Automated Generation of Stand-Alone Source Codes for Software Libraries

Lucas Machi, Henry L. Carscadden,
Chris J. Kuhlman, Dustin Machi, and S. S. Ravi

University of Virginia, Charlottesville, VA
[lhm4v],[hlc5v],[cjk8gx],[dm8qs],[ssr6nh]@virginia.edu

Abstract

Networks are pervasive in society: infrastructures (e.g., telephone), commercial sectors (e.g., banking), and biological and genomic systems can be represented as networks. Consequently, there are software libraries that analyze networks. Containers (e.g., Docker, Singularity), which hold both runnable codes and their execution environments, are increasingly utilized by analysts to run codes in a platform-independent fashion. Portability is further enhanced by not only providing software library methods, but also the driver code (i.e., `main()` method) for each library method. In this way, a user only has to know the invocation for the `main()` method that is in the container. In this work, we describe an automated approach for generating a `main()` method for each software library method. A single intermediate representation (IR) format is used for all library methods, and one IR instance is populated for one library method by parsing its comments and method signature. An IR for the `main()` method is generated from that for the library method. A source code generator uses the `main()` method IR and a set of small, hand-generated source code templates—with variables in the templates that are automatically customized for a particular library method—to produce the source code `main()` method. We apply our approach to two widely used software libraries, SNAP and NetworkX, as examplars, which combined have over 400 library methods.

Keywords: *automated source code generation, software libraries, network science, cyberinfrastructure*

1 Introduction

1.1 Background and Motivation

Networks and graphs are pervasive in society. Humans form friendship networks, follower networks on social media, and collaboration networks. Interactions among organizations such as banking, finance, and historic preservation can be characterized as networks. Infrastructures, such as the land line telephone system, form networks. Geographical features form networks, such as networks of canals (see [8, 9, 13]). Consequently, analysts in a range of disciplines use many of the same network science approaches to understand networks.

This suggests the need for a cyberinfrastructure (CI) for network science whereby analysts can share computational codes and data, and all analysts can benefit from using these resources.

A *cyberinfrastructure* (CI) consists of computing systems, data storage systems, advanced instruments and data repositories, and visualization environments, all linked by high speed computer networks to make possible innovation and discoveries not otherwise possible [16, 21]. Currently, there is no general-purpose CI for network science.

A CI for network science called *net.science* is being constructed; it was initially released in June 2021. An overview of the *net.science* system is given in [1]. The system is composed of the following elements: (*i*) an infrastructure that manages data and software, and provides execution environments for software; (*ii*) common services such as workflow and visualization services; (*iii*) computational codes (which we call *tasks*) that perform operations on networks and network-related data; (*iv*) network and associated data; (*v*) a web application (web app) for interacting with *net.science* through web browser screens; and (*vi*) an application programming interface (API) that is used to access *net.science* resources. (Third-party codes and the web app use the API.)

In this paper, we are concerned with computational codes (i.e., tasks) and their execution environments within the CI. Such tasks may run serially or in parallel, giving rise to a number of considerations and requirements. First, the CI runs each code in its own execution environment in order to isolate it; that is, each code runs within a Singularity container. Second, each code runs as a separate process and is invoked with command-line arguments (CLAs); *net.science* invokes codes based on parameters specified by users. This enables testing of codes inside and outside of containers, and inside and outside of *net.science*. Third, this adds to the modularity of the overall system, where individual codes and collections of codes can be added, updated, or removed atomically. Fourth, individual codes make reasoning about compositions of operations more straight-forward. This is useful since many steps must be used inside *net.science* to invoke a single task; examples of such steps include setting up a workspace, copying files into it, running the code, assigning and updating metadata for provenance, and error handling. Fifth, for the case of multiple hardware systems, if a code is required to run on another remote platform, the containerization step ensures that the software that must be copied to a destination is well-defined, and will run on the destination system (with appropriate hardware).

