

Exploring Data Structures and Tools for Computations on Graphs and Networks

K.A. Hawick

Institute of Information and Mathematical Sciences

Massey University – Albany, North Shore 102-904, Auckland, New Zealand

Email: k.a.hawick@massey.ac.nz

Tel: +64 9 414 0800 Fax: +64 9 441 8181

August 2007, Revised April 2009

Abstract

Many important applications problems can be formulated in terms of network or graph models. There are several well known approaches to storing and manipulating graph data using conventional data structures such as adjacency matrices and edge list arrays. Some new powerful possibilities are available using associative and other sophisticated data structures that are built-in to some modern programming languages. This article reviews these ideas and their use in a library and an associated tool-set for measuring graph and network properties on small and large scale networks from sources including: computer networks; biological networks; socio-nets; physical models; and other simulated systems. Directed graph networks pose particular problems of their own for efficient storage and traversal algorithms. Some techniques of interest such as graph colouring; reachability analysis; enumerating circuits and loops; and eigen-spectral analysis are computationally expensive but have different tradeoffs for optimal efficiency. A primary “neighbours” data structure of “to-arcs” is used, coupled with various support data structures and routines for conversion between different sparse and dense network data storage mechanisms. A collection of algorithms and prototype implementations in the programming language **D** are given along with a discussion of design and implementation issues for these libraries and program tools that use them.

Keywords: graph; graph algorithms; D language; adjacency list; data structure.

1 Introduction

Graphs form an important data structure for implementing many network based applications problems. There are still rather few software packages available for manipulating graphs and for efficiency reasons it is often useful to embed custom graph algorithms into a simulation program. This note provides a brief review of graph ideas, software and implementation issues in support of complex graph and network simulations and other calculations. Particular focus is on a prototype graph calculation library and set of programs for implementing single directional arcs as a neighbours list. This system was prototyped in the relatively recent “D” Programming language.

Definitions of graphs and the mathematical terminology for describing them and their properties is given in Gould’s book on Graph Theory [1] which provides many good links and references into the historical graph research literature - including classic work by: Harary [2] on graphical enumeration; Erdos and Renyi [3] on random graphs; Dijkstra [4] on all-pairs algorithms; and Floyd [5] on shortest path algorithms. Other useful general works include: Sedgewick’s book on graph algorithms implemented in Java [6]; Hartsfield and Ringel [7] on pearls in graph theory; Newman *et al.* on random graphs [8]; and the book by Newman, Barabasi *et al.* on the structure and dynamics of networks [9].

A great deal of early work was done on random graphs [10–12] with more recent work looking at percolation and associated issues [13]. There are a number of important graph problems that have attracted recent renewed interest in the literature [14]. Not least of these are the problems understanding complex and scal-

ing properties [15] of the World Wide Web [16] and the underpinning Internet networks. Understanding the properties of clustering and graph topology [17] through techniques such as graph colouring [18] and distance measurements [19] are also important. A lot of recent work has been triggered by interest in the small-world network phenomenon [20] whereby the distance properties of a whole network can be dramatically changed by just a few changed links [21].

Another important trigger for recent work has been the fast simulation and enumeration capabilities offered by parallel computing techniques and in particular by commodity data parallelism that can be used to investigate graph and network structures on Graphical Processing Units (GPUs) [22,23]. Over and above specialist parallel processing and parallel algorithms however, it is necessary to establish some suitable data structures, file formats and data exchange mechanisms for graph and network data. Discussion of this issue around some prototype implementations of a neighbour list oriented data structure is the focus of this present article.

Some discussion on the necessary data structures for implementing graphs and networks is given in section 2 as background material. Ideas on suitable programming languages are offered in 3 with a list of the prototype D programs developed given in section 4. A brief discussion of ideas on graph model generation algorithms is given in section 5; some graph metrics and measurement algorithmic ideas are presented in section 6; a discussion of graph file formats is laid out in section 7. A summary and ideas for future work are given section 8 with some conclusions.

