View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CWI's Institutional Repository

# Executable Pseudocode for Graph Algorithms

Breanndán Ó Nualláin
University of Amsterdam
bon@science.uva.nl

## ABSTRACT

Algorithms are written in pseudocode. However the implementation of an algorithm in a conventional, imperative programming language can often be scattered over hundreds of lines of code thus obscuring its essence. This can lead to difficulties in understanding or verifying the code. Adapting or varying the original algorithm can be laborious.

We present a case study showing the use of Common Lisp macros to provide an embedded, domain-specific language for graph algorithms. This allows these algorithms to be presented in Lisp in a form directly comparable to their pseudocode, allowing rapid prototyping at the algorithm level.

As a proof of concept, we implement Brandes' algorithm for computing the betweenness centrality of a graph and see how our implementation compares favourably with state-of-the-art implementations in imperative programming languages, not only in terms of clarity and verisimilitude to the pseudocode, but also execution speed.

## Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Graph algorithms; E.1 [**Data Structures**]: Graphs and networks; D.3.3 [**Language Constructs and Features**]: Patterns; D.2.3 [**Coding Tools and Techniques**]: Control Structures

## General Terms

Algorithms, Design, Languages

## Keywords

Graph algorithms, Lisp macros, Pseudocode, Verification

## 1. INTRODUCTION

Much effort is invested in ensuring that programs faithfully implement the algorithms on which they are based. Test-driven development [7], Software Verification [13] and a variety of other methodologies have been developed in efforts to achieve this goal.

But what could be better than a computer program that not only resembles the algorithm upon which it is based so closely as to inspire confidence in its implementation, but also runs with an efficiency competitive with more verbose implementations in lower-level programming languages?

We present a proof of concept of a Common Lisp library for programming in this manner and argue that it fulfils the above desiderata as well as having further advantages for pedagogy and experimentation in the field of algorithms.

Pseudograph is a Common Lisp library which provides this functionality permitting graph algorithms to be written in a manner similar to their pseudocode.

## 2. PSEUDOCODE

The *lingua franca* for presenting algorithms is pseudocode. Pseudocode is a jargon intended to be understood by practising professional programmers and computer scientists but lacking any formal semantics or standard. Students of data structures and algorithms learn pseudocode from their textbooks [10]. Computer scientists use it to publish descriptions of novel algorithms [12].

Pseudocode describes algorithms in a way which is programming-language independent. Machine-level implementation details are omitted, as are any consideration of data abstraction and error handling.

Although pseudocode is intended to be read by humans, not by machines, some attempts have been made to design programming languages which have more of a natural-language nature [16, 1]. Indeed, the syntax of the Python programming language [26] has been praised for its clarity and the natural way in which it can express algorithms [21].

## 3. LISP APPROACHES

In his book Practical Common Lisp [27, pp. 250–252], Peter Seibel presents an approach to using Common Lisp's `tagbody` and `go` special operators to "*translate*" algorithms from pseudocode into working Lisp code which is subsequently manually refactored into more natural Lisp code.

As an example he translates Donald Knuth's Algorithm S from the Art of Computer Programming [19, p. 142].

First the algorithm is translated into Lisp code which although "*not the prettiest*" is a verifiably "*faithful translation of Knuth's algorithm*". Subsequently the code is manually refactored, checking at each step, until it no longer resembles Knuth's recipe but still gives confidence that it indeed implements it.

This approach goes against the grain. Lisp is a programmable programming language that we should be able to bend to our will, shaping the language from the bottom up until it can express the problems we are tackling at the level at which they are expressed [17].

Lisp is the ultimate extensible programming language, deriving this power from the Lisp macro. Lisp macros allow us to create new binding constructs and control constructs, alter evaluation order, define data languages and improve code readability. Lisp macro programming is the extension of

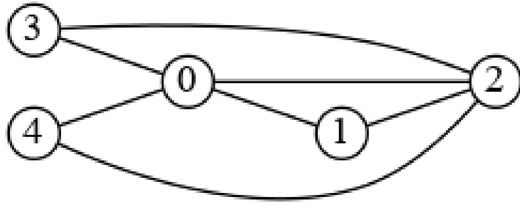


**Figure 1: An undirected graph**

the Lisp language by developing domain-specific languages (DSL) [15] embedded within Lisp itself.

