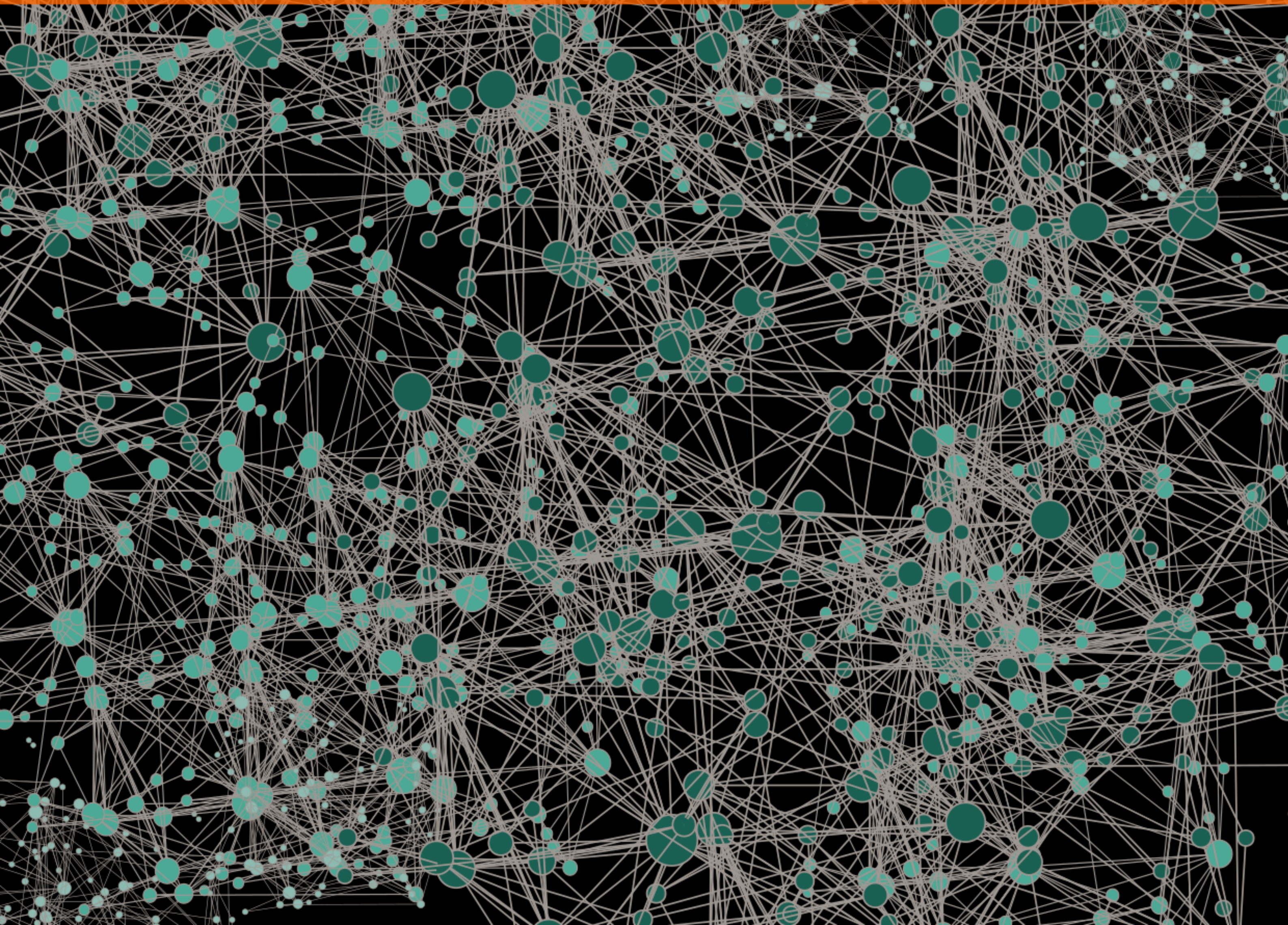
LinkPred

A High Performance Library for Link Prediction in Complex Networks

Said Kerrache



Contents

1. Introduction

The problem of determining the likelihood of existence of a link between two nodes in a network is called *link prediction*. Such prediction is made possible thanks to the existence of a topological structure in most real life networks. In other words, the topologies of networked systems such as the World Wide Web, the Internet, metabolic networks and human society are far from random, which implies that partial observations of these networks can be used to infer information about undiscovered interactions. Significant research efforts have been invested into the development of link prediction algorithms, and some researchers have made the implementation of their methods available to the research community. However, these implementations are often written in different languages and use different modalities of interaction with the user, which hinders their effective use. LinkPred is a high performance parallel and distributed link prediction library that includes the implementation of the major link prediction algorithms available in the literature by development from scratch and wrapping or translating existing implementations. The library offers a unified interface that facilitates the use and comparison of link prediction algorithms by researchers as well as practitioners.

1.1 Design principles

LinkPred is designed with the following guiding principles:

- Ease of use: LinkPred borrows heavily from the STL design and aims at offering an elegant and powerful interface. C++ users with minimum experience using STL will find the interface of LinkPred to be very familiar. Furthermore, the use of templates allows for greater flexibility when using LinkPred and allows for its integration within a variety of contexts.
- Extensibility: LinkPred was not only designed for practitioners of link prediction, but also fo researchers in the field. The library is designed in a way that allows developers of new link prediction algorithms to easily integrate their code into the library and take advantage of the existing functionalities such network data structures and performance evaluation algorithms.

- Efficiency: the data structures used and implemented in LinkPred are all chosen and designed to achieve the best possible performance. Additionally, most code in LinkPred is parallelized using OpenMp, which allows to take advantage of shared memory architectures. Furthermore, a significant portion of the predictors support distributed processing using MPI allowing the library to handle very large networks (hundreds of thousands to millions of nodes).

1.2 Functionalities

LinkPred provides the following functionalities:

- Basic data structures to efficiently store and access network data.
- Basic graph algorithms such graph traversal and shortest path algorithms.
- Implementation of several topological similarity index predictors, for example: common neighbors, Adamic-Adard index and Jackard index among other predictors (a full list is available in the library documentation).
- Implementation of several state-of-the-art link predictors, such as SBM, HRG, FBM and KAB (a full list is available in the library documentation).
- Test data generation from ground truth networks.
- Performance evaluation functionalities.

1.3 Requirements

The following softwares are used by LinkPred:

- A C++14 compliant compiler (required). Note that strict compliance with the standard C++14 is enforced during compilation and that C+11 compliance is not enough to build the library.
- The GNU Scientific Library (GSL) (required). LinkPred was tested with the version 2.1, but earlier versions might work as well.
- OpenMP (optional, default on): LinkPred works with OpenMP 3.0 or higher as it uses loop parallelization for STL iterators.

R Unfortunately, Visual C++ only supports OpenMp 2.0. LinkPred can still be compiled by disabling OpenMP, bu parallelism cannot be used when compiling with Visual C++.

- MPI (optional, default on): To take advantage of distributed architectures, several predictors as well as performance evaluation routines can run distributively using MPI. Although optional, it is strongly recommended
- Intel Math Kernel Library (MKL) (optional, default off): LinkPred was tested with the version 2016, but earlier versions might work as well. The MKL library is used by some prediction algorithms that incorporate linear algebra calculations. This library is nonetheless optional, since LinkPred offers replacements of the required methods. The replacement code is, however, a naive one and may result in significant loss of performance in the said algorithms.

1.4 Installation

LinkPred is distributed as source code that can be used to build the library using CMake. In the default setting, the building process is as follows:

1. Create a build directory in the root of the LinkPred directory:

   ```
   $ mkdir build
   ```

2. Configure the library:

   ```
   $ cd build
   $ cmake ../
   ```

   R Build options can be set by editing the file `CMakeLists.txt` or through the user interface if GUI CMake is used.

3. Build the library:

   ```
   $ make
   ```

4. Build documentation (optional): this step requires Doxygen and generates documentation in HTML and Latex:

   ```
   $ make doc
   ```

5. If you want to install the library:

   ```
   $ make install
   ```

   To install the library system-wide, you may need root privilege:

   ```
   $ sudo make install
   ```

   If you prefer a local install instead (which is usually the case when working on institution-wide HPC clusters/supercomputers), you need to set the install directory in the configuration step (Step 2 above):

   ```
   $ cmake -DCMAKE_INSTALL_PREFIX=YOUR_PATH ../
   ```

The `examples` directory contains sample code that can be used as a start point for using the library.

1.5 Quick start

The following example shows how to use the Common Neighbors link predictor to predict links in a network that is loaded from file. The network file must have the following format (one edge per line):

```
1       2
2       4
2       8
2       14
3       2
3       4
3       8
3       9
```

Here, we consider two scenarios: In the first one, we would like to compute the score for all non-existing links, whereas in the second we want to find out the k top links (those more likely to be missing).

To compute the score of all non-existing links, proceed as follows. In a file named `cne.cpp`, type the following code[1]:

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  auto net = UNetwork<>::read("Infectious.edges");
  UCNEPredictor<> predictor(net);
  predictor.init();
  predictor.learn();
  std::cout << "#Start\tEnd\tScore\n";
  for (auto it=net->nonEdgesBegin();it!=net->nonEdgesEnd();++it){
    auto i = net->getLabel(net->start(*it));
    auto j = net->getLabel(net->end(*it));
    double sc = predictor.score(*it);
    std::cout << i << "\t" << j << "\t" << sc << std::endl;
  }
  return 0;
}
```

R In this code, the predictor is instantiated with default template parameters. You may use non-default parameters if needed.

Compile your code. For example, if your compiled LinkPred with MPI an OpenMP enabled:

```
$ mpiCC cne.cpp -o cne -fopenmp -lLinkPred
```

R If you face any dialect-related complaints from the compiler, you may need to add the option: `-std=c++14`. Also, depending on the LinkPred functionalities used in your code, you may need to additionally link against the MKL library (using `-lmkl_rt`) and/or gsl (using `-lgsl -lgslcblas`).

If you built LinkPred without MPI and OpenMP, compile as follows:

```
$ g++ cne.cpp -o cne -lLinkPred
```

Run your code:

```
$ ./cne
```

R Make sure that the library is located in the load path. Under Linux, you may need to set the environment variable `LD_LIBRARY_PATH`. In the case of a default system-wide install, LinkPred will be installed to the default directory, which is already in the load path. You may, however, need to refresh the ld cache by running:

[1]This code is available in the examples directory.

```
$ sudo ldconfig
```

To get the top *k* links, type the following code[2] in a file named `cnetop.cpp`:

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  std::size_t k = 10;
  auto net = UNetwork<>::read("Infectious.edges");
  UCNEPredictor<> predictor(net);
  predictor.init();
  predictor.learn();
  std::vector<typename UNetwork<>::EdgeType> edges;
  edges.resize(k);
  std::vector<double> scores;
  scores.resize(k);
  k = predictor.top(k, edges.begin(), scores.begin());
  std::cout << "#Start\tEnd\tScore\n";
  for (std::size_t l = 0; l < k; l++) {
    auto i = net->getLabel(net->start(edges[l]));
    auto j = net->getLabel(net->end(edges[l]));
    std::cout << i << "\t" << j << "\t" << scores[l] <<std::endl;
  }
  return 0;
}
```

Compile the code and run it as previously shown.

1.6 Third-party software

LinkPred includes modified and/or translated versions of the following software sources:

- HRG code [2]: we used the implementation available at `http://tuvalu.santafe.edu/~aaronc/hierarchy/hrg_20120527_predictHRG_v1.0.4.zip`.
- SBM code [5]: we used the C code provided by the authors at `http://seeslab.info/media/filer_public/eb/ae/ebaee03f-a53a-430f-a4a1-6b713d36e91e/rgraph-2.0.1.tar.gz`.
- FBM code [10]: we translated the Matlab code provided by the authors into C++.
- HyperMap (HYP) code [12, 13]: we used the code provided by the authors at `http://www.cut.ac.cy/eecei/staff/f.papadopoulos/?languageId=2`.
- CG_DESCENT: a conjugate gradient method with guaranteed descent[6].
- *plfit*: a C++ implementation of Clauset, Shalizi and Newman [3] method for fitting power law distributions written by Tamas Nepusz. The code is available at `http://tuvalu.santafe.edu/~aaronc/powerlaws/`.

1.7 Data

Two small networks are included with the library and can be used with the example programs:

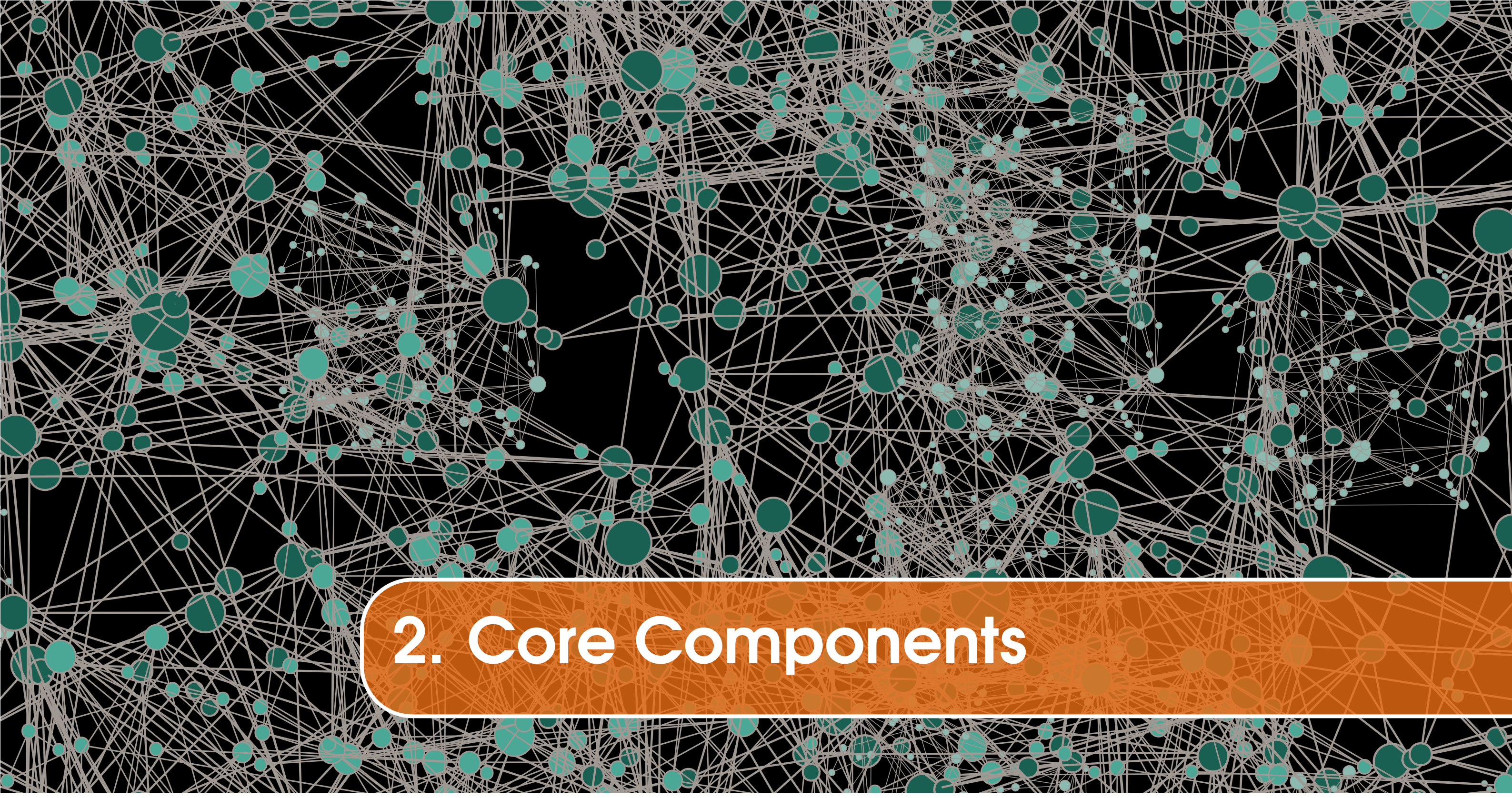
[2]This code is available in the examples directory.

- Zakaray's Karate Club[17] (file: `Zakarays_Karate_Club.edges`): A social network that represents friendships between members of a karate club at an American university. The data was collected in the 1970s by Wayne Zachary and is available at `http://konect.uni-koblenz.de/networks/ucidata-zachary`.
- Infectious[7] (file: `Infectious.edges`): Face-to-face interaction between visitors of the exhibition INFECTIOUS: STAY AWAY in 2009 at the Science Gallery in Dublin. A link indicates that a face-to-face interaction took place for more than 20 seconds. The dataset is available at `http://konect.uni-koblenz.de/networks/sociopatterns-infectious`.

More data can be found in the following public data repositories [1, 8, 9, 14, 15, 18, 19].

1.8 Citation

If you use LinkPred in your research, kindly cite the references of the algorithms you used and cite LinkPred as: Said Kerrache. "LinkPred: A High Performance Library for Link Prediction in Complex Networks". In: Submitted (2019).



