Contents lists available at ScienceDirect
Computer Physics Communications
journal homepage: www.elsevier.com/locate/cpc

APINetworks: A general API for the treatment of complex networks in arbitrary computational environments[☆]

Alfonso Niño[*], Camelia Muñoz-Caro, Sebastián Reyes

SciCom Research Group, Escuela Superior de Informática. Universidad de Castilla-La Mancha, Paseo de la Universidad 4, 13071 Ciudad Real, Spain

ARTICLE INFO

Article history:
Received 1 July 2014
Received in revised form
2 May 2015
Accepted 21 May 2015
Available online xxxx

Keywords:
API
Software
Complex networks
Heterogeneous networks
Dynamic networks

ABSTRACT

The last decade witnessed a great development of the structural and dynamic study of complex systems described as a network of elements. Therefore, systems can be described as a set of, possibly, heterogeneous entities or agents (the network nodes) interacting in, possibly, different ways (defining the network edges). In this context, it is of practical interest to model and handle not only static and homogeneous networks but also dynamic, heterogeneous ones. Depending on the size and type of the problem, these networks may require different computational approaches involving sequential, parallel or distributed systems with or without the use of disk-based data structures. In this work, we develop an Application Programming Interface (APINetworks) for the modeling and treatment of general networks in arbitrary computational environments. To minimize dependency between components, we decouple the network structure from its function using different packages for grouping sets of related tasks. The structural package, the one in charge of building and handling the network structure, is the core element of the system. In this work, we focus in this API structural component. We apply an object-oriented approach that makes use of inheritance and polymorphism. In this way, we can model static and dynamic networks with heterogeneous elements in the nodes and heterogeneous interactions in the edges. In addition, this approach permits a unified treatment of different computational environments. Tests performed on a C++11 version of the structural package show that, on current standard computers, the system can handle, in main memory, directed and undirected linear networks formed by tens of millions of nodes and edges. Our results compare favorably to those of existing tools.

Program summary

Program title: APINetworks 1.0

Catalogue identifier: AEWZ_v1_0

Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AEWZ_v1_0.html

Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland

Licensing provisions: Standard CPC licence, http://cpc.cs.qub.ac.uk/licence/licence.html

No. of lines in distributed program, including test data, etc.: 33736

No. of bytes in distributed program, including test data, etc.: 541147

Distribution format: tar.gz

Programming language: Standard ANSI C++11.

Computer: Workstation.

Operating system: Linux, Windows.

Classification: 6.3.

Nature of problem: The computational modeling and handling of complex systems, described as a network of interacting elements, is nowadays a topic of paramount importance. In the general case, it is necessary to represent static or dynamic, time dependent, sets of heterogeneous entities related through

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author. Tel.: +34 926295300; fax: +34 926295354.
E-mail address: alfonso.nino@uclm.es (A. Niño).

http://dx.doi.org/10.1016/j.cpc.2015.05.017
0010-4655/© 2015 Elsevier B.V. All rights reserved.

heterogeneous interactions. In turn, and depending on the size and nature of the problem, different computational approaches may be required. Thus, we can resort to sequential, parallel or distributed systems and to disk-based data structures. Different tools are available that satisfy one or several of these requirements. However, a unified approach for dealing with heterogeneous networks in arbitrary computational environments is still pending.

Solution method: To address the above issues, we have developed an Application Programming Interface: APINetworks. The API is organized in several packages where the one responsible for the network structure acts as the core element. We resort to an object-oriented approach, that makes use of inheritance and polymorphism. In this way, we can model static and dynamic networks and include heterogeneous agents in the nodes and heterogeneous interactions in the edges. In addition, this approach permits a unified treatment, transparent to the user, of different computational environments: sequential, parallel, distributed, memory-based or disk-based.

Running time: Examples provided take a few seconds each.

1. Introduction

The last decade has seen a great development of the structural and dynamic study of complex systems under the prism of a networked system. From this standpoint, actual systems can be described as a set of, possibly, heterogeneous entities or agents interacting in, possibly, different ways. This point of view leads to the description of a general system as a network of nodes, representing the entities or agents, connected by some links or edges, representing the interactions.

Strictly speaking, network studies date back to the introduction of graph theory by Leonard Euler in the famous Königsberg bridges problem [1]. Later, in the late 1950s, Erdös and Rényi (ER) made a breakthrough in classical mathematical graph theory by describing a network with complex topology as a random graph [2]. This work triggered a set of intensive studies since then to the present day. It is interesting to note that the main topological characteristic of an ER network is that its node-degree variation follows a Poisson distribution. This means that almost all nodes have the same degree (connectivity). The ER random graph model was the dominant complex network model for nearly half a century. However, the situation changed when detailed information on very large-scale real-world networks was available at the end of the 20th century. Until then, the use of networks was mainly associated with the social scientists work. Thus, for instance, in 1933 Jacob Moreno introduced the sociogram [3], a network where individuals are the nodes and the social relationships the edges. Moreno showed how to use sociograms to derive specific information about the structure and behavior of a social system. Another inflexion point was the introduction of the concept of weak ties by Mark Granovetter in the 1970s [4]. Weak ties are links between different social clusters that proved to be essential for information dissemination. In addition, they have a point of contact with the small-world characteristic introduced in the social sciences by Milgram in the late 1960s [5]. The small-world property implies a small distance (measured in number of edges) between any pair of nodes with respect to the number of nodes, N, in the network (actually, the distance grows with the logarithm of N).