2 Data Structures

Graph data can be stored in computer memory in a number of ways - including as lists or adjacency matrices. Memory layout is primarily chosen for algorithmic efficiency but is not unrelated to how data needs to be traversed in file input and output. This section introduces the neighbour data structure which is a compact memory efficient way of storing the main graph structural information.

Graph data comes in many forms and formats. It is a non-trivial issue to design a completely general and extensible graph data file format as so many different applications need to decorate the nodes and arcs with different data types as payloads. Formats such as GML [24] and GraphML [25] are good attempts at a mark-up oriented file format that supports different

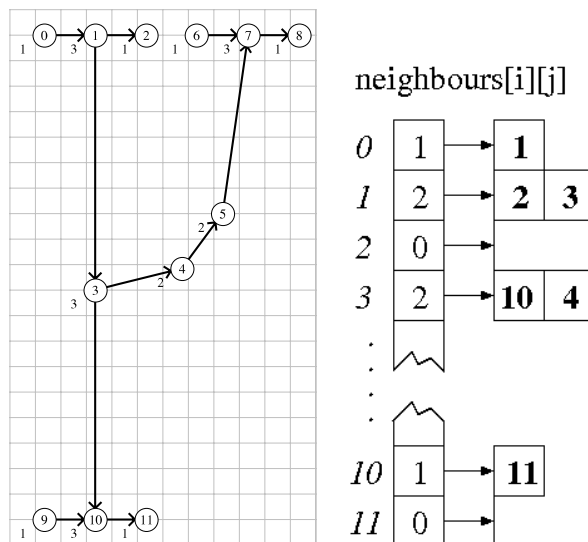


Figure 1: Left) Test Graph (k-rune) with 12 vertices and 11 di-arcs and Right) the neighbours data array to store the graph structure.

data type payloads. For the purposes of simulating network properties however a lightweight, minimalist format that encodes the graph structure is useful. It is also valuable if such a format can be read and written quickly from and to files in a way that maps closely to a memory structure and which does not require large parsing processing overheads.

The “neighbours” file format (with file ending “.nbr”) is designed to map closely to the neighbours data structure in memory and to encode that absolute minimal information to specify the structure of a graph or network. The notion is that many algorithms involve a traversal of the form “for each neighbour of each vertex...” and that furthermore the out-list of each vertex is often the most used.

The **neighbours data structure** is shown in figure 1 and is essentially a list of “to-arcs” or destination nodes for all the output arcs of each node in the graph. This can be implemented efficiently as an integer array-of-arrays in languages like C or D. In the case of C it is necessary to store the size of each neighbour list as a separate integer for each node, whereas D arrays carry their size built-in as a `.length` field. In D therefore it is easy to number the entries in each list $0..N_{\text{out-degree}} - 1$, but in C they can be indexed $1..N_{\text{out-degree}}$ with the convention that the 0th entry is used for the list length. In both cases this structure maps closely to the information in the neighbours file (shown in figure 2) which can be stepped through to allocate space for the lists dynamically on input.

Other important aspects on a practical library and set of implemented graph calculation tools relate to the software engineering capabilities of different languages. The D language, like C/C++ is a compiled language and it is possible to pre-compile a library of appropriate functions and procedures that can support an imperative of Object-Oriented program. The neighbours library and tools reported here were implemented as a set of imperative functions without an explicit OO class structure. This was for efficiency reasons, since in most cases the calculations require traversing large neighbour lists. Even the overhead of indirectly referencing nodes or arcs via an object structure is perceptible in slowing down algorithms such as circuit enumeration which have a high computational complexity.

The programs were implemented in two parts:

1. A library of useful auxiliary functions and data type definitions;
2. Explicit (and generally quite short) command line programs that operate with neighbours `.nbr` files.

The neighbour utilities were incorporated into a compiled D library that could be linked against the programs described in section 4. Some notable D programming issues for this library are discussed below.