Here we put macros to work to define such a DSL for expressing graph algorithms which are written in pseudocode. Graph algorithms can then be written in the DSL in their natural form once and for all, not requiring any further rewriting or refactoring.

# 4. GRAPH ALGORITHMS

Graph theory is a field of mathematics dealing with relations between objects [8]. An undirected graph is a set of nodes (or vertices) together with a set of edges (unordered pairs) on these nodes. Such a graph is usually depicted as in Figure 1; nodes as numbered circles and edges as links joining them.

Many other kinds of graphs can be considered such as directed graphs and graphs with weights or other attributes associated with their nodes and/or edges but for the purposes of this article we will limit ourselves to undirected graphs of a simple nature as in Figure 1.

In recent years graph theory has found application as the underlying model for the study of Complex Networks [23], such as social, computer and biological networks. With the recent increase in size of available networks and of the associated data sets, much research has been focusing on algorithms for computing properties of graphs.

As a case study, we consider one such graph property. The *betweenness centrality* of a node in a graph is a measure of the network load that passes through that node. It is the accumulated total number of messages passing through that node when every pair of nodes sends and receives a single message along each shortest path connecting the pair. This notion was first introduced in graph theory by Anthonisse in 1971 [4] who named it the "*rush*" of the graph. In the world of sociology, betweenness centrality was introduced by Linton Freeman in 1977 [14] and has become an important measure of the relative influence of members of social networks.

A straightforward algorithm based on all-pairs shortest paths can calculate the betweenness centrality in time $O(n^3)$ where $n$ is the number of nodes in the graph. Several incremental improvements were proposed but a breakthrough came in 2001 in the form of Brandes' algorithm [9], which runs in $O(nm)$, where $m$ is the number of edges in the graph. Brandes' algorithm consists of two phases: the computation of the lengths and numbers of shortest paths between all pairs; and then, the summing of all pair dependencies. Brandes' innovation was to recognise that the dependencies could be summed cumulatively by maintaining several attributes at each node of the graph.

# 5. EXECUTABLE PSEUDOCODE

In order to express Brandes and similar algorithms, we employ Common Lisp macros to build a DSL for graph algorithms. This DSL is PSEUDOGRAPH.

Central to PSEUDOGRAPH are two macros, `vlet` and `elet` which allow attributes on vertices and edges, respectively, to be declared, initialised, assigned and updated. Together with further macros for iteration over nodes and edges, they form the core of PSEUDOGRAPH.

In PSEUDOGRAPH the $n$ vertices of a graph are represented by the consecutive integers $[0, n)$. Its undirected edges are represented by unordered pairs of vertices. PSEUDOGRAPH uses straightforward implementations of queues and stacks.

Brandes' algorithm as originally presented in his paper [9, p. 10] is shown in Figure 2. We have highlighted some sections in colour. These highlighted sections correspond to the similarly coloured sections in the PSEUDOGRAPH implementation of the algorithm in Figure 3. An explication of the usage in the various sections of the implementation follows.

The code sections which are not highlighted are merely Common Lisp code containing calls to underlying stack and queue libraries.

The sections highlighted in red make use of PSEUDOGRAPH macros for iteration over the nodes of a graph (`do-nodes`) and over the elements of stacks (`do-stack`) and queues (`do-queue`). Note how `do-stack` (respectively `do-queue`) pops an element from the given stack (queue) and binds a variable to that element for the body of the macro. This corresponds directly to Brandes' pseudocode for

**while** S not empty **do**
  pop w ← S;
  ...
**end while**

The three sections highlighted in blue are the header parts of uses of PSEUDOGRAPH's `vlet` macro (*"vertex let"*). `vlet` takes care of the initialisation of and assignment to the vertex attributes of a graph. Like Common Lisp's `let` special form, `vlet` binds values to the specified variables in the lexical scope of its body, `vlet` differing in that it binds a vector of (copies of) the given value to each variable. The size of these vectors is given by the first argument to `vlet`. For example, the `vlet` in the second blue block makes three vectors of size $n$, each element of the first vector containing (a copy of) an empty queue, each element of the second the `fixnum` 0 and each element of the third the value of the variable `unfound`. These values are then bound in parallel to the variables `pred`, `sigma` and `dist`, respectively.

