By *Jure Leskovec*

STANFORD UNIVERSITY

SNAP for C++
SNAP for Python
SNAP Datasets
What's new
People
Papers
Projects
Citing SNAP
Links
About
Contact us

Open positions

Open research positions in **SNAP** group are available here.

Quick Introduction to SNAP

Graph and Network Types

SNAP supports *graphs* and *networks*. *Graphs* describe topologies. That is nodes with unique integer ids and directed/undirected/multiple edges between the nodes of the graph. *Networks* are graphs with data on nodes and/or edges of the network. Data types that reside on nodes and edges are simply passed as template parameters which provides a very fast and convenient way to implement various kinds of networks with rich data on nodes and edges.

Graph types in SNAP:

```
TUNGraph: undirected graph (single edge between an unordered pair of nodes)
TNGraph: directed graph (single directed edge between an ordered pair of nodes)
TNEGraph: directed multi-graph (multiple directed edges between a pair of nodes)
```

Network types in SNAP:

```
TNodeNet<TNodeData>: like TNGraph, but with TNodeData object for each node
TNodeEDatNet<TNodeData, TEdgeData>: like TNGraph, but with TNodeData on each node
  and TEdgeData on each edge
TNodeEdgeNet<TNodeData, TEdgeData>: like TNEGraph, but with TNodeData on each node
  and TEdgeData on each edge
TNEANet: like TNEGraph, but with attributes on nodes and edges. The attributes
  are dynamic in that they can be defined at runtime
TBigNet<TNodeData>: memory efficient implementation of TNodeNet that avoids
  memory fragmentation and handles billions of edges with sufficient RAM
  being available
```

An example of how to create a network with rich data on its nodes can be found in `imdbnet.h`, which is part of SNAP. `imdbnet.h` provides an example of a IMDB actors-to-movies bipartite network with movie and actor data on the nodes of the network.

Graph Creation

Example of how to create and use a directed graph:

```
// create a graph
PNGraph Graph = TNGraph::New();
Graph->AddNode(1);
Graph->AddNode(5);
Graph->AddNode(32);
Graph->AddEdge(1,5);
Graph->AddEdge(5,1);
Graph->AddEdge(5,32);
```

Nodes have explicit (and arbitrary) node ids. There is no restriction for node ids to be contiguous integers starting at 0. In a multi-graph `TNEGraph` edges have explicit integer ids. In `TUNGraph` and `TNGraph` edges have no explicit ids -- edges are identified by a pair node ids.

SNAP uses smart-pointers (`TPt`) so there is not need to explicitly free (delete) graph objects. They self-destruct when they are not needed anymore. Prefix `P` in the class name stands for a pointer, while `T` means a type.

Networks are created in the same way as graphs.

Iterators

Many SNAP operations are based on node and edge iterators which allow for efficient implementation of algorithms that work on networks regardless of their type (directed, undirected, graphs, networks) and specific implementation.

Some examples of iterator usage are shown below:

```
// create a directed random graph on 100 nodes and 1k edges
PNGraph Graph = TSnap::GenRndGnm<PNGraph>(100, 1000);
// traverse the nodes
for (TNGraph::TNodeI NI = Graph->BegNI(); NI < Graph->EndNI(); NI++) {
```

```
  printf("node id %d with out-degree %d and in-degree %d\n",
    NI.GetId(), NI.GetOutDeg(), NI.GetInDeg());
}
// traverse the edges
for (TNGraph::TEdgeI EI = Graph->BegEI(); EI < Graph->EndEI(); EI++) {
  printf("edge (%d, %d)\n", EI.GetSrcNId(), EI.GetDstNId());
}
// we can traverse the edges also like this
for (TNGraph::TNodeI NI = Graph->BegNI(); NI < Graph->EndNI(); NI++) {
  for (int e = 0; e < NI.GetOutDeg(); e++) {
    printf("edge (%d %d)\n", NI.GetId(), NI.GetOutNId(e));
  }
}
```