3.1 Neighbour Arrays and Memory

All the utilities work with input/output data in the form of `int [][] neighbours` data arrays. In the case of functions that read data from file or which produce a new neighbours set from an old one, they dynamically allocate the necessary memory and return a new neighbours structure. The algorithms involved in the programs described in this suite were mostly compute bound and not memory bound. This meant there was not a huge advantage in working with pre-declared “neighbour buffers” but that it was more elegant to program dynamically allocated neighbours arrays as needed and to rely on the D garbage collector to clean up after any old unused structures. This would not necessarily be the best strategy for memory-bound problems nor for ports of these algorithms to parallel programming languages and systems.

3.2 Strings and File I/O Routines

Filenames in C/C++ are often manipulated in terms of arrays of characters terminated by a null or ‘0’ `char` whereas in D, strings are manipulated as dynamic arrays of characters with an associated and explicit length attribute. It was convenient to use many of the file handling utilities of C/C++ that are also available from D, some conversion between the two string representations was necessary. Generally D support overloaded function names well, so it was possible to make a “master version” of the requisite file routine and supply various overloaded functions on top of it to support the use of file pointers, filenames and so forth in different ways. In hindsight it would have been simpler to use the D file I/O system and the D string representation throughout.

A useful feature is to employ constructs like:

```
char [][] words = f.readLine().split();
```

whereby a line read from a file can be very easily tokenised and each word processed appropriately using `toInt()` for example.

The standard idiomatic form of traversing a neighbours file or data structure is therefore embodied by the code in figure 4 which shows some D source code demonstrating how to load a “neighbours file” dynamically allocating memory as a single loop. The D feature of assigning values to the `.length` field of the array is used to trigger appropriate memory reallocation methods inside the D array class apparatus. This works surprisingly well for even large graphs that might fit into the memory of a contemporary desktop computer. Even in cases where the structure might occupy around $10^6 - 10^7$ nodes this is feasible. An alternative model would be to make two passes through the file, the first just to tally how many neighbours each vertex node has prior to allocating appropriate memory for each neighbour index list.

The D mechanism for dynamically reallocating an array size is also convenient for making a single pass of a neighbours structure to generate the reverse neighbours or node sources. Figure 5 shows how this can be coded. This likely has some memory allocation overhead but is compact in terms of source code. In a non-garbage collected language a two pass algorithm is likely more efficient, with the first pass to compute the sizes of the reverse lists and the second to populate them. A third pass is needed in principle if the original forward arc neighbour list must subsequently be deallocated.

The D language offers some useful built-in capabilities such as associative arrays. This capability can be used to good effect to make a compact routine to identify and eliminate duplicate arcs by building frequency tables using an associative array of integers and the `int[int]myarray;` syntax. Figure 6 illustrates how the associative arrays might be coded for this.

One useful algorithm for efficiently obtaining the pair-wise hop distances between nodes is that of Floyd [5], and which is given as Java source code in Sedgewick's book [6]. Figure 7 gives our D version of this algorithm encoded to return an adjacency-like array with the pair-wise distances given in the $i - j$ 'th positions.

This code follows our philosophy of returning a simple but uncompressed memory-allocated structure that can then be reduced by summing rows or columns or can be directly looked up or saved easily to file. The D language makes it very easy to write memory allocation code this way without needing copious asserts to ensure there are no null-pointers returned by malloc or realloc.

Component labelling is another important algorithm. There are a number of variations possible, depending upon the different sort of connectivity patterns in the graph there are different tradeoffs available from different algorithms. A simple algorithm that is adequate for smallish graphs of arbitrary pattern is given in 8. Multiple sweeps pass through the integer labels ensuring a unique label results for each connected component of cluster of nodes. This makes use of the built-in `min` function of D. More sophisticated component-labelling algorithms lend themselves to parallel implementation as well [27].

D lends itself well to implementation of simple house keeping routines such as that for counting the number of components present using the unique component labels. Figure 9 shows a D code using the associative array feature for this – which conveniently turns a “hole-ridden” integer space into a compact sequence if we go back and relabel the components by their original label sequence number. This of course makes other subsequent sorting or selection operations on the components a lot easier to manage. Some examples might be pulling out the n 'th largest component or computing some property such as centre of gravity for clusters of vertices embedded in a physical space.