As well as binding initial values to its variables, `vlet` defines a number of macros which have bindings local to the body of the `vlet`. These local macros permit operations assigning values to elements of the locally defined vectors such as updating, incrementing, enqueueing, *etc.*. Uses of these local macros are highlighted in yellow in the figures. Examples of the behaviour of these local macros appearing in Brandes' algorithm are as follows.

- The form `(dist v)` accesses the `v`th element of the vector `dist`
- The form `(sigma start = 1)` assigns the value `1` to the `start`th element of the vector `sigma`
- The form `(sigma w += (1+ (dist v)))` increments the `w`th element of the `sigma` vector by one plus the `v`th

**Algorithm 1:** Betweenness centrality in unweighted graphs

```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V;   σ[s] ← 1;
    d[t] ← −1, t ∈ V;   d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            end
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
            end
        end
    end
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + σ[v]/σ[w] · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
    end
end
```

**Figure 2: Brandes' algorithm in pseudocode (from [9, p. 10], colouring added).**

```
(defun brandes (G)
  (let ((n (node-count G))
        (unfound -1))
    (vlet n ((bc 0.0))
      (do-nodes (start n)
        (let ((S (make-stack))
              (Q (make-queue)))
          (vlet n ((pred  (make-queue))
                   (sigma 0)
                   (dist  unfound))
            (sigma start = 1)
            (dist start = 0)
            (enqueue start Q)
            (while ((v (qpop Q)))
              (push v S)
              (do-nodes (w (neighbours G v))
                (when (= (dist w) unfound)
                  (enqueue w Q)
                  (dist w = (1+ (dist v))))
                (when (= (dist w) (1+ (dist v)))
                  (sigma w += (sigma v))
                  (pred w enqueue v))))
            (vlet n ((delta 0.0))
              (do-stack (w S)
                (do-queue (v (pred w))
                  (delta v += (* (/ (sigma v) (sigma w))
                                 (1+ (delta w)))))
                (unless (= w start)
                  (bc w += (delta w)))))))
      (bc))))
```

**Figure 3: Brandes' algorithm in Common Lisp "executable pseudocode"**

element of the `dist` vector.

- The form `(pred w enqueue v)` adds the value `v` to the queue in the `w`th element of `pred`. `vlet` allows the use of other operations similar to `enqueue` in its place permitting access and update of other data structures where required.

- Finally, the form `(bc)` returns the entire vector `bc`

## 5.1 Advantages

The PSEUDOGRAPH DSL provides a number of advantages over traditional implementations.

The local macros in the `vlet` body permit the expression of the operations in the pseudocode of Brandes' algorithm in a manner sufficiently similar to the original pseudocode to allow immediate, *at-a-glance* comparison. This gives confidence that the program is indeed a faithful implementation of the algorithm in pseudocode form.

The resulting code is succinct, a mere 30 lines. In fact the resulting code is almost line-for-line equivalent to the original pseudocode. As a comparison, the implementation of Brandes' algorithm in the Boost Graph Library [28] runs to over 600 lines of `C++`. Checking that this code faithfully implements the pseudocode is not a trivial task and certainly not as immediate as checking the PSEUDOGRAPH code.

Since the PSEUDOGRAPH code is executable, experimentation with graph algorithms can be carried out at the pseudocode level. A computer scientist wishing to investigate variants of a graph algorithm can immediately edit, execute and experiment without having to carry out intricate coding in a large code base. In this way, Lisp's advantage as a vehicle for rapid prototyping [25], with incremental development, program introspection and read-eval-print loop allowing short development cycles can be brought right to the algorithm level.

The PSEUDOGRAPH DSL provides a concise API surface area. The correct coding of a variety of graph algorithms on this DSL gives confidence in the correctness in the underlying Lisp code.

