Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

APINetworks Java. A Java approach to the efficient treatment of large-scale complex networks

Camelia Muñoz-Caro*, Alfonso Niño, Sebastián Reyes, Miriam Castillo

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 21 June 2016
Accepted 24 June 2016
Available online 15 July 2016

Keywords:
API
Complex networks
Java
BFS
Modeling
Network science

ABSTRACT

We present a new version of the core structural package of our Application Programming Interface, APINetworks, for the treatment of complex networks in arbitrary computational environments. The new version is written in Java and presents several advantages over the previous C++ version: the portability of the Java code, the easiness of object-oriented design implementations, and the simplicity of memory management. In addition, some additional data structures are introduced for storing the sets of nodes and edges. Also, by resorting to the different garbage collectors currently available in the JVM the Java version is much more efficient than the C++ one with respect to memory management. In particular, the G1 collector is the most efficient one because of the parallel execution of G1 and the Java application. Using G1, APINetworks Java outperforms the C++ version and the well-known NetworkX and JGraphT packages in the building and BFS traversal of linear and complete networks. The better memory management of the present version allows for the modeling of much larger networks.

New version program summary

Program title: APINetworks Java 1.0

Program Files doi: http://dx.doi.org/10.17632/3pzd5v4chp.1

Licensing provisions: Apache License 2.0

Programming language: Java

Journal Reference of previous version: Comput. Phys. Commun. **196** (2015) 446

Does the new version supersede the previous version? Yes

Nature of problem: Due to the availability of large data collections, the computational modeling and analysis of large-scale complex networks are becoming a topic of great interest. However, no single computational solution does exist to model and analyze efficiently large networks in different computational environments, especially when the networks are heterogeneous and dynamic.

Solution method: To tackle the above problem, we have developed an Application Programming Interface, APINetworks, for the treatment of complex networks in arbitrary computational environments. By resorting to object-orientation and, in particular, to inheritance and polymorphism, APINetworks allows to describe heterogeneous and dynamic networks in arbitrary computational environments. Originally, a C++ version of the core structural package was developed.

Reasons for the new version: A Java version seems very attractive over the C++ because of the ease of implementation of object-oriented designs; the portability of the code; the simplicity of memory management; and the availability of tools for parallel and distributed computing. In addition, the use of Java's automatic garbage collection permits an efficient memory management, which, for a given amount of RAM memory, allows larger networks to be build and analyzed.

Summary of revisions: The APINetworks Java version introduces some specializations to the general design presented in [1]. In particular, we make use of bounded generics [2] to allow for the use of an integer as node or edge key in network modeling classes. In this form, we can make explicit use of key handling methods in these classes without losing the generality provided by generics. To such an end, we introduce an interface `Indexable`, defining a `getKey()` method returning the integer used for identification

* Corresponding author. Fax: +34 926295354.
E-mail address: camelia.munoz@uclm.es (C. Muñoz-Caro).

http://dx.doi.org/10.1016/j.cpc.2016.06.020

purposes. This interface is implemented by the `Node` and `Edge` generic interfaces, introduced in [1], and in all its descendent classes. By defining the generic type in a given class as `<T extends Indexable>`, we allow for the use of the `getKey()` method through any reference of the generic type. This capability is especially useful for node or edge processing within data structures.

When working with nodes on a network, each node needs to store appropriate references to their incident edges. In the previous C++ APINetworks version [1] we make use of a linked list as defined in the C++ standard template library. In Java, we have a similar option: the `LinkedList` class implemented in the standard Java API. However, `LinkedList` is a double linked list. So, two references are used for each element stored in the list: one to refer to the previous element and other to the next. To reduce the amount of memory used and to increase the efficiency of graph-related algorithms, which usually only needs to traverse the list of edges of each node, we have developed a singly linked list. Therefore, each node of the list stores only a reference to the next node and a data element. The singly linked list, `APINetworksList`, handles any generic element implementing the `Indexable` interface. To allow traversal of the list, we have developed an iterator. Therefore, the `APINetworksList` class implements the `Iterable` interface of the Java standard API, and an inner class `APINetworksListIterator`. This last implements the `Iterator` interface of the Java standard API. In short, the `APINetworksList` class returns an iterator than can be handled as any iterator of any other data structure of the Java API.