In the late 1990s, the technological advances made possible to collect (and process) information regarding large networks such as collaboration networks, communication networks, the World Wide Web or the Internet. In contrast with the ER model, the analysis of these networks showed that their degree distribution follows, at least approximately, a power law distribution. This implies that the networks self-organize into a scale-free structure, a feature unpredicted by the random network model. Thus, for instance, the World Wide Web [6], the Internet [7], and the citation patterns of scientific publications [8,9] exhibit this behavior.

In this context, two seminal works are considered triggers of the present interest in the study of complex networks. The first one is the work by Watts and Strogatz on small-world networks [10]. This work showed how a regular lattice network can be transformed into a random network, exhibiting the small-world property, just by randomly reconnecting a small fraction of edges. The second is the one by Barabási and Albert on the generation of scale-free networks by a process of preferential attachment [11]. These initial studies led to the discovery that many natural and artificial networks follow a power-law distribution. In fact, this behavior is observed in systems as diverse as communication, social, economic, technological, biological, or transportation networks, to name just a few. A detailed survey is found in [12]. The observation of a common behavior pattern in networks, independent of the network nature, has led to the establishment of a series of general principles and to a unified treatment of networked systems, defining the new field of complex networks [13].

The interest in the modeling and treatment of complex networks fostered the development of multiple software tools. Thus, on one hand, we find general platforms, running on a single computer, with visualization and analysis capabilities such as Tulip [14], Gephi [15], Cytoscape [16] or Pajek [17]. On the other, we have several software packages used as programming environments in different languages such as, for instance, NetworkX [18] for Python.

General platforms are extremely useful for network visualizations and for applying already established, and therefore, already implemented, analysis techniques. However, the treatment of very large networks is a problem, since the capabilities of a single computer are limited. In addition, the use of new, non-standard, techniques can be troublesome. Software packages, on the contrary, are well suited for implementing new methods and techniques. However, as shown in Section 2.1, the different available packages are usually tailored to specific kinds of work or computational environments.

In this work, we present the design and initial implementation of an Application Programming Interface (API) for the modeling and treatment of general networks in arbitrary computational environments. Focusing in the network structural component, the design allows the modeling of static and dynamic networks, and the use of heterogeneous agents in the nodes and heterogeneous interactions in the edges. Using an object-oriented approach, and making use of inheritance and polymorphism, the API defines a unified treatment, transparent to the user, for different computational environments: sequential, parallel, distributed,

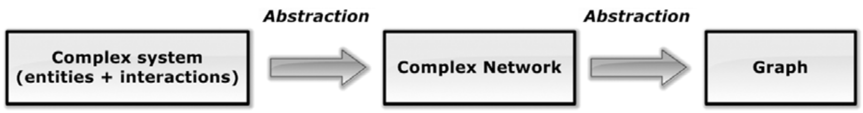

Fig. 1. Conceptual representation of the relationship between complex system, complex networks and graphs as considered in this work.

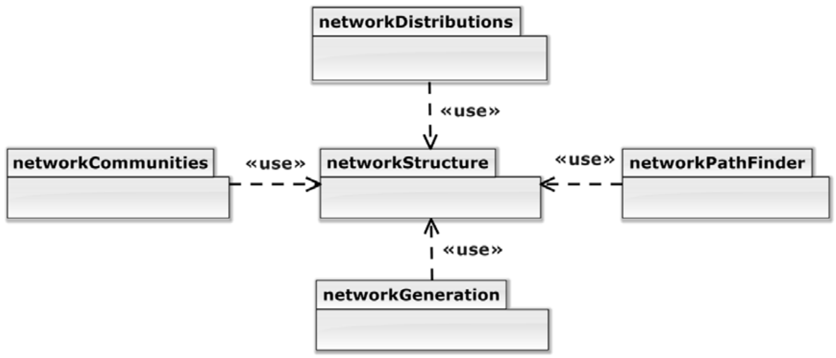

Fig. 2. UML package diagram showing the proposed organization of functional and structural components for the API.

memory-based or disk-based. A first sequential C++ implementation is provided, and its performance tested and discussed.

2. Methodology

2.1. Analysis of the problem

Our basic approach is described in the diagram shown in Fig. 1. There, we observe how a general system, composed of different entities (agents in the agents-based modeling lingo) and interactions, is abstracted as a network of interrelated elements. In turn, the network is abstracted, from a topological standpoint, as a graph. The existence of an intermediate step when going from the system to the graph is important. In fact, this step determines the need for implementing specific, and potentially different, entities and interactions on the same network.

On a network, the number of operations and procedures we can apply is huge. This implies structural and functional issues. To deal with this situation, our starting point is to decouple structure from function in the API. Hence, we organize the API as a set of components, minimizing the coupling between them. Thus, modification or extension of capabilities in one component does not affect another. This approach is illustrated in Fig. 2, as a UML package diagram [19]. In this diagram, the API components correspond to different packages. Here, a package is a logical container or namespace that groups related elements together. For the sake of clarity, Fig. 2 includes packages for some of the most common tasks performed on networks: generation of different kinds of networks [20], handling of degree probability distributions [21], path searches [22], or communities detection [23]. All these cases exhibit a “use” relationship with the package responsible for building the basic core structure of the network. That means that any functional package needs to use some network provided by the structural counterpart. Therefore, the structural component should be developed in first place.

Regarding the above structural problem, it is worth having a clear picture of the current state of the art. Thus, it is interesting to review briefly the characteristics of the available software packages useful for treatment of complex networks. Here, we include some tools designed for graph description and handling, although not specially oriented to complex networks. The result of our survey is collected in Table 1, where we have included representative examples of the different kinds of available software tools. The list does not try to be exhaustive since the number of network/graph oriented software packages is simply huge. Instead, we have included representative examples of the different kinds of tools.