2. Core Components

This chapter is concerned with the basic building blocks of LinkPred. Some of these components, for instance the network data structures, are essential for an optimal use of the library. Other components can be very useful for building new efficient link prediction algorithms. For a first reading, we invite the reader to study Section 2.1 and come back for the remaining sections at a later time or when necessary.

2.1 The undirected network data structure

At the heart of LinkPred lies the class `UNetwork`, which represents an undirected network. This is a data structure designed to efficiently represent immutable graphs (graphs that once created are not modified). It offers efficient access to nodes, edges and non-existing edges as well.

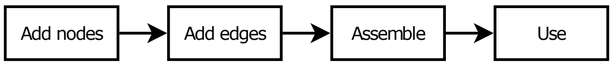


Figure 2.1: Example network.

The life cycle of a network has two distinct phases:

- **Pre-assembly**: In this phase, it is possible to add nodes and edges to the network. It is also possible to access nodes and translate external labels to internal IDs and vice versa. However, most functionalities related to accessing edges are not yet available. As a result, the network at this stage is practically unusable. To be able to use the network, it is first necessary to assemble it.
- **Post-assembly**: Once assembled, no new nodes or edges can be added (or removed) to the network. The network is now fully functional and can be passed as argument to any method that requires so.

R Attempting to add new nodes or edges after assembling the network produces an exception. On the other hand, due to performance considerations, no such checks are made in methods that prerequire assembly. Therefore, using a network before assembling it may result in unspecified behavior.

2.1.1 Building the network

To build a network, we first create an empty network, named for instance `net`, by calling the default constructor:

```
UNetwork<> net;
```

Most classes in LinkPred manipulate networks through smart pointers for efficient memory management. To create a shared pointer to a `UNetwork` object:

```
auto net = std::make_shared<UNetwork<>>();
```

Notice that the class `UNetwork` is a class template, which is here instantiated with the default template arguments. In this default setting, the labels are of type `std::string`, whereas internal IDs are of type **unsigned int**, but `UNetwork` can be instantiated with a number of other data types if wanted. For instance, the labels can be of type **unsigned int**, which may reduce storage size in some situations.

Adding nodes is achieved by calling the method `addNode`, which takes as parameter the node label and returns an `std::pair` containing, respectively, the node ID and a Boolean which is set to true if the node is newly inserted, false if the node already exists. The nodes IDs are guaranteed to be contiguous in $0, \dots, n-1$, where n is the number of nodes. Inserting a node that already exists has no effect.

```
auto res = net.addNode(label);
auto id  = res.first; // This the node ID
bool inserted = res.second; // Was the node inserted or did it
    already exist?
```

The method `addEdge` is used to create an edge between two nodes specified by their IDs (**not their labels**):

```
net.addEdge(i, j);
```

A possible and shorter way to create edges without the need for adding nodes beforehand or storing externally their IDs is as follows[1]:

```
net.addEdge(net.addNode(labelI).first, net.addNode(labelJ).first)
    ;
```

Loops are not allowed, and attempting to add one results in an exception. Adding the same edge more than one time, including the case where both an edge (i, j) and its inverse (j, i) are inserted, has no effect.

The last step in building the network is to assemble it:

```
net.assemble();
```

The method `assemble` initializes the internal data structures and makes the network ready to be used.

The class `UNetwork` offers also a static method that reads the network data from file:

[1]Notice that the ID assigned to a node depends on the order in which this node is added to the network. Therefore, depending on the order in which function arguments are processed (which is implementation-dependent), the nodes may be assigned different internal IDs when using this code. This, however, has no effect whatsoever on the results.

```
std::string fileName = "Infectious.edges";
auto net = UNetwork<>::read(fileName);
```

The file must be in text format with each line specifying an edge. No comments are allowed in the file. An example input file is the following:

```
1 2
1 3
2 4
3 5
2 6
```

■ **Example 2.1** Consider the network shown in Figure 2.2.

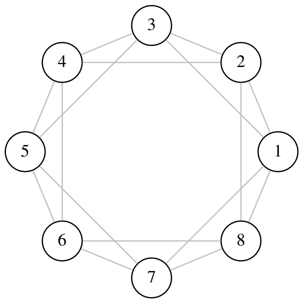


Figure 2.2: Example network.

The code below shows how to build this network:

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  int n = 8;
  UNetwork<unsigned int> net; // Labels are of type unsigned int
  std::cout << "Label\tID\tNew?" << std::endl;
  for (int i = 1; i <= n; i++) {
    auto res = net.addNode(i);
    std::cout << i << "\t" << res.first << "\t" << res.second <<
       std::endl;
  }
  for (int i = 1; i <= n; i++) {
    net.addEdge(net.getID(i), net.getID(i % n + 1));
    net.addEdge(net.getID(i), net.getID((i + 1) % n + 1));
  }
  net.assemble();
  std::cout << "Printing␣network:" << std::endl;
  net.print();
  return 0;
}
```

This is the output of this code:

```
Label   ID      New?
1       0       1
2       1       1
3       2       1
```

```
4       3       1
5       4       1
6       5       1
7       6       1
8       7       1
Printing network:
1       2
1       3
1       7
1       8
2       3
2       4
2       8
3       4
3       5
4       5
4       6
5       6
5       7
6       7
6       8
7       8
```

The following is another version of the code that builds the same network:

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  int n = 8;
  UNetwork<unsigned int> net;
  for (int i = 1; i <= n; i++) {
    net.addEdge(net.addNode(i).first, net.addNode(i % n + 1).
        first);
    net.addEdge(net.addNode(i).first, net.addNode((i + 1) % n +
        1).first);
  }
  net.assemble();
  std::cout << "Printing␣network:" << std::endl;
  net.print();
  return 0;
}
```

■

2.1.2 Accessing nodes

Nodes can be accessed through iterators provided by `nodesBegin()` and `nodesEnd()`. The order of iteration is that of internal IDs, which is also the order of insertion. For convenience, the iterator points to a pair, the first element of which is the internal ID, whereas the second is the external label.

```
std::cout << "ID\tLabel" << std::endl;
for (auto it = net.nodesBegin(); it != net.nodesEnd(); ++it) {
  std::cout << it->first << "\t" << it->second << std::endl;
}
```

Alternatively, one can iterate over labels (in increasing order) in a similar way using the iterators `labelsBegin()` and `labelsEnd()`:

```
std::cout << "Label\tID" << std::endl;
for (auto it = net.labelsBegin(); it != net.labelsEnd(); ++it) {
  std::cout << it->first << "\t" << it->second << std::endl;
}
```

It is also possible to translate labels to IDs and vice versa using `getID(label)` and `getLabel(id)` respectively.

Oftentimes, one would want to iterate over a random sample of nodes instead of the whole set. This can be easily done using the two methods:

```
RndNodeIterator rndNodesBegin(double ratio, long int seed) const
RndNodeIterator rndNodesEnd() const
```

The method `rndNodesBegin` takes two parameters: the ratio of nodes contained in the sample (must be in $[0,1]$) and a seed for the random number generator. For example, the following for loop iterates over about half the nodes and skips the other half. The nodes are accessed in increasing order of their IDs:

```
double ratio = 0.5;
long int seed = 777;
std::cout << "ID\tLabel" << std::endl;
for (auto it = net.rndNodesBegin(ratio, seed); it != net.
    rndNodesEnd(); ++it) {
  std::cout << it->first << "\t" << it->second << std::endl;
}
```

Notice that `ratio` specifies the probability that a node gets selected. Because of the random nature of the selection process, the actual number of nodes selected may be different from `ratio` $\times n$.

The methods above can be used to access nodes data even before the networks is assembled. After assembling the network, more functionalities become available. For instance, it is possible to access nodes degrees:

```
std::cout << "ID\tDegree" << std::endl;
for (auto it = net.nodesDegBegin(); it != net.nodesDegEnd(); ++it
    ) {
  std::cout << it->first << "\t" << it->second << std::endl;
}
```

The iterator returned by `nodesDegBegin()` points to a pair where the first element is the node ID and the second element is its degree. It is also possible to obtain the degree of a given node using `getDeg(id)`:

```
std::cout << "ID\tDegree" << std::endl;
for (auto it = net.nodesBegin(); it != net.nodesEnd(); ++it) {
  std::cout << it->first << "\t" << net.getDeg(it->first) << std
      ::endl;
}
```

2.1.3 Accessing edges

Information on edges can only be accessed after assembling the network. One way to access edges is to iterate over all edges in the network. This can be done using the method `edgesBegin()` and `edgesEnd()`. Obtaining the start and end nodes of an edge is accomplished by means of the two **static** methods `start` and `end`:

```
std::cout << "Start\tEnd" << std::endl;
for (auto it = net.edgesBegin(); it != net.edgesEnd(); ++it) {
  std::cout << net.start(*it) << "\t" << net.end(*it) << std::
      endl;
}
```

As it is the case with nodes, it is possible to access a random sample of edges:

```
double ratio = 0.5;
long int seed = 777;
std::cout << "Start\tEnd" << std::endl;
for (auto it = net.rndEdgesBegin(ratio, seed); it != net.
    rndEdgesEnd(); ++it) {
  std::cout << net.start(*it) << "\t" << net.end(*it) << std::
      endl;
}
```

LinkPred offers the possibility to iterate over negative links in the same way one iterates over positive edges. This can be done using the method `nonEdgesBegin()` and `nonEdgesEnd()`:

```
std::cout << "Start\tEnd" << std::endl;
for (auto it = net.nonEdgesBegin(); it != net.nonEdgesEnd(); ++it
    ) {
  std::cout << net.start(*it) << "\t" << net.end(*it) << std::
      endl;
}
```

It is also possible to iterate over a randomly selected sample of negative links:

```
double ratio = 0.5;
long int seed = 777;
std::cout << "Start\tEnd" << std::endl;
for (auto it = net.rndNonEdgesBegin(ratio, seed); it != net.
    rndNonEdgesEnd(); ++it) {
  std::cout << net.start(*it) << "\t" << net.end(*it) << std::
      endl;
}
```

R Negative edges are not stored in memory for obvious performance reasons. As a result, instead of $O(1)$ in the case of positive edges iterators, the incrementation operator (`++`) for negative links iterators has a higher running time, which depends on the network density.

The neighbors of a given node can be accessed by means of the two methods `neighborsBegin(id)` and `net.neighborsEnd(id)`:

```
unsigned int i = 0;
std::cout << "Start\tEnd" << std::endl;
```

```
for (auto it = net.neighborsBegin(i); it != net.neighborsEnd(i);
    ++it) {
  std::cout << net.start(*it) << "\t" << net.end(*it) << std::
      endl;
}
```

Notice that the iterator points to the edges adjacent to the node and not directly to its neighbors. The neighbors are always located at the end of these edges, whereas the node passed to `neighborsBegin` is stored as the starting node.

■ **Example 2.2** Consider the network shown in Figure 2.3.

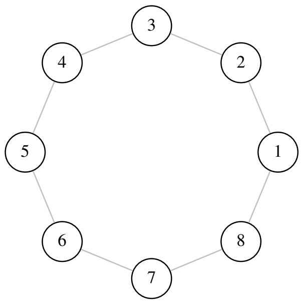


Figure 2.3: Example network.

The following code iterates over the neighbors of all nodes and a random sample of negative links:

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  int n = 8;
  UNetwork<unsigned int> net;
  for (int i = 1; i <= n; i++) {
    net.addEdge(net.addNode(i).first, net.addNode(i % n + 1).
        first);
  }
  net.assemble();

  std::cout << "Positive␣links:" << std::endl;
  std::cout << "Start\tEnd" << std::endl;
  for (auto it = net.nodesDegBegin(); it != net.nodesDegEnd(); ++
      it) {
    for (auto nit = net.neighborsBegin(it->first);
        nit != net.neighborsEnd(it->first); ++nit) {
      std::cout << net.getLabel(net.start(*nit)) << "\t"
          << net.getLabel(net.end(*nit)) << std::endl;
    }
  }

  std::cout << "Random␣negative␣links:" << std::endl;
  double ratio = 0.2;
  long int seed = 777;
  std::cout << "Start\tEnd" << std::endl;
  for (auto it = net.rndNonEdgesBegin(ratio, seed);
        it != net.rndNonEdgesEnd(); ++it) {
      std::cout << net.getLabel(net.start(*it)) << "\t"
          << net.getLabel(net.end(*it)) << std::endl;
   }
   return 0;
}
```

The following is the output of this code:

```
Positive links:
Start   End
2       1
2       3
1       2
1       8
3       2
3       4
4       3
4       5
5       4
5       6
6       5
6       7
7       6
7       8
8       1
8       7
Random negative links:
Start   End
2       4
1       3
3       7
3       8
4       8
```

■

2.2 The directed network data structure

To represent directed networks, LinkPred offers the class `DNetwork`, which offers a very similar interface to `UNetwork`.

■ **Example 2.3** For example, the code below shows how to create the directed network shown in Figure 2.4 and iterate over the neighbors of all nodes as well as a random sample of negative links:

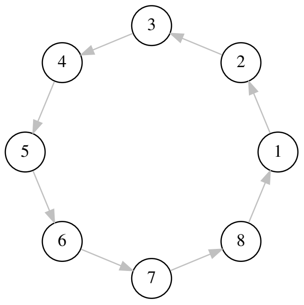


Figure 2.4: Example of a directed network.

```cpp
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  int n = 8;
  DNetwork<unsigned int> net;
  for (int i = 1; i <= n; i++) {
    net.addEdge(net.addNode(i).first, net.addNode(i % n + 1).
        first);
  }
  net.assemble();

  std::cout << "Positive␣links:" << std::endl;
  std::cout << "Start\tEnd" << std::endl;
  for (auto it = net.nodesDegBegin(); it != net.nodesDegEnd(); ++
      it) {
    for (auto nit = net.neighborsBegin(it->first);
        nit != net.neighborsEnd(it->first); ++nit) {
      std::cout << net.getLabel(net.start(*nit)) << "\t"
          << net.getLabel(net.end(*nit)) << std::endl;
    }
  }

  std::cout << "Random␣negative␣links:" << std::endl;
  double ratio = 0.2;
  long int seed = 777;
  std::cout << "Start\tEnd" << std::endl;
  for (auto it = net.rndNonEdgesBegin(ratio, seed);
      it != net.rndNonEdgesEnd(); ++it) {
    std::cout << net.getLabel(net.start(*it)) << "\t"
        << net.getLabel(net.end(*it)) << std::endl;
  }
  return 0;
}
```

The following is the output of this code:

```
Positive links:
Start   End
2       3
1       2
3       4
4       5
5       6
```

```
6       7
7       8
8       1
Random negative links:
Start   End
2       1
2       8
3       2
3       1
3       7
5       2
5       8
7       6
8       2
```

■

2.3 Maps

Maps are a useful way to associate data to nodes or edges. Two types of maps are available in LinkPred: *node maps* (class `NodeMap`) and *edge maps* (class `EdgeMap`), both member of `UNetwork`. The first assigns data to the nodes of the network, whereas the latter maps data to edges.

Creating a node map is achieved by calling the method `createNodeMap` on the network object. This is a template method with the mapped data type as the only template argument. For example, to create a node map with data type **double** over the network `net`:

```
auto nodeMap = net.template createNodeMap<double>();
```

To obtain a smart pointer (`std::shared_ptr`) to a node map, the method `createNodeMapSP` must be called instead:

```
auto nodeMapSP = net.template createNodeMapSP<double>();
```

Creating an edge map can be done in a similar way:

```
auto edgeMap = net.template createEdgeMap<double>();
auto edgeMapSP = net.template createEdgeMapSP<double>();
```

Both `NodeMap` and `EdgeMap` offer the same interface, which in fact is similar to `std::map`. This includes the operator `[]`, the methods `at`, `begin`, `end`, `cbegin` and `cend`. From the performance point of view, `NodeMap` offers constant time access to mapped values, whereas `EdgeMap` requires logarithmic time access ($O(\log m)$, m being the number of edges).

■ **Example 2.4** The following code shows how to create and use node and edge maps:

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  int n = 8;
  UNetwork<> net;
  for (int i = 1; i <= n; i++) {
    net.addEdge(net.addNode(i).first, net.addNode(i % n + 1).
        first);
    net.addEdge(net.addNode(i).first, net.addNode((i + 1) % n +
        1).first);
  }
  net.assemble();

  int i = 0;
  auto nodeMap = net.template createNodeMap<double>();
  for (auto it = net.nodesBegin(); it != net.nodesEnd(); ++it) {
    nodeMap[it->first] = i++ / 2.0;
  }

  std::cout << "ID\tValue" << std::endl;
  for (auto it = net.nodesBegin(); it != net.nodesEnd(); ++it) {
    std::cout << it->second << "\t" << nodeMap.at(it->first) <<
        std::endl;
  }

  i = 0;
  auto edgeMap = net.template createEdgeMap<double>();
  for (auto it = net.edgesBegin(); it != net.edgesEnd(); ++it) {
    edgeMap[*it] = i++ / 2.0;
  }

  std::cout << "Start\tEnd\tValue" << std::endl;
  for (auto it = net.edgesBegin(); it != net.edgesEnd(); ++it) {
    std::cout << net.getLabel(net.start(*it)) << "\t"
        << net.getLabel(net.end(*it)) << "\t" << edgeMap.at(*it)
        << std::endl;
  }
  return 0;
}
```

The following is the output of this code:

```
ID      Value
2       0
1       0.5
3       1
4       1.5
5       2
6       2.5
7       3
8       3.5
Start   End     Value
2       1       0
2       3       0.5
2       4       1
2       8       1.5
1       3       2
1       7       2.5
1       8       3
3       4       3.5
3       5       4
4       5       4.5
4       6       5
```

```
5       6       5.5
5       7       6
6       7       6.5
6       8       7
7       8       7.5
```

■

2.3.1 Sparse maps

If a node map is sparse, that is, has non-default values only on a small subset of the elements, it is better to use sparse node and edge maps. To create a sparse node map:

```
auto nodeSMap = net.template createNodeSMap<double>(0.0);
```

Notice that the methods takes as input one parameter that specifies the default value of the map (in this case, it is 0.0). Hence, in this example any node which is not explicitly assigned a value is assumed to have the default value 0.0. To obtain a smart pointer (`std::shared_ptr`) to a sparse node map, the method `createNodeSMapSP` must be called instead:

```
auto nodeSMapSP = net.template createNodeSMapSP<double>(0.0);
```

Sparse maps use $O(\log(k))$ space and time to sore and access data, where k is the number of elements explicitly assigned elements (having non-default value).