PSEUDOGRAPH can also be used for educational purposes. Krishnamurthi points out how "*Lisp programmers have long used macros to extend their language*" before continuing to lament the "*paucity of effective pedagogic examples of macro use*" [20]. Since PSEUDOGRAPH has a *common vocabulary* with text books on graph algorithms, students can readily implement and experiment with them at the pseudocode level leading to better understanding.

And finally, the process can be reversed and pseudocode in LaTeX format can be generated and pretty printed from the PSEUDOGRAPH implementation of an algorithm allowing the results of such experimentation to be included in documentation and reports with the guarantee that no errors have been introduced during transcription.

One could consider making the syntax available within `vlet` even more like the corresponding pseudocode by permitting expressions such as `(sigma[s] = 1)` for $\sigma[s] = 1$. We choose not to do this for the same reasons that we prefer the syntax of Jonathan Amsterdam's `iterate` package [3] over the syntax of Common Lisp's `loop` facility [29].

Where Lisp's `loop` facility provides a complete "*Pascalish*" sublanguage for carrying out iteration, the `iterate` package provides a more naturally Lisp-like syntax which is more easily embedded, extended and customised. In our view `loop` could be seen as `iterate` taken too far but we acknowledge this may be a matter of taste. We feel that our *executable pseudocode* is similar enough to the original pseudocode for *at-a-glance* comparison, while still enjoying all the advantages of Lisp syntax as enumerated by Amsterdam in [2].

## 5.2 Performance

With all of the advantages listed above, one might be forgiven for thinking that the resulting PSEUDOGRAPH code would be unable to compete for speed with carefully coded implementations in lower level imperative programming languages. However nothing is further from the truth.

Note, for example, that the initialisation of variables in the `vlet` macro also permits any keyword arguments accepted by `make-array` [29] such as type declarations, for example:

```
(vlet n ((cb   0.0     :element-type 'float)
         (dist unfound :element-type 'fixnum)
  ...
```

These particular declarations allow a Lisp compiler to introduce optimizations for the vertex attributes.

We performed initial comparisons of our PSEUDOGRAPH code for Brandes with implementations in NetworkX, igraph and Boost Graph Library.

NetworkX [22] is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It advertises itself as "*high-productivity software*" and it lives up to that billing by allowing the programmer to quickly set up and experiment with graphs of various sorts. However although Python shines in ease of use, it sacrifices performance. Comparison showed our implementation of Brandes in PSEUDOGRAPH to be between 2 and 3 times faster that the corresponding NetworkX implementation.

igraph [18, 11] is a collection of network analysis tools with the emphasis on efficiency, portability and ease of use. It has front ends for Python, C and R. Although the code appears fast, the igraph implementation of Brandes algorithm fails to run on graphs with 64 or more nodes due to data structure limitations. This precluded a meaningful comparison with PSEUDOGRAPH.

To get an indication of the performance of our PSEUDOGRAPH implementation of Brandes' algorithm we turned to the Boost Graph Library [28]. Boost is a collection of `C++` libraries that are open source and peer reviewed. Their libraries are widely regarded as being efficient and of high quality. In many areas Boost libraries are *de facto* standards and we take them to be the state-of-the-art yardstick against which we could get a first indication of the speed of our code.

We generated graphs of various sizes following the Newman-Watts-Strogatz model [24]. This is a model of so-called *small-world* graphs, a type of particular interest since it frequently occurs in social networks and other complex networks. The graphs are generated by connecting the nodes in a ring, then connecting each node to a number of its nearest neighbours, then rewiring each edge randomly with a fixed probability.