Technical Challenges: One challenge is due to the sizes of software libraries that are integrated into *net.science*. SNAP [14] and NetworkX [10] libraries are well-known and highly used, which makes them prime candidates for inclusion in the CI. SNAP has some 151 network methods and NetworkX has more than 270. In *net.science*, each of these operations must be a stand-alone code. A primary challenge is how to accomplish this goal—incorporating these 400-plus methods into the CI—in an automated fashion. Given the effort needed to tackle the first challenge, we also want to devise a solution whose artifacts can be used for other purposes.

1.2 Contributions

1. An approach for automatically generating stand-alone codes for the methods of a software library. Because each method of a software library must be a stand-alone executable code in *net.science*, each library method requires a `main()` method, with appropriate CLAs, that is invoked from the command line and that, in turn, calls a particular library function. A conceptual view is provided in Figure 1. A user selects a method to execute, and the CI stages data and invokes the appropriate `main()` method. The approach is programming language (PL) agnostic (the implementation is not PL agnostic).

The inputs and outputs for a library method often are in volatile memory. When these methods are used in isolation within a stand-alone code, the contents of in-memory input variables must be read from file(s) and the contents of in-memory output variables must be written to file(s). Thus, the arguments for the `main()` method are often different from those

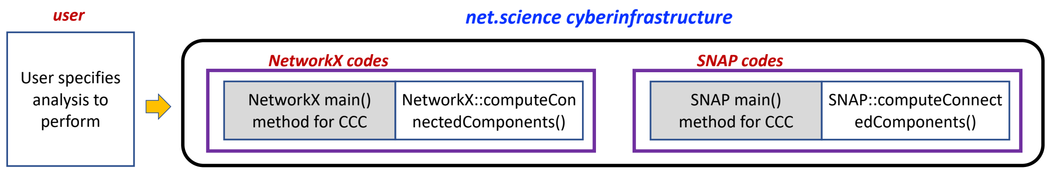

user
User specifies analysis to perform
net.science cyberinfrastructure
NetworkX codes
NetworkX main() method for CCC
NetworkX::computeConnectedComponents()
SNAP codes
SNAP main() method for CCC
SNAP::computeConnectedComponents()

Figure 1: Conceptual view of packaging library methods. Each library method (in white within the CI box) has a corresponding `main()` method (in gray in the CI box), resulting in a stand-alone code. Here, the software library method computes the connected components of a graph. There are 151 and 270 methods in SNAP and NetworkX, respectively.

for the library method itself (e.g., different in number and types). For example, an input to a library method to compute the degree distribution of a graph will be an object that contains the graph, in-memory. The `main()` method, however, must take as arguments the name of the file that contains the graph data, the type (i.e., format) of the graph in file (e.g., edge list, adjacency list), and the directionality of edges (i.e., directed or undirected). The `main()` method must also know the input/output argument types in a library method's signature, because this dictates which method variables, if any, are inputs (and therefore may be read from file), are outputs (and therefore may be written to file), and are both inputs and outputs (and therefore may be read from and written to files). Our example is graph libraries, but the approach is more general, and extensible; e.g., see Section 3.5 on extensibility.

2. A software system that implements the approach for network structural analysis libraries. Our Python-based software system uses two steps to auto-generate the `main()` method's source code for each library method. In the first step, the source code and/or comments files are parsed to produce intermediate representations (IRs) of the library method and of the required `main()` method's source code. This parser can vary for different software libraries, and we confront this situation in our work. (Additional explanation regarding this is provided in Item 3 below.) In the second step, the source code for the `main()` method is generated by a source code generator (SCG), from the IRs. Other inputs are software library-specific datatypes, and source code templates that are customizable by the SCG for the particular information within the IRs. The tailored templates of (Python) source code snippets are composed by the SCG to construct the `main()` method. The SCG is PL and software library agnostic; these dependencies are contained in the code templates and datatypes. This two-step process is wrapped in a **for** loop to iterate over all the methods in a software library.

3. Case studies for NetworkX and SNAP. Stand-alone codes for 151 SNAP methods and 270 NetworkX method have been generated using this approach. These methods are of two types: structural analysis codes to generate characteristics of a graph, and graph generators. There are differences between these two libraries, even though both have Python interfaces. Snap.py is the Python-wrapped library that encloses SNAP, a C++ library. Therefore, Version 5 of Snap.py has input, output, and in-out arguments in method signatures, in addition to return arguments. NetworkX, by comparison, is more "Pythonic" in design and hence arguments in method signatures are of signature type input only, and outputs are return variables. Both libraries were selected for inclusion into *net.science* because they are well-known and have large user bases. They are also complementary: NetworkX has more methods and SNAP can handle very large networks. Finally, source code commenting is done differently in the two libraries, so this forced us to have multiple related parsing approaches.

