SNAP: Small-world Network Analysis and Partitioning User's Guide Version 0.2

Kamesh Madduri
College of Computing
Georgia Institute of Technology
kamesh@cc.gatech.edu

June 1, 2008

Contents

1 Introduction **3**

2 Installation **4**
2.1 Configuring SNAP 4

3 Using the SNAP framework **5**

4 SNAP development **8**

1 Introduction

SNAP (Small-world Network Analysis and Partitioning) is an extensible parallel framework for exploratory analysis and partitioning of large-scale networks.

SNAP is implemented in C, uses POSIX threads and OpenMP primitives for parallelization, and targets sequential, multicore, and symmetric multiprocessor platforms. Our intent with SNAP is to provide a simple and intuitive interface for network analysis and application design, hiding the parallel programming complexity from the user. In addition to path-based, centrality, and community identification queries on large-scale graphs, we support commonly-used preprocessing kernels and quantitative measures that consider the global network topology.

Figure 1 gives an overview of the SNAP framework. SNAP can process weighted and unweighted graphs in several common text formats, and uses different data structures for storing the graph, depending on the problem we are looking to solve. On top of the graph data structures, we have efficient parallel implementations of fundamental graph kernels such as graph traversal, shortest paths, spanning tree, and connected components. The parallelism in these kernels is fine-grained and relies on the low diameter of small-world graphs. In addition, we consider the unbalanced degree distribution in the graph while distributing work among processors.

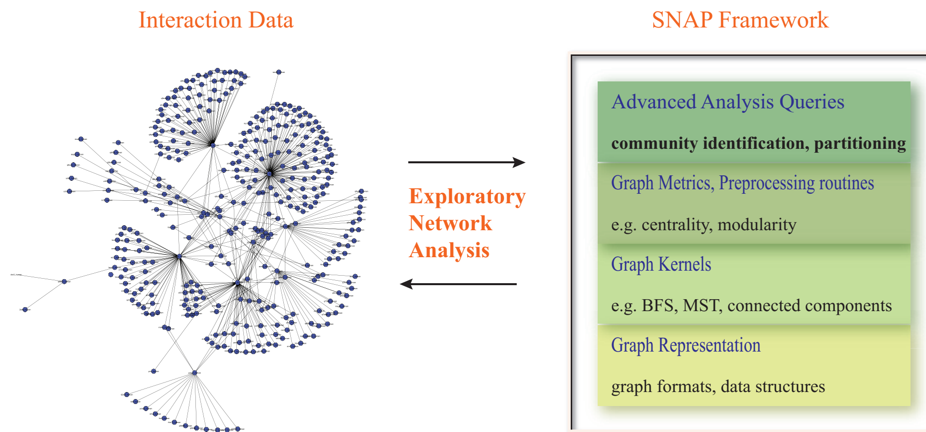

Figure 1: The SNAP Graph Framework

A novel contribution of SNAP is a collection of parallel implementations of social network analysis routines and queries. In particular, we designed new parallel approaches for the problem of community identification in large-scale networks. SNAP provides efficient implementations of these approaches; through a combination of algorithm engineering for small-world networks and parallel speedup, we show that these are typically 10-100X faster than state-of-the-art algorithms [1]. For more information on SNAP and research publications, please visit the SNAP home page [2].

For SNAP installation guidelines, please see Section 2. Section 3 briefly describes usage of the SNAP framework with a simple C code listing.

2 Installation

SNAP needs to be built from source files on various platforms. To compile and install SNAP 0.2, you will need an ANSI C compiler, and an environment providing standard UNIX tools, such as make and grep. For parallel support, the compiler should support OpenMP.

SNAP has been tested on Linux, AIX, Solaris, and Cygwin. It works on 32-bit as well as 64-bit platforms.

To install SNAP, follow the steps below:

1. Download snap-0.2.tar.gz from the SNAP Sourceforge home page.

2. Unpack the distribution.
   `% gunzip snap-0.2.tar.gz`
   `% tar xvf snap-0.2.tar`
   These commands unpack the SNAP package into a sub-directory `snap-0.2`.