R Any element that is explicitly assigned, even with the default value, is stored in memory. Hence, you should avoid explicit assignment with the default value as it unnecessarily increases the size of the map.

■ **Example 2.5** The following code shows how to create and use a sparse node map:

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  int n = 8;
  UNetwork<unsigned int> net;
  for (int i = 1; i <= n; i++) {
    net.addEdge(net.addNode(i).first, net.addNode(i % n + 1).
        first);
    net.addEdge(net.addNode(i).first, net.addNode((i + 1) % n +
        1).first);
  }
  net.assemble();

  auto nodeSMap = net.template createNodeSMap<double>(-1.0);
  nodeSMap[2] = 2.0;
  nodeSMap[3] = 3.0;

  std::cout << "ID\tValue" << std::endl;
  for (auto it = net.nodesBegin(); it != net.nodesEnd(); ++it) {
    std::cout << it->second << "\t" << nodeSMap.at(it->first) <<
        std::endl;
  }
  return 0;
}
```

The following is the output of this code:

```
ID      Value
2       -1
1       -1
3       2
4       3
5       -1
6       -1
7       -1
8       -1
```

■

2.4 Graph traversal

LinkPred provides two classes for graph traversal: `BFS`, for Breadth First traversal, and `DFS` for Depth First traversal. They both inherit from the abstract class `GraphTraversal`, which declares one virtual method `traverse`. It takes as parameter the source node, from where the traversal starts, and a reference to a `NodeProcessor` object which is in charge of processing nodes sequentially as they are visited.

```
/**
 * Abstract graph traversal.
 */
template<typename NetworkT = UNetwork<>, typename NodeProcessor =
    Collector<NetworkT>> class GraphTraversal {
protected:
  std::shared_ptr<NetworkT const> net;
public:
  GraphTraversal(std::shared_ptr<NetworkT const> net) : net(net)
      {
  }
...
  /**
   * Traverse the graph.
   * @param srcNode The source node.
   * @param processor The node processor.
   */
  virtual void traverse(typename NetworkT::NodeIdType srcNode,
      NodeProcessor & processor) = 0;
};
```

The class `NodeProcessor` is a template argument of `GraphTraversal` and is required to implement the method `bool process(typename NetworkT::NodeIdType const & i)`, which processes node *i* and returns true if the traversal must continue, false otherwise. Notice, however, that independently of the return value of `process`, only the nodes in the same connected component as the source node are visited by `BFS` and `DFS`.

The library offers two useful implementations of `NodeProcessor`: `Counter`, which simply counts the visited nodes, and `Collector`, which collects the visited nodes' IDs into a queue in the order of their visit. `Collector` is the default value for the template argument `NodeProcessor`. The two classes `Counter` and `Collector` are shown below.

```
/**
 * A class that counts nodes during traversal.
 */
template<typename NetworkT = UNetwork<>> class Counter {
protected:
  std::size_t count = 0;
public:
...
  /**
   * Node processing.
   */
  bool process(typename NetworkT::NodeIdType const & i) {
    count++;
    return true;
  }

  /**
   * @return The nodes count.
   */
  std::size_t getCount() const {
    return count;
  }

  /**
   * Reset the nodes count to 0.
   */
  void resetCount() {
    count = 0;
  }
};
```

```
/**
 * A class that collects nodes during traversal.
 */
template<typename NetworkT = UNetwork<>> class Collector {
protected:
  std::queue<typename NetworkT::NodeIdType> visited;
public:
...
  /**
   * Node processing.
   */
  bool process(typename NetworkT::NodeIdType const & i) {
    visited.push(i);
    return true;
  }

  /**
   * @return The visited nodes.
   */
  const std::queue<typename NetworkT::NodeIdType>& getVisited()
     const {
    return visited;
  }
};
```

■ **Example 2.6** Consider the network shown in Figure 2.5.

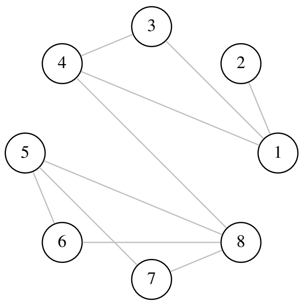


Figure 2.5: Example network.

The code below shows how to traverse this graph using BFS and DFS classes. For BFS, we collect the nodes, whereas for DFS we only count them.

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  auto net = UNetwork<>::read(std::string(argv[1]));
  // BFS
  BFS<> bfs(net);
  Collector<> col;
  bfs.traverse(net->getID(1), col);
  auto visited = col.getVisited();
  std::cout << "BFS:" << std::endl;
  while (!visited.empty()) {
    auto i = visited.front();
    visited.pop();
    std::cout << net->getLabel(i) << std::endl;
  }
  // DFS
  DFS<UNetwork<>, Counter<>> dfs(net);
  Counter<> counter;
  dfs.traverse(net->getID(1), counter);
  std::cout << "DFS␣visited␣" << counter.getCount() << "␣nodes"
     << std::endl;
  return 0;
}
```

Here is the output of this code:

```
BFS:
1
2
3
4
8
5
6
7
DFS visited 8 nodes
```

■

2.5 Shortest paths

The LinkPred library contains an implementation of Dijkstra's algorithm for solving the shortest path problem[2]. To use it, it is first necessary to define a length (or weight) map that specifies the length associated with every edge in the graph. A length map is simply a map over the set of edges which can take integer as well as double values. It can therefore be created using the template methods `createEdgeMap()` and `createEdgeMapSP()` defined in the class `UNetwork` (see Section 2.3). The method `createEdgeMap` returns an `EdgeMap` object, whereas `createEdgeMapSP` returns a smart pointer (`std::shared_ptr`) to an `EdgeMap` object. For example, in order to create a length map taking double values, one can proceed as follows:

```
auto length = net->template getEdgeMapSP<double>();
for (auto it = net->edgesBegin(); it != net->edgesEnd(); ++it) {
  (*length)[*it] = 1;
}
```

The next step is to create a `Dijkstra` object and register the length map:

```
Dijkstra<> dijkstra(net);
auto lengthMapId = dijkstra.registerLengthMap(length);
```

The identifier `lengthMapId` is used to uniquely identify the registered length map. It is possible to register multiple length maps with the same `Dijkstra` object, and once there is no more need for a given length map, it can be unregistered as follows:

```
dijkstra.unregisterLengthMap(lengthMapId);
```

The class `Dijkstra` offers two methods for computing distances:

1. `getShortestPath` : This method computes and returns the shortest path between two nodes and its length:

   ```
   auto res = dijkstra.getShortestPath(i, j, lengthMapId);
   auto path = res.first; // This is the shortest path
   auto dist = res.second; // This is its length
   ```

2. `getDist` : Computes the distance between a source node and all other nodes. The returned value is a node map, where each node is mapped to a pair containing the distance from the source node and the number of edges in the corresponding shortest path:

   ```
   auto distMap = dijkstra.getDist(i, lengthMapId);
   auto res = distMap->at(j);
   double dist = res.first; // Distance between i and j
   std::size_t nbHops =  res.second; // Number of hops along the
        shortest path
   ```

Both methods run Dijkstra's algorithm, except that `getShortestPath` stops once the destination node is reached, whereas `getDist` continues until all reachable nodes are visited. The distance and number of hops assigned to disconnected couples are passed as the last two arguments of these two methods. By default, they are assigned the values `std::numeric_limits<double>::infinity()` and `std::numeric_limits<std::size_t>::max()` respectively.

[2]The current implementation uses a binary heap instead of a Fibonacci heap. Consequently, the performance of the implementation is $O(nm + n^2 \log^2 m)$ instead of $O(nm + n^2 \log m)$.

■ **Example 2.7** Consider the network shown in Figure 2.6.

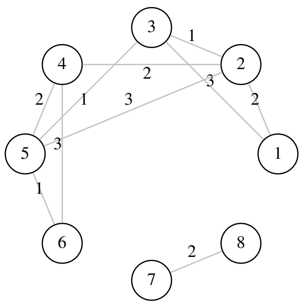


Figure 2.6: Example network with an associated length map.

In the following code, we first compute the shortest path from 1 to 6, then compute the shortest path distances from 1 to all other nodes.

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  auto net = UNetwork<>::read(std::string(argv[1]));
  auto length = net->template createEdgeMapSP<double>();
  int i = 1;
  for (auto it = net->edgesBegin(); it != net->edgesEnd(); ++it,
     i++) {
    (*length)[*it] = (13 * i) % 3 + 1;
  }
  Dijkstra<> dijkstra(net);
  auto lengthMapId = dijkstra.registerLengthMap(length);
  {
    auto res = dijkstra.getShortestPath(net->getID("1"), net->
       getID("6"),
        lengthMapId);
    auto path = res.first;
    auto dist = res.second;
    std::cout << "Path:␣";
    for (auto it = path->begin(); it != path->end(); ++it) {
      std::cout << net->getLabel(*it) << "␣";
    }
    std::cout << "\ndist:␣" << dist << std::endl;
  }
  {
    auto distMap = dijkstra.getDist(net->getID("1"), lengthMapId)
       ;
    std::cout << "dist:␣" << std::endl;
    for (auto it = net->nodesBegin(); it != net->nodesEnd(); ++it
       ) {
      auto res = distMap->at(it->first);
      std::cout << it->second << "␣:␣" << res.first << ",␣" <<
         res.second
          << std::endl;
    }
  }
  return 0;
}
```

Here is the output of this code:

```
Path: 1 2 4 6
dist: 5
dist:
1 : 0, 0
2 : 2, 1
3 : 3, 1
4 : 4, 2
5 : 4, 2
6 : 5, 3
7 : inf, 18446744073709551615
8 : inf, 18446744073709551615
```

■

2.5.1 Memory management

Computing shortest-path distances in large networks require not only considerable time but also significant space resources. Consequently, efficient management of memory is necessary to render the task feasible in such situations. The abstract class `NetDistCalculator` provides an interface for an additional layer over the class `Dijkstra` which facilitates its use and can serve to manage memory usage. A `NetDistCalculator` object is associated with a single length map and provides two methods for computing distances:

- `getDist(i, j)` : Computes and returns the distance between the two nodes *i* and *j*. The return value is an `std::pair` , with the first element being the distance, whereas the second is the number of hops in the shortest path joining the two nodes.
- `getDist(i)` : Computes and returns a node map containing the distances from node *i* to all other nodes in the network.

LinkPred includes two implementations of `NetDistCalculator` : `ESPDistCalculator` (exact shortest path distance calculator) and `ASPDistCalculator` (approximate shortest path distance calculator). In what follows, a description of the former is given, whereas the latter implementation is presented in the next section (2.5.2).

The class `ESPDistCalculator` implements the interface `NetDistCalculator` and returned the exact shortest path distances as computed by `Dijkstra` . Additionally, it caches the computed results for better performance. The constructor of `ESPDistCalculator` takes three parameters: a `Dijkstra` object, a length map and a third parameter of type `CacheLevel` , which is an enumeration of the available caching strategies:

- `NoCache` : The results computed by `Dijkstra` are discarded immediately after the call to the method has ended. This minimizes memory consumption, but is very inefficient from the time perspective. This strategy should only be used when memory is scarce and the couples for which the distance is to be computed are few in number and have a few or no nodes in common.
- `NodeCache` : In this strategy, the distances from a single node to all other nodes (a distance node map) are kept in cache and replaced in case of a cache miss. Moderate memory use is incurred from using this scheme, and if the couples between which the distances to be computed are grouped according to their

starting or ending node, time requirements are optimal in the sens that no results are wasted. Therefore, this strategy should be used with large network with the precaution of ordering couples as explained.

- `NetworkCache`: In this scheme, any computed distance map is kept in cache, which results in maximum memory consumption and minimal computation time. This should be with small to average-size networks.

■ **Example 2.8** In the following code, we compute the distance between all couples in the network of Figure 2.6.

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  auto net = UNetwork<>::read(std::string(argv[1]));
  auto length = net->template createEdgeMapSP<double>();
  int i = 1;
  for (auto it = net->edgesBegin(); it != net->edgesEnd(); ++it,
      i++) {
    (*length)[*it] = (13 * i) % 3 + 1;
  }
  Dijkstra<> dijkstra(net);
  ESPDistCalculator<> calc(dijkstra, length, NetworkCache);
  std::cout << "Src\tDst\tDist" << std::endl;
  for (auto sit = net->nodesBegin(); sit != net->nodesEnd(); ++
      sit) {
    for (auto dit = sit + 1; dit != net->nodesEnd(); ++dit) {
      std::cout << sit->second << "\t" << dit->second << "\t"
          << calc.getDist(sit->first, dit->first).first << std::
              endl;
    }
  }
  return 0;
}
```

The output of this code is as follows:

```
Src     Dst     Dist
1       2       2
1       3       3
1       4       4
1       5       4
1       6       5
1       7       inf
1       8       inf
2       3       1
2       4       2
2       5       2
2       6       3
2       7       inf
2       8       inf
3       4       3
3       5       1
3       6       2
3       7       inf
3       8       inf
4       5       2
```

```
4       6       3
4       7       inf
4       8       inf
5       6       1
5       7       inf
5       8       inf
6       7       inf
6       8       inf
7       8       2
```

■

2.5.2 Approximate shortest path distances

Computing exact distances in very large networks can be time consuming, and resorting to approximations may be necessary. `ASPDistCalculator` is an implementation of `NetDistCalculator` that computes approximate shortest path distances of exact ones. The approximation works as follows. A set $\mathscr{L}$ of nodes called *landmarks* is selected, and the distance from each landmark to all other nodes is pre-computed and stored in memory. The distance between any two nodes i, j is then approximated by:

$$d_{ij} \simeq \min_{k \in \mathscr{L}} [d_{ik} + d_{kj}]. \tag{2.1}$$

The landmarks are passed to `ASPDistCalculator` object using the method `setLandmarks`. Of course, by increasing the number of landmarks, more precision can be obtained, be it though at a higher computational and memory cost. The choice of the landmarks is left to the user.

■ **Example 2.9** In the following code, we compute the approximate distances between all couples in the network of Figure 2.6 using 30% of nodes as landmarks.

```
#include "linkpred.hpp"
#include <iostream>
int main(int argc, char*argv[]) {
  auto net = UNetwork<>::read(std::string(argv[1]));
  auto length = net->template createEdgeMapSP<double>();
  int i = 1;
  for (auto it = net->edgesBegin(); it != net->edgesEnd(); ++it,
      i++) {
    (*length)[*it] = (13 * i) % 3 + 1;
  }
  Dijkstra<> dijkstra(net);
  ASPDistCalculator<> calc(dijkstra, length);
  double landmarkRatio = 0.3;
  long int seed = 777;
  std::vector<typename UNetwork<>::NodeIdType> landmarks;
  std::cout << "Landmarks:" << std::endl;
  for (auto it = net->rndNodesBegin(landmarkRatio, seed);
      it != net->rndNodesEnd(); ++it) {
    landmarks.push_back(it->first);
    std::cout << it->second << std::endl;
  }
  calc.setLandmarks(landmarks.begin(), landmarks.end());
  std::cout << "Src\tDst\tDist" << std::endl;
  for (auto sit = net->nodesBegin(); sit != net->nodesEnd(); ++
     sit) {
    for (auto dit = sit + 1; dit != net->nodesEnd(); ++dit) {
      std::cout << sit->second << "\t" << dit->second << "\t"
          << calc.getDist(sit->first, dit->first).first << std::
             endl;
    }
  }
  return 0;
}
```

The output of this code is as follows:

```
Landmarks:
1
4
Src     Dst     Dist
1       2       2
1       3       3
1       4       4
1       5       4
1       6       5
1       7       inf
1       8       inf
2       3       5
2       4       2
2       5       4
2       6       5
2       7       inf
2       8       inf
3       4       3
3       5       5
3       6       6
3       7       inf
3       8       inf
4       5       2
4       6       3
4       7       inf
4       8       inf
5       6       5
5       7       inf
5       8       inf
6       7       inf
6       8       inf
7       8       inf
```

■

3. Predictors

In this chapter, we cover the link prediction algorithms available in LinkPred. The library offers a unified interface for all link prediction algorithms which simplifies the use and comparison of different prediction methods. This interface is presented first in this chapter. The two subsequent sections present the available prediction algorithms for undirected networks and directed networks respectively. We end the chapter with an explanation on how to implement your own link prediction algorithm so that it can be used with LinkPred classes.