2 Related Work

Operations on networks: network structural analysis codes. There are several structural analysis libraries, including: Gunrock [24], a GPU-based graph analysis software system; graph-tool [18]; NetworKit [20], an OpenMP-based graph structural analysis library; igraph [7], a Python-wrapped C++ implementation of many methods in NetworkX and advanced network plotting; NetworkX [10], a Python-based system; PEGASUS [11], a Hadoop-based implementation; and SNAP [14], a C++ implementation with multithreading, along with Snap.py, which is a Python wrapper around SNAP. We have selected initially the methods in NetworkX and SNAP libraries for the reasons specified in Section 1.2.

Automated source code generation. The term automated source code generation is broad. This can include generating source code from scratch (e.g., [4, 6]), or from pseudo-code [30], code completion for smaller snippets of code [3], and error correcting of code [27]. Earlier on, research was on how to generate source code in an automated fashion from design patterns [4] and unified modeling language (UML) diagrams [12, 19]. Currently, there is much activity in using artificial intelligence (AI) approaches (e.g., neural networks) for automated code generation [2, 25], and combined natural language processing and AI methods [3, 15, 17, 22, 23, 28, 29]. Overviews of various approaches are discussed in [6, 26]. The references cited above fall into two categories: earlier works that are now more straight-forward, and new approaches that are quite sophisticated and are difficult to apply to large problems. Our approach lies between these two extremes.

3 System Description

3.1 Overview

The problem considered in our work and our solution approach are outlined below.

Given: A commented software library whose methods compute properties of graphs.

Required: A set of stand-alone codes (i.e., `main()` methods) that are invoked from the command line, one code per library method, where inputs such as graphs are read from files and outputs are written to files.

Solution approach: Figure 2 briefly describes our solution approach; one can think of this as the activity diagram associated with our approach. Square boxes in blue denote files, either data or source code. The rounded boxes in black represent (Python) codes that automatically construct the `main()` method for each library method. The commented library source code and/or separate method descriptions, at the left, are input to a parser that constructs IRs of the library method and of the `main()` method. The second component, namely the SCG, reads in the two IRs, along with other inputs, and produces the `main()` method corresponding to a library method (at right).

In this work, we focus on software libraries implemented in, or wrapped by, Python code. The remainder of this section details our approach and implementation.

3.2 Input: A Software Library

The software libraries that we integrate take different approaches to code documentation. Snap.py Version 5 provides a text file for each library method, separate from library method source code. These are the files that we parse to generate IRs. Snap.py Version 5 has method signature types because Snap.py is a wrapper for the underlying C++ implementation, SNAP. (Newly released Version 6 of Snap.py is more “Pythonic.”) There are rigorous rules for the structure of code comments that are imposed on developers, enabling automated parsing.

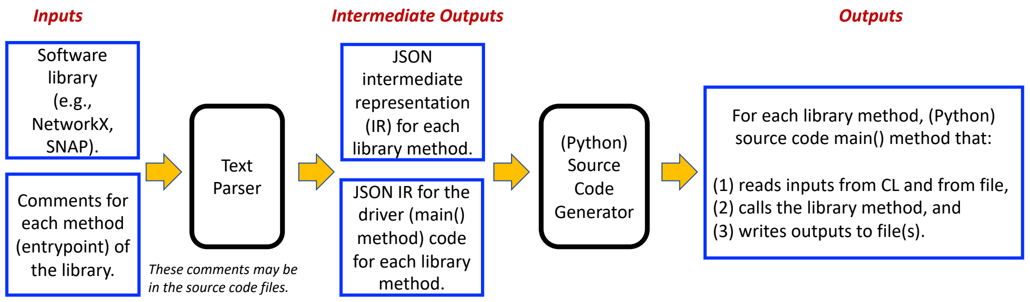

Inputs
Intermediate Outputs
Outputs
Software library (e.g., NetworkX, SNAP).
Comments for each method (entrypoint) of the library.
Text Parser
These comments may be in the source code files.
JSON intermediate representation (IR) for each library method.
JSON IR for the driver (main() method) code for each library method.
(Python) Source Code Generator
For each library method, (Python) source code main() method that:
(1) reads inputs from CL and from file,
(2) calls the library method, and
(3) writes outputs to file(s).