We observe that the available software packages fall into three different categories, represented by three sections in Table 1. The first approach, see first block in Table 1, corresponds to systems designed to work on a single computer or commodity cluster. That is, to work on systems where interprocess communication is not a main concern. This is the usual approach found in packages for handling complex networks or in tools oriented to graph treatment. In all these tools, the data structures used to represent the graph underlining the network are assumed to fit in main memory. Therefore, treatment of networks, or graphs, is fast, but the size of the problem is limited by the available memory. This option is unsuitable for very large problems. The second case, corresponds to the approach adopted by GraphChi, see Table 1. Here, a single computer is used, but the data structures associated with the network are essentially managed on-disk [24]. This allows the treatment of much larger networks. The counterpart comes from the slow disk access when compared to memory access. In this case, it is essential to optimize on-disk operations. GraphChi does this by resorting to a clever design of a sliding-window algorithm [24]. Finally, the third group, last block of Table 1, corresponds to software systems oriented to the treatment of networks in distributed computational environments. In this case, connection between the different, possibly heterogeneous, computing elements is achieved through a conventional network. The main problem to solve in these systems is load balancing the workload among the different computers over the network. Here, Google’s Pregel paved the way [25], followed by several open-source initiatives, see Table 1. The main advantage of these tools is its scalability, which allows for the treatment of extremely large networks. These software packages apply the Bulk Synchronous Parallel (BSP) model of distributed computation introduced by Leslie Valiant [26]. BSP is simple to implement in distributed systems. However, it works on iterations (supersteps), which imply a costly synchronization. In addition, BSP needs to store two versions of the information used: one from the current and one from the previous superstep. In these Pregel-like systems, algorithms are implemented in a vertex-centric way. Standard graph algorithms under this computation model can incur in unnecessary inefficiencies. These are mainly associated with slow convergence (excessive number of supersteps) and high communication cost. Several optimization techniques have been

Table 1
Characteristics of several representative network/graph software tools.

Name	Language	Objects in nodes and edges	Parallel distributed memory version	Parallel shared memory version	On-disk data structures
(a) Network	R	Only attributes	No	No	No
(b) Graph	R	Only attributes	No	No	No
(c) JGraphT	Java	Yes	No	No	No
(d) JUNG	Java	Yes	No	No	No
(e) NetworkX	Python	Yes	No	No	No
(f) igraph	C/C++	Yes	No	No	No
(g) GraphLab	C++	Only attributes	Yes	Yes	No
(h) Boost GL	C++	Only attributes	Yes	No	No
(i) Annas	Java	Only attributes	No	No	No
(j) STINGER	C	Yes	No	Yes	No
(k) SNAP	C	No	No	Yes	No
(l) GraphChi	C++, Java	Only attributes	No	Yes	Yes
(m) Pregel[a]	C++	No	Yes	No	No
(n) GPS[a]	Java	Yes	Yes	No	No
(o) Hama-Graph[a]	Java	Yes	Yes	Yes	No
(p) Giraph[a]	Java	Yes	Yes	Yes	Yes

(a) Network. A Network Package for Managing Relational Data in R. http://cran.r-project.org/web/packages/network/index.html.
(b) Graph. A package to handle graph data structures. http://svitsrv25.epfl.ch/R-doc/library/graph/html/00Index.html.
(c) JGraphT. Java graph library that provides mathematical graph-theory objects and algorithms. https://github.com/jgrapht/jgrapht.
(d) JUNG. Java Universal Network/Graph Framework. http://jung.sourceforge.net/.
(e) NetworkX. A software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks [18].
(f) igraph. Library for creating and manipulating graphs [32]. http://igraph.org/.
(g) GraphLab. A graph-based, high performance, distributed computation framework written in C++. https://dato.com/products/create/open_source.html.
(h) Boost. The Boost Graph Library. http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/index.html.
(i) Annas. Java framework designed for developers and researchers in the fields of Graph Theory. https://sites.google.com/site/annasproject/.
(j) STINGER: Spatio-Temporal Interaction Networks and Graphs Extensible Representation. http://www.stingergraph.com/.
(k) SNAP: Small-world Network Analysis and Partitioning. http://snap-graph.sourceforge.net/.
(l) GraphChi. Disk-based graph computation engine [24]. https://github.com/GraphChi.
(m) Pregel. Model and library for fault-tolerant parallel processing of graphs [25].
(o) GPS. System for scalable, fault-tolerant, and easy-to-program execution of algorithms on extremely large graphs [37]. http://infolab.stanford.edu/gps/
(p) Hama-Graph. Graph computing framework. http://hama.apache.org/.
(q) Giraph. Iterative graph processing system built for high scalability. http://giraph.apache.org/.
Last access to all URL's April 2015.
[a] These systems use the Bulk Synchronous Parallel model [26] of distributed computation.

proposed to reduce these costs [27]. In that work, the authors found a clear performance improvement when the optimizations are used on several fundamental graph algorithms such as: identification of strongly and weakly connected components, computation of a minimum spanning forest, graph coloring, and approximate maximum weight matching.

Our survey, see Table 1, shows that very few systems allow association of true objects, i.e., encapsulations of attributes (data) and procedures (methods), on nodes and edges. When this object association is not possible, we cannot use nodes and edges of different type on the same network since polymorphism can only be used through class inheritance. In addition, Table 1 also shows that the collected software tools are oriented to specific computational conditions. None is designed to address, at least putatively, several computational environments.