The D code in figure 10 shows that is relatively straight-forward to use the unique cluster labels to split the neighbours list for the whole graph apart into separate neighbours lists – one for each connected component. This is particularly useful for analysing

individual components or their properties.

Sometimes it is useful to be able to iterate over each component and have a handy list of all vertices in it. The code in figure 11 constructs this. D lets us sort this list of whole arrays according to the number of vertices (size) of each cluster.

When loading components or graphs from separate files or sources it is also useful to be able to reconstruct a coherently numbered whole graph in the form of a unified neighbours list. The code in figure 12 does this. It makes use of the `.dup` operation which duplicate-copies subarrays to build up a new clean data structure.

The code shown in figure 13 removes a list of specified vertex nodes from the structure coherently while healing the gaps in the consecutive node numbering space. This is useful in applications such as interactive graph drawing where the graph can shrink when nodes are deleted. Generally this seems a better approach than trying to maintain adjacency arrays and the like where some rows and columns are no longer valid.

4 D Implementation Programs

A suite of programs was developed to make use of this library of routines. In some cases the program was nothing more than a very thin wrapper for a single routine. The philosophy was to support easy exchange of graph and network information between application programs - and especially simulations - that were likely written, developed and maintained at different times and which might not necessarily be written in D.

Much of the work reporting in the articles: [28–38] made use of these programs.

- `neighbours_analysis.d` does cluster and histogram analysis of a neighbours network file
- `neighbours_edit.d` extracts the one (largest) cluster from a neighbours file
- `neighbours_split.d` makes 1... named files from the (separate) descending order sized component clusters
- `neighbours_unique.d` makes all the arcs unique in a `.nbr` file, removing duplicates
- `neighbours_merge.d` combines several `.nbr` files into one with a single numbering scheme

- `neighbours_prune.d` removes dangling leaves and branches with no circuits
- `neighbours_circuits.d` report on the circuits present in the graph found in a single `.nbr` file

5 Graph Generator Models

There are many possible graph generation algorithms and indeed being able to investigate the properties of such models was a motivating factor for development of the neighbours library and related programs. There is not space to discuss them all but some key models are: random graph models, scale free models and small-world systems [30], boolean networks [35] and spatially embedded systems [28].

The following graph generator “test programs” are part of the D suite accompanying the neighbours library:

- `generate_NK.d` generates variations of a Kauffman NK network including pair-wise mixed $< K >$ nets
- `generate_arb_config.d` generates r -connected points from radial proximities
- `generate_preferential.d` generates a scale-free preferential attachment network
- `generate_lattice.d` generates hypercubic lattice using usual Lengths specifier
- `generate_SW` is an initial working prototype for generating a small-world model based on a lattice

Devising new and interesting graphs with “different” or unique properties is an ongoing area for further research.

6 Graph Measurement Metrics

There are a number of useful things to characterise or measure about a network: Figure 14 gives the D source code names for some of these obvious simple properties. The variable names are used consistently in the D programs and library routines.

Some static graph properties can be investigated using simple book-keeping techniques. A set of D programs was developed to include:

- `neighbours_indegree.d` computes and histograms the vertex in-degrees
- `neighbours_outdegree.d` computes and histograms the vertex out-degrees
- `neighbours_extract_inputs.d` assuming a `.nbr` file to be “to-arcs” or outputs, this constructs the “from-arcs” or inputs
- `neighbours_components.d` labels and computes cluster statistics from a `.nbr` file and makes `.comp` compound `nbr` file
- `neighbours_allpairs.d` computes Dijkstra all pairs distance for a `.nbr` file

More sophisticated programs were developed to calculate expensive properties such as the number of circuits [30, 31, 39–42] or the path-lengths present [43]. The clustering coefficient [44] is also an interesting property that characterises network structure and for which parallel computations are possible [45].