Figure 2: Overall automated approach for building a stand-alone code for a (structural analysis) library method.

NetworkX has comments in the code base, and we parse/interpret these files in a different way. NetworkX is a more organic, community-driven code base, and as such, its commenting of library methods varies more across methods. Consequently, it is difficult to automate the process of parsing content and generating IRs directly from comments in the code. The method signature, for example, cannot be used by itself because variable types are required and inference of these is often not possible; Section 3.3 below lists the required parameters.

3.3 Text Parser

A text parser parses a file that describes a library method. For Snap.py, this individual file has a well-defined format and parsing is automated; the outputs are IRs for the library method and the `main()` method. For NetworkX, the parsing process is more elaborate, as follows. A human reads a library method's source code comments, and then manually enters values into a file that is an abbreviated IR. A software tool then converts the contents of this abbreviated file into a library method IR. The parser used for Snap.py is then used to build the `main()` method IR from the library method IR. This process for NetworkX is much faster than inspecting the NetworkX source code and manually producing both IRs.

The general structure of the two parser codes is the same for NetworkX and Snap.py, but specific commands (e.g., regular expression syntax) are specific to a library. The parsing operations cover the following pieces of information: library method's name; the method's textual description; library method's signature from which we obtain all method arguments and return values; for each library method argument, the variable type and the method signature type (i.e., input, output, or in-out); and for each return value, the variable type.

3.4 Intermediate Representation

The parser generates instances of IRs in JSON format. Figure 3 provides an IR for the `main()` method for the Snap.py method GetKCoreEdges(). This listing is given in two-column format. The keys in the JSON object are given in alphabetical order. Starting at the upper left, internal variables that are in the library method's signature, but do not appear as command line arguments in the `main()` method, are provided first. Second, information about the return variable for the library method is listed. Third, the library method's name is given. Fourth, the six method signature variables—the CLAs—for the `main()` method are enumerated in the figure (these start towards the bottom of the left column and continue on the top of the right column of Figure 3). Lastly, a summary of the number of different types of arguments is given, along with the parser version number.

```
{
  "date": "23jan2020",
  "internalVariablesInDriver": [
    {
      "argumentLongDescription": "A vector of
                (order, number of edges of the given order) pairs.",
      "argumentShortDescription": ", a vector of (int, int) pairs",
      "argumentTypeInMethodSignature": "output",
      "argumentVariableName": "CoreIdSzV",
      "argumentVariableType": "TIntPrV",
      "defaultArgumentValue": "--"
    },
    {
      "argumentLongDescription": "A Snap.py graph or a network.",
      "argumentShortDescription": "--",
      "argumentTypeInMethodSignature": "input",
      "argumentVariableName": "Graph",
      "argumentVariableType": "graph",
      "defaultArgumentValue": "--"
    }
  ],
  "isThereALibraryReturnType": true,
  "isThereAReturnType": false,
  "libraryMethodReturnTypeVariable": {
    "returnType": "int",
    "returnTypeDescription": "The number of cores."
  },
  "methodName": "GetKCoreEdges",
  "methodReturnTypeVariable": {
    "returnType": "None",
    "returnTypeDescription": "--"
  },
  "methodSignatureVariables": [
    {
      "argumentLongDescription": "name of graph file
                        containing graph to operate on.",
      "argumentShortDescription": "--",
      "argumentTypeInMethodSignature": "input",
      "argumentVariableName": "inputFile_Graph",
      "argumentVariableType": "str_infileName",
      "defaultArgumentValue": "--"
    },
    {
      "argumentLongDescription": "graph type is Snap type,
                      so one of:  PNGraph, PUNGraph, PNEANet.",
      "argumentShortDescription": "--",
      "argumentTypeInMethodSignature": "input",
      "argumentVariableName": "graphType",
      "argumentVariableType": "str",
      "defaultArgumentValue": "--"
    },
    {
      "argumentLongDescription": "col index in graphName
                     for the src (source) node of the edge (src, des).",
      "argumentShortDescription": "--",
      "argumentTypeInMethodSignature": "input",
      "argumentVariableName": "srcCol",
      "argumentVariableType": "int",
      "defaultArgumentValue": "--"
    },
    {
      "argumentLongDescription": "col index in graphName
                  for the des (destination) node of the edge (src, des).",
      "argumentShortDescription": "--",
      "argumentTypeInMethodSignature": "input",
      "argumentVariableName": "desCol",
      "argumentVariableType": "int",
      "defaultArgumentValue": "--"
    },
    {
      "argumentLongDescription": "output file name for variable CoreIdSzV",
      "argumentShortDescription": "for an output argument",
      "argumentTypeInMethodSignature": "input",
      "argumentVariableName": "outputFile_CoreIdSzV",
      "argumentVariableType": "str_outfileName",
      "defaultArgumentValue": "--"
    },
    {
      "argumentLongDescription": "output file name for variable
                                    libraryMethodReturnTypeVariable",
      "argumentShortDescription": "for a *library method* return argument",
      "argumentTypeInMethodSignature": "input",
      "argumentVariableName": "outputFile_libraryMethodReturnTypeVariable",
      "argumentVariableType": "str_outfileName",
      "defaultArgumentValue": "--"
    }
  ],
  "methodTextDescription": "Returns the number of edges in each core of
    order K (where K=0, 1, ...). Stores pairs (K, number of edges) in *CoreIdSzV*.",
  "numberOfMethodSignatureArguments": 6,
  "numberOfMethodSignatureInOutArguments": 0,
  "numberOfMethodSignatureInputArguments": 6,
  "numberOfMethodSignatureOutputArguments": 0,
  "versionNumber": "0.00"
}
```

