Rigel: Fast and Scalable Analysis of Massive Social Graphs

Implementation

Rigel is split into two phases - the embedding phase, which is written in C++, and the querying phase, which is written in Python.

Rigel's embedding phase is entirely self-contained, using only the C++ Standard Template Library. It has no dependencies outsides of the STL. We have compiled it successfully using G++ 4.1.2.

Rigel's querying phase is written with Python 2.7+ and has no dependencies outside of the Python standard library.

Basic Usage

Below is a sample snippet of how to use Rigel. `###` indicates omitted filenames for brevity. Please see below for details on what each command does.

Embedding phase example

```
// Landmark embedding
$ ./rigel -b 16 -e -1 -i 16 -L 100 -l ### -o ### -r ### -t ### -u 1000 -x 10 -y -1

// Non-landmark embedding
$ ./rigel -b 16 -e -1 -i 16 -L 100 -l ### -o ### -r ### -t ### -u 1000 -x 10 -y 0
```

Querying phase example

```
$ python queryrigel.py ### 1 -1
Enter ID of 2 nodes: 10 5
Rigel estimates the distance between 10 and 5 to be 5.587465.

Enter ID of 2 nodes: 6 93
Rigel estimates the distance between 6 and 93 to be 6.433285.

Enter ID of 2 nodes: 1043 9713
Rigel estimates the distance between 1043 and 9713 to be 5.909692.

Enter ID of 2 nodes: exit
```

Embedding Phase (C++)

Compiling

Because Rigel is entirely self-contained, only CMake is necessary to build.

```
$ make
```

Usage Details

Rigel is designed to be used as a command line tool. It is called from the command line and fed the arguments listed and detailed below. Please note that Rigel expects the nodes of the graph to be labeled from 0 - (N - 1), where N is the number of nodes in the graph.

Rigel is called in two steps. The first step is the initial bootstrapping phase where we embed our landmark nodes into the system. A separate Rigel call will be made once that step is done which will embed the rest of the nodes against these landmarks.

An example call to Rigel can be seen below. `###` are omitted filenames for brevity.

```
// Landmark embedding
```

```
$ ./rigel -b 16 -e -1 -i 16 -L 100 -l ### -o ### -r ### -t ### -u 1000 -x 10 -y -1

// Non-landmark embedding
$ ./rigel -b 16 -e -1 -i 16 -L 100 -l ### -o ### -r ### -t ### -u 1000 -x 10 -y 0
```

The first command will embed the landmarks of a graph with 1000 nodes into a 10 dimensional hyperbolic space with curvature 0 (making it equivalent to Euclidean space). It uses 100 landmarks and aligns each non-landmark against 16 of these landmarks. It is run serially.

The second command will embed the non-landmarks serially.

Rigel can be run serially or in parallel. If run in parallel, certain files must be generated per thread/machine (details below). At a high level, the non-landmark nodes of the graph will be partitioned into "T" close to equal parts, where T is the number of threads that will run. Each thread when calling Rigel will pass in an integer (the `-y` flag) indicating which partition it will embed (indexed starting from 0).

Input Details

-b (integer)

The number of landmarks against which a non-landmark node will be aligned. For example, if we pass in the integer 10, each non-landmark node will align itself to 10 randomly selected landmarks.

-e (integer)

The curvature of the hyperbolic space (negative integer).

-i (integer)

The number of "primary" landmarks. The "primary" landmarks is a subset of the landmarks that are the first to be embedded. The "secondary" landmarks (the rest of the landmarks) will be embedded against this set of primaries. Default value 16.

-L (integer)

The parameter following this flag is an integer, indicating the number of landmarks used. If the flag and parameter are not provided, the default value (100) will be used.

-l (string)

The parameter following this flag is the pathname of the distance matrix. The distance matrix is an L x N matrix, where L is the number of landmarks and N is the number of nodes. It should be stored in plain text, and have L lines. Each line has N values. For the i-th line, the j-th value indicates the distance between the i-th landmark and the node with ID j.

-o (string)

The prefix of the output files to be generated. When embedding landmarks, the emitted output files will be suffixed with `.land` and `.time`. When embedding non-landmarks the emitted output files will be suffixed with `.land` and `.coord`.

-r (string)

The filename of the file that contains a single column list of the IDs of the landmark nodes. The first "i" (where "i" is the integer passed into the -i flag) id's of this file will be the primary landmarks.

-t (string)

The prefix of two files that are suffixed by `.num` and `.ord`. The `.num` file contains one line: the "partition id" followed by the number of nodes. If you are running Rigel serially, the file would look like:

```
0 [# of nodes]
```

If you are running Rigel in parallel, you will have to assign each thread/machine a unique integer.