In the previous C++ APINetworks implementation, the set of nodes and edges of a network can be represented through a generic growable array, among other possibilities. However, in Java, the data models used in arrays and generics are not fully compatible, since arrays are covariant and generics invariant [2]. Thus, it is not possible to allocate generic arrays. The solution is to allocate arrays of class `Object` and cast them to the generic type. Using our integer key as index for the array, the access to specific elements is done in constant time. In addition, in the present Java APINetworks version, we have introduced the use of hash tables [3] with the integer key of nodes or edges as hash code [3]. Again, access to specific elements is done in constant time.

Another key point for the present APINetworks Java version is the efficiency of the automatic memory management. Thus, we test the behavior of the different GCs implemented in the JVM, using the demanding case of building a complete, fully connected, network. Here, each node is connected to every other. Thus, for n nodes, we have $\mathrm{m} = \mathrm{n}(\mathrm{n} - 1)/2$ edges and the relationship between nodes and edges is quadratic, $m = O(n^2)$. Different data structures are available in APINetwoks Java for representing the set of nodes and edges in networks. Here, we consider the array-based one, since the simplicity of the memory access used in arrays makes it the most efficient data structure. For the tests, we consider the Parallel GC, which is the standard one included in the current Java 1.8 distribution, as well as the two concurrent GCs available: the Concurrent Mark Sweep (CMS) collector and the Garbage First (G1) collector. For each of them, we build complete networks ranging from 10^3 to $20 \cdot 10^3$ nodes in increments of 10^3. The results, obtained in an Octa-Core Intel® Xeon® E5-2630 v3 (2.4 GHz) with 48 GB of heap memory for the Java API, show that the G1 garbage collector gives the best performance followed by CMS and last, by the Parallel GC. Our data show that the relative difference between the garbage collectors increases with network size. In particular, for the largest network ($20 \cdot 10^3$ nodes) the G1 and CMS collectors use only a 47% and 64% of the Parallel GC time, respectively. These values correspond to a speedup (defined as the quotient of the Parallel GC time to the other collectors time) of 2.1 and 1.6 for G1 and CMS, respectively.

The performance of APINetworks Java is tested against the previous C++ version and two popular tools in the field: NetworkX [4], in Python, and JGraphT [5], in Java. As tests, we use two different network operations. The first is a basic one: the construction of the network in the linear and complete cases. The complete case has been already introduced. In the linear one, each node is linked only to the previous one. Thus, for n nodes, we have $\mathrm{m} = \mathrm{n} - 1$ edges and the relationship between nodes and edges is linear, $m = O(n)$. The second test uses the Breadth First Search (BFS) traversal of a network, which for n nodes and m edges exhibits $O(n + m)$ time complexity [3, 6]. In all cases, we have used the G1 garbage collector for the Java APIs: APINetworks Java and JGraphT. In addition, we have selected array-based data structures for APINetworks in its Java and C++ versions. NetworkX and JGraphT use hash tables. With JGraphT, we have used the package build-in lineal and complete graph generators. In all cases, we build networks of increasing size until the system memory is exhausted.

For the linear case, we build different networks starting with 10^6 nodes and using an increment of 10^6 nodes. The results are collected in Fig. 1 case (a). We observe the lineal dependence of the running time versus the number of nodes, consequence of the linear relationship between the number of edges and nodes ($\mathrm{m} = \mathrm{n} - 1$). The variation, in all cases, fits well a linear function of the type $\mathrm{Time} = \mathrm{a} \cdot \mathrm{n}$. The worst case is NetworkX with a coefficient of determination $\mathrm{r}^2 = 0.983$. On the other hand, Fig. 1 case (a) shows that APINetworks Java is the most time efficient package followed by APINetworks C++, JGraphT, and NetworkX. On the other hand, APINetworks Java builds networks with as much as 200 million nodes (in 67 seconds) versus the 80 million of APINetworks C++, the 100 million of JGraphT, and the 50 million of NetworkX. For the largest networks built with APINetworks C++, JGraphT and NetworkX, APINetworks Java is 1.8, 4.0, and 19.5 times faster, respectively. The large difference with NetworkX can be attributed to the interpreted nature of Python.