Graph analysis is a useful approach to many applications problems and not least complex systems [46, 47]. An interesting set of open issues concerns the identification of communities – or areas of strong connectivity in an otherwise weakly connected graph. Spectral methods may give insights into these [48, 49].

Some experimental D codes to handle complex eigenvalues [50] were also constructed as part of the neighbours suite:

- `neighbours_eigenspec.d` compute eigenvalues of a network and their spectral density
- `neighbours_complex_eigenspec.d` computes (complex) eigenvalues of a network and (2d) spectral density across the complex plane

In this area and others, there is still outstanding work to do a systematic investigation of how bulk properties of various networks vary statistically with N , M and the generation algorithm control parameters such as K , p , r .

7 Graph File Formats

Various attempts have been made to establish a file format for graph information. This is not trivial as ultimately many applications use a core graph concept but decorate the nodes and arcs with diverse information. The graph markup language (GML) [24] is one

format still in use, despite it having been invented before the wide promulgation of extensible markup language (XML). GraphML [25] is another markup language for graph information and with some promise, although a simple and minimalist XML-based format appears to be eluding the community just because of the temptation to incorporate too much information over and above the core structural data.

This present article has described the very simple textual file format for neighbours lists. Some variations of this might be to provide composite neighbours lists to give forward and reverse arc information or to include some grouping to indicate separate components.

Some other format ideas I have found useful are inclusion of spatial information such as x-y coordinates in a 2-D space or x-y-z information in a 3-D space. It is not obvious what the best way to incorporate these into a format. The `.graph` file format invented for use in the GraViz graph drawing and visualisation program [26] used integer x-y-z information associated with each vertex node. This unfortunately introduces assumptions about the scale of the embedding space, and probably normalised floating point values would be more generally useful. Nevertheless for some simulations where the graph results from a simulation that itself is defined on an integer coordinate space – such as from an array index mapping – this was useful.

Simulations programs such as those for generating diffusion limited aggregation (DLA) or cluster-cluster aggregation (DCLA) models [51] made use of these techniques. The following D programs were developed to convert between formats:

- `graph_to_neighbours.d` extracts the structural information from a GraViz `.graph` file to make a `.nbr` file
- `icoord_to_neighbours.d` makes a `nbr` file from an x-y or x-y-z integer coordinates file (eg from DLA or DCLA)
- `coordinates_to_neighbours.d` makes `.nbr` file from a DLA style int xyz coordinates file

Generally however, the design of a forward scalable graph file format and a comprehensive discussion of the associated issues is beyond the scope of this present article.

8 Discussion and Conclusions

We have presented a selection of D source code fragments for manipulating graph data in the form of a neighbours list or array of destination “to-arcs” for each vertex in the graph. This has proved a compact and convenient form to load from file and store to file and allows more memory-intensive structures such as adjacency tables to be easily constructed but only when needed. Many of the algorithms of common interest are structured to have a “for each vertex...for each connected vertex...” as their outermost loops, and so the neighbours list works well in many cases.

The D language as found to quite well suited to this sort of application and for development of library routines. The GNU `gdc` compiler was used mostly and generally D offers a combination of elegance and performance efficiency that is not met by C, C++ or Java. Unfortunately during the course of this work D has waned in general popularity as a programming language internationally. This is partially due to improvements in other languages but mostly I believe because the D programming support libraries have not had the effort expended in them rapidly enough to allow D to really take off. I rather regrettably find myself reimplementing many of the elegant constructs I prototyped in D in C++ again. The type checking offered by `writeln` in D is now also offered at some level by C and C++ compilers for `printf`. The various built-ins such as: arrays sorts; dynamic memory re-allocation; and associative arrays, can of course be implemented in C++ user classes anyway. The more elegant pointer and reference handling apparatus in D is still attractive and I believe is better done than in C# or Java but does not appear to have been a strong enough selling feature for the majority of programmers. At the end of the day developing code is expensive and one wants to believe the platform will widely available for at least a decade to be worth maintaining effort in it. I still hope D may experience a recovery in popularity however.