3.0.1 The predictor interface

All link predictors for undirected networks must inherit from the abstract class `ULPredictor` shown below. It declares three important virtual methods that must be implemented by the derivative classes:

- The method `void init()` : This method is used to initialize the state of the predictor, including any internal data structures. Depending on the predictor, this method may be left empty if no such initialization is required.
- The method `void learn()` : In algorithms that require learning, it is in this method that the model is built. The learning is separated from prediction, because usually the model is independent from the set of edges to be predicted. Notice that even if the algorithm does not require any learning, this method must still be implemented (it can be left empty).
- The method `double score(EdgeType const & e)` : returns the score for the edge `e` (usually a negative edge).

In addition to these three basic methods, `ULPredictor` declares the following three methods:

- The method `void predict(EdgesRandomIteratorT begin, EdgesRandomIteratorT end, ScoresRandomIteratorT scores)` : In this method, the edges to be predicted are passed to the predictor in the form of a range (`begin` , `end`) in addition to a third parameter (`scores`) to which the scores are written. All iterators must allow random access, and the memory for storing the scores must already be allocated.

- The method `std::pair<NonEdgeIterator, NonEdgeIterator> predictNeg( ScoresRandomIteratorT scores)` predicts the score for all negative (non-existing) links in the network. The scores are written into the random output iterator `scores`. The method returns a pair of iterators begin and end to the range of non-existing links predicted by the method.
- The method `std::size_t top(std::size_t k, EdgesRandomOutputIteratorT eit, ScoresRandomIteratorT sit)` finds the k negative edges with the top score. The edges are written to the output iterator `eit`, whereas the scores are written to `sit`. The scores are written in the same order as the edges. The method returns the number of negative edges inserted. It is the minimum between k and the number of negative edges in the network. Ties are broken randomly.

The class `ULPredictor` offers default implementations for the methods `top`, `predict` and `predictNeg`. Sub-classes may use these implementations or redefine them to achieve better performance.

```
template<...> class ULPredictor {
  virtual void init() = 0;
  virtual void learn() = 0;
  virtual double score(EdgeType const & e);
  virtual void predict(EdgesRandomIteratorT begin,
     EdgesRandomIteratorT end, ScoresRandomIteratorT scores);
  virtual std::pair<typename NetworkT::NonEdgeIterator, typename
     NetworkT::NonEdgeIterator> predictNeg(ScoresRandomIteratorT
     scores);
  virtual std::size_t top(std::size_t k,
     EdgesRandomOutputIteratorT eit, ScoresRandomIteratorT sit);
};
```

The abstract class `DLPredictor` plays the same role as `ULPredictor` but for link predictors in directed networks. It offers the same interface as the latter but with different default template arguments and methods implementation.

3.1 Link predictors for undirected networks

Most link predictors proposed in the literature apply to undirected networks. In this section, we present the main prediction algorithms implemented in LinkPred. We divide these into topological ranking methods and global methods. Topological ranking methods (or local methods) are fast and can scale to very large networks. On the other hand, global methods, which although produce good prediction results, are usually limited to small to medium sized networks because of their high computational requirements.

3.1.1 Topological-ranking methods

Topological-ranking methods use local topological information to assign scores to edges [11, 16]. Since they do not require learning, they are in general computationally efficient, and depending on the type of network, may produce highly precise predictions. A large number of topological measures have been proposed by researchers, and implementing all of them can be an arduous task. Nevertheless, LinkPred contains the implementation of the most important measures found in the literature.

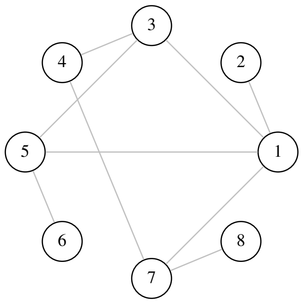


Figure 3.1: Example network.

1. **Adamic-Adar index (ADA)**: In this method, a couple (i,j) is assigned the score:

$$s_{ij} = \sum_{k \in \Gamma_{ij}} \frac{1}{\log(\kappa_k)}, \tag{3.1}$$

where Γ_{ij} is the set of nodes adjacent to both i and j (set of common neighbors of i and j), and κ_k is the degree of node k. If i and j have no common neighbors, their score is set to 0. Eq. (3.18) is well defined, because $\kappa_k \neq 1$ (since k is a common neighbor of i and j, its degree must be at least 2).

■ **Example 3.1** Consider the network of Figure 3.1. The Adamic-Adar index score of $(3,7)$ is:

$$\frac{1}{\log(\kappa_1)} + \frac{1}{\log(\kappa_4)} = \frac{1}{\log(4)} + \frac{1}{\log(2)} \approx 2.1640.$$

The score of $(5,8)$ is zero, since the two nodes have no common neighbors. ■

The Adamic-Adar index method is implemented in the class `ADAPredictor`.

2. **Common neighbors (CNE)**: In this approach, the score of a couple (i,j) is simply the number of common neighbors of i and j:

$$s_{ij} = |\Gamma_{ij}|. \tag{3.2}$$

■ **Example 3.2** For the network shown in Figure 3.1, the score of $(3,7)$ is 2, whereas the score of $(5,8)$ is zero, since thy have no common neighbors. ■

The common neighbors index method is implemented in the class `CNEPredictor`.

3. **Cannistraci resource allocation** index (CRA): The score of a couple (i,j) is given by:

$$s_{ij} = \sum_{k \in \Gamma_{ij}} \frac{|\Gamma_k \cap \Gamma_{ij}|}{\kappa_k},$$

where Γ_k is the set of nodes adjacent to node k.

■ **Example 3.3** For the network shown in Figure 3.2, the score given by CRA to $(3,7)$ is $\frac{1}{5} + \frac{1}{3} = 0.53$. ■

The predictor CRA is implemented by the class `CRAPredictor`.

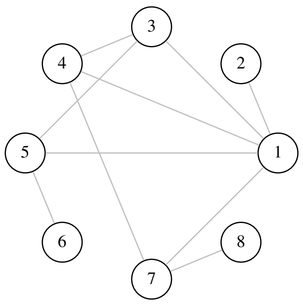


Figure 3.2: Example network.

4. **Hub depromoted index (HDI)**: In this approach, the score of a couple (i, j) is given by:

$$s_{ij} = \frac{|\Gamma_{ij}|}{\max(\kappa_i, \kappa_j)}. \tag{3.3}$$

■ **Example 3.4** For the network shown in Figure 3.1. The score of $(1,4)$ is 0.5, and so is the score of $(4,8)$. ■

The hub depromoted index method is implemented in the class `UHDIPredictor`.

5. **Hub promoted index (HPI)**: In this approach, the score of a couple (i, j) is given by:

$$s_{ij} = \frac{|\Gamma_{ij}|}{\min(\kappa_i, \kappa_j)}. \tag{3.4}$$

■ **Example 3.5** For the network shown in Figure 3.1, the score of $(1,4)$ is 1, and so is the score of $(4,8)$. ■

The hub promoted index method is implemented in the class `UHPIPredictor`.

6. **Jackard index (JID)**: In this approach, the score of a couple (i, j) is given by:

$$s_{ij} = \frac{|\Gamma_{ij}|}{\kappa_i + \kappa_j - |\Gamma_{ij}|}. \tag{3.5}$$

If i and j have no common neighbors, the score 0 is assigned.

■ **Example 3.6** For the network shown in Figure 3.1. The score of $(1,4)$ is 0.5, and so is the score of $(4,8)$. ■

The Jackard index method is implemented in the class `UJIDPredictor`.

7. **Local path index (LCP)**: This method can be thought of as higher order correction to common neighbors. Instead of considering only paths of length two between the two nodes i and j (which is equal to the number of common neighbors), the number of paths of length three, Π^3_{ij}, is also considered:

$$s_{ij} = |\Gamma_{ij}| + \varepsilon \Pi^3_{ij}, \tag{3.6}$$

where ε is an algorithm parameter, which usually takes small values (1e-3 for instance). It is worth mentioning that the computation of paths of length three increases the computational complexity of LCP compared to the other topological-ranking methods.

■ **Example 3.7** For the network shown in Figure 3.1, and taking $\varepsilon = 0.001$. The score of $(3,7)$ is $2+0.001\times 1 = 2.001$, and that of $(5,8)$ is $0+0.001\times 1 = 0.001$. ■

The local path index method is implemented in the class `ULCPPredictor`. The default value of ε is 1e-3. To read the value ε use the method `double getEpsilon()const`, and to modify it use `void setEpsilon(double epsilon)`.

8. **Leicht-Holme-Newman index (LHN)**: In this approach, the score of a couple (i,j) is given by:

$$s_{ij} = \frac{|\Gamma_{ij}|}{\kappa_i \kappa_j}. \tag{3.7}$$

■ **Example 3.8** For the network shown in Figure 3.1. The score of $(1,4)$ is 0.25, whereas the score of $(4,8)$ is 0.5. ■

The Leicht-Holme-Newman index method is implemented in the class `ULHNPredictor`.

9. **Preferential attachment index (PAT)**: In this approach, the score of a couple (i,j) is simply given by:

$$s_{ij} = \kappa_i \kappa_j. \tag{3.8}$$

■ **Example 3.9** For the network shown in Figure 3.1. The score of $(1,4)$ is 8, whereas the score of $(4,8)$ is 2. ■

The preferential attachment index method is implemented in the class `UPATPredictor`.

10. **Resource allocation index (RAL)**: In this approach, the score of a couple (i,j) is given by:

$$s_{ij} = \sum_{k\in\Gamma_{ij}} \frac{1}{\kappa_k}. \tag{3.9}$$

■ **Example 3.10** For the network shown in Figure 3.1. The score of $(1,4)$ is 0.67, whereas the score of $(4,8)$ is 0.33. ■

The resource allocation index method is implemented in the class `URALPredictor`.

11. **Salton index (SAI)**: In this approach, the score of a couple (i,j) is given by:

$$s_{ij} = \frac{|\Gamma_{ij}|}{\sqrt{\kappa_i \kappa_j}}. \tag{3.10}$$

■ **Example 3.11** For the network shown in Figure 3.1. The score of $(1,4)$ is $1/\sqrt{2} = 0.707$, and so is the score of $(4,8)$. ■

The Salton index method is implemented in the class `USAIPredictor`.

12. **Sorensen index (SOI)**: In this approach, the score of a couple (i,j) is given by:

$$s_{ij} = \frac{|\Gamma_{ij}|}{\kappa_i + \kappa_j}. \tag{3.11}$$

■ **Example 3.12** For the network shown in Figure 3.1. The score of $(1,4)$ is 0.33, and so is the score of $(4,8)$. ■

The Sorensen index method is implemented in the class `USOIPredictor`.

13. **Sum of degrees index (SUM)**: A popularity index where the score for couple (i,j) is simply given by:

$$s_{ij} = \kappa_i + \kappa_j. \tag{3.12}$$

■ **Example 3.13** For the network shown in Figure 3.1. The score of $(1,4)$ is 6, whereas the score of $(4,8)$ is 3. ■

The preferential attachment index method is implemented in the class `USUMPredictor`.

3.1.2 Global predictors

1. **Hierarchical Random Graph (HRG)** [2] is a probabilistic model where a hierarchical structure consisting of a binary tree is used to predict connection probabilities. The leaves of the binary tree represent the nodes of the network, whereas internal nodes correspond to nested clusters. Each cluster is assigned the probability of a link existing between its children. The probability of two nodes being connected is then determined by finding their lowest common ancestor. This algorithm is implemented by the class `UHRGPredictor` (which is actually a C++ wrapper around the code provided by the authors). This algorithms requires three parameters: a seed passed to the constructor and used to initialize the internal random generator, the number of bins which can be accessed and modified by `int getNumBins()const` and `void setNumBins(int numBins)`, and the number of samples which can be read and set using `int getNumSamples()const` and `void setNumSamples(int numSamples)`. The default value for `numBins` is 25 and that of `numSamples` is 10000.
2. **Stochastic block model (SBM)** [5] is a probabilistic model, where the nodes are divided into non-overlapping partitions. A matrix Q specifies the partition-to-partition connection probability, which is the probability that a node from one partition connects to a node from the other one. SBM can be used to detect both missing and spurious links. It produces excellent results in general, but its high computational cost limits its use to small networks. This algorithm is implemented by the class `USBMPredictor` (which is actually a C++ wrapper around the C code provided by the authors). This algorithms requires two parameters: a seed passed to the constructor and used to initialize the internal random generator, and the maximum number of iterations. The latter can be read and modified by `std::size_t getMaxIter()const` and `void setMaxIter(std::size_t maxIter)` respectively. The default value of `maxIter` is 10000.
3. **Fast blocking model (FBM)** [10] uses as greedy search strategy to efficiently partition the network into communities, reducing hence the computation complexity of the graph partitioning task. The link densities within and between communities are used to estimate the connection probability between nodes. Despite producing results that are in general slightly lower than those of SBM, its low computational requirements make FBM a good choice for average-size networks. FBM is implemented in the class `UFBMPredictor` (a C++ translation of the Matlab code provided

by the authors). This class requires a single parameter, which is the maximum number of iterations. It can be read and set using `std::size_t getMaxIter()const` and `void setMaxIter(std::size_t maxIter)` respectively. The default value of `maxIter` is 50.

4. **HyperMap (HYP)** [12, 13]: The *Popularity×Similarity Optimization* (PSO) and its variant E-PSO are complex network models that assume the existence of a hidden hyperbolic space that controls the topology of real networks. The likelihood of connection between nodes is a trade-off between the similarity of the nodes and their popularity, and it is the behavior of the connection probability with respect to nodes popularity that gives the hidden metric space its hyperbolic geometry. In these models, every is assigned a radial coordinate r_i and an angular coordinate θ_i. The probability that two nodes i, j connect is then given by:

$$p_{ij} = \frac{1}{1+e^{(d_{ij}-R)/T}}, \tag{3.13}$$

where R and T are model parameters and d_{ij} is the hyperbolic distance between i and j given by:

$$d_{ij} \approx r_i + r_j + \frac{2}{\zeta}\ln(\theta_{ij}/2), \tag{3.14}$$

where θ_{ij} is angular distance between i and j given by $\theta_{ij} = \pi - |\pi - |\theta_i - \theta_j||$. The parameter $\zeta = \sqrt{-K}$ with K representing the curvature of the hyperbolic plane.
The HyperMap algorithm embeds the network according to the E-PSO model by assuming that r_i are equal degree and using the *Metropolis-Hastings* algorithm to find the coordinates θ_i that maximize the local likelihood L_i,

$$L_i = \prod_{1\leq j\leq i} (p_{ij})^{a_{ij}}[1-(p_{ij})]^{1-a_{ij}} \tag{3.15}$$

HyperMap is implemented in the class `UHYPPredictor` (a wrapper around the code provided by the authors with the additional feature of fitting power law distribution to estimate the exponent γ as explained below). It receives a random number generator seed in its constructor. Additionally, it has five parameters required by the E-PSO model:
   (a) m: represents the average number of nodes with which new nodes connect. It is set to the minimum degree in the network. After calling `init`, it is possible to get and set m using `double getM() const` and `void setM(double m)` respectively.
   (b) L: represents the average number of nodes with which old nodes connect and is set to $L = (\langle k\rangle - 2m)/2$, where $\langle k\rangle$ is the average node degree. After calling `init`, it is possible to get and set L using `double getL() const` and `void setL(double L)` respectively.
   (c) γ: is the exponent of the power-law degree distribution. In our implementation, it is calculated from the degree distribution of the training set

network using *plfit*, a C++ implementation of Clauset, Shalizi and Newman [3] method for fitting power law distributions written by Tamas Nepusz (`http://tuvalu.santafe.edu/~aaronc/powerlaws/`). After calling `init`, it is possible to get and set γ using `double getGamma()const` and `void setGamma(double gamma)` respectively.

(d) T: controls the average clustering. It can be read and modified using `double getT() const` and `void setT(double T)` respectively. The default value is 0.8.

(e) $\zeta = \sqrt{-K}$ where K is the curvature of the hyperbolic plane. It is set by default to 1 and can be read and modified using `double getZeta()const` and `void setZeta(double zeta)`.

HyperMap gives good results in Internet networks and networks with similar properties, but it has a high a computational cost in general.