Therefore, to reach the goals of time efficiency, generality and user transparency, we propose here an API satisfying the following requirements:

i. *It should be fully object-oriented to ease its use and maintenance, to allow portability between different languages and to achieve full, individual modeling of networks, nodes, and edges.*
ii. *It should be able to model different network types such as: directed, undirected, weighted, and unweighted.*
iii. *It should allow heterogeneous, generic objects on nodes and edges.*
iv. *It must be able to handle dynamic, evolving, networks (i.e., edges and nodes should be able to appear and disappear from the network at any time).*
v. *It should work on multicore architectures.*
vi. *It should work on distributed environments.*
vii. *It should be able to handle on-disk data structures.*
viii. *For the sake of efficiency, it should be written in a compilable mainstream language.*

2.2. Software design

The first step in the development of the proposed API is to build the structural core of the system, *i.e.*, the networkStructure package depicted in Fig. 2.

As indicated in the previous section, the API will be fully developed under the object-oriented paradigm. Therefore, individual classes will describe networks, nodes, edges, elements/agents and interactions. In this form, each class will provide a full set of methods to handle its attributes and to obtain the desired functionality. Using this approach, dynamic, evolving, networks can be described by including in the corresponding classes an appropriate set of methods. First, we include the usual methods needed to create edges and its associated nodes. Second, we introduce methods for removing nodes and delete their incident edges. Finally, we add methods for removing specific edges. To achieve the desired independent behavior of classes, we apply the adapter design pattern [28] to hide the existence of inner classes from the user.

We represent the set of nodes and edges forming a network, using the object-oriented incidence list approach as described in [29]. Therefore, two collections (from now on, we will call them "lists"), one for nodes and one for edges, are maintained by the network. In turn, each node keeps a list of references to its adjacent edges. In the case of directed networks, each node splits the edge list in two: the in-list, which collects the incoming edges to the current node, and the out-list, containing the outgoing edges to the current node. To enhance performance, in particular when using array-based data structures, nodes and edges are identified

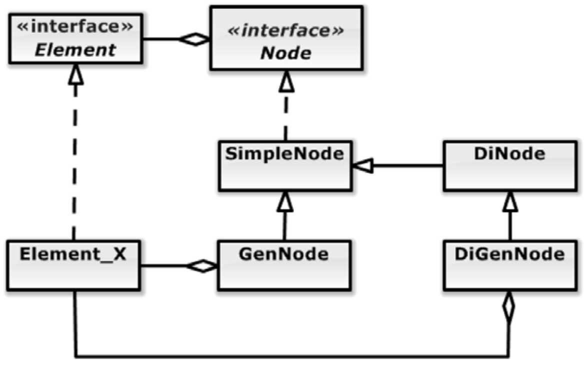

Fig. 3. UML class diagram defining the proposed model for the uniform description of different nodes, edges, elements and interactions.

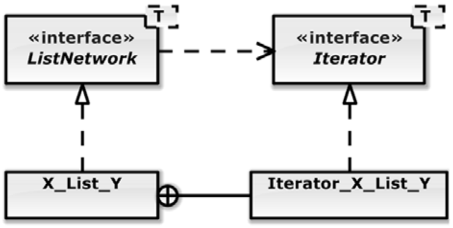

Fig. 4. UML class diagram defining the model proposed to achieve generalization of network description in different computational environments. The nested relationship between classes X_List_Y and Iterator_X_ List_Y indicates that the latter is an inner class of the former.

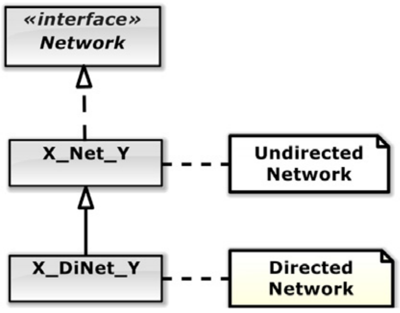

Fig. 5. UML class diagram defining the model proposed to achieve uniform treatment of different network types.

by an integer. This integer is used as key for the different data structures.

Within this general framework, three main points need addressing:

i. *How to achieve heterogeneity in the entities associated with nodes and edges.*
ii. *How to work transparently in different computational environments.*
iii. *How to use the same syntax to interact with different kinds of networks.*

Let us consider these points in order.

To allow heterogeneous elements/agents on nodes and interactions on the edges, we resort to polymorphism as shown in the UML class diagram of Fig. 3. Therefore, we define the functional behavior of nodes and elements in the interfaces *Node* and *Element*. In addition, using an aggregation relationship, class *Node* includes a class *Element* reference as an attribute. This attribute refers to the object describing the element associated with the node. *Element* is extended (inherited) in as many descendant classes, Element_X in Fig. 3, as needed. Thanks to polymorphism, see section 14.2 in Ref. [30], the parent class reference can refer to any object of a descendant class. In turn, each descendant class can implement a customized version of the original methods defined in the interface. In this form, we can associate objects of different classes to nodes of the same network. The same technique is applied to edges and interactions.

Fig. 3 also shows the existence of four node classes, all of them descendants of the *Node* parent class. The SimpleNode and DiNode classes represent nodes without associated elements. On the other hand, GenNode and DiGenNode are classes that include an element as an attribute. SimpleNode and GenNode describe unoriented nodes, i.e., nodes of undirected networks where incident edges are all equal. In turn, DiNode and DiGenNode represent directed network nodes, where incident edges are divided in incoming and outgoing edges. These sets of edges are stored in a data structure that is handled through the *ListNetwork* interface depicted in Fig. 4.