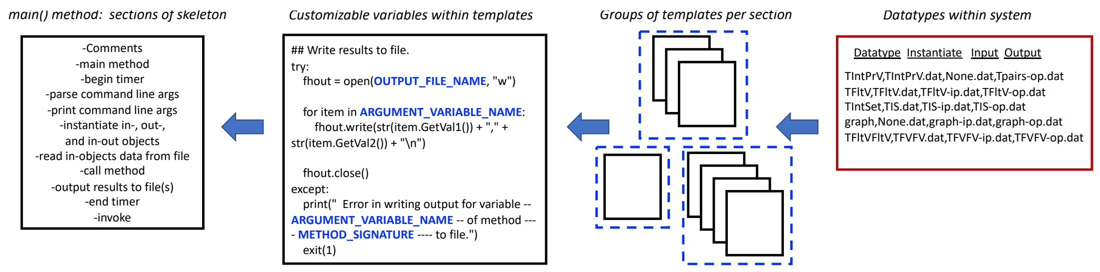
Figure 3: Intermediate representation (IR) for the `main()` method that calls the GetKCoreEdges() method within Snap.py. The JSON file contents are split over two columns. A few of the lines in the file have hard returns in them, for visualizing in this figure.

Each CLA for the `main()` method has a name; long and short descriptions; a variable type (e.g., int, str, float, TIntPrV) that is either a primitive type of the PL, a data structure, or a class; a method signature type (input, output, or in-out); and a default value ("–" means no value). Some arguments are straight-forward. For example, the "argumentVariableName" of "desCol" in the right column of Figure 3 is the column index in the graph file that contains a node ID (the destination node ID, des) for each edge that is represented as the node pair (src, des). This index is an int and its signature type is input.