5. **Shortest-path predictor (SHP)**: This predictor assigns scores to node couples according to their shortest-path distance:

$$s_{ij} = \frac{1}{d_{ij}}. \tag{3.16}$$

It can also assign scores to existing edges:

$$s_{ij} = \frac{1}{\bar{d}_{ij}}, \tag{3.17}$$

where $\bar{d}_{ij}$ is the distance between i and j obtained by first removing the edge (i,j). The shortest-path predictor is implemented by the class `USHPPredictor`. The distances are computed using Dijkstra's algorithm, which is executed during the `predict` method. The distances are therefore not pre-calculated, but rather computed on-demand according to the set of edges to be predicted. Nevertheless, to improve the time performance the distances can be cached. The cache strategy can be read set using the method `CacheLevel getCacheLevel()const` and `void setCacheLevel(CacheLevel cacheLevel)` respectively (see Section 2.5.2 for more details).

6. **Scalable popularity-similarity[kab16] (KAB)**: A link prediction method that assumes that the likelihood of the existence of a link depends on popularity, similarity and local attraction. The algorithm pre-weights the graph with a specific weight map that factors out non-similarity factors and uses it to find similarity between non=connected nodes. The result is then used to assign scores to non-existing edges.

3.2 Link predictors for directed networks

LinkPred contains the implementations of several link prediction algorithms that work on directed networks. These are basically adaptations of topological-ranking methods shown earlier for the undirected case:

1. **Directed Adamic-Adar index (DADA)**: In this method, a couple (i,j) is assigned

the score:

$$s_{ij} = \sum_{k \in \Gamma_{i \to j}} \frac{1}{\log(\kappa_k)}, \tag{3.18}$$

where $\Gamma_{i \to j}$ is the set of nodes k such that there exists and edge between i and k and an edge between k and j. Here, κ_k is the degree of node k (the sum of out and in-degrees). If $\Gamma_{i \to j}$ is empty, the score is set to 0. Eq. (3.18) is well defined, because $\kappa_k \neq 1$. The directed Adamic-Adar index method is implemented in the class `DADAPredictor`.

2. **Directed common neighbors (DCNE)**: In this approach, the score of a couple (i, j) is simply the size of the set $\Gamma_{i \to j}$. The directed common neighbors index method is implemented in the class `DCNEPredictor`.
3. **Directed hub depromoted index (DHDI)**: In this approach, the score of a couple (i, j) is given by:

$$s_{ij} = \frac{|\Gamma_{i \to j}|}{\max(\kappa_i^{out}, \kappa_j^{in})}, \tag{3.19}$$

where κ_i^{out} is the out-degree of i, and κ_j^{in} is the in-degree of j. The hub depromoted index method is implemented in the class `DHDIPredictor`.
4. **Directed hub promoted index (DHPI)**: In this approach, the score of a couple (i, j) is given by:

$$s_{ij} = \frac{|\Gamma_{i \to j}|}{\min(\kappa_i^{out}, \kappa_j^{in})}, \tag{3.20}$$

The hub promoted index method is implemented in the class `DHDIPredictor`.
5. **Directed Jackard index (DJID)**: In this approach, the score of a couple (i, j) is given by:

$$s_{ij} = \frac{|\Gamma_{i \to j}|}{\kappa_i^{out} + \kappa_j^{in} - |\Gamma_{i \to j}|}. \tag{3.21}$$

If $\Gamma_{i \to j}$ is empty, the score 0 is assigned. The directed Jackard index method is implemented in the class `DJIDPredictor`.
6. **Directed local path index (DLCP)**: This method can be thought of as higher order correction to directed common neighbors. It assigns the score:

$$s_{ij} = |\Gamma_{i \to j}| + \varepsilon \Pi^3_{i \to j}, \tag{3.22}$$

where Π^3_{ij} stands for the number of directed paths of length three, and ε is an algorithm parameter, which usually takes small values (1e-3 for instance). The directed local path index method is implemented in the class `DLCPPredictor`. The default value of ε is 1e-3. To read the value ε use the method `double getEpsilon()const`, and to modify it use `void setEpsilon(double epsilon)`.

7. **Directed Leicht-Holme-Newman index (DLHN)**: In this approach, the score of a couple (i,j) is given by:

$$s_{ij} = \frac{|\Gamma_{i \to j}|}{\kappa_i^{out} \kappa_j^{in}}. \tag{3.23}$$

The directed Leicht-Holme-Newman index method is implemented in the class `DLHNPredictor` .

8. **Directed preferential attachment index (DPAT)**: In this approach, the score of a couple (i,j) is simply given by:

$$s_{ij} = \kappa_i^{out} \kappa_j^{in}. \tag{3.24}$$

The directed preferential attachment index method is implemented in the class `DPATPredictor` .

9. **Directed Salton index (DSAI)**: In this approach, the score of a couple (i,j) is given by:

$$s_{ij} = \frac{|\Gamma_{i \to j}|}{\sqrt{\kappa_i^{out} \kappa_j^{in}}}. \tag{3.25}$$

The directed Salton index method is implemented in the class `DSAIPredictor` .

10. **Directed Sorensen index (DSOI)**: In this approach, the score of a couple (i,j) is given by:

$$s_{ij} = \frac{|\Gamma_{i \to j}|}{\kappa_i^{out} + \kappa_j^{in}}. \tag{3.26}$$

The directed Sorensen index method is implemented in the class `DSOIPredictor` .

3.3 Implementing a new link prediction algorithm

The first step in implementing a new link prediction algorithm is to inherit from `ULPredictor` and implement the necessary methods. For a minimal implementation, the three methods `init` , `learn` and `score` must at least be defined. If you want to achieve better performance you may want to redefine the three other methods (`top` , `predict` and `predictNeg`).

■ **Example 3.14** Suppose you want to create a very simple link prediction algorithm that assigns as score to (i,j) the score $\kappa_i + \kappa_j$, the sum of the degrees of the two nodes[1]. In a file named `msupredictor.hpp`, write the following code:

```
#ifndef MSUPREDICTOR_HPP_
#define MSUPREDICTOR_HPP_
#include <linkpred.hpp>
/**
 * @brief A sum-of-degrees link predictor.
 */
```

[1]LinkPred already contains a sum-of-degree predictor named `USUMPredictor` .

```
class MSUPredictor: public LinkPred::LPredictor<> {
  using LinkPred::LPredictor<>::net;
  using LinkPred::LPredictor<>::name;
public:
  using EdgeType = typename LinkPred::LPredictor<>::EdgeType;
  MSUPredictor(std::shared_ptr<NetworkT const> net) :
      LinkPred::LPredictor<>(net) {
    name = "MSU";
  }
  virtual void init();
  virtual void learn();
  virtual double score(EdgeType const & e);
  virtual ~MSUPredictor() = default;
};
#endif /* MSUPREDICTOR_HPP_ */
```

R The abstract class `ULPredictor` is in fact a class template, but in the code above we are extending it with the default template parameters. Although a bit restrictive, this approach is the quickest and easiest way to add a new predictor.

In a file named `msupredictor.cpp` write the implementation of the abstract methods:

```
#include "msupredictor.hpp"
void MSUPredictor::init() {}
void MSUPredictor::learn() {}
double MSUPredictor::score(EdgeType const & e) {
  auto i = net->start(e);
  auto j = net->end(e);
  return net->getDeg(i) + net->getDeg(j);
}
```

Note that this predictor does not require initialization or learning. This predictor is now ready to be used with LinkPred classes and methods. ■

4. Performance Evaluation

Performance evaluation is a crucial phase in the development of new link prediction algorithms as well as in the study of their effectiveness for a given type or family of networks. LinkPred offers a set of tools that help streamlining the performance evaluation procedure. This includes data setup functionalities, which can be used to create test data, efficient implementations of the most important performance measures used in link prediction literature, and and helper classes that facilitates the comparative evaluation of multiple link prediction algorithms using multiple performance measures.

4.1 Data setup

To measure the performance of a link prediction algorithm, it is presented with a distorted version of a fully known network that serves as ground truth data. The distortions to which the network is subjected can be categorized into three types[1]:

1. Removing existing links.
2. Adding new links.
3. A combination of the above.

The task of the link prediction algorithm is to determine the actual status of couples based on the observed relationships. Notice that traditionally, the role of link prediction methods has been limited to detecting missing links. As a result, most link prediction algorithms can only handle distortions of the first kind (link removal). Nevertheless, LinkPred offers the possibility of performing all three types of distortions. Before presenting the relevant classes and methods, some terminology is of the order:

- *The reference network* is the ground truth network used to measure the performance of the algorithm. This network is unknown to the algorithm.
- *The observed network* is the network obtained after distortion and presented to the algorithm.
- *A true positive link* is a link that is present in the reference network as well as the observed network.

[1]Notice that distortions are limited to edges. No nodes are added or removed from the network

- *A true negative link* is a link that is missing from the reference network as well as the observed network.
- *A false positive link* is a link that is missing from the reference network but present in the observed network.
- *A false negative link* is a link that is present in the reference network but missing form the observed network.

Notice that depending on the type of distortions applied to the network, some of the sets defined above (true positive links, true negative links, false positive links and false negative links) may be empty. For instance, if the network is only modified by removing existing links, the set of false positive links contains no elements[2].

The data used for performance evaluation is stored within a class named `TestData`. This class provides a smart pointer to the reference network via `getRefNet()`, a smart pointer to the observed network via `getObsNet()` and the following ranges:

- The set of positive links included in the test set: `posBegin()` and `posEnd()`.
- The set of negative links included in the test set: `negBegin()` and `negEnd()`.

A `TestData` object can be created by calling the constructor:

```
TestData testData(refNet, obsNet, remLinks, addLinks, tpLinks,
    tnLinks, posClass, negClass);
```

The two last arguments are of the type `LinkClass`:

```
enum LinkClass {
  TP, /**< True positive link. */
  FN, /**< False negative link. */
  FP, /**< False positive link. */
  TN /**< True negative link. */
};
```

and are used to specify the set of links used, respectively, as positive instances and negative instances in the test set. This allows for instance to consider non-existing links as the positive instances.

It is clear from the constructor's signature that `TestData` is intended to be merely a container to store the test data elements together. To generate the test data, LinkPred provides the class `NetworkManipulator`, which contains a set of static methods that can be used to that end.

The first of these method distorts the network by removing existing links:

```
static TestData<NetworkT, std::vector<EdgeType>>
    createTestDataRem(NetworkCSP refNet, double remRatio, bool
    keepConnected, bool aTP, double tpRatio, bool aTN, double
    tnRatio, long int seed, bool preGenerateTPN = true);
```

The parameters of this method are as follows:

- `refNet`: A constant shared pointer to the reference network.
- `remRatio`: Value between 0 and 1 that specifies the percentage of edges to be removed.

[2]It is important to keep in mind that the class of a link as defined here is determined solely by specifying the reference and observed networks and is independent of any classification results. Hence, if a classifier is used on the network, a true negative link may for example be classified as positive and will therefore constitute a false positive instance. This may render the discussion a bit confusing by times but is necessary to keep in line with existing conventions.

- `keepConnected` : Specifies whether to keep the network connected. If the reference network is disconnected or the ratio of edges to be removed is too large to keep the network connected, an exception is raised.
- `aTP` : Specifies whether to use all true positive links in the test set.
- `tpRatio` : Ratio of true positive links to be used in the test set. This parameter is only relevant when `aTP` is false.
- `aTN` : Specifies whether to use all true negative links in the test set.
- `tnRatio` : Ratio of true negative links to be used in the test set. This parameter is only relevant when `aTN` is false.
- `seed` : The random number generator's seed.
- `preGenerateTPN` : Whether to pre-generate true positives and true negatives.

The set of false negative links is used as the set of positive instances in the test set, whereas the set of true negative links is used as the set of negative instances.

The second method distorts the network by adding new links:

```
static TestData<NetworkT, std::vector<EdgeType>>
    createTestDataAdd(NetworkCSP refNet, double remRatio, bool aTP
    , double tpRati, bool aTN, double tnRatio, long int seed, bool
     preGenerateTPN = true);
```

The parameters of this method are as follows:

- `refNet` : A constant shared pointer to the reference network.
- `addRatio` : Value between 0 and 1 that specifies the percentage of edges to be added.
- `aTP` : Specifies whether to use all true positive links in the test set.
- `tpRatio` : Ratio of true positive links to be used in the test set. This parameter is only relevant when `aTP` is false.
- `aTN` : Specifies whether to use all true negative links in the test set.
- `tnRatio` : Ratio of true negative links to be used in the test set. This parameter is only relevant when `aTN` is false.
- `seed` : The random number generator's seed.
- `preGenerateTPN` : Whether to pre-generate true positives and true negatives.

The set of true positive links is used as the set of positive instances in the test set, whereas the set of false positive links is used as the set of negative instances.

The last method is more flexible and can be used to create a new network by both adding and removing links. The method starts first by removing existing links, then proceeds to add new links:

```
static TestData<NetworkT, std::vector<EdgeType>> createTestData(
    NetworkCSP refNet, double remRatio, double addRatio, bool
    keepConnected, bool aTP, double tpRatio, bool aTN, double
    tnRatio, LinkClass posClass, LinkClass negClass, long int seed
    , bool preGenerateTPN = true);
```

The parameters of this method are as follows:

- `refNet` : A constant shared pointer to the reference network.
- `remRatio` : Value between 0 and 1 that specifies the percentage of edges to be removed.
- `addRatio` : Value between 0 and 1 that specifies the percentage of edges to be added.

- `keepConnected` : Specifies whether to keep the network connected. If the reference network is disconnected or the ratio of edges to be removed is too large to keep the network connected, an exception is raised.
- `aTP` : Specifies whether to use all true positive links in the test set.
- `tpRatio` : Ratio of true positive links to be used in the test set. This parameter is only relevant when `aTP` is false.
- `aTN` : Specifies whether to use all true negative links in the test set.
- `tnRatio` : Ratio of true negative links to be used in the test set. This parameter is only relevant when `aTN` is false.
- `posClass` : Indicates which links will be considered the positive links.
- `negClass` : Indicates which links will be considered the negative links.
- `seed` : The random number generator's seed.
- `preGenerateTPN` : Whether to pre-generate true positives and true negatives.

The class `NetworkManipulator` offers two other methods for generating test data by comparing two snapshots of the same network: `createTestDataSeq` and `createTestDataSeqInter` .

R If the parameter `preGenerateTPN` is set to false, edges are only generated on-demand. The class `TestData` can also *stream* edges without storing them in memory. This is particularity useful for very large networks. The streamed edges are accessed through the following methods of the class `TestData` :

```
auto posStrmBegin() const;
auto posStrmEnd() const;
auto negStrmBegin() const;
auto negStrmEnd() const;
```

■ **Example 4.1** Consider the network shown in the right side of Figure 4.1. The following code creates a distorted version of this network by adding and removing edges. The resulting network is shown in the right side of Figure 4.1.

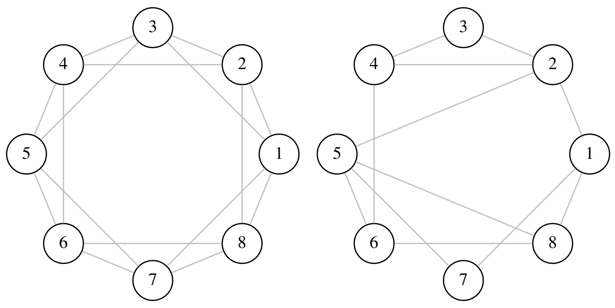


Figure 4.1: Example of network distortion. To the right, the reference network. To the left, the observed network.