Polymorphism is also used to achieve a uniform network description in different computational environments. The key point is to associate each computational environment with specific versions of the data structures used to store nodes and edges. As shown in Fig. 4, a basic functional interface, *ListNetwork*, is defined as an abstract, generic, class. Here, we specify the general behavior of the data structures. Concrete classes inheriting from the interface represent particular implementations of this specification, see Fig. 4. These classes are labeled as X_ List_Y, where X represents the computational model used (for instance, S: sequential; P: parallel). In turn, Y identifies the data structure used to represent the “list” of nodes or edges (for instance, arrays or associative maps). Any descendant class needs to implement an iterator that allows traversing the elements stored. Thus, we define a generic *Iterator* interface based on the Iterator pattern [28]. As shown by the dependence relationship in Fig. 4, the *ListNetwork* interface uses *Iterator* in its specification. Classes descending from *ListNetwork* implement its own iterator as an inner class, which inherits from *Iterator*, see Fig. 4. The figure also shows that any of these concrete classes is defined as an inner class of the corresponding X_List_Y class.

On the other hand, polymorphism is also used to provide a uniform treatment of different kinds of networks, as shown in Fig. 5. Thus, a *Network* interface is defined to specify the general behavior of any network. From this interface, we derive by inheritance a class defining the basic structure and functionality of an undirected, simple network. This class is used as parent class for the different kinds of networks, see Fig. 5. The meaning of X in Fig. 5 is the same as in Fig. 4. Y, on the other hand, indicates how the associated data structure works (in main memory, on-disk, or distributed). It is interesting to note that no specific classes are needed for weighted and unweighted networks. The weight, or any other attribute, can be associated with the interactions stored in the edges. Therefore, only directed and undirected networks need to be specified.

The current version of the API provides two network classes: S_Net_InCore and S_ DiNet_InCore for describing, in main memory, undirected and directed networks, respectively. For both classes, it is possible to select associative maps or array-based data structures for edge and node lists just by using the appropriate constructor, see below. For associative maps, we use red–black trees-based data structures. Red–black trees guarantee a logarithmic variation, with the number of data, of the time used by insertion, removal and find operations. On the other hand, array-based data structures perform insertion, removal and find operations in constant time, independently of the number of data. In addition, arrays offer the benefit of a more efficient use of cache memory. The counterpart is the fixed, static, amount of memory that arrays use. Therefore, associative map-based data structures are useful for dynamic networks, where the number of nodes or edges changes greatly.

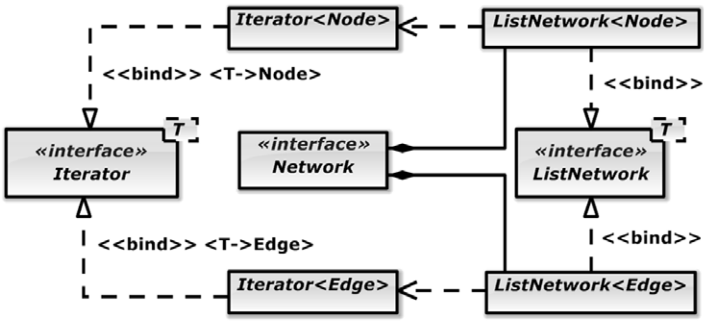

Fig. 6. UML class diagram showing the general structure of the API.

Static networks, however, can be very efficiently handled by array-based data structures. For higher flexibility, we allow resizing of the arrays as described in [31].

Finally, a general overview of the whole system is shown in the UML class diagram of Fig. 6, which collects the basic interactions between interfaces. We can observe that every network includes two realizations of the generic, parameterized, *ListNetwork* interface: one for nodes and other for edges. In turn, each of them includes a particular realization of the generic *Iterator* interface. Thanks to polymorphism, the system structure does not depend on any specific realization of the interfaces. This allows abstraction from the specific nature of the underlying classes, which define the computational environment and the kind of network, see Figs. 4–5. In this form, we achieve the desired independence of structure and function.

Comparison of our design with those of previous packages is difficult due to the lack of specific information. Usually, the provided information focuses on algorithmic performance rather than in software structure (see, for instance, GraphChi [24] and igraph [32]). The information available on the parallel Boost Graph Library [33] specifies the use of an interface that abstracts away the details of particular graph data structures, similarly to our design. However, only properties (data) can be associated with vertices and edges. On the other hand, Ref. [25] indicates that Pregel uses a vertex abstraction allowing objects to be stored on the vertices. Our design uses a similar approach but on both vertices and edges. In summary, when compared to previous packages and tools, the specific advantages of the present design are:

i. *It allows inclusion of heterogeneous objects on nodes and edges of the same network.*
ii. *The same functional interface can be used to build and handle networks in different computational working environments (for instance, in a single machine or in a distributed system).*
iii. *The same functional interface can be used to represent the vertices and edges of a graph with different data structures on different storages (for instance, on main memory or on-disk).*

2.3. Implementation

The described network structural package is implemented as a namespace in ANSII C++11. In addition, data structures are dynamically allocated in the heap by using the *new* operator. Thus, all auxiliary data structures are defined and allocated when needed. The associated memory is released when it is no longer necessary.

