Chapter 1

SNAP

Contributed by Justin DeBenedetto

1.1 Background

The Stanford Network Analysis Project (SNAP) [1] has been actively developed since 2004. It was developed in order to handle large graphs efficiently, include many common graph algorithms, and allow dynamic network changes. The original developers felt that there was a need for a tool which satisfied all of these criteria simultaneously. They felt that each of the existing tools failed in one or more of those categories.

SNAP is available in C++ and as a Python module. It is open source and available at the http://snap.stanford.edu/ website. This website has both the C++ and Python available for download as well as a large network database collection consisting of over 50 large network datasets. These networks include online social networks, communication networks, scientific citation networks, and collaboration networks.

SNAP has 8 graph types included, 20 graph generation methods, and over 100 graph algorithms. They consider NetworkX to be similar to SNAP and compare against it. They find that SNAP runs one to two orders of magnitude faster while using fifty times less memory. Both NetworkX and SNAP can be used for Python and are intended to be run on a single machine. They do point out that NetworkX has more flexibility if that is required by the user.

1.2 Expressing Graphs

Graphs can be read in from files in various ways. The graph file can have one edge per line (source target) or one node per line (source target1 target2 ...). SNAP also has built in methods for loading directly from other established systems such as DyNet and Pajek.

The user can also build graphs easily. There are twenty graph generators available within SNAP and the user is able to define their own as well. If they prefer to build their graphs one node or edge at a time, there are functions that handle that as well. Regardless of how the graph is read into SNAP, it can be saved into SNAP's own binary representation for faster loading and saving.

SNAP organizes its various graph structures into "containers". Each container provides the same outward appearance to functions calling them in order to allow algorithms to be written once and run on all types. Internally, each container stores the graphs in a way which is efficient for the specific type of graph. SNAP also distinguishes between labelled and unlabelled edges/nodes,

referring to those without labels as graphs and those with labels as networks. The types of containers are described below [1]:

Graph Containers	
TUNGraph	Undirected graphs
TNGraph	Directed graphs
TNEGraph	Directed multigraphs
TBPGraph	Bipartite graphs
Network Containers	
TNodeNet	Directed graphs with node attributes
TNodeEDatNet	Directed graphs with node and edge attributes
TNodeEdgeNet	Directed multigraphs with node and edge attributes
TNEANet	Directed multigraphs with dynamic node and edge attributes

SNAP seeks to take advantage of the benefits of vectors and hash tables where they can. This interaction can be seen in the following figure taken from [1]:

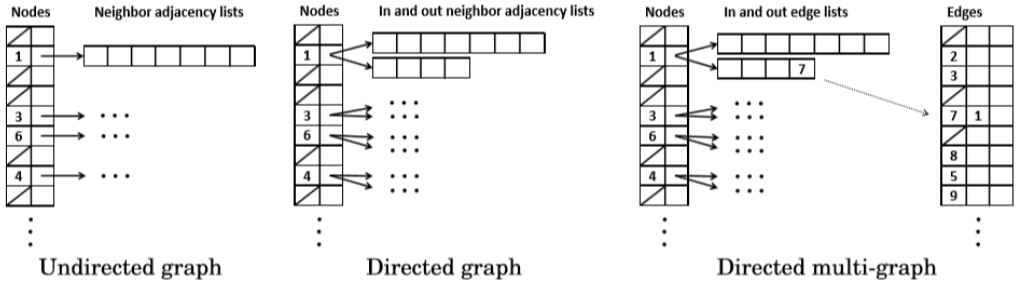

Fig. 2. A diagram of graph data structures in SNAP. Node ids are stored in a hash table, and each node has one or two associated vectors of neighboring node or edge ids.

1.3 Syntax

A list of common methods used in SNAP, taken from their documentation [1], is shown here:

Table II. Common Graph and Network Methods.