Other arguments are more involved. For example, the next "argumentVariableName" is "outputFile_CoreIdSzV." In our system, *outputFile* is a keyword, and it specifies that the variable name is associated with an output file. Furthermore, the suffix, here "CoreIdSzV," denotes the variable name whose contents are written to this output file. This suffix will be a variable name or a return type that appears elsewhere in this IR. In this case, "argumentVariableName" of "CoreIdSzV" appears near the beginning of this listing, as an internal variable. Its argument type, in turn, is "TIntPrV" and this is a datatype that is linked to particular types of templates. Templates are described in Section 3.5 below, but we note here that type "TIntPrV" dictates the contents of file "outputFile_CoreIdSzV." Finally, the "argmentVariableType" for variable "outputFile_CoreIdSzV" is "str_outfileName." This also contains multiple pieces of information. First, the variable type is string ("str"), and second, it is for an output filename ("outfileName").

3.5 Source Code Generator

Per Figure 2, the source code generator (SCG) constructs a library `main()` method from an IR. Before addressing the SCG, we describe the approach for generating the source code for the `main()` method. Figure 4 provides the conceptual view. On the left, there is a Python `main()` method source code file, which we call a *skeleton* because initially the file merely contains a name (text string) for each of the sections depicted, that in totality, constitute the `main()` method. Each text string is to be overwritten by one or more template files (shown in the second part of the figure, from the left).


main() method: sections of skeleton
Customizable variables within templates
Groups of templates per section
Datatypes within system

```
-Comments
-main method
-begin timer
-parse command line args
-print command line args
-instantiate in-, out-,
and in-out objects
-read in-objects data from file
-call method
-output results to file(s)
-end timer
-invoke
```

```
## Write results to file.
try:
   fhout = open(OUTPUT_FILE_NAME, "w")

   for item in ARGUMENT_VARIABLE_NAME:
      fhout.write(str(item.GetVal1()) + "," +
str(item.GetVal2()) + "\n")

   fhout.close()
except:
   print("  Error in writing output for variable --
ARGUMENT_VARIABLE_NAME -- of method ---
- METHOD_SIGNATURE ---- to file.")
   exit(1)
```

```
Datatype  Instantiate  Input  Output
TIntPrV,TIntPrV.dat,None.dat,Tpairs-op.dat
TFltV,TFltV.dat,TFltV-ip.dat,TFltV-op.dat
TIntSet,TIS.dat,TIS-ip.dat,TIS-op.dat
graph,None.dat,graph-ip.dat,graph-op.dat
TFltVFltV,TFVFV.dat,TFVFV-ip.dat,TFVFV-op.dat
```

Figure 4: Templating approach for building `main()` method source code from IRs. The SCG orchestrates the work right-to-left.

A template file may contain keywords that need to be overwritten for a particular library method. For example, there may be a keyword ARGUMENT_VARIABLE_NAME that is overwritten with the name of a particular variable from the IR. Variables are shown in bold blue in Figure 4. This particular template shows how the data associated with a particular type are written to file. The methods "GetVal1" and "GetVal2" are methods associated with a particular Snap.py class. Thus, templates are PL-dependent and may also be software library dependent because a library may have particular classes and methods that must be used in a `main()` method.

In the third graphic from the left, groups of templates are provided; there is one group for each heading in the skeleton. Groups are shown within blue dashed boxes. So, before a particular template is specialized, that template must be selected from the group that corresponds to the heading of interest in the skeleton. This decision is based on the contents of the `main()` method IR, and the SCG may select multiple templates from a group (e.g., if there are multiple output files).

There is a datatypes file at the right in Figure 4 that contains an extensible listing of datatypes (including data structures and class names) in the system. For example, datatype TIntPrV is a datatype in the Snap.py library, and is shown in Figure 3, first column, tenth line. For each datatype, the datatypes file in Figure 4 contains three filenames, each of which contains source code: (i) the template filename containing code for instantiating an object of this datatype; (*ii*) the template filename containing code for populating an instantiated object of this datatype (when the method signature type is input or in-out), which involves reading data from a file; and (*iii*) the template filename containing code for writing the contents of this variable to an output file when an object of this datatype is of method signature type output or in-out, or is a return type. The datatypes may be inputs in selecting the correct templates from a group.

It is evident now that the ultimate Python source code `main()` method file is a composition of template source code files. This design is a compromise between having flexibility in constructing (or composing) a `main()` method, and having too much flexibility that demands that the SCG has to build the `main()` method line-by-line with all of the error-prone syntax specifications that this entails.