2.4. Using the API

The network structural package is defined in the networkStructure namespace. We can access the available classes and interfaces using the scope resolution operator (::) on the namespace. Alternatively, we can use the namespace implicitly by making

# using namespace networkStructure;

Now, the program must include the definition (header) file of the classes to use. Thus, for an undirected network we do:

#include “S_Net_InCore.h”;

The next step is to declare a pointer to the interface Network, which is implicitly included through the S_Net_InCore header,

Network *myNetwork;

When needed, the Network pointer can be polymorphically used to instantiate an object of any descendant class, calling the corresponding constructor. For instance

myNetwork = new S_Net_InCore();

This constructor selects an associative map-based data structure for edge and node lists. The array-based version can be selected by creating the object as follows,

myNetwork = new S_Net_InCore(n, m);

where n and m are integers representing the maximum number of nodes or edges, respectively.

Now, all the functionality defined in the Network interface is available just by using a polymorphic call to the method of interest. For instance, to create an edge with the corresponding nodes and include all of them in the network we do

myNetwork->addEdge($n1$, $n2$);

As described in the API documentation, this method uses the $n1$ and $n2$ integers to define two nodes and the corresponding edge. In fact, a list of edges defined in this form is the simplest way to build a network.

It is interesting to note that the way to call any method defined in Network is unique, independently of the real nature of the object (that is, if it belongs to the S_Net_InCore, S_ DiNet_InCore, or any other class implementing the Network interface).

3. Performance

The approach proposed here for modeling complex networks is flexible and general. However, the pointers used to refer to node, edge and adjacency lists as well as to the different polymorphic references introduce time and memory overheads over simpler, but less general, options. Therefore, the performance of the current approach needs to be determined. To such an end, we conduct several tests to estimate the amount of time and space (memory) used as a function of the network size.

In all the tests, the system we use is a Quad-Core AMD Opteron^{TM} 2376 (2.3 GHz) with 16 GB. This is a compute node of a cluster running under the Rocks 6.1.1 distribution (based on CentOs 6.5) [34]. Compilation is carried out with the 4.7.2 version of the gcc compiler. We include the −std = C++11 option to request the use of the C++11 standard. In addition, we apply the −O3 compiling option to get the highest level of code optimization.

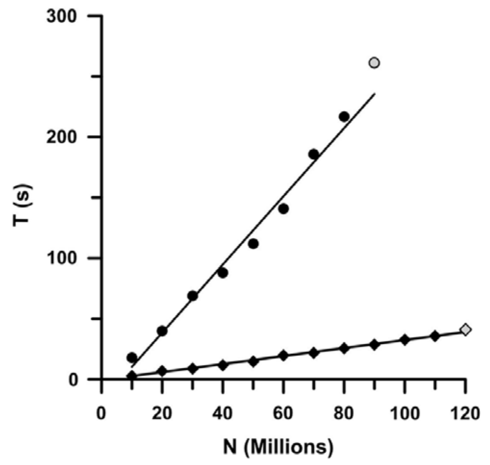

Fig. 7. Time (in seconds) used to build networks including only nodes, as a function of the number of nodes (N, in millions). Diamonds and circles mark the experimental data for array and map-based data structures, respectively. The gray diamond and the circle at the end of the *x*-axis represent networks that have exhausted the available computer memory. Solid lines correspond to linear fits obtained for networks fully maintained in memory. The fits are extrapolated to the last point of their set.

3.1. Relative performance of associative maps versus array-based data structures

We tackle, in first place, the performance of associative maps versus arrays when used for representing lists of elements. To such an end, we build a network containing just nodes. Thus, we only have the data structure used to represent the node list. In all cases, the number of nodes is increased until main memory is exhausted, and virtual memory begins to be used. In this form, we determine the maximum network size as a function of memory.

In all the tests, the nodes are randomly added to the network. To such an end, the Mersenne twister 19937 pseudorandom number generator [35] in its 64 bits version is used. To increase the cache performance in the map-base case, we use the FSBAllocator 2 of the Juha Nieminen's Fixed-Size Block (FSB) allocator suite [36].

The results show that the main memory (16 GB) is exhausted with 110 and 80 million nodes for the array and map-based cases, respectively. On the other hand, Fig. 7 shows the time variation of the two cases as a function of the number of nodes. The figure includes fits obtained using data of networks stored in main memory. We observe that both cases fit very well a linear variation. In fact, the squared correlation coefficient, r^2, is found to be 0.9967 and 0.9860 for the array and map-based cases, respectively. The linear time variation of the array-based data structure is a consequence of the constant time used to access an element in an array. In contrast, the map-based case uses logarithmic time for processing an element. The variation is, therefore, linearithmic, but it is well approximated by a straight line with a relatively high slope. Our results show that, for the same number of nodes, the map-based data structure is 6–9 times slower than the array-based. It is interesting to note that for the small-sized networks used in many studies (one million nodes at most) both approaches work below the one-second threshold.

3.2. Relative performance of undirected versus directed networks in linear and complete graphs

Here, we consider the relative performance of the directed and undirected network representations in two limiting cases. The first one corresponds to a linear network. In this case, each node is linked only to the previous one. Therefore, for N nodes, we have $N - 1$ edges and a linear relationship between nodes and edges. The other limiting case is the fully connected, complete, network. Now, each node is connected to every other. Thus, for N nodes, we have $N(N-1)/2$ edges and a quadratic relationship between nodes and edges. Using these two limiting cases, we perform several tests, as described in the following points, with the most efficient implementation for node and edge lists: the array-based one. In all the tests, networks are built using an edge list with randomly generated pairs of nodes for each edge. Again, the Mersenne twister 19937 pseudorandom number generator [35] in its 64 bits version is used.

We find that, in the linear case, a maximum of 37 and 25 million nodes can be stored in main memory for undirected and directed networks, respectively. Taking into account that in a linear network we have, in practice, as many edges as nodes, the previous values correspond to 74 and 50 million elements. Roughly speaking, this translates into about 4.5 and 3.1 million elements (or 2.25 and 1.55 million nodes) per GB.