There remain many interesting graph related simulation problems to work on, and it seemed worthwhile writing up experiences with this software project, so that some aspects can serve for future work on graph and network simulation calculations. There is some hope for the data-parallel languages such as NVIDIA’s proprietary Compute Unified Device Architecture (CUDA) [52] for the Graphical Processing Unit accelerator devices, and for the emerging Open Compute Language (OpenCL) [53] also targeted at accelerator devices. Both these languages are strongly C/C++/D syntax and concept based and some of the

ideas discussed in this present note will hopefully find reuse [27, 45] there.

Acknowledgements

Thanks to H.A.James and A.Leist for useful discussions and comments on the “neighbours” software tools described in this article.

References

- [1] Gould, R.: Graph Theory. The Benjamin/Cummings Publishing Company (1988)
- [2] Harary, F., Palmer, E.M.: Graphical Enumeration. New York, Academic Press (1973)
- [3] Erdős, P., Rényi, A.: On random graphs. *Publicationes Mathematicae* **6** (1959) 290–297
- [4] Dijkstra, E.W.: A note on two problems in connexion with graphs. *Numerische Mathematik* **1** (1959) 269–271
- [5] Floyd, R.W.: Algorithm 97: Shortest Path. *Communications of the ACM* **5** (1962) 345
- [6] Sedgewick, R.: Algorithms in Java. Addison-Wesley (2002) ISBN: 978-0201361209.
- [7] Hartsfield, N., Ringel, G.: Pearls in Graph Theory A Comprehensive Introduction. Academic Press (1990)
- [8] Newman, M.E.J., Strogatz, S.H., Watts, D.J.: Random graphs with arbitrary degree distribution and their applications. *Phys. Rev. E* **64** (2001)
- [9] Newman, M., Barabasi, A.L., Watts, D.J.: The Structure and Dynamics of Networks. Princeton University Press (2006)
- [10] Bollobas, B.: Random Graphs. Academic Press, New York (1985)
- [11] Burda, Z., Jurkiexicz, J., Krzywicki, A.: Statistical mechanics of random graphs. *Physica A* **344** (2004) 56–61
- [12] Jackson, S., Luczak, T., Rucinski, A.: Random Graphs. Wiley (2000)
- [13] Callaway, D.S., Newman, M.E.J., Strogatz, S.H., Watts, D.J.: Network robustness and fragility: Percolation on random graphs. *Phys. Rev. Lett.* **85** (2000)
- [14] Barabasi, A.L.: Linked - The New Science of Networks. Number ISBN 0-7382-0667-9. Perseus (2002)
- [15] Barabasi, A.L., Albert, R.: Emergence of scaling in random networks. *Science* **286** (1999) 509–512
- [16] Donato, D., Laura, L., Leonardi, S., Millozzi, S.: Simulating the webgraph: a comparative analysis of models. *IEEE Computing in Science & Engineering* (2004) 84–89
- [17] Abdo, A.H., de Moura, A.P.S.: Clustering as a measure of the local topology of networks. Technical report, Universidade de São Paulo, University of Aberdeen (2008)
- [18] Barbosa, V.C., Ferreira, R.G.: On the phase transitions of graph coloring and independent sets. *Physica A* **343** (2004) 401–423
- [19] Zwick, U.: Exact and approximate distances in graphs - a survey. In: Proc. 9th Annual European Symposium on Algorithms, Springer-Verlag (2001) 33–48
- [20] Watts, D.J.: Small worlds: the dynamics of networks between order and randomness. Princeton University Press (1999)
- [21] Robins, G., Alexander, M.: Small worlds among interlocking directors: network structure and distance in bipartite graphs. *Computational & Mathematical Organization Theory* **10** (2004) 69–94
- [22] Harish, P., Narayanan, P.: Accelerating large graph algorithms on the GPU using CUDA. In Aluru, S., Parashar, M., Badrinath, R., Prasanna, V., eds.: High Performance Computing - HiPC 2007: 14th International Conference, Proceedings. Volume 4873., Goa, India, Springer-Verlag (2007) 197–208
- [23] Leist, A., Playne, D., Hawick, K.: Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. *Concurrency and Computation: Practice and Experience* **21** (2009) 2400–2437 CSTN-065.
- [24] Himsolt, M.: Gml: A portable graph file format. (1997)
- [25] Brandes, U., Eiglsperger, M., Lerner, J.: Graphml primer. Technical report, Uni. Konstanz, Germany (2007)
- [26] Hawick, K.: Interactive graph algorithm visualization and the graviz prototype. Technical Report CSTN-061, Computer Science, Massey University (2008)
- [27] Hawick, K.A., Leist, A., Playne, D.P.: Parallel Graph Component Labelling with GPUs and CUDA. Technical Report CSTN-089, Massey University (2009) Accepted (July 2010) and to appear in the Journal Parallel Computing.
- [28] Hawick, K., James, H.: Small-world effects in wireless agent sensor networks. *Int. J. Wireless and Mobile Computing* **4** (2010) 155–164 ISSN (Online): 1741-1092 - ISSN (Print): 1741-1084.
- [29] Hawick, K., James, H.: Managing community membership information in a small-world grid. Technical report, Computer Science, Massey University (2004) CSTN-002.
- [30] Leist, A., Hawick, K.A.: Circuits as a classifier for small-world network models. In: Proc. WORLD-COMP 2009 International Conference on Foundations of Computer Science (FSC 09) Las Vegas, USA. Number CSTN-003 (2009)