For the complete case, we build networks starting with 10^3 nodes, incrementing the size in steps of 10^3 nodes. The results are shown in Fig. 1 case (b). Here, we observe non-linear variations of the running time with the number of nodes due to the quadratic relationship between the number of edges and nodes ($\mathrm{m} = \mathrm{n}(\mathrm{n} - 1)/2$). The variation, now, fits well a quadratic function of the type $\mathrm{Time} = \mathrm{a} \cdot \mathrm{n}^2$. Now, the worst case is JGraphT, which exhibits a coefficient of determination $\mathrm{r}^2 = 0.923$. As in the linear case, APINetworks Java is again the most efficient tool followed by APINetworks C++, NetworkX, and JgraphT. APINetworks Java processes, in 105 seconds, networks with up to 27 thousand nodes (350 million nodes + edges). This value can be compared to the 17 thousand nodes of APINetworks C++, and the 13 thousand nodes of NetworkX. For JGraphT, we have considered only data up to 4 thousand nodes, Fig. 1 case (b), since the running time becomes too large. For the largest APINetworks C++, NetworkX, and JGraphT networks considered, APINetworks Java is 7.0, 10.4, and 320.0 times faster, respectively.

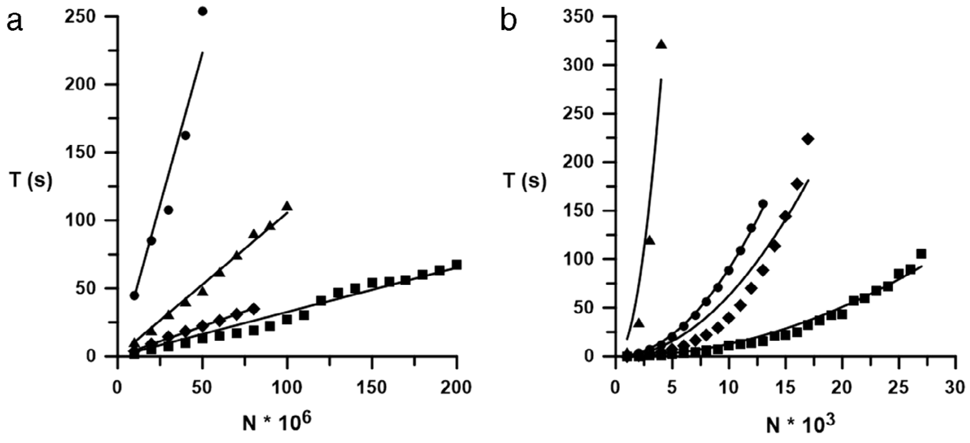

Fig. 1. Time (in seconds) used to build linear networks, case (a), and complete networks, case (b), as a function of the number of nodes, N, and the networks platform used. Squares represent APINetworks Java. Diamonds, triangles, and circles correspond to APINetworks C++, JGraphT and NetworkX, respectively.

With respect to the BFS traversal in the linear case, Fig. 2 case (a) collects the results obtained in the comparative study. First, we observe a linear variation of the running time with the number of nodes. This is a consequence of the linear dependence between the number of nodes, n, and the number of edges, m = n − 1, and the $O(n+m)$ asymptotic complexity of the BFS procedure. Clearly, in the linear case, the asymptotic complexity [6] of BFS is,

$$O\left(n+m\right)=O\left(n+n-1\right)=O\left(n\right). \tag{1}$$

Despite the oscillations observed in Fig. 2 case (a), the four curves fit extremely well a Time = a · n linear function. In fact, the worst fit is found for the APINetworks Java case with a coefficient of determination $r^2 = 0.942$. With respect to the relative performance, Fig. 2 case (a) shows that APINetworks Java is again the most efficient tool followed by APINetworks C++, JGraphT, and NetworkX. APINetworks Java needs 69 seconds to traverse the largest, 200 million nodes network. In particular, in relative terms, APINetworks Java is 1.2, 2.2, and 17.8 times faster than JGraphT, APINetworks C++, and NetworkX, respectively, for the largest network used by each package.