The comparison was carried out on a standard Linux 3.19.2 distribution on a 2.9 GHz Intel i7 processor. PSEUDOGRAPH ran on Steel Bank Common Lisp version 1.2.8 with compiler settings `(optimize (speed 3) (safety 0))`. Boost version 1.57 was compiled on `gcc 4.9.2` with the `-O3` compiler set-

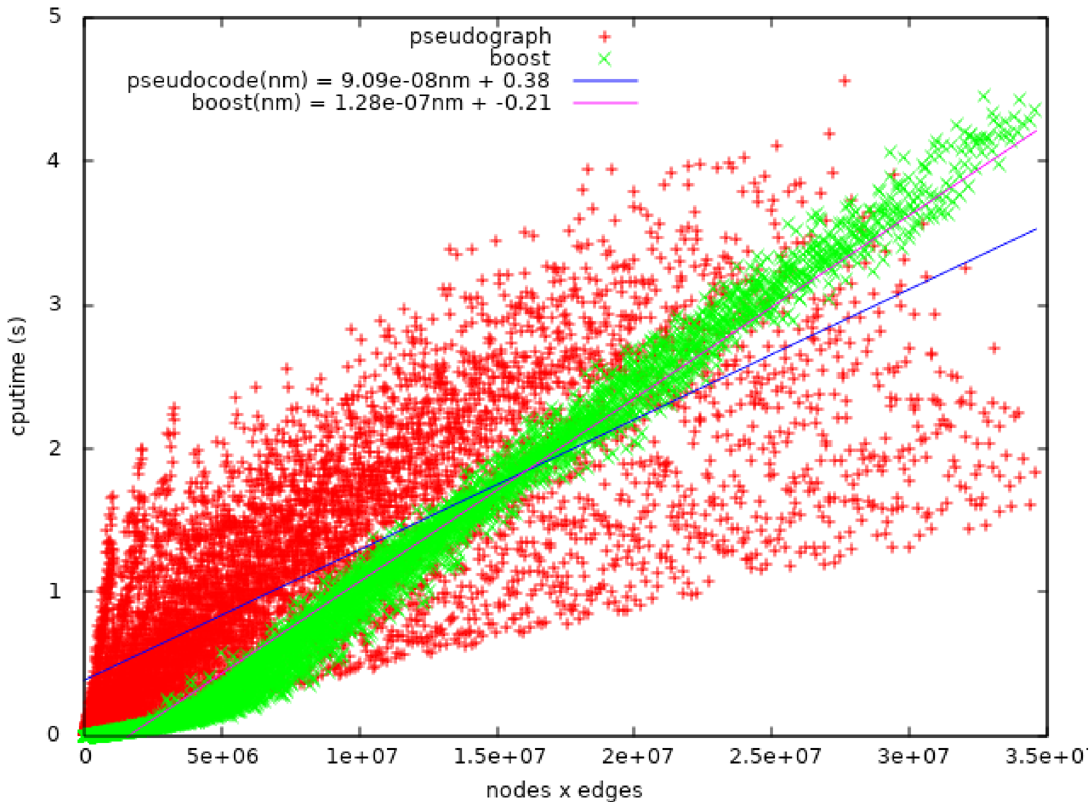


**Figure 4: Comparison of pseudograph with Boost on small-world graphs.**

ting. In both cases, these are the most aggressive compiler settings. The times shown are net run times after the graphs have been loaded into the native data structures.

The results are shown in the graph in Figure 4. On the x-axis is the product ($nm$) of the number of nodes and the number of edges of the graph. On the y-axis, the CPU time in seconds. Since Brandes is $O(mn)$, we expect the worst-case times on the graph to be roughly linear. Each data point is a run of either the PSEUDOGRAPH or Boost implementation of the betweenness centrality algorithm on an instance of a small-world graph generated according to the Newman-Watts-Strogatz model for $n \in \{10, 20, \ldots, 300\}$, $k \in \{3, 4, \ldots, n-1\}$, $p \in \{0.1, 0.2, \ldots, 0.9\}$. We make two observations on the basis of these results.

As can be seen, PSEUDOGRAPH is competitive with Boost, in some cases slower, in some faster. Linear regression on the two sets of data points (indicated by the two straight lines in Figure 4) shows the average PSEUDOGRAPH run time to be increasing more slowly than that of Boost with a crossover point at about $mn =$ `1.6e7`. This suggests that for larger graphs, PSEUDOGRAPH will continue to outperform Boost on average.