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  int n = 8;
  auto net = std::make_shared<UNetwork<>>();
  for (int i = 1; i <= n; i++) {
    std::string il = std::to_string(i);
    std::string jl = std::to_string(i % n + 1);
    net->addEdge(net->addNode(il).first, net->addNode(jl).first);
    jl = std::to_string((i + 1) % n + 1);
    net->addEdge(net->addNode(il).first, net->addNode(jl).first);
  }
  net->assemble();
  auto testData = NetworkManipulator<>::createTestData(net, 0.4,
     0.3, true, true, 0, true, 0, FN, TN, 777);
  std::cout << "Reference␣network:\n";
  testData.getRefNet()->print();
  std::cout << "Observed␣network:\n";
  testData.getObsNet()->print();
  std::cout << "Positive␣links:" << std::endl;
  for (auto it = testData.posBegin(); it != testData.posEnd(); ++
     it) {
    std::cout << net->getLabel(net->start(*it)) << "\t" << net->
       getLabel(net->end(*it)) << std::endl;
  }
  std::cout << "Negative␣links:" << std::endl;
  for (auto it = testData.negBegin(); it != testData.negEnd(); ++
     it) {
    std::cout << net->getLabel(net->start(*it)) << "\t" << net->
       getLabel(net->end(*it)) << std::endl;
  }
  return 0;
}
```

The following is the output of this code:

```
Reference network:
2       1
2       3
2       4
2       8
1       3
1       7
1       8
3       4
3       5
4       5
4       6
5       6
5       7
6       7
6       8
7       8
Observed network:
2       1
2       3
2       4
1       6
1       7
1       8
3       4
4       6
```

```
5       6
5       7
5       8
6       8
Positive links:
6       7
7       8
2       8
4       5
3       5
1       3
Negative links:
2       5
2       6
2       7
1       4
1       5
3       6
3       7
3       8
4       7
4       8
```

■

For performance purposes and to avoid redundant computations, link prediction results are stored in an object of the class `PredResults`. The constructor of this class takes int two parameters a `TestData` object and an `std::shared_ptr` to a link predictor. The most important methods provided by this class are:

- `bool isPosComputed()const`: Check whether the positive links scores have been computed.
- `void compPosScores()`: Compute the scores of positive links. The method performs the computation only once.
- `bool isNegComputed()const`: Check whether the negative links scores have been computed.
- `void compNegScores()`: Compute the scores of negative links. The method performs the computation only once.
- `SortStatus getNegSortStatus()const`: Return the sort status of negative links scores. The type `SortStatus` is an enumeration containing the following values:

```
enum SortStatus {
  None, /**< Not sorted. */
  Inc, /**< Sorted in increasing order. */
  Dec /**< Sorted in decreasing order. */
};
```

- `void sortNeg(SortStatus negSortStatus)`: Sort the negative links scores according to the specified sorting direction. The method only sorts the scores if necessary.
- `SortStatus getPosSortStatus()const`: Return the sort status of positive links scores.
- `void sortPos(SortStatus posSortStatus)`: Sort the positive links scores according to the specified sorting direction. The method only sorts the scores if necessary.

4.2 Performance measures

All performance measures in LinkPred inherit from the abstract class `PerfMeasure`. Every performance should be uniquely identified by its name, which can be passed as parameter to the constructor. The most important method in the class `PerfMeasure` is `eval` which evaluates the value of the performance measure given an object `predResult` (see Section 4.1). The results of the performance measure are written to an object of type `PerfResults` passed as the second parameter of the method. The class `PerfResults` is defined as `std::map<std::string, double>`. This allows the possibility of associating several result values with a single performance measure.

An important class of performance measures are performance curves such as the receiver operating characteristic (ROC) curve and the precision-recall (PR) curve. These are represented by the abstract class `PerfCurve`, which inherits from the class `PerfMeasure`. The class `PerfCurve` defines a new virtual method:

```
virtual std::vector<std::pair<double, double>> getCurve(std::
    shared_ptr<PredResultsT>& predResults) = 0;
```

which returns the performance curve in the form of an `std::vector` of points. Each data point is represented by an `std::pair`, where the first element is the x coordinate, whereas the second element is the corresponding y coordinate. Although not mandatory, the area under the curve, computed using numerical integration, is the typical performance value associated with a performance curve, and its is that value which is returned by the method `eval`.

LinkPred includes the implementation of number of performance measures, including the most important performance curves (ROC and PR) and a generic (parameterized) curve class. These performance measures are presented in the rest of this section.

4.2.1 Receiver operating characteristic curve (ROC)

One of the most important performance measure used in the field of link prediction is the receiver operating (ROC) curve, in which the true positive rate (recall) is plotted against the false positive rate. The ROC curve can be computed using the class `ROC`. The following code show how to calculate the ROC curve and the associate area under the curve. Notice that the two operations are independent of each other, and if the AUC is the only result required, it is enough (and computationally better) to call the method `eval`. An example ROC curve obtained using this code is plotted in Figure 4.2.

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  std::string netFileName(argv[1]);
  auto fullNet = UNetwork<>::read(netFileName, false, true);
  auto testData = NetworkManipulator<>::createTestData(fullNet,
      0.3, 0, true, true, 0, true, 0, FN, TN, 777);
  testData.lock();
  auto predictor = std::make_shared<UHRGPredictor<>>(testData.
      getObsNet(),  333);
  predictor->init();
  predictor->learn();
  auto predResults = std::make_shared<PredResults<>>(testData,
      predictor);
  auto roc = std::make_shared<ROC<>>("ROC");
  auto curve = roc->getCurve(predResults);
  std::cout << "#x\ty\n";
  for (std::size_t i = 0; i < curve.size(); i++) {
    std::cout << curve[i].first << "\t" << curve[i].second << std
        ::endl;
  }
  PerfResults res;
  roc->eval(predResults, res);
  std::cout << "#ROCAUC:␣" << res.at(roc->getName()) << std::endl
      ;
  return 0;
}
```

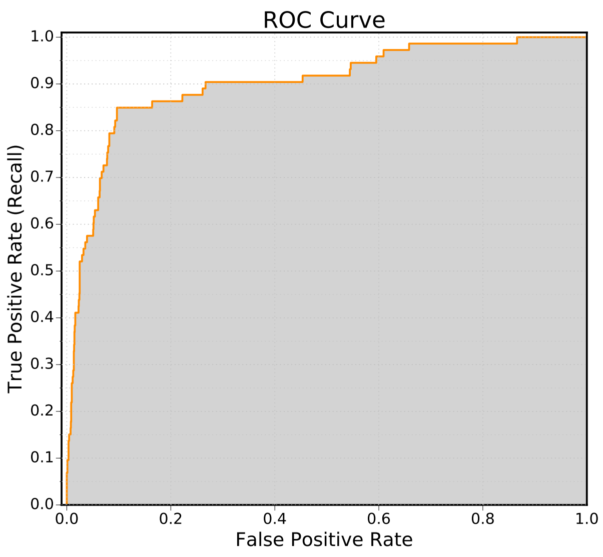


Figure 4.2: Example ROC curve. The area under the curve (shown in gray) is the value associated with this performance curve.

The default behavior of the `ROC` performance measure is to compute the positive and negative edge scores and then compute the area under the curve. This may lead to memory issues with large graphs. To compute the ROCAUC without storing both types of scores, the class `ROC` offers a method that *streams* scores without storing them. To enable this method, call `setStrmEnabled(`**bool**`)` on the `ROC` object. To specify which scores to steam use the method `setStrmNeg(`**bool**`)`. By default the negative scores are streamed, while the positive scores are stored. Passing **false** to `setStrmNeg` switches this.

■ **Example 4.2** This is an example of using the streaming method with `ROC`.

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char *argv[]) {
  std::string netFileName(argv[1]);
  auto refNet = UNetwork<>::read(netFileName);
  auto testData = NetworkManipulator<>::createTestData(refNet,
      0.1, 0, false, true, 0, true, 0, FN, TN, 777, false);
  testData.lock();
  auto predictor = std::make_shared<UADAPredictor<>>(testData.
     getObsNet());
  predictor->init();
  predictor->learn();
  auto predResults = std::make_shared<PredResults<>>(testData,
     predictor);
  auto roc = std::make_shared<ROC<>>("ROC");
  roc->setStrmEnabled(true);
  PerfResults res;
  roc->eval(predResults, res);
  std::cout << "#ROCAUC␣(streaming):␣" << res.at(roc->getName())
     << std::endl;
  return 0;
}
```

■

In addition to consuming little memory, the streaming method supports distributed processing (in addition to shared memory parallelism), which makes it suitable for large networks (see Chapter 5).

4.2.2 Precision-recall curve

The precision-recall (PR) curve is also a widely used measure of performance of link prediction algorithms. In this curve, the precision is plotted as a function of the recall. The PR curve can be computed using the class `PR`. The area under the PR curve can be computed using two integration methods:

- The trapezoidal rule which assumes a linear interpolation between the PR points.
- Nonlinear interpolation as proposed by Jesse Davis and Mark Goadrich [4].

The second method is more accurate, as linear integration tends to overestimate the area under the curve [4]. Furthermore, the implementation of Davis-Goadrich nonlinear interpolation in LinkPred ensures little to no additional cost compared to the trapezoidal method. Nevertheless, the user can choose the integration method using the method `void setInterpolMethod(InterpolMethod interpolMethod)`. The active integration method can be queried using `InterpolMethod getInterpolMethod()const`. The type `InterpolMethod` is a public enumeration of the class `PR` with two possible values:

```
enum InterpolMethod {
LIN, /**< Linear interpolation (Trapezoidal rule). */
DGI /**< Davis-Goadrich nonlinear interpolation. */
};
```

By default, the integration method used is `DGI`.

The following code shows how to calculate the PR curve and the associate area under the curve. Notice that the two operations are independent of each other, and if the AUC is the only result required, it is enough to call the method `eval`. An example PR curve obtained using this code is plotted in Figure 4.3.

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  std::string netFileName(argv[1]);
  auto fullNet = UNetwork<>::read(netFileName, false, true);
  auto testData = NetworkManipulator<>::createTestData(fullNet,
     0.3, 0, true, true, 0, true, 0, FN, TN, 777);
  testData.lock();
  auto predictor = std::make_shared<UHRGPredictor<>>(testData.
     getObsNet(), 333);
  predictor->init();
  predictor->learn();
  auto predResults = std::make_shared<PredResults<>>(testData,
     predictor);
  auto pr = std::make_shared<PR<>>("PR");
  auto curve = pr->getCurve(predResults);
  std::cout << "#x\ty\n";
  for (std::size_t i = 0; i < curve.size(); i++) {
    std::cout << curve[i].first << "\t" << curve[i].second << std
       ::endl;
  }
  PerfResults res;
  pr->eval(predResults, res);
  std::cout << "#PRAUC:␣" << res.at(pr->getName()) << std::endl;
  return 0;
}
```

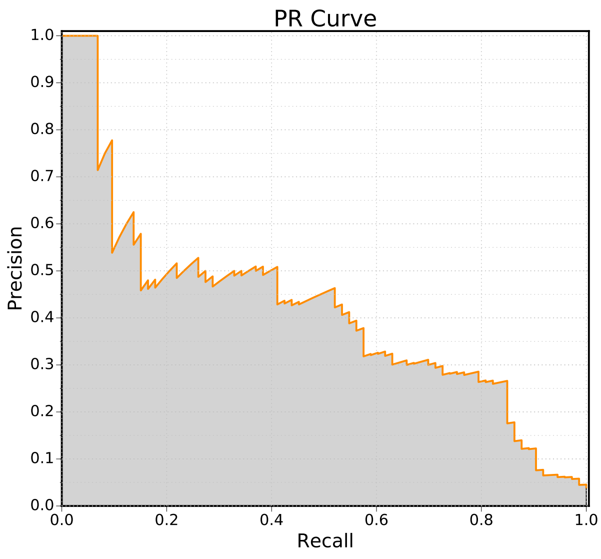


Figure 4.3: Example PR curve. The area under the curve (shown in gray) is the value associated with this performance curve.

4.2.3 General performance curves

LinkPred offers the possibility of calculating general performance curves using the class `GCurve`. A performance curve is in general defined by giving the *x* and *y* coordinates functions. These are passed as parameters -in the form of lambdas- to the constructor of the class `GCurve`. The associated performance value is the area under the curve computed using the trapezoidal rule (linear interpolation). For example, the ROC curve can be defined as:

```
GCurve<> cur(fpr, rec, "ROC");
```

The two first parameters of the constructors are lambdas having the signature:

```
double(std::size_t tp, std::size_t fn, std::size_t tn, std::
    size_t fp, std::size_t P, std::size_t N)
```

where:

- `tp` : Number of true positives.
- `fn` : Number of false negatives.
- `tn` : Number of true negatives.
- `fp` : Number of false positives.
- `P` : Number of positives. Notice that: `P` = `tp` + `fn` .
- `N` : Number of negatives. Here also: Notice that: `N` = `tn` + `fp` .

LinkPred contains the definition of several useful lambdas that can be used to define performance curves. These are defined in the name space `PerfLambda` :

- Recall (`rec`):

$$\frac{tp}{P}. \tag{4.1}$$

- False positive rate (`fpr`):

$$\frac{fp}{N}. \tag{4.2}$$

- Precision (`pre`):

$$\frac{tp}{tp+fp}. \tag{4.3}$$

- False negative rate (`fnr`):

$$\frac{fn}{P}. \tag{4.4}$$

- True negative rate (`tnr`):

$$\frac{tn}{N}. \tag{4.5}$$

- False omission rate (`fmr`):

$$\frac{fn}{tn+fn}. \tag{4.6}$$

- Accuracy (`acc`):

$$\frac{tp+tp}{P+N}. \tag{4.7}$$

- False discovery rate (`fdr`):

$$\frac{fp}{tp+fp}. \tag{4.8}$$

- Negative predictive value (`npv`):

$$\frac{tn}{tn+fn}. \tag{4.9}$$

Notice that some of these functions may be undefined for certain boundary values of the threshold, and therefore particular care must be taken when using them with `GCurve` . In particular, the curve, and consequently the area under it, may become undefined in some cases. For instance, it is possible to define the PR curve using `GCurve` in the same way we previously defined the ROC curve:

```
GCurve<> pr(rec, pre, "PR");
```

However, there are two important reasons to avoid such practice. First, as explained before, the area under the curve may be undefined in some cases. Second, the class `PR` offers a more accurate method for calculating the area under the curve (Davis-Goadrich interpolation) than the method used by `GCurve` (trapezoidal rule).

The following code shows how to calculate the ROC curve with negatives replacing positives (we denote this curve by NROC) and the associated area under the curve. This can be achieved by using `GCurve` with `fnr` (false negative rate) as the *x*-coordinates and `tnr` (true negative rate) as the *y*-coordinates. An example NROC curve obtained using this code is plotted in Figure 4.4.

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  std::string netFileName(argv[1]);
  auto fullNet = UNetwork<>::read(netFileName, false, true);
  auto testData = NetworkManipulator<>::createTestData(fullNet,
     0.3, 0, true, true, 0, true, 0, FN, TN, 777);
  testData.lock();
  auto predictor = std::make_shared<UHRGPredictor<>>(testData.
     getObsNet(), 333);
  predictor->init();
  predictor->learn();
  auto predResults = std::make_shared<PredResults<>>(testData,
     predictor);
  auto nroc = std::make_shared<GCurve<>>(PerfLambda::fnr,
     PerfLambda::tnr, "NROC");
  auto curve = nroc->getCurve(predResults);
  std::cout << "#x\ty\n";
  for (std::size_t i = 0; i < curve.size(); i++) {
    std::cout << curve[i].first << "\t" << curve[i].second << std
       ::endl;
  }
  PerfResults res;
  nroc->eval(predResults, res);
  std::cout << "#NROCAUC:␣" << res.at(nroc->getName()) << std::
     endl;
  return 0;
}
```

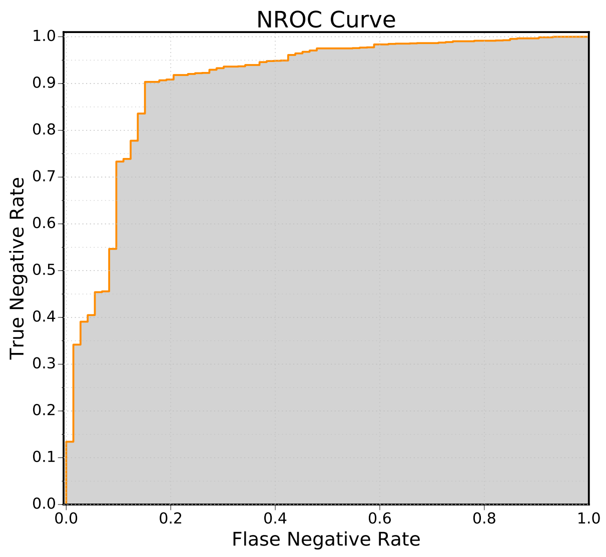


Figure 4.4: Example ROC curve with negative instances replacing positive ones. The area under the curve (shown in gray) is the value associated with this performance curve.

4.2.4 Top precision

The top precision measure is defined as the ratio of true positives within the top l scored edges, $l > 0$ being a parameter of the measure (usually l is set to the number of links removed from the network). Top precision is implemented by the class `TPR`, and since it is not a curve measure, this class inherits directly from `PerfMeasure`. The class `TPR` offers two approaches for computing top-precision. The first approach requires computing the score of all negative links, whereas the second approach calls the method `top` of the predictor. The first approach is in general more precise than the second one but may require more memory and time. The reason behind this is that it is possible to write efficient implementations of the method `top` (finding top scored edges) for most prediction algorithms. Indeed for most link predictors computing top scored edges does not require generating true negative links nor computing their scores. As a result, the second approach is the performance measure of choice for very large networks. Note that if `ROC` or `PR` are requested, it is better to use the first approach, since the computation of these two performance measures require the computation of the scores of all negative links anyways. To toggle between the two approaches simply call `setUseTopMethod`.

R The implementation of the method `top` in link predictors may be biased based on node IDs. This may skew the results when computing top-precision. To obtain unbiased results over multiple runs, it is advised to reshuffle the node IDs from run to run. This can be done by simply calling the method `shuffle` on the reference network between runs.

The following code shows how to use the class `TPR` using the first approach.

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  std::string netFileName(argv[1]);
  auto fullNet = UNetwork<>::read(netFileName, false, true);
  auto testData = NetworkManipulator<>::createTestData(fullNet,
     0.3, 0, true, true, 0, true, 0, FN, TN, 777);
  testData.lock();
  auto predictor = std::make_shared<UHRGPredictor<>>(testData.
     getObsNet(),  333);
  predictor->init();
  predictor->learn();
  auto predResults = std::make_shared<PredResults<>>(testData,
     predictor);
  auto tpr = std::make_shared<TPR<>>(testData.getNbPos(), "TPR");
  PerfResults res;
  tpr->eval(predResults, res);
  std::cout << "TPR:␣" << res.at(tpr->getName()) << std::endl;
  return 0;
}
```

In the next code, we compute top-precision using the method `top`:

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  std::string netFileName(argv[1]);
  auto fullNet = UNetwork<>::read(netFileName, false, true);
  auto testData = NetworkManipulator<>::createTestData(fullNet,
     0.3, 0, true, true, 0, true, 0, FN, TN, 777);
  testData.lock();
  auto predictor = std::make_shared<URALPredictor<>>(testData.
     getObsNet());
  predictor->init();
  predictor->learn();
  auto predResults = std::make_shared<PredResults<>>(testData,
     predictor);
  auto tpr = std::make_shared<TPR<>>(testData.getNbPos(),"TPRT");
  tpr->setUseTopMethod(true);
  PerfResults res;
  tpr->eval(predResults, res);
  std::cout << "TPRT:␣" << res.at(tpr->getName()) << std::endl;
  return 0;
}
```

4.3 Performance evaluation classes

LinkPred offers two helper classes that simplify the task of evaluating and comparing the performance of link prediction algorithms: `PerfEvaluator` and `PerfEvalExp`.

4.3.1 The class `PerfEvalExp`

The class `PerfEvalExp` allows to evaluate the performance of several link predictors based on several performance measures. The experimental setting in `PerfEvalExp` consists in removing a certain ratio of existing links from a reference network and presenting the algorithms with the obtained network (the observed networks). The performance measures specified by the user are then applied to assess the predictive power of the algorithms. The parameters of the experiment are passed to `PerfEvalExp` as an instance of the **struct** named `PerfEvalExpDesc`. These include the reference network, the number of iterations, the range of removal ratios (defined as `ratioStart`, `ratioEnd` and `ratioStep`), the ratio of false negative links and true negative links used in the test set, etc.