- [31] Hawick, K., James, H.: A fast code for enumerating circuits and loops in graphs. Technical Report CSTN-013, Massey University (2005)
- [32] Hawick, K.A., James, H.A.: Performance, scalability and object-orientation in discrete graph-based simulation models. In: *Int. Conf. on Modeling, Simulation and Visualization Methods (MSV'05)*, Las Vegas, USA (2005)
- [33] Hawick, K.A., James, H.A.: Node importance ranking and scaling properties of some complex road networks. Technical report, Information and Mathematical Sciences, Massey University, Albany, North Shore 102-904, Auckland, New Zealand (2005)
- [34] Hawick, K.A., James, H.A., Scogings, C.J.: Simulating large random boolean networks. Technical Report CSTN-039, Information and Mathematical Sciences, Massey University, Albany, North Shore 102-904, Auckland, New Zealand (2007)
- [35] Hawick, K., James, H., Scogings, C.: Structural Circuits and Attractors in Kauffman Networks. In Abbas, H.A., Randall, M., eds.: *Proc. Third Australian Conference on Artificial Life*. Volume 4828 of LNCS., Springer (2007) 189–200 978-3-540-76930-9.
- [36] Hawick, K.A., James, H.A., Scogings, C.J.: Circuits, Attractors and Reachability in Mixed-K Kauffman Networks. Technical Report CSTN-046; arXiv:0711.2426, Massey University (2007)
- [37] Hawick, K.: Eigenvalue spectra measurements of complex networks. In H.Arabnia, ed.: *Proc. Int. Conf on Scientific Computing (CSC'08)*, Las Vegas (2008) CSTN-051.
- [38] Hawick, K.: Spectral analysis of attractors in gene-regulatory network models. In: *Proc. WORLD-COMP 2009 International Conference on Foundations of Computer Science (FCS 09)* July, Las Vegas, USA. Number CSTN-058 (2009)
- [39] Saunders, S., Takaoka, T.: Improved shortest path algorithms for nearly acyclic graphs. *Electronic Notes in Theoretical Computer Science* **42** (2001)
- [40] Tarjan, R.: Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing* **2** (1973) 211–216
- [41] Tiernan, J.C.: An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM* **13** (1970) 722–726
- [42] Johnson, D.B.: Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing* **4** (1975) 77–84
- [43] Pettie, S., Ramachandran, V.: A shortest path algorithm for real-weighted undirected graphs. In: to appear *SIAM J. Computing*. (2002)
- [44] Schank, T., Wagner, D.: Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications* **9** (2005) 265–275
- [45] K.A.Hawick, A.Leist, D.P.Playne: Mixing multi-core cpus and gpus for irregular graph and network calculations. Technical report, Computer Science, Massey University (2010)
- [46] Li, F., Li, X.: On the integrity of graphs. In: *Proc IASTED Conf. on Parallel and Distributed Computing and Systems*. Number 439-148 (2004)
- [47] Li, L., Alderson, D., Tanaka, R., Doyle, J.C., Willinger, W.: Towards a theory of scale-free graphs: Definition, properties, and implications. In: *Proc. Symp. on Complex Systems Engineering*, The Rand Corporation, Santa Monica, USA. (2007)
- [48] Claussen, J.C.: Offdiagonal complexity: A computationally quick complexity measure for graphs and networks. *Physica A* **375** (2007) 365–373
- [49] Farkas, I.J., Derenyi, I., Barabasi, A.L., Vicsek, T.: Spectra of “real-world” graphs: Beyond the semi-circle law. *Phys. Rev. E* **64** (2001) 026704
- [50] Hawick, K.: Detecting and labelling wireless community network structures from eigen-spectra. In: *Proc. International Conference on Wireless Networks (ICWN'10)*. Number CSTN-083, Las Vegas, USA (2010) ICW5189.
- [51] Hawick, K.: Simulating and visualising sedimentary cluster-cluster aggregation. In: *Proc. International Conference on Modeling, Simulation and Visualization Methods (MSV'10)*. Number CSTN-012, Las Vegas, USA (2010) MSV3277.
- [52] NVIDIA® Corporation: CUDA™ 2.0 Programming Guide. (2008) Last accessed November 2008.
- [53] Khronos Group: OpenCL - Open Compute Language (2008)

```

int [][] loadNeighboursFromFile( File f ){
    int [][] neighbours;
    int N;
    char [] line = f.readLine();
    N = toInt( line );
    neighbours.length = N;
    int num;
    for( int k=0;k<N;k++){
        char [][] words = f.readLine().split();
        neighbours[k].length = toInt( words[0] );
        for(int n=0;n<neighbours[k].length;n++){
            neighbours[k][n] = toInt( words[ n + 1 ] );
        }
    }
    return neighbours;
}