Nodes	
`AddNode`	Adds a node
`DelNode`	Deletes a node
`IsNode`	Tests, if a node exists
`GetNodes`	Returns the number of nodes
Edges	
`AddEdge`	Adds an edge
`DelEdge`	Deletes an edge
`IsEdge`	Tests, if an edge exists
`GetEdges`	Returns the number of edges
Graph Methods	
`Clr`	Removes all nodes and edges
`Empty`	Tests, if the graph is empty
`Dump`	Prints the graph in a human readable form
`Save`	Saves a graph in a binary format to disk
`Load`	Loads a graph in a binary format from disk
Node and Edge Iterators	
`BegNI`	Returns the start of a node iterator
`EndNI`	Returns the end of a node iterator
`GetNI`	Returns a node (iterator)
`NI++`	Moves the iterator to the next node
`BegEI`	Returns the start of an edge iterator
`EndEI`	Returns the end of an edge iterator
`GetEI`	Returns an edge (iterator)
`EI++`	Moves the iterator to the next edge

When using the Python library, each of those methods can be called in standard Python way. If G is a SNAP graph, then one can simply type "G.AddNode(1)" to add a node to the graph with node ID 1. A similar style is used for the other methods available where they are called on a graph object and often have node IDs as parameters.

1.4 Key Graph Primitives

Some of the key graph primitives employed by SNAP are the iterators and adding and deleting nodes and edges as shown above in the common methods. These allow the user to build a graph or remove parts of a graph as well as iterate through the existing nodes or edges in the graph. Iterating is employed in many graph algorithms, so iterators are returned easily to the user masking the container specific implementations of how the iterators function.

1.5 Execution Model

SNAP is designed to be run on a single machine and most of the included algorithms run sequentially on a single core. This allows the user to run multiple instances of SNAP at once if desired. When

the user runs their program, the chosen graph container determines the underlying data structure, and the methods they call run in the same fashion on top of this regardless of which underlying data structure is used. This allows maximum flexibility in switching container types in a written program, however the user cannot easily convert one container type into another during a single execution run.

Most of their provided benchmarks and algorithms run sequentially and they aim to minimize memory requirements through using vectors and hash tables. When the user wishes to parallelize their code, a single machine is assumed and multiple cores are used for the desired parallelism. To use the parallel version, there is another system built on top of SNAP called RINGO [2]. RINGO uses a similar Python module setup while modifying the C++ backend to implement the parallelism that provides a speed boost. Since this system has its own properties, the details are not described here.

1.6 Examples

The SNAP tutorial [1] provides the following code to get started:

Get degree distribution pairs (out-degree, count):

```
>>> snap.GetOutDegCnt(G9, CntV)
>>> for p in CntV:
>>>     print "degree %d: count %d" % (p.GetVal1(), p.GetVal2())
```

Generate a Preferential Attachment graph on 100 nodes and out-degree of 3:

```
>>> G10 = snap.GenPrefAttach(100, 3)
```

Define a vector of floats and get first eigenvector of graph adjacency matrix:

```
>>> EigV = snap.TFltV()
>>> snap.GetEigVec(G10, EigV)
>>> nr = 0
>>> for f in EigV:
>>>     nr += 1
>>>     print "%d: %.6f" % (nr, f)
```

Get an approximation of graph diameter:

```
>>> diam = snap.GetBfsFullDiam(G10, 10)
```

Count the number of triads:

```
>>> triads = snap.GetTriads(G10)
```

Get the clustering coefficient:

```
>>> cf = snap.GetClustCf(G10)
```

From these examples we can see how easy to is to generate a graph and obtain many common pieces of information about its properties. The graph generator used in this example is the preferential attachment graph generation method which grows a graph by adding one node at a time and attaching the new node to existing nodes with probability proportional to the degree of the existing node. There are many other graph generation methods that can be called, all of them using the same style syntax and only requiring a few parameters.

1.7 Conclusion

SNAP aims to provide a flexible framework for graph processing on a single machine and does well in this domain. It maintains much of the flexibility that NetworkX possesses while making huge gains in terms of reducing memory footprint and reducing runtime. Most of these gains can be attributed to the internal representations specific to each container, using vectors and hash

tables. Like NetworkX, SNAP includes many of the most common graph algorithms, generators, and functionality that make it easy to get started. Customization and adding new algorithms is relatively fast and portable due to the unified calls to any container, which allow an algorithm to be written once and applied to any valid container while still receiving the performance boost associated with the relevant container. SNAP has been cited many times and remains under active development.

Bibliography

[1] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[2] Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1105–1110. ACM, 2015.