Further, it is noticeable that there is a greater variance in the PSEUDOGRAPH data points than in those of Boost. This might suggest that the performance of PSEUDOGRAPH is less predictable than that of Boost.[1]

In order to find an explanation, we look more closely at runs for a fixed number of nodes. Holding $n$ fixed at 600 and varying only the parameter $k$, generates graphs with a fixed number of nodes but a varying number of edges. The run times of neither implementation was sensitive to the value of $p$ so it was held fixed at 0.3. These results, shown in Figure 5, are typical of those for graphs of other sizes.

As we can see, the Boost and PSEUDOGRAPH implementations have differing performance characteristics for graphs which are sparse (having relatively few edges) and dense (having relatively many edges). While the run time of Boost increases uniformly with the density of edges, PSEUDOGRAPH's run time peaks before decreasing for very dense graphs. We have no explanation for this behaviour but speculate that it may be a property of `bitsets` a compact representation PSEUDOGRAPH uses for subsets of integers in a range $[0, n)$ [5]. This bears further investigation. In any case, it appears that this particular performance characteristic leads to the larger variation of PSEUDOGRAPH run times in Figure 4.

Given the many other advantages of PSEUDOGRAPH stated above, we regard the result that it can compete with the Boost `C++` library as very encouraging, especially since little attempt has been made at optimising the PSEUDOGRAPH code as yet. Currently, standard, straightforward representations are used for stacks and queues. The structure of undirected graphs is represented as adjacency lists of sets of nodes, which are represented as `fixnums`. These sets are represented as `bitsets`.

We emphasise however that this initial test, while encouraging, is by no means conclusive. We plan full tests and comparisons on other kinds of graphs and larger graphs.

[1] One might think that garbage collection is respobsible. However no garbage collection was triggered during the runs.

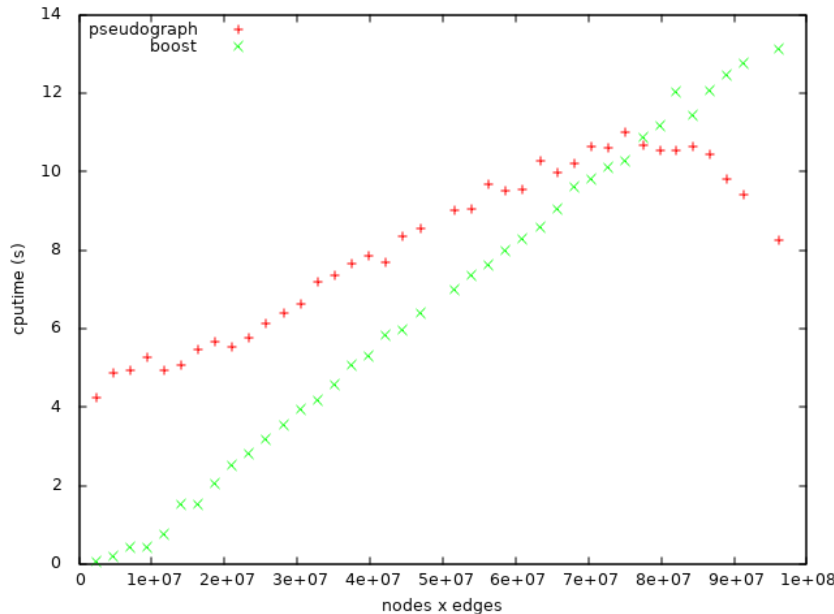


**Figure 5: Comparison of pseudograph with Boost on small-world graphs,** $n = 600$**,** $p = 0.3$

# 6. CONCLUSIONS AND FUTURE WORK

We have shown a proof of concept for our approach to programming graph algorithms by presenting an implementation of Brandes' algorithm which is not only comparable *at a glance* to the pseudocode of the original algorithm, but also competitive in speed with state-of-the-art code for the algorithm written in optimised `C++`.

To use the words of Krishnamurthi [20], we feel that this approach represents '*…a rare "sweet-spot" in the readability-performance trade-off.*' Despite the long-standing popular belief to the contrary, Lisp has been shown to be competitive with lower-level programming languages for scientific numerical computing [31, 30]. Our results now show that this competitive performance need not be limited to numerical computing.