3. Configure SNAP for your platform.
   `% mkdir snap-build-0.2`
   `% cd snap-build-0.2`
   `% ../snap-0.2/configure --prefix=${SNAPDIR}`
   As done above, we recommend building SNAP in a directory separate from the source tree you unpacked in Step 2. Here we use a directory named `snap-build-0.2`.
   The configure script tries to detect your operating system and CPU, and chooses a compiler and default compiler optimization flags accordingly. These choices will be displayed when the script finishes. If you wish to specify these flags yourself, or if configure is unable to choose a compiler or default flags, see Section 2.1 for more details.

4. Compile SNAP.
   `% make`
   The compile step should take about two to five minutes on current systems.

5. Install SNAP in `${SNAPDIR}`.
   `% make install`
   This step installs the SNAP library in the specified path. Check for the library in the path to make sure that the build completed successfully.

2.1 Configuring SNAP

The Configure step can be customized in many ways. The most commonly used options are discussed below. For a complete list, run (from the build tree in our previous example):
`% ../snap-0.2/configure --help`.

Build parallel programs
To build with OpenMP support, use the `--enable-openmp` option.

Overriding the default compiler and/or flags
To specify the compiler and/or compiler flags yourself, define the `CC` and/or `CFLAGS` environment variables accordingly. For example, to use the Intel C compiler to build SNAP with opti-

mization flags:
`% ../snap-0.2/configure CC=icc CFLAGS='-O3' ...` Please let us know if you find compiler and optimization flag combinations that improve on the defaults. The Configure step will try to auto-detect a Fortran 77 compiler to build some code associated with parallel random number generation. To override the F77 compiler or F77 compiler flags, use the F77 or FFLAGS environment variables, as with CC/CFLAGS above.

3 Using the SNAP framework

This section introduces main aspects of the SNAP framework, with a C code listing below. This serves as a simple example to test your installation.

```
#include "graph_defs.h"
#include "graph_gen.h"
#include "graph_kernels.h"
#include "graph_metrics.h"
#include "utils.h"

int main(int argc, char** argv) {

    char *infilename, *outfilename, *graph_type;
    FILE* fp;
    graph_t* g;

    long src;
    int curArgIndex;
    long numSrcs;
    int est_diameter;
    long i, j;
    long num_vertices_visited;

    /* Step 1: Parse command line arguments */
    if (argc < 3) {
        fprintf(stdout, "\nUsage: ./eval_BFS (-src <vertex ID (0 to n-1)>)"
                        "-infile <graph filename>"
                        " (-graph <graph type> -outfile <output filename>)\n");

        usage_graph_options();

        fprintf(stdout, "Using the -src option, specify the vertex ID to run BFS from.
                        "A random vertex is selected if the src is not specified.\n\n");
        exit(-1);
    }

    curArgIndex = 0;
    infilename  = (char *) calloc(500, sizeof(char));
    outfilename = (char *) calloc(500, sizeof(char));
    graph_type  = (char *) calloc(500, sizeof(char));

    strcpy(outfilename, "/tmp/results.out");
```

```
src = -1;

while (curArgIndex < argc) {

    if (strcmp(argv[curArgIndex],"-src")==0) {
        src = atol(argv[++curArgIndex]);
    }

    if (strcmp(argv[curArgIndex],"-infile")==0) {
        strcpy(infilename, argv[++curArgIndex]);
    }

    if (strcmp(argv[curArgIndex], "-outfile")==0) {
        strcpy(outfilename, argv[++curArgIndex]);
    }

    if (strcmp(argv[curArgIndex], "-graph")==0) {
        strcpy(graph_type, argv[++curArgIndex]);
    }
    curArgIndex++;
}

fp = fopen(infilename, "r");
if (fp == NULL) {
    fprintf(stderr, "Error! Could not open input file. Exiting ...\n");
    exit(-1);
}
fclose(fp);

fp = fopen(outfilename, "w");
if (fp == NULL) {
    fprintf(stderr, "Error! Could not write to output file. Exiting ...\n");
    exit(-1);
}
fclose(fp);

graph_ext_check(infilename, graph_type);

fprintf(stdout, "\n");
fprintf(stdout, "Input Graph File    : %s\n", infilename);
fprintf(stdout, "Output Graph File   : %s\n\n", outfilename);

/* Step 2: Generate graph */
g = (graph_t *) malloc(sizeof(graph_t));
graph_gen(g, infilename, graph_type);

fprintf(stdout, "No. of vertices     : %ld\n", g->n);
fprintf(stdout, "No. of edges        : %ld\n\n", g->m);

if (src == -1)
```

```
        src = lrand48() % g->n;

    assert((src >= 0) && (src < g->n));
    fprintf(stdout, "Source vertex       : %ld\n\n", src);

    /* Step 3: Run algorithm */

    /* Assuming a low diameter graph */
    /* change the est_diameter value for high diameter graphs */
    est_diameter = 100;
    num_vertices_visited = BFS_parallel_frontier_expansion(g, src, est_diameter);

    /* Step 4: Write output to file */
    fp = fopen(outfilename, "w");
    fprintf(fp, "Breadth-first search from vertex %ld\n", src);
    fprintf(fp, "No. of vertices visited: %ld\n", num_vertices_visited);

    /* Step 5: Clean up */
    free(infilename);
    free(outfilename);
    free(graph_type);

    free_graph(g);
    free(g);

    return 0;
}
```