For the complete network case, we find that a maximum of 8 and 7 thousand nodes can be stored in main memory for undirected and directed networks, respectively. Taking into account the relationship between the number of edges and the number of nodes in a complete network, these values correspond approximately to 32 and 24 million elements. In rough terms, this amounts to 2 and 1.5 million elements (or 500 and 430 nodes) per GB.

In addition, we consider the variation of network building time versus number of nodes. Fig. 8 collects the results for linear networks, case (a), and for complete networks, case (b). As in Fig. 7, we include least squares fits obtained from the data corresponding to networks fully stored in main memory. Fig. 8, case (a), shows that for a linear network, the undirected and directed cases fit well a straight line. The squared correlation coefficients, r^2, are found 0.9992 and 0.9985 for undirected and directed networks, respectively. On the other hand, case (b) of Fig. 8 shows a non-linear variation of time with the number of nodes. The data fit now a quadratic variation, with r^2 values of 0.9999 and 0.9998 for undirected and directed networks, respectively. The observed variation is a consequence of the relationship between nodes and edges: linear in linear networks and quadratic in complete networks.

With respect to the building time difference between undirected and directed networks, the data obtained for networks stored in memory show that linear directed networks are between 1.35–1.43 times slower than undirected ones. In the case of complete networks, directed networks are slower than undirected ones by a factor of 1.27–1.42.

3.3. Relative performance of APINetworks with respect to previous software packages

Finally, another point of interest is the performance of the present API against existing representative software tools. As shown in Table 1, the number of options is overwhelming. Therefore, we choose representative examples of well-known and current packages implemented in languages different from the one used here, C++. In particular, we have selected the ubiquitous NetworkX Python package and the recent Java JGraphT library. For the considered packages, we have determined the time taken to build linear and complete networks as a function of the number of nodes. To such an end, we select array-based data structures for APINetworks. NetworkX and JGraphT use hash tables by default. In the case of JGraphT, we resort to their linear and complete graph generators to build the networks.

For the lineal case, Fig. 9 case (a), we increase the network size in steps of 5 million nodes until the system memory is exhausted. As previously shown, APINetworks can handle networks up to

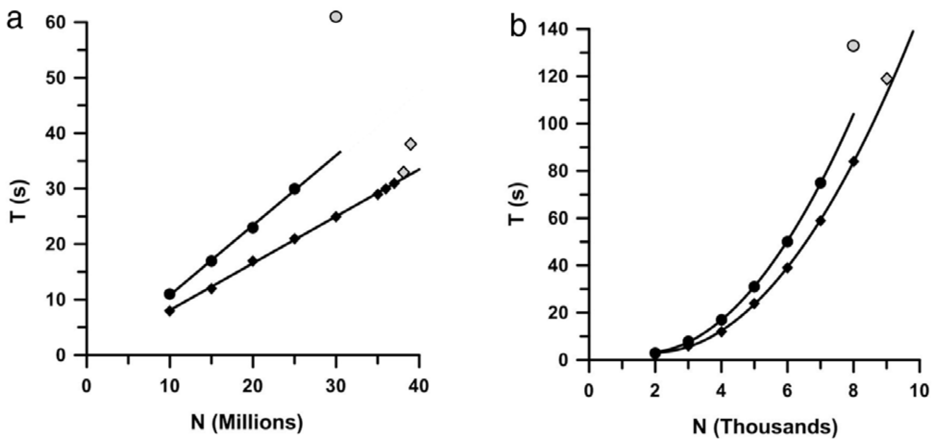

Fig. 8. Time (in seconds) used to build linear networks, case (a), and complete networks, case (b), as a function of the number of nodes, N. Diamonds and circles correspond to the experimental data for undirected or directed networks, respectively. The gray diamond and the circle at the end of the *x*-axis represent networks that have exhausted the available computer memory. Solid lines correspond to fits obtained for networks fully maintained in memory. The fits are extrapolated to the last point of their set.

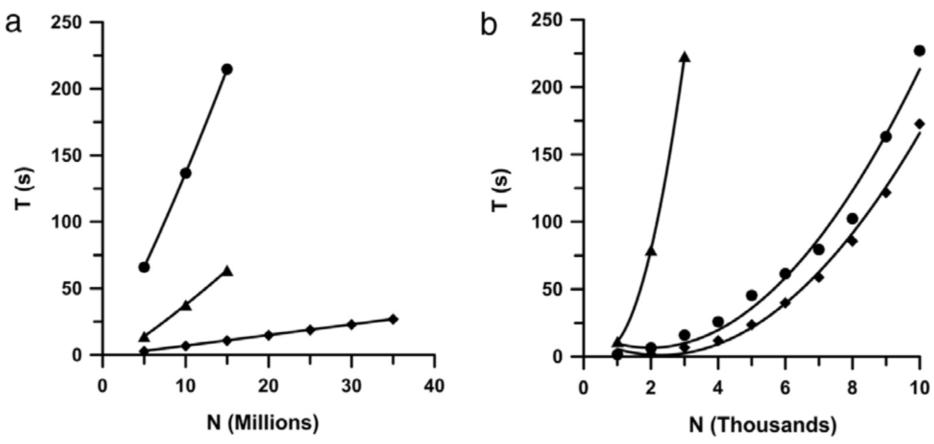

Fig. 9. Time (in seconds) used to build linear networks, case (a), and complete networks, case (b), as a function of the number of nodes, N, by the packages APINetworks (Diamonds), NetworkX (Circles) and JGraphT (Triangles). Solid lines correspond to polynomial fits to the data included in the plot.