The range of graph algorithms which we can express with the machinery of PSEUDOGRAPH as it stands is surprisingly large. As we have seen, Brandes' algorithm requires maintaining several vertex attributes. Other graph algorithms such as Floyd-Warshall requires the maintenance of edge attributes. This can be achieved using PSEUDOGRAPH's `elet` macro (*edge let*). Directed graphs can also be represented. Moreover, such is the flexibility of the approach that novel pseudocode constructs can readily be added to PSEUDOGRAPH.

We are continuing to implement other graph algorithms using this approach and experimenting with optimizations of the code *under the hood*. The bottom-up nature of the PSEUDOGRAPH code with clean interfaces makes it easy to vary such representations independently of each other.

We plan to release our work under an open-source license in the form of the PSEUDOGRAPH library which will also be packaged and submitted to Quicklisp [6].

# 7. ACKNOWLEDGMENTS

The author is grateful to Leen Torenvliet and Markus Pfundstein for helpful discussions.

# 8. REFERENCES

[1] Adobe. Lingo language support center. `http://www.adobe.com/support/director/lingo.html`.
[2] J. Amsterdam. Don't loop, iterate. Technical report, MIT Artificial Intelligence Laboratory, May 1990.
[3] J. Amsterdam. The iterate manual. Technical Report A. I. MEMO 1236, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, Oct. 1990.
[4] J. Anthonisse. *The rush in a directed graph*. Stichting Mathematisch Centrum Amsterdam. Afdeling Mathematische Besliskunde, 1971.
[5] J. Arndt. *Matters Computational*. Springer, 2011. `http://www.jjj.de/fxt/#fxtbook`.
[6] Z. Beane. Quicklisp library manager for common lisp. `http://www.quicklisp.org/`.
[7] K. Beck. *Test-driven Development: By Example*. Addison-Wesley, 2003.
[8] B. Bollobás. *Modern Graph Theory*. Graduate texts in mathematics. Springer, 1998.
[9] U. Brandes. A faster algorithm for betweenness centrality. *Mathematical Sociology*, 25(2):163–177, 2001.
[10] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
[11] G. Csárdi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.
[12] DMTCS. Discrete mathematics & theoretical computer science. `http://www.dmtcs.org/dmtcs-ojs/index.php/dmtcs`.

[13] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[14] L. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40:35–41, 1977.

[15] D. Ghosh. DSL for the uninitiated. *Commun. ACM*, 54(7):44–50, July 2011.

[16] D. Goodman. *The Complete HyperCard Handbook*. Bantam Books, 1987.

[17] P. Graham. *On Lisp: Advanced Techniques for Common Lisp*. An Alan R. Apt book. Prentice Hall, 1994.

[18] igraph. igraph. `http://igraph.org/`.

[19] D. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, MA., 1981.

[20] S. Krishnamurthi. Educational pearl: Automata via macros. *J. Funct. Program.*, 16(3):253–267, 2006.

[21] S. McConnell. *Code Complete*. DV-Professional. Microsoft Press, 2009.

[22] NetworkX. Networkx. `http://networkx.github.io/`.

[23] M. Newman. *Networks: An Introduction*. Oxford University Press, March 2010.

[24] M. E. J. Newman and D. J. Watts. Renormalization group analysis of the small-world network model. *Physics Letters A*, 263:341–346, 1999.

[25] K. M. Pitman. Accelerating hindsight: Lisp as a vehicle for rapid prototyping. *ACM SIGPLAN Lisp Pointers*, VII(1–2):14–21, January 1994. `http://www.nhplace.com/kent/PS/Hindsight.html`.

[26] Python. The Python programming language. `https://www.python.org`.

[27] P. Seibel. *Practical Common Lisp*. Apress, Berkely, CA, USA, 1st edition, 2012.

[28] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley Professional, 2001.

[29] G. L. Steele, Jr. *COMMON LISP: the language*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 2nd edition, 1990.

[30] D. Verna. Beating C in scientific computing applications. In *Third European Lisp Workshop*, Nantes, France, July 2006. `http://lisp-ecoop06.bknr.net/home`.

[31] D. Verna. How to make Lisp go faster than C. *IAENG International Journal of Computer Science*, 32(4), Dec. 2006.