The above code listing performs a parallel BFS from a user-defined source vertex. The various SNAP routines called are discussed below.

Initialization

Step 1 of the above code initializes the graph and other query-dependent data structures, and parses the input arguments.

Graph file formats

We support two text formats for reading graphs: the SNAP format and the DIMACS file format. The DIMACS file format is defined as follows. Comment-lines begin with the character 'c'. Vertex identifiers are 1-indexed, and the graph is assumed to be undirected with integer edge weights. The first non-comment line in the file specifies the number of vertices and edges in the graph (e.g., 'p sp 10 100' initializes a graph with 10 vertices and 100 edges). Every line following the problem line defines an edge (e.g., 'e 2 3 10' specifies an edge from vertex 2 to 3 of weight 10).

The SNAP graph format is very similar to the DIMACS format, but is more extensive. The problem line lets the user specify if the graph is directed or undirected, as well as the weight type (32 and 64-bit integers, float, or double) and the vertex indexing order (0-indexed or 1-indexed). Edges are specified in the format 'headID tailID weight'

Synthetic problem instances can be defined using a configuration file. For instance, an undirected, unweighted Erdos-Renyi random graph of 100,000 vertices and 1 million edges can be generated with the following configuration file:

```
# A sample configuration file for generating random graphs
# (lines beginning with a # symbol indicate comments)
# The number of vertices is given by n
n 100000
# The number of edges is denoted by m
m 1000000
# Set undirected to 1 for a undirected graph, 0 otherwise
undirected 1
# Edge weights: 0 for unweighted, 1 for integer weight,
# 2 for long weights, 3 for floats, 4 for doubles
weight_type 0
# Max edge weight
# max_weight 100
# Min edge weight
# min_weight 0
```

Running a kernel/analysis query

Once the graph is generated, SNAP routines can be invoked to answer queries (step 3). Here, we use the routine *BFS_parallel_frontier_expansion* to perform parallel BFS.

Building the code

Suppose the above code is saved to a file named *testBFS.c*. By linking to the SNAP library, we can build an executable *eval_BFS* as follows:

```
% ${CC} ${CFLAGS} testBFS.c -I${SNAPDIR}/include -o eval_BFS -L${SNAPDIR}/lib
```

The environment variables need to be set up correctly. Alternately, please see driveBFS.c and Makefile.orig files in the *test* directory for an example of integrating new code into SNAP using the autoconf framework.

Executing the program

Once the code is built, it can be executed on the command line by specifying the required input arguments.

```
% ./eval_BFS
```

4 SNAP development

SNAP is designed to be modular and extensible, and allows users to add new code easily. The SNAP repository is organized as follows. *src* is the primary source directory of the distribution, and is sub-divided into *graph_generation*, *graph_kernels*, *graph_metrics*, and *graph_partitioning*. The *sprng2.0* directory consists of portable, parallel pseudo-random number generators and are primarily used for synthetic network generation. A few driver routines to execute the SNAP functions are located in the *test* folder, while all the required header files are in the *include* directory. New routines that enhance the functionality of SNAP and are independent of existing routines can be added to the corresponding sub-directory in *src*. High-level algorithms that call existing SNAP routines can be created by the user and built independently, as described in the previous section.

References

[1] D.A. Bader and K. Madduri. SNAP, Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2008)*, Miami, FL, April 2008.

[2] K. Madduri. SNAP: Small-world network analysis and partitioning, 2008. `http://snap-graph.sourceforge.net`.