the naming system and directories of ICE, the object structure and the main desgin choices. As described in the Get started with ICE vignette, ICE workflow is split into four distinct phases:

- `Parsing`
- `Input Data Generation`
- `Simulation`
- `Processing`

File Naming and Hierarchy

File Naming

ICE uses a unified file naming convension that is a hybrid of 'Hyphenated' and 'under_score', derived from the attributes of an object and other attributes that affects the object, as follows:

- first letter(s) before the first hyphen is the object name
- hyphens separate sections of the name (regular expressions), each of which represent an argument or a subsection of the attributes
- between two consecutive hyphens, under scores separates the label of an attribute and its setting.

This naming convension allow for easy and persistent description of the object meta-data.

File Hierarchy

ICE generates input data automatically when the `GenSimData` function is called, as described in Get started with ICE vignette. All files and subdirectories of the generated data are placed under the directory specified by the `store` argument in **config.yml**:

```
default:
  dirs:
    store: "data/"
```

During parsing phase and depending on the configurations under the `catalogue` section, permutations of catalogues are generated. Each permutation represent a catalogue of its own, for which a subdirectory under `data/` is created, to encompasse all the input data related to the catalogue.

Consider the following example:

```
default:
  catalogue:
    size: [500, 1000]
    v: [0.02,0.04, 0.06]
    d: "zipf"
    d.param:
      s: [0.85]
      ig: 0.0
```

Two permutations of catalogues will be created here, one for catalogue of 500 items, with a directory named `N_500-d_zipf-params_s_0.85_ig_0-v_0.02_0.04_0.06`; and the other of 1000 items with a directory named `N_1000-d_zipf-params_s_0.85_ig_0-v_0.02_0.04_0.06`. Notice how the name follows the convension descried in Section File Naming

Following on, input data that is dependent on the catalogue is placed in the directory of the respective catalogue. Those include:

- catalogue original data: an object of name `cc` that lists all the items' objects but with empty vectors of publishers/subscribers
- origins: each origin object is a single permutation, derived in a similar way to the permutations of catalogues
- surrogates: each surrogate object is a single permutation, derived in a similar way to the permutations of catalogues
- DNS maps: each DNS map corresponds to a single permutation
- Edge to Origin maps: also known as surrogate to origin maps

Object structure

Catalogue

catalogue object `cc.RData` is a list of length equals the `size` specification of the catalogue. Each element in the list is an item object, that is a list of:

- `i` the rank of the object, also act as the identifier
- `s` subscribers of the item (`NULL`)
- `p` publishers of the item (`NULL`)
- `v` bitrate and/or storage volume of the item
- `f` popularity of the item
- `l` publishers with locally cached copy

Origins

For a single catalogue, e.g. `N_500-d_zipf-params_s_0.85_ig_0-v_0.02_0.04_0.06`, multiple origins permutations might be generated; depending on the number of graphs in **config.yml** and the settings of the parameters in the `origins` section.

For example `origins` section in **config.yml**:

```
default:
  graph:
    gdir: "data-raw/zoo"
    gset: [1]
    augument: NULL
  origin:
    opt: [1]
```

```
    k: [2]
    a: 0.1
```

This will results in generating one `O` object stored in a filename `O-g_Abilene-opt_1-k_2-a_0.1-tests_50.RData`. Another object `O` of origins publishing `1000` items will also be generated and stored under the same name, but in the directory of the `1000` items catalogue, `N_1000-d_zipf-params_s_0.85_ig_0-v_0.02_0.04_0.06`

Each `O` is a 2-level list, first level corresponds is of length equal to the settig of `tests` in **config.yml**, each instance represent an independent test and contains a list of origins objects selected in this test. Each origin object is a list of:

- `p` node label
- `v` node storage capacity
- `i` items published by `p`

Surrogates

Similar to Origins, a surrogate permutation is created that co-exist with a single permutation of origins. For example:

```
default:
  graph:
    gdir: "data-raw/zoo"
    gset: [1]
    augument: NULL
  origin:
    opt: [1]
    k: [2]
    a: 0.1
  surrogate:
    opt: [1]
    k: [2,4]
    a: 0.1
```

will result in set of two permutations: `E-g_Abilene-a_0.1-k_2-opt_1-o_opt_1-k_2-a_0.1-tests_50` and `E-g_Abilene-a_0.1-k_4-opt_1-o_opt_1-k_2-a_0.1-tests_50`. The structure of surrogates is the same as that of origins.

Publishers (Origins + Surrogates)

Once both origins and surrogates are generated, a merge of the two is created that corresponds to `publishers` in the network. Following the examples above, these are:

- `P-g_Abilene-o_opt_1-k_2-a_0.1-e_a_0.1-k_2-opt_1-tests_50`, and
- `P-g_Abilene-o_opt_1-k_2-a_0.1-e_a_0.1-k_4-opt_1-tests_50`.

Subscribers

subs permutations are much like the publishers, they are specific for the network. The list however has an additional outer level that corresponds to the specific load instance on the network.

Pub/Sub state

Pub/Sub state is pretty much derived from the catalogue object `cc.RData`, but they populate the `s` and `p` parameters with the set of publishers and subscribers, depending on the permutations involved.