```
template<typename NetworkT=UNetwork<>> struct PerfeEvalExpDescp{
  ...
  std::shared_ptr<NetworkT> refNet;
  std::size_t nbTestRuns = 1;
  double ratioStart = 0.1;
  double ratioEnd = 0.1;
  double ratioStep = 0.1;
  bool keepConnected = false;
  double fnRatio = 1;
  double tnRatio = 1;
  bool timingEnabled = false;
  long int seed = 0;
  std::ostream* out = &std::cout;
};
```

`PerfEvalExp` requires also a callback object to create link predictors and performance measures. This object must implement the interface `PEFactory`, which contains two methods one for creating link predictors `getPredictors` and the other for creating performance measures `getPerfMeasures`:

```
template<...> class PEFactory {
public:
  virtual std::vector<std::shared_ptr<LPredictorT>> getPredictors
     (std::shared_ptr<NetworkT const> obsNet) = 0;
  virtual std::vector<std::shared_ptr<PerfMeasureT>>
     getPerfMeasures(TestDataT const & testData) = 0;
};
```

■ **Example 4.3** The following code shows how to use `PerfEvalExp` to compare three link prediction methods (ADA, JID and RAL) using top-precision (calling the `top` method). The experiment is repeated ten times and the default ratio of removed edges is used (0.1).

```
#include <linkpred.hpp>
#include <iostream>
#include <vector>
#include <memory>
using namespace LinkPred;
class Factory: public PEFactory<> {
public:
  virtual std::vector<std::shared_ptr<LPredictor<>>>
     getPredictors(std::shared_ptr<UNetwork<> const> obsNet) {
    std::vector<std::shared_ptr<LPredictor<>>> prs;
    prs.push_back(std::make_shared<UADAPredictor<>>(obsNet));
    prs.push_back(std::make_shared<UJIDPredictor<>>(obsNet));
    prs.push_back(std::make_shared<URALPredictor<>>(obsNet));
    return prs;
  }
  virtual std::vector<std::shared_ptr<PerfMeasure<>>>
     getPerfMeasures(TestData<> const & testData) {
    std::vector<std::shared_ptr<PerfMeasure<>>> pms;
    auto tpr = std::make_shared<TPR<>>(testData.getNbPos(), "TPRT
       ");
    tpr->setUseTopMethod(true);
    pms.push_back(tpr);
    return pms;
  }
  virtual ~Factory() = default;
};
int main(int argc, char*argv[]) {
  PerfeEvalExpDescp<> ped;
  ped.refNet = UNetwork<>::read("Infectious.edges");
  ped.nbTestRuns = 10;
  ped.seed = 777;
  auto factory = std::make_shared<Factory>();
  PerfEvalExp<> exp(ped, factory);
  exp.run();
  return 0;
}
```

The output of this code is as follows:

```
# n: 410 m: 2765
#ratio  TPRTADA TPRTJID TPRTRAL
0.10    0.3225  0.3225  0.3297
0.10    0.3225  0.3696  0.3514
0.10    0.3370  0.3406  0.3406
0.10    0.3188  0.3406  0.3297
0.10    0.3007  0.3623  0.3297
0.10    0.3188  0.3406  0.3442
0.10    0.3406  0.3370  0.3551
0.10    0.3116  0.3225  0.3442
0.10    0.3841  0.3696  0.3949
0.10    0.3406  0.3478  0.3696
#Time: 688.532 ms
```

■

Enabling timing in `PerfEvalExp` (by setting `timingEnabled` to true), results in three time measures being calculated:

- `ITN` (Init Time Nano): The time spent in the method `init` in nanoseconds.
- `LTN` (Learn Time Nano): The time spent in the method `learn` in nanoseconds.
- `PTN` (Predict Time Nano): The time spent in the method `predict` in nanoseconds.

Since different predictors split the processing differently between the three methods, time comparison should be based on the sum of the three methods rather than that of a single one.

■ **Example 4.4** The following code shows how to use `PerfEvalExp` to compare the time performance of two algorithms ADA and HRG (note that no learning performance measures are used). The experiment is repeated ten times and the default ratio of removed edges is used (0.1).

```
#include <linkpred.hpp>
#include <iostream>
```

```
#include <vector>
#include <memory>
using namespace LinkPred;
class Factory: public PEFactory<> {
public:
  virtual std::vector<std::shared_ptr<LPredictor<>>>
     getPredictors(std::shared_ptr<UNetwork<> const> obsNet) {
    std::vector<std::shared_ptr<LPredictor<>>> prs;
    prs.push_back(std::make_shared<UADAPredictor<>>(obsNet));
    prs.push_back(std::make_shared<UHRGPredictor<>>(obsNet, 333))
        ;
    return prs;
  }
  virtual std::vector<std::shared_ptr<PerfMeasure<>>>
     getPerfMeasures(TestData<> const & testData) {
    std::vector<std::shared_ptr<PerfMeasure<>>> pms;
    return pms;
  }
  virtual ~Factory() = default;
};
int main(int argc, char*argv[]) {
  PerfeEvalExpDescp<> ped;
  ped.refNet = UNetwork<>::read("Zakarays_Karate_Club.edges");
  ped.nbTestRuns = 10;
  ped.seed = 777;
  ped.timingEnabled = true;
  auto factory = std::make_shared<Factory>();
  PerfEvalExp<> exp(ped, factory);
  exp.run();
  return 0;
}
```

The output of this code is as follows:

```
# n: 34 m: 78
#ratio  ITNADA  ITNHRG  LTNADA  LTNHRG  PTNADA  PTNHRG  TTNADA  TTNHRG
0.10    344     608972  145     2242406166      32789   16438   33278
   2243031576
0.10    309     137817  74      1528222331      9713    22062   10096
   1528382210
0.10    305     139456  78      1505922677      8470    16297   8853
   1506078430
0.10    315     181203  77      2240478339      8308    21287   8700
   2240680829
0.10    465     142991  80      1861618290      9255    17122   9800
   1861778403
0.10    293     141502  76      1623068910      9964    19323   10333
   1623229735
0.10    319     145763  80      1628975047      10742   27264   11141
   1629148074
0.10    298     146415  78      1974976422      9563    18218   9939
   1975141055
0.10    287     142000  80      1729851112      9260    15397   9627
   1730008509
0.10    290     146339  78      2402748124      9593    15082   9961
   2402909545
#Time: 18745.5 ms
```

■

R The time spent in computing the performance measures is not computed as it is not part of the link prediction task.

R Enabling timing causes automatically the computation of scores for all links in the test set. If you add a performance measure that does not require these scores but rather calls directly the link predictor methods (such as top-precision with the option `useTopMethod` enabled), then the time spent in these methods is not measured.

4.3.2 The class `PerfEvaluator`

The class `PerfEvaluator` offers more flexibility than `PerfEvalExp`, since it takes the object `TestData` as input. However, `PerfEvalExp` performs a single iteration comparison, and it is up to he user to repeat the experiment. To use the class `PerfEvaluator`, we proceed as follows:

1. First, the `TestData` object is passed as parameter to the constructor:
   ```
   PerfEvaluator<> perf(testData);
   ```
2. Add the link predictors:
   ```
   perf.addPredictor(std::make_shared<ADAPredictor<>>(testData
       .getObsNet()));
   perf.addPredictor(std::make_shared<CNEPredictor<>>(testData
       .getObsNet()));
   ```
3. Add the performance measures:
   ```
   perf.addPerfMeasure(std::make_shared<ROC<>>());
   perf.addPerfMeasure(std::make_shared<PR<>>());
   ```
   Notice that this step can be exchanged or interleaved with the previous one.
4. Run the evaluation (the predictors are initialized by the performance evaluator):
   ```
   perf.eval();
   ```
   The evaluator can be set to take time measurements by enabling timing before running the method `eval`:
   ```
   perf.setTimeEnabled(true); // Enable timing. Timing is
       disabled by default.
   perf.eval();
   ```
   Three time measures are calculated by `PerfEval`:
   - `ITN` (Init Time Nano): The time spent in the method `init` in nanoseconds.
   - `LTN` (Learn Time Nano): The time spent in the method `learn` in nanoseconds.
   - `PTN` (Predict Time Nano): The time spent in the method `predict` in nanoseconds.

   Since different predictors split the processing differently between the three methods, time comparison should be based on the sum of the three methods rather than that of a single one.

5. Finally, retrieve the performance values. The class `PerfEval` provides a range for retrieving results: `resultsBegin()` and `resultsEnd()`. The provided iterator points to a pair the first element of which is the name of the performance measure (of type `std::string`), and the second element is the value of the measure (of type **double**). Since there are several predictors, the name of the performance measures results reported is the concatenation of the performance name (`measure->getName()`) and the predictor's name (`predictor->getName()`).

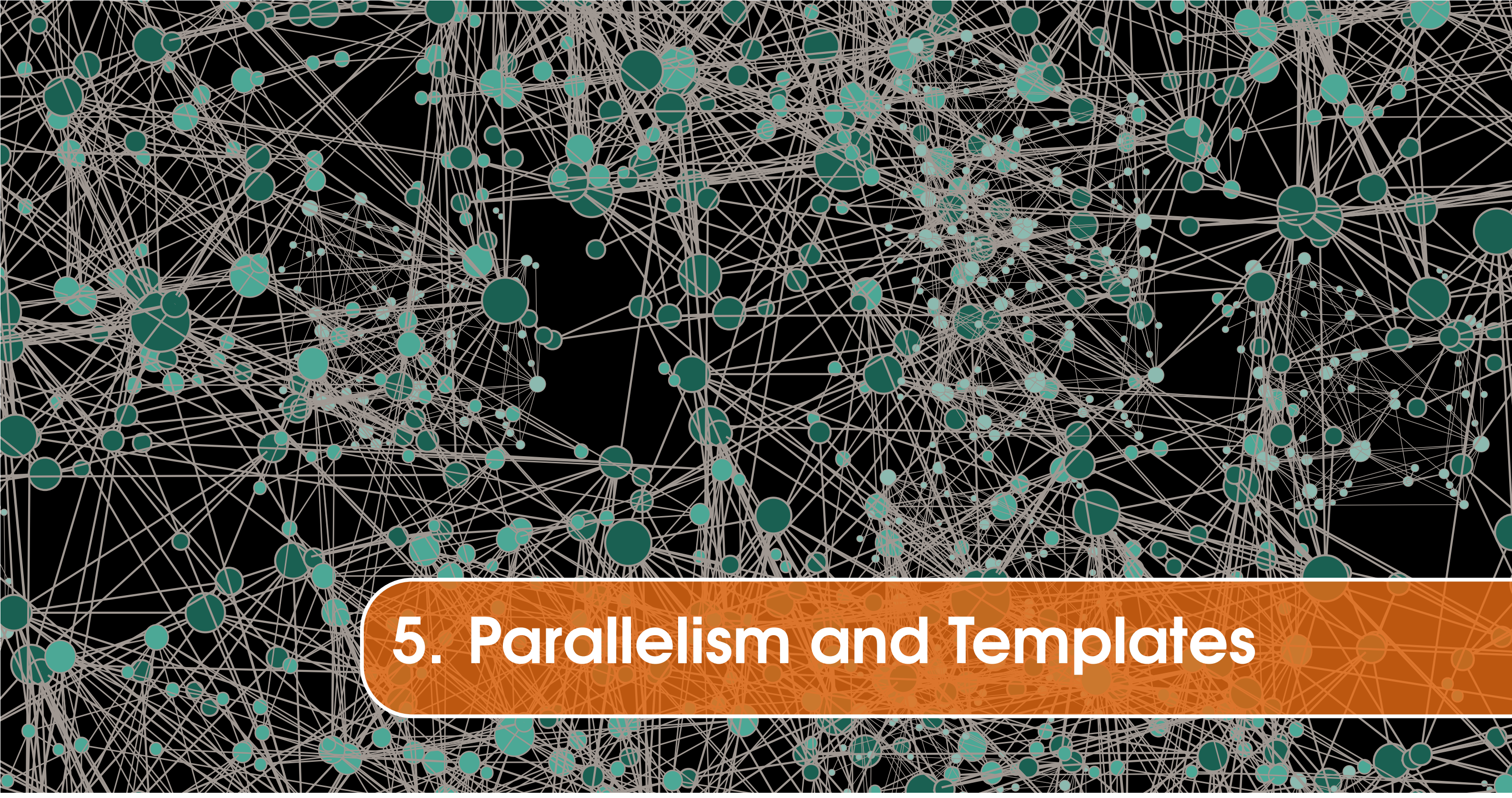
The following code shows how to use `PerfEval` to evaluate the performance of two link predictors using two measures.

```
#include "linkpred.hpp"
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  std::string netFileName(argv[1]);
  long int seed = std::atol(argv[2]);
  std::size_t nbTests = std::atol(argv[3]);
  RandomGen rng(seed);
  auto fullNet = UNetwork<>::read(netFileName, false, true);
  for (std::size_t i = 0; i < nbTests; i++) {
    auto testData = NetworkManipulator<>::createTestData(fullNet,
        0.1, 0, true, true, 0, true, 0, FN, TN, rng.getInt());
    PerfEvaluator<> perf(testData);

    perf.addPredictor(std::make_shared<UADAPredictor<>>(testData.
       getModNet()));
    perf.addPredictor(std::make_shared<UCNEPredictor<>>(testData.
       getModNet()));

    perf.addPerfMeasure(std::make_shared<ROC<>>());
    perf.addPerfMeasure(std::make_shared<PR<>>());

    perf.eval();

    if (i == 0) {
      std::cout << "#";
      for (auto it = perf.resultsBegin(); it != perf.resultsEnd()
         ; ++it) {
        std::cout << it->first << "\t";
      }
      std::cout << std::endl;
    }
    for (auto it = perf.resultsBegin(); it != perf.resultsEnd();
       ++it) {
      std::cout << std::setprecision(4) << it->second << "\t";
    }
    std::cout << std::endl;
  }
  return 0;
}
```

This is an example output of this code:

```
#ADAPR  ADAROC  CNEPR   CNEROC
0.8064  0.9694  0.8036  0.9637
0.7427  0.9878  0.6466  0.9704
```

```
0.8079  0.9687  0.7557  0.9573
0.7892  0.991   0.7871  0.9839
0.8453  0.9703  0.8122  0.9677
0.729   0.8982  0.6902  0.8964
0.7356  0.9768  0.6272  0.9557
0.8014  0.9641  0.7294  0.9504
0.7796  0.9614  0.7378  0.9624
0.6416  0.9348  0.5392  0.9221
```



5. Parallelism and Templates

This chapter deals with time performance issues and how to harness the power of LinkPred on parallel/distributed machines. This may become a necessity when dealing with large data as predicting links in large networks can be time and memory consuming even for the most efficient of algorithms. We will also discuss template arguments of LinkPred classes and how to add new instantiations to the library.

5.1 Parallelism

LinkPred offers two types of parallelism, shared memory parallelism using OpenMP, and distributed parallelism via MPI. These two types of parallelism can be used in conjunction or separately, or completely disabled at compilation time.

5.1.1 Shared memory parallelism

Most LinkPred classes support shared memory parallelism using OpenMP for the computationally intensive parts of their code. Enabling parallelism can result in significant improvement in running time. However, depending on the algorithms implemented in the methods under consideration, parallelism may result in different degrees of speedup. For example, Figure 5.1 shows the running time speed up obtained using parallel execution of four link predictors on the Yeast network. Once can see that different algorithms exhibit different speed ups depending on the details of the prediction procedure.

Instead of a using a ”global switch” to enable and disable parallelism, LinkPred offers a fine grain control of parallelism at the object level. This allows more flexibility to handle different use scenarios. To turn on parallelism for a predictor, we need to call the method `setParallel`. Notice that by default parallelism is turned off in all classes.