Furthermore, when new datatypes are introduced, new templates are added, and the SCG source code is not altered. This extensibility is important because with large software libraries like NetworkX, it would be onerous to understand all methods and all of their requirements before designing the SCG. With our approach, this detailed global view need not be constructed because the system can be extended as needed.

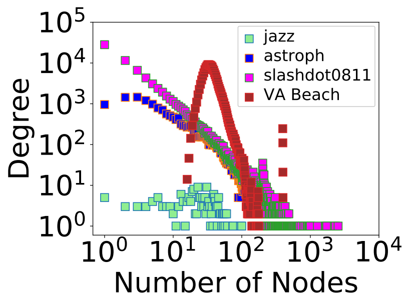
With the descriptions thus far in this subsection, we can now simply describe the SCG code. The SCG orchestrates the manipulations of files that are represented in Figure 4, using the types of IRs identified in Figure 2, in order to produce a software library method's particular `main()` method. Note that SCG is largely PL- and software library-agnostic; and the PL- and library-specific information is confined to the templates and datatypes file.

3.6 Output: `main()` Method and Validation

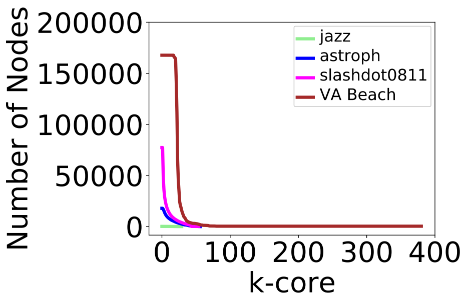
The output from the SCG is the library method's corresponding `main()` method, shown at the far right of Figure 2. This approach was validated in three ways: (*i*) by comparing the contents of automatically generated `main()` method source codes against manually constructed ones; (*ii*) by comparing outputs from invocations with the automatically generated `main()` methods against outputs from manually generated `main()` methods, and (*iii*) by verifying output files from sufficiently small graphs.

4 Case study

We illustrate the use of the results of our approach for generating `main()` methods using two of the methods from each of the NetworkX and Snap.py libraries. These are illustrative of the 400+ library methods for which we have generated `main()` methods. One of these library methods produces the degree distribution while the other produces the k-core distribution for a network. The degree distribution is the frequency count of the number of (distance-1) neighbors of nodes in a graph. A k-core is a subgraph of an original graph in which every node has degree at least k. The k-core distribution is the number of nodes in the k-core (graph), for varying k. We first generated, for both libraries, all of the stand-alone codes using the approach described in the paper. We then executed the resulting auto-generated stand-alone codes to obtain the structural analysis results on several networks; these results are shown in Figure 5. Degree distributions are provided in Figure 5a and k-core distributions are provided in Figure 5b. These data tell us many things, e.g., that the astroph and slashdot0811 have scale-free degree distributions and the VA Beach network has a random graph-like structure. From the numbers of nodes and edges for each graph that are given in the caption, it is observed that these data span four orders of magnitude in numbers of nodes and numbers of edges. We have used a great many of these resulting codes on other networks, for other projects, on networks up to billions of edges [5].



(a) degree distributions



(b) k-core distributions

Figure 5: (a) Degree distributions and (b) k-core distributions for four networks. With each network is listed $(n; e)$, where n is its number of nodes and e is its number of edges: jazz (198; 2,742); astroph (17,903; 196,972); slashdot0811(77360; 469,180); and Virginia Beach (167,722; 3,245,840). The first three networks are undirected; the last is directed. These properties were produced with the stand-alone codes that were generated using the procedures in this work.

5 Summary, Limitations, and Future Work

We have described an approach for automatically constructing large numbers of stand-alone codes (i.e., `main()` methods) from software library methods. These methods are containerized and incorporated into a cyberinfrastructure called *net.science*. We have integrated two well-known and well-used software libraries, NetworkX and SNAP, into *net.science*. While the approach is useful for software libraries, modifications are needed to enable more seemless integration of individual software codes. Also, we are investigating other approaches so that a single parser would suffice for all codes. In an expanded version, we will provide an example `main()` method generated by this approach. We will also provide timing data for developing these methods by hand, versus using the automated approach.