35 million nodes. However, the other two packages only admit networks up to 15 million nodes. In addition, APINetworks is faster generating the networks than the other two packages. On the other hand, we find NetworkX to be the slowest tool. This fact can be attributed to the interpreted nature of Python.

The results for complete networks are collected in case (b) of Fig. 9. Here, we increase the network size in steps of 1 thousand nodes. We observe that APINetworks and NetworkX are both able to handle networks up to 10 thousand nodes. Again, APINetwork is more efficient when building the network, but in the present case, the difference with NetworkX is not very large. In average, APINetwork is 1.8 times faster. JGraphT, on the other hand, behaves in a very different way. In fact, we have been able to build networks up to 8 thousand nodes. However, the time needed to build the network increases dramatically with the problem size (3572 s for the 8 thousand nodes case, more than thirty times slower than the other two packages). Therefore, in Fig. 9 case (b) we only include the first three points, to keep the scale of the plot within reasonable limits.

These results show that the network representation proposed in the present API compares favorably to typical existing packages.

Acknowledgments

This work has been co-financed by FEDER funds (European Union) and the Consejería de Educación, Cultura y Deportes de la Junta de Comunidades de Castilla-La Mancha (grant #PEII-2014-020-A).

References

[1] L. Euler, Commentarii Academiae Scientarum Imperialis Petropolitanae 8 (1736) 128–140.
[2] P. Erdös, A. Rényi, Publ.Math. Inst. Hung. Acad. Sci. 5 (1959) 17–60.
[3] J.L. Moreno, Who Shall Survive?, Beacon House Inc., New York, 1934.
[4] M.S. Granovetter, Am. J. Sociol. 78 (6) (1973) 1360–1380.
[5] S. Milgram, Psychol. Today 2 (1967) 60–67.
[6] R. Albert, H. Jeong, A.-L. Barabási, Nature 401 (1999) 130–131.
[7] M. Faloutsos, P. Faloutsos, C. Faloutsos, Comput. Commun. Rev. 29 (1999) 251–262.
[8] D.J. de, S. Price, Science 149 (1965) 510–515.
[9] S. Redner, Eur. Phys. J. B 4 (1998) 131–134.
[10] D.J. Watts, S.H. Strogatz, Nature 393 (1998) 440–442.
[11] A.-L. Barabási, R. Albert, Science 286 (1999) 509–512.
[12] L. da, F. Costa, O.N. Oliveira Jr., G. Travieso, F.A. Rodrigues, P.R.V. Boas, L. Antiqueira, M.P. Viana, L.E.C. da Rocha, Adv. Phys. 60 (3) (2011) 329–412.
[13] M.E.J. Newman, Networks: An Introduction, University Press, Oxford, 2010.
[14] Tulip: http://tulip.labri.fr/TulipDrupal/; last access April, 2015.
[15] Gephi: https://gephi.org/; last access April, 2015.
[16] Cytoscape: http://www.cytoscape.org/; last access April, 2015.
[17] Pajek: http://mrvar.fdv.uni-lj.si/pajek/; last access April, 2015.
[18] NetworkX: http://networkx.github.io/; last access April, 2015.
[19] R. Miles, K. Hamilton, Learning UML 2.0, O'Reilly Media, 2006.
[20] J. Prettejohn, M.J. Berryman, M.D. McDonnell, Front. Comput. Neurosci. 5 (2011) http://dx.doi.org/10.3389/fncom.2011.00011.
[21] A. Niño, C. Muñoz-Caro, Phys. Rev. E. 88 (2013) 032805.
[22] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, third ed., The MIT Press, Cambridge, Massachusets, 2009.
[23] S. Fortunato, Phys. Rep. 486 (2010) 75–174.

[24] A. Kyrola, G. Blelloch, C. Guestrin, OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation, 2012, pp. 31–46.

[25] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Proceedings of the ACM SIGMOD International Conference on Management of data, 2010, pp. 135–146.

[26] L.G. Valiant, Commun. ACM 33 (8) (1990) 103–111.

[27] S. Salihoglu, J. Widom, Optimizing graph algorithms on pregel-like systems, in: 40th International Conference on Very Large Data Bases, 1–5 September 2014, Hangzhou, China.

[28] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

[29] M.T. Goodrich, R. Tamassia, M.H. Goldwasser, Data Structures and Algorithms in Java, 6th Edition International Student Version, John Wiley & Sons, Inc., 2014.

[30] B. Meyer, Object-Oriented Software Construction, second ed., Prentice Hall, 2000.

[31] R. Sedgewick, K. Wayne, Algorithms, fourth ed., Pearson education, 2011.

[32] W.-S. Han, J. Lee, M.-D. Pham, J.X. Yu, Proceedings VLDB Endowment. 3 (1) (2010) pp. 449–459.

[33] N. Edmonds, A. Breuer, D. Gregor, A. Lumsdaine, Single-Source Shortest Paths with the Parallel Boost Graph Library. The Ninth DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ, 2006.

[34] Rocks 6.1.1.: http://www.rocksclusters.org/wordpress/; last access April, 2015.

[35] M. Matsumoto, T. Nishimura, ACM Trans. Model. Comput. Simul. 8 (1) (1998) 3–30.

[36] FSB: Fixed-Size Block Allocator suite for C++: http://warp.povusers.org/FSBAllocator/; last access April, 2015.

[37] S. Salihoglu, J. Widom, Proceedings of the 25th International Conference on Scientific and Statistical Database Management, 2013.