```
predictor->setParallel(true);
```

The same applies for performance measures:

```
measure->setParallel(true);
```

There are other classes in LinkPred that support parallelism, and they can all be set to run their code in parallel using the method `setParallel`.

In general, and especially in the case where several classes are set to run parallel code, it is important to allow for nested parallelism:

```
omp_set_nested(1);
```

This should typically be done at the start of the `main` function, or at least before running the LinkPred code.

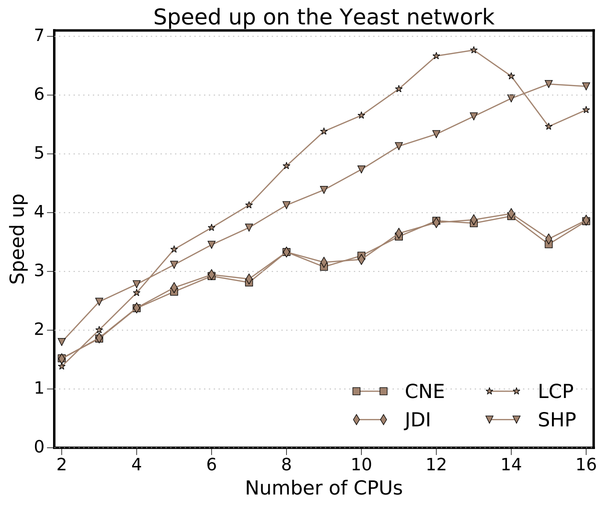


Figure 5.1: Runtime speed up of several link predictors on the Yeast network.

Choosing the level at which parallelism should be activated is important to achieve the best possible performance. In what follows, we present a number of scenarios that LinkPred users may face and the suggested parallelization strategies.

1. Running a single predictor on a single network: in this case, parallelism should activated at the predictor level as it is the only to take advantage of the parallel execution capability.
2. Evaluating a single link predictor: In general, performance evaluation involves running the link predictor multiple times with different training and test sets. Enabling parallelism at the predictor can lead to some improvement (depending on the type of the predictor). A better strategy, however, would be to parallelize the execution at the outer level, that is executing the predictor with different test data in parallel.
3. Evaluating the performance of several link predictors: This is similar to the previous case, except that we can run the predictors in parallel. This can be beneficial if the predictors have comparable runtime, but if there is a large discrepancy between runtimes it is better to prallelize at the predictor level or over test runs.

The following code how to compute the scores for all negative links fo a network using CNE predictor in parallel.

```
#include <linkpred.hpp>
#include <iostream>
#include <algorithm>
using namespace LinkPred;
```

```
int main(int argc, char*argv[]) {
  omp_set_nested(1); // enable nested parallelism
  auto net = UNetwork<>::read("Infectious.edges");
  UCNEPredictor<> predictor(net);
  predictor.setParallel(true);
  predictor.init();
  predictor.learn();
  std::vector<double> scores;
  scores.resize(net->getNbNonEdges());
  auto its = predictor.predictNeg(scores.begin());
  std::cout << "#Start\tEnd\tScore\n";
  std::size_t i = 0;
  for (auto it = its.first; it != its.second; ++it, i++) {
    std::cout << net->getLabel(net->start(*it)) << "\t"<< net->
        getLabel(net->end(*it)) << "\t" << scores[i] << std::endl;
  }
  return 0;
}
```

The following is an extract of this code's output:

```
#Start  End     Score
100     10      0
100     11      0
100     113     7
100     12      0
100     13      0
100     14      0
100     15      0
100     16      0
100     107     10
100     23      0
...
```

5.1.2 Distributed parallelism

Distributed parallelism is implemented in LinkPred using MPI (Message Passing Interface) unless this deactivated during compilation time (the corresponding flag is `WITH_MPI`). Several (but not all) link predictors offer distributed implementations of the methods `predictNeg` and `top`. Note, however, that the network data structure is not distributed, and consequently, each processor must have access to the whole network data either by reading it from file or otherwise.

The `ROC` performance measure supports distributed processing when using the streaming method, and the same applies to `TPR` (top-precision) when using the `top` method. In both cases, the result of the performance measure is only available at processor 0. The default methods in `ROC` and `TPR` as well as the `PR` class dot not support distributed parallelism.

Similarly to shared memory parallelism, distributed processing is controlled at the object level. To activate/deactivate distributed processing for a given predictor just set the attributed `distributed`. For example:

```
predictor->setDistributed(true);
```

The following code shows how to find the top *k* edges distributively.

```
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char*argv[]) {
  MPI_Init(&argc, &argv);
  std::size_t k = 10;
  auto net = UNetwork<>::read("Infectious.edges");
  URALPredictor<> predictor(net);
  predictor.setComm(MPI_COMM_WORLD); // Optional when using the
     default communicator MPI_COMM_WORLD
  predictor.setDistributed(true);
  predictor.init();
  predictor.learn();
  std::vector<typename UNetwork<>::EdgeType> edges;
  edges.resize(k);
  std::vector<double> scores;
  scores.resize(k);
  k = predictor.top(k, edges.begin(), scores.begin());

  int procID;
  MPI_Comm_rank(MPI_COMM_WORLD, &procID);
  if (procID == 0) {
    std::cout << "#Start\tEnd\tScore\n";
  }
  for (std::size_t i = 0; i < k; i++) {
    std::cout << net->getLabel(net->start(edges[i])) << "\t"
        << net->getLabel(net->end(edges[i])) << "\t" << scores[i]
        << std::endl;
  }
  MPI_Finalize();
  return 0;
}
```

If you compile this code into the executable `ratop`, for example, the program can be run as follows:

```
mpirun -n 4 ./raltop
```

This is an example output of this code:

```
#Start  End     Score
169     178     0.912642
144     142     0.886008
51      39      0.985052
265     297     0.811915
300     295     0.806431
197     237     0.836456
257     299     0.864928
257     294     0.887479
261     292     0.973033
389     367     0.965622
```

R In this example, you may need to set `OMP_NUM_THREADS` to 1, because we are not using shared memory parallelism. On Linux, this can be achieved by running the command `export OMP_NUM_THREADS=1`.

The following shows how to compute the area under the ROC in a distributed way:

```cpp
#include <linkpred.hpp>
#include <iostream>
using namespace LinkPred;
int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  int procID = 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &procID);
  std::string netFileName(argv[1]);
  auto refNet = UNetwork<>::read(netFileName);
  auto testData = NetworkManipulator<>::createTestData(refNet,
     0.1, 0, false, true, 0, true, 0, FN, TN, 777, false);
  testData.lock();
  auto predictor = std::make_shared<UADAPredictor<>>(testData.
     getObsNet());
  predictor->init();
  predictor->learn();
  auto predResults = std::make_shared<PredResults<>>(testData,
     predictor);
  auto roc = std::make_shared<ROC<>>("ROC");
  roc->setComm(MPI_COMM_WORLD); // Optional when using the
     default communicator MPI_COMM_WORLD
  roc->setParallel(true);
  roc->setDistributed(true);
  roc->setStrmEnabled(true);
  PerfResults res;
  roc->eval(predResults, res);
  if (procID == 0) {
    std::cout << "#ROCAUC␣(streaming):␣" << res.at(roc->getName()
      ) << std::endl;
  }
  MPI_Finalize();
  return 0;
}
```

If you compile this code into the executable `rocstrmdist`, for example, the program can be run as follows:

```
mpirun -n 4 ./rocstrmdist AS_Internet.edges
```

This is an example output of this code:

```
#ROCAUC (streaming): 0.8466
```

5.2 Templates

LinkPred is mainly a pre-instantiated template library, a design choice that offers fast compilation while keeping the library extensible and customizable. The default templates arguments used in the pre-instantiated classes are chosen to give the best performance possible, but it is often the case that classes are instantiated with arguments other then the default ones. For example, the class `UNetwork` is instantiated with `std::string` as the default type for nodes labels, and it is also instantiated with **unsigned int**. The latter is a more restrictive option but may be useful for saving memory, especially that most network datasets have integer node IDs.

R You can find the list of pre-instantiated classes in the file `include/instantiations.hpp`.

The class templates instantiations found in `instantiations.hpp` can be readily used. If you want to instantiate a class with a template argument other than those already available, you have to edit the file `instantiations.hpp`.

■ **Example 5.1** If you want to instantiate the class `UNetwork` with **short int**, locate the following section in `instantiations.hpp`:

```
#ifdef UNETWORK_CPP
template class UNetwork<>;
template class UNetwork<unsigned int>;
#endif
```

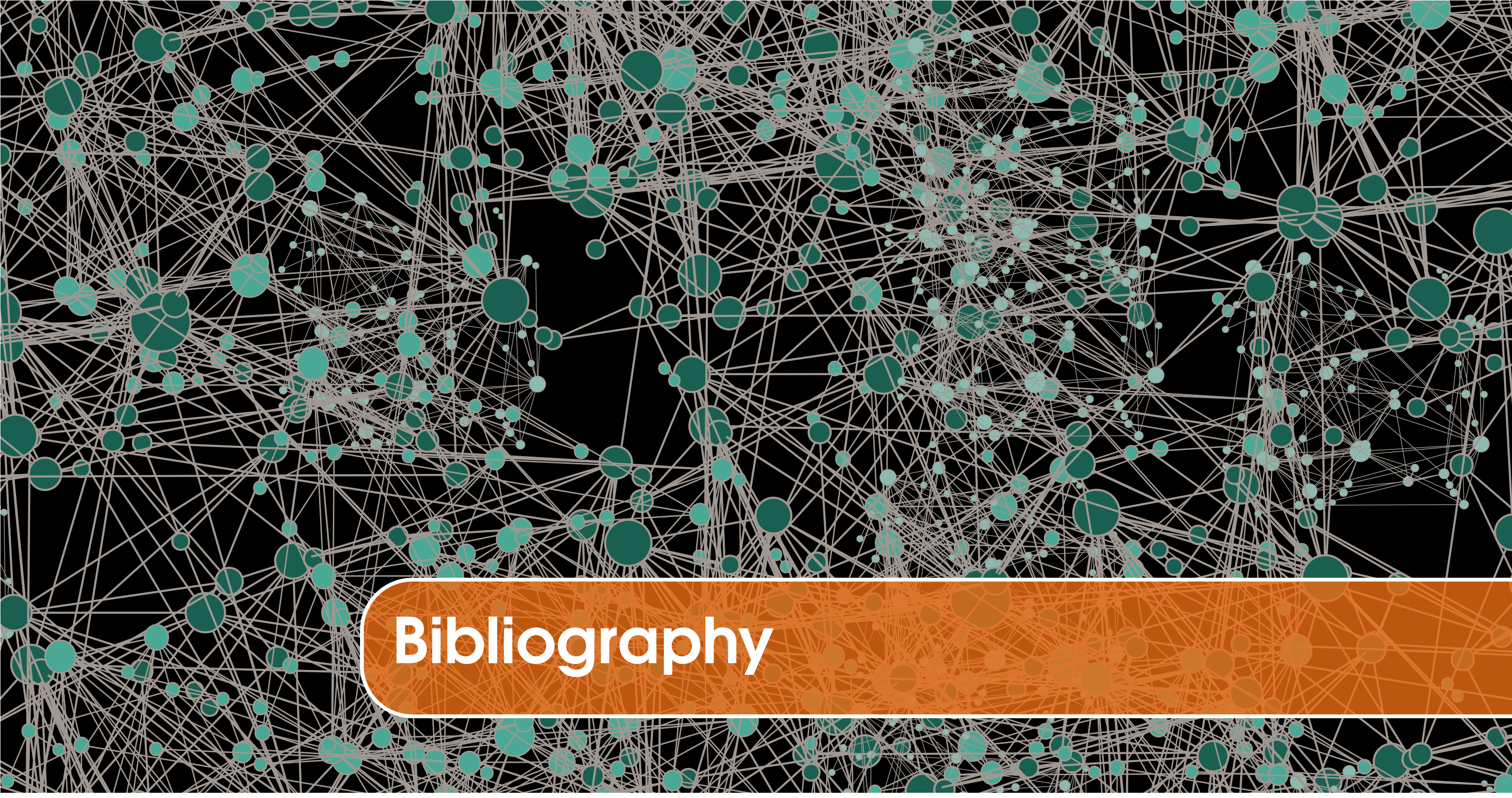
and change it to (stay within the `ifdef`):

```
#ifdef UNETWORK_CPP
template class UNetwork<>;
template class UNetwork<unsigned int>;
template class UNetwork<short int>;
#endif
```

You need than to recompile the library to use the new instantiation. ■

R Note that adding a new instantiation may require to add other new instantiations in classes that use it. For example, if you want to use the new instantiation `UNetwork<short int>` with the CNE predictor, you need to add a new instantiation of the class `UCNEPredictor` as follows:

```
#ifdef UCNEPREDICTOR_CPP
template class UCNEPredictor<>;
template class UCNEPredictor<UNetwork<>, typename
    UNetwork<>::NonEdgeIterator>;
template class UCNEPredictor<UNetwork<short int>,
    typename UNetwork<short int>::NonEdgeIterator>; //
    new instantiation
#endif
```



Bibliography

[1] Vladimir Batagelj and Andrej Mrvar. *Pajek Datasets*. `http://vlado.fmf.uni-lj.si/pub/networks/data`. 2006 (cited on page 10).

[2] Aaron Clauset, Cristopher Moore, and Mark EJ Newman. “Hierarchical structure and the prediction of missing links in networks”. In: *Nature* 453.7191 (2008), pages 98–101 (cited on pages 9, 38).

[3] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. “Power-law distributions in empirical data”. In: *SIAM review* 51.4 (2009), pages 661–703 (cited on pages 9, 40).

[4] Jesse Davis and Mark Goadrich. “The Relationship Between Precision-Recall and ROC Curves”. In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML ’06. Pittsburgh, Pennsylvania, USA: ACM, 2006, pages 233–240. ISBN: 1-59593-383-2 (cited on page 53).

[5] Roger Guimerà and Marta Sales-Pardo. “Missing and spurious interactions and the reconstruction of complex networks”. In: *Proceedings of the National Academy of Sciences* 106.52 (2009), pages 22073–22078 (cited on pages 9, 38).

[6] William W. Hager and Hongchao Zhang. “Algorithm 851: CG_DESCENT, a conjugate gradient method with guaranteed descent”. In: *ACM Trans. Math. Softw.* 32.1 (Mar. 2006), pages 113–137. ISSN: 0098-3500 (cited on page 9).

[7] Lorenzo Isella et al. “What’s in a crowd? Analysis of face-to-face behavioral networks”. In: *Journal of Theoretical Biology* 271.1 (2011), pages 166–180. ISSN: 0022-5193 (cited on page 10).

[8] Jérôme Kunegis. “KONECT: The Koblenz Network Collection”. In: *Proceedings of the 22Nd International Conference on World Wide Web*. WWW ’13 Companion. Rio de Janeiro, Brazil: ACM, 2013, pages 1343–1350. ISBN: 978-1-4503-2038-2. URL: `http://konect.uni-koblenz.de/networks` (cited on page 10).

[9] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014 (cited on page 10).

[10] Zhen Liu et al. “Correlations between community structure and link formation in complex networks”. In: *PloS one* 8.9 (2013) (cited on pages 9, 38).

[11] Linyuan Lü and Tao Zhou. “Link prediction in complex networks: A survey”. In: *Physica A: Statistical Mechanics and its Applications* 390.6 (2011), pages 1150–1170 (cited on page 34).

[12] Fragkiskos Papadopoulos, Constantinos Psomas, and Dmitri Krioukov. “Network mapping by replaying hyperbolic growth”. In: *IEEE/ACM Transactions on Networking (TON)* 23.1 (2015), pages 198–211 (cited on pages 9, 39).

[13] Fragkiskos Papadopoulos et al. “Popularity versus similarity in growing networks”. In: *Nature* 489.7417 (2012), pages 537–540 (cited on pages 9, 39).

[14] Ryan A. Rossi and Nesreen K. Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization”. In: *AAAI*. 2015. URL: `http://networkrepository.com` (cited on page 10).

[15] Damian Szklarczyk et al. “STRING v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets”. In: *Nucleic Acids Research* 47.D1 (Nov. 2018), pages D607–D613. ISSN: 0305-1048 (cited on page 10).

[16] Peng Wang et al. “Link prediction in social networks: the state-of-the-art”. In: *Science China Information Sciences* 58.1 (2015), pages 1–38 (cited on page 34).

[17] Wayne W Zachary. “An information flow model for conflict and fission in small groups”. In: *Journal of anthropological research* 33 (1977), pages 452–473 (cited on page 10).

[18] R. Zafarani and H. Liu. *Social Computing Data Repository at ASU*. 2009. URL: `http://socialcomputing.asu.edu` (cited on page 10).

[19] Marinka Zitnik et al. *BioSNAP Datasets: Stanford Biomedical Network Dataset Collection*. `http://snap.stanford.edu/biodata`. Aug. 2018 (cited on page 10).