All graph and network datatypes define node and edge iterators. In general graph/network data types use the following functions to return various iterators:

```
BegNI(): iterator to first node
EndNI(): iterator to one past last node
GetNI(u): iterator to node with id u
BegEI(): iterator to first edge
EndEI(): iterator to one past last edge
GetEI(u,v): iterator to edge (u,v)
GetEI(e): iterator to edge with id e (only for multigraphs)
```

In general node iterators provide the following functionality:

```
GetId(): return node id
GetOutDeg(): return out-degree of a node
GetInDeg(): return in-degree of a node
GetOutNId(e): return node id of the endpoint of e-th out-edge
GetInNId(e): return node id of the endpoint of e-th in-edge
IsOutNId(int NId): do we point to node id n
IsInNId(n): does node id n point to us
IsNbhNId(n): is node n our neighbor
```

In addition, iterators over networks also allow for easy access to the data:

```
GetDat(): return data type TNodeData associated with the node
GetOutNDat(e): return data associated with node at endpoint of e-th out-edge
GetInNDat(e): return data associated with node at endpoint of e-th in-edge
GetOutEDat(e): return data associated with e-th out-edge
GetInEDat(e): return data associated with e-th in-edge
```

For additional information on node (`TNodeI`) and edge (`TEdgeI`) iterators see the implementation of various graph and network data types in `graph.h` and `network.h`.

Input/Output

With SNAP it is easy to save and load networks in various formats. Internally SNAP saves networks in compact binary format but functions for loading and saving networks in various other text and XML formats are also available (see `gio.h`).

For example:

```
// generate a network using Forest Fire model
PNGraph G = TSnap::GenForestFire(1000, 0.35, 0.35);
// save and load binary
{ TFOut FOut("test.graph"); G->Save(FOut); }
{ TFIn FIn("test.graph"); PNGraph G2 = TNGraph::Load(FIn); }
// save and load from a text file
TSnap::SaveEdgeList(G2, "test.txt", "Save as tab-separated list of edges");
PNGraph G3 = TSnap::LoadEdgeList("test.txt", 0, 1);
```

Manipulating Graphs and Networks

SNAP provides rich functionality to efficiently manipulate graphs and networks. Functions are implemented as a part of namespace `TSnap`. All functions support any graph/network type.

For example:

```
// generate a network using Forest Fire model
PNGraph G = TSnap::GenForestFire(1000, 0.35, 0.35);
// convert to undirected graph TUNGraph
PUNGraph UG = TSnap::ConvertGraph<PUNGraph>(G);
```

```
// get largest weakly connected component of G
PNGraph WccG = TSnap::GetMxWcc(G);
// get a subgraph induced on nodes {0,1,2,3,4,5}
PNGraph SubG = TSnap::GetSubGraph(G, TIntV::GetV(0,1,2,3,4));
// get 3-core of G
PNGraph Core3 = TSnap::GetKCore(G, 3);
// delete nodes of degree 10
TSnap::DelDegKNodes(G, 10);
```

For more examples of how to work with networks see example applications in `SnapSamples` directory of the SNAP distribution. Also refer to description of files to see what functionality is implemented in various SNAP files.

Computing Structural Properties

SNAP provides rich functionality to efficiently compute structural properties of networks. Functions are implemented as a part of namespace `TSnap`. All functions support any graph/network type.

For example

```
// generate a Preferential Attachment graph on 1000 nodes and node out degree of 3
PUNGraph G = TSnap::GenPrefAttach(1000, 3);
// get distribution of connected components (component size, count)
TVec<TPair<TInt, TInt> > CntV; // vector of pairs of integers (size, count)
TSnap::GetWccSzCnt(G, CntV);
// get degree distribution pairs (degree, count)
TSnap::GetOutDegCnt(G, CntV);
// get first eigenvector of graph adjacency matrix
TFltV EigV; // vector of floats
TSnap::GetEigVec(G, EigV);
// get diameter of G
TSnap::GetBfsFullDiam(G);
// count the number of triads in G, get the clustering coefficient of G
TSnap::GetTriads(G);
TSnap::GetClustCf(G);
```

For more examples of how to work with networks see example applications in `SnapSamples` directory of the SNAP distribution. Also refer to description of files to see what functionality is implemented in various SNAP files.