The `.ord` file contains a single-column list of the id's of every non-landmark node. The full suffix is actually `[partition id].ord`. If you are running Rigel serially, the `.ord` file should actually be suffixed by `0.ord`. If you are running Rigel in parallel, the suffix integer should match the first integer indicated in the `.num` file.

-u (integer)

The number of nodes in the graph.

-x (integer)

The dimension of the coordinate space.

-y (integer)

If set to -1, Rigel will embed landmarks. Otherwise, it should be set to a positive integer indicating which partition of the graph nodes it will embed (0-indexed).

Output Details

.land

The .land file contains the coordinates of the landmarks. The file will have L lines, where L is the number of landmarks used. The first column of each row is the ID of the landmark (labeled from 0 - (N-1)). The following columns are the coordinates for that node.

.coord

The .coord file is formatted similarly to the .land file, but instead of storing landmark coordinates it stores non-landmark coordinates.

.time

The .time file contains the time it took the system to embed.

Embedding API (C++)

To have the embedding process automated for you, please call the `main()` function in `rigel.cc/h`, passing it the appropriate parameters described above using an `argv` array (don't forget the corresponding `argc`).

`node.cc/h`

The `node` header and implementation file define a class `Node` which is a wrapper around a 1D `double` array. It has one constructor which initializes the array to a size defined by the global variable `DIMENSIONS`, found in `rigel.cc/h` and fills the array with zeroes. The array is `public` and can be accessed as a member variable under the name `vec`.

It has one method called `Refresh` which takes no parameters - calling it will "refresh" the underlying array to all zeroes.

`rigel.cc/h`

`double linear_dist(double *a, double *b)`

Given two `double` arrays it will return the associated Euclidean distance, treating each index value as a coordinate.

`double pb_dist(double *a, double *b)`

Given two `double` arrays it will return the associated hyperbolic distance, treating each index value as a coordinate.

`void simplex_downhill1(double **simplex, double *values, int d, double ftol, double (*obj)(double *, int, int), int *num_eval, int stuff)`

Responsible for global optimization of the initial embedding for the primary landmarks (the integer passed into the flag `-i`). The `values` array indicates the current difference between the estimated distances to the other landmarks and the true distance. `d` indicates the dimension of the system. `ftol` indicates the tolerance of the estimation versus the ground truth. `obj` is the function against which you are trying to optimize. `stuff` is the node ID which you are trying to optimize.

`void simplex_downhill2(double **simplex, double *values, int d, double ftol, double (*obj)(double *, int, int), int *num_eval, int stuff)`

Responsible for local optimization of all non-primary landmark nodes.

`double fit_p1(double *array, int num, int backup)`

This function computes the error between the estimated distance and the ground truth when performing global optimization for the primary landmarks. `array` is the coordinates of the particular landmark. `num` is the product of the integer passed into the `-i` parameter and the dimension of the system.

`double fit_p5(double *array, int num, int test_node)`

Computes the error between the estimated distance and the ground truth for the non-primary landmark nodes.

Querying Phase (Python 2.7+)

The querying API can be used as a command line tool or called from a program written/can interface with Python.

Command Line

```
$ python queryrigel.py [coordinate file prefix] [# of partitions] [curvature]
```

The script will then load in the `.land` and `.coord` files and prompt for two node IDs. Upon receiving two node IDs it will print out the estimated distance.

Programmatic API

The API consists of one simple function, `query()`, whose signature looks like this:

```
def query(coordinates, source, destination, curvature)
```

`coordinates` is expected to be a `dict` that maps an integer node ID to a `list` of coordinates. `source` and `destination` are node IDs, and `curvature` is the curvature of the hyperbolic space.

Publications

Xiaohan Zhao, Alessandra Sala, Haitao Zheng, Ben Y. Zhao. Efficient Shortest Paths on Massive Social Graphs. Proceedings of the The 7th International Conference on Collaborative Computing: Networking, Applications, and Worksharing (CollaborateCom) (Invited Paper). Orlando, FL, Oct 2011.

Xiaohan Zhao, Alessandra Sala, Haitao Zheng, Ben Y. Zhao. Fast and Scalable Analysis of Massive Social Graphs. In arXiv preprint arXiv:1107.5114.

Xiaohan Zhao, Alessandra Sala, Christo Wilson, Haitao Zheng, Ben Y. Zhao. Orion: Shortest Path Estimation for Large Social Graphs. Proceedings of The 3rd Workshop on Online Social Networks (WOSN). Boston, MA, June 2010.