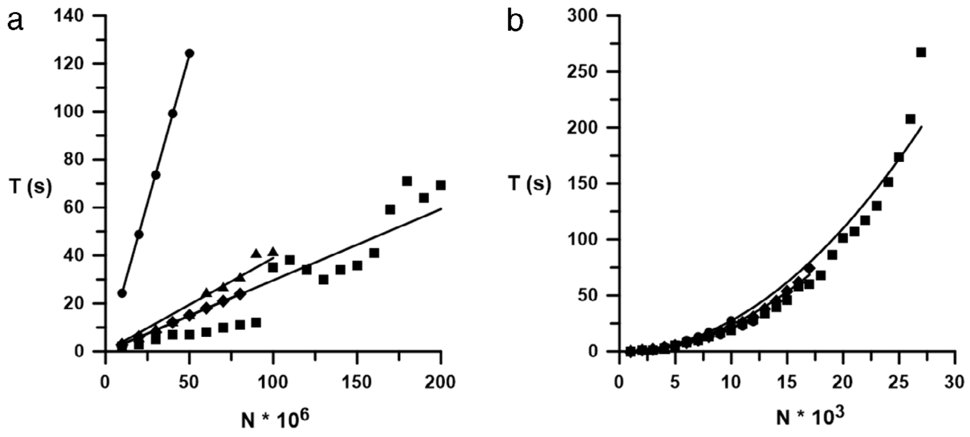

Fig. 2. Time (in seconds) used to perform a BFS traversal in linear networks, case (a), and complete networks, case (b), as a function of the number of nodes, N, and the networks platform used. Squares represent APINetworks Java. Diamonds, triangles, and circles correspond to APINetworks C++, JGraphT and NetworkX, respectively.

Finally, the results for the BFS traversal of complete networks are collected in Fig. 2 case (b). Now, we observe a non-linear variation of the running time versus the number of nodes. This is due to the quadratic dependence between the number of nodes, n, and the number of edges, m = n · (n − 1)/2, and the O(n + m) asymptotic complexity of the BFS procedure. The asymptotic complexity [6] of BFS in the current case is,

$$O\left(n+m\right)=O\left(n+\frac{n\cdot(n-1)}{2}\right)=O\left(n^2\right) \tag{2}$$

In all cases, the results fit well a Time = a · n^2 quadratic function. The worst result is obtained for NetworkX with a coefficient of determination $r^2 = 0.901$. Now, we observe that the efficiency, in decreasing order, of the different packages is: JGraphT, APINetworks Java, and, with almost identical trend, APINetworks C++ and NetworkX. However, the results obtained for JGraphT are consistently too small. It seems that the built-in complete network is identified as such by the JGraphT BFS routine and only exploration of the nodes adjacent to the first one is allowed. Being a complete network, this implies all the nodes are already visited. So, JGraphT results cannot be compared to the other packages. On the other hand, APINetworks Java needs only 267 seconds to traverse the 27 thousand nodes complete network. For the largest networks considered in each case, APINetworks Java is 0.95 and 1.2 times faster than NetworkX and APINetworks C++, respectively. However, only in the two last NetworkX cases APINetworks Java is slightly slower (an 6%, in the worst case).

Acknowledgments

The authors wish to thank the Consejería de Educación y Ciencia de la Junta de Comunidades de Castilla-La Mancha [grant # PEII-2014-020-A]. The economic support of the Universidad de Castilla-La Mancha is also acknowledged.

References

[1] A. Niño, C. Muñoz-Caro, S. Reyes, Computer Physics Communications, 196 (2015) 446-454
[2] J. Bloch, Effective Java. Second edition. Addison-Wesley, 2008
[3] M. T. Goodrich, R. Tamassia, M. H. Goldwasser, Data Structures and Algorithms in Java. 6th edition, International Student Version, Wiley, 2014
[4] NetworkX: http://networkx.github.io/; last access June 2016
[5] JgraphT: Java graph library: http://jgrapht.org/; last access June 2016
[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms 3rd Edition. The MIT Press, 2009