Acknowledgments: We thank the anonymous reviewers for their helpful feedback. This work is partially supported by UVA Strategic Investment Fund number SIF160 and NSF grants OAC-1916805, CMMI-1916670 and IIS-1908530.

References

[1] N. K. Ahmed, R. A. Alo, et al. net.science: A cyberinfrastructure for sustained innovation in network science and engineering. In *Gateway Conference*, 2020.

[2] T. Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *EICS*, 2018.

[3] M. Brockschmidt, M. Allamanis, A. Gaunt, and O. Polozov. Generative code modeling with graphs. In *ICLR*, 2019.

[4] F. Budinsky, M. Finnie, P. Yu, and J. Vlissides. Automatic code generation from design patterns. *IBM Systems Journal*, pages 1–25, 1996.

[5] H. L. Carscadden, L. Machi, A. Kishore, et al. ExecutionManager: A software system to control execution of third-party software that performs network computations. In *WSC*, 2021.

[6] J. Cruz-Benito, S. Vishwakarma, F. Martin-Fernandez, and I. Faro. Automated source code generation and auto-completion using deep learning: Comparing and discussing current language model-related approaches. *AI*, 2:1–16, 2021.

[7] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems, 2006.

[8] J. C. Davis. *The Human Story: Our History, From the Stone Age to Today.* HarperCollins, New York, NY, 2004.

[9] G. Grullon, S. Underwood, and J. P. Weston. Comovement and investment banking networks. *Journal of Financial Economics*, 113:73–89, 2014.

[10] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *SciPy2008*, pages 11–15, 2008.

[11] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system–implementation and observations. In *ICDM*, pages 229–238, 2009.

[12] D. Kundu, D. Samanta, and R. Mall. Automatic code generation from unified modelling language sequence diagrams. *IET Software*, 7:12–28, 2013.

[13] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

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

[15] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiský, F. Wang, and A. Senior. Latent predictor networks for code generation. In *ACL*, pages 599–609, 2016.

[16] Cyberinfrastructure vision for 21st century discovery, 2007. Report by the National Science Foundation Cyberinfrastructure Council.

[17] M. R. Parvez, S. Chakraborty, B. Ray, and K.-W. Chang. Building language models for text with named entities. In *ACL*, pages 2373–2383, 2018.

[18] T. P. Peixoto. The graph-tool python library. *figshare*, 2014.

[19] S. Philippi. Automatic code generation from high-level petri-nets for model driven systems engineering. *Journal of Systems and Software*, 79:1444–1455, 2006.

[20] C. Staudt, A. Sazonovs, and H. Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2014.

[21] C. A. Stewart, S. Simms, B. Plale, M. Link, D. Y. Hancock, and G. C. Fox. What is cyberinfrastructure. In *ACM SIGUCCS Fall Conference: Navigation and Discovery*, pages 37–44, 2010.

[22] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang. A grammar-based structural CNN decoder for code generation. In *AAAI*, pages 7055–7062, 2019.

[23] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang. Treegen: A tree-based transformer architecture for code generation. In *AAAI*, pages 8984–8991, 2020.

[24] Y. Wang, Y. Pan, et al. Gunrock: Gpu graph analytics. *ACM Trans. Parallel Comput.*, 4(1), 2017.

[25] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin. Code generation as a dual task of code summarization. In *NeurIPS*, pages 6559–6569, 2019.

[26] D. A. Wheeler, J. D. Birdwell, and F. L. Loaiza. A partial survey on AI technologies applicable to automated source code generation. Technical report, Institute for Defense Analyses, 2019.

[27] M. Yasunaga and P. Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *ICML*, volume 119, pages 10799–10808, 2020.

[28] P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. In *ACL*, pages 440–450, 2017.

[29] P. Yin, C. Zhou, J. He, and G. Neubig. StructVAE: Tree-structured latent variable models for semi-supervised semantic parsing. In *ACL*, pages 754–765, 2018.

[30] R. Zhong, M. Stern, and D. Klein. Semantic scaffolds for pseudocode-to-code generation. In *ACL*, pages 2283–2295, 2020.