```

Figure 4: D Code for loading a neighbours file and dynamically allocating the neighbours data structure.

```

// reverse the sense of the arcs, returning "neighbours-from"
int [][] reverseNeighbours( int [][] neighbours ){
    int N = neighbours.length;
    int [][] inp; inp.length = N;

    for(int i=0;i<N;i++){
        for(int j=0;j<neighbours[i].length;j++){
            int k = neighbours[i][j];
            inp[k].length = inp[k].length + 1;
            inp[k][length-1] = i;
        }
    }
    return inp;
}

```

Figure 5: D Code for generating the reverse-neighbours structure using D's dynamic memory reallocation syntax

```

int [][] uniqueNeighbours( int [][] neighbours ){
    int N = neighbours.length;
    int [][] inp; inp.length = N;

    for(int i=0;i<N;i++){

        int [ int ] freq;
        for(int j=0;j<neighbours[i].length;j++){
            int n = neighbours[i][j];
            ++freq[n];
        }
        inp[i] = freq.keys.sort;
    }
    return inp;
}

```

Figure 6: D Code for eliminating duplicated arcs from the neighbours structure.

```

/* compute the Floyd pair distances in an adjacency structure */
int [][] computeFloydDistances(int [][] neighbours){
    int n = neighbours.length;
    int [][] adjacency; adjacency.length = n;

    for(int i=0;i<n;i++){// initialise adjacency matrix
        adjacency[i].length = n;
        for(int j=0;j<n;j++){
            adjacency[i][j] = 0;
        }
    }
    for(int i=0;i<n;i++){ // we end up with self arcs of length 1 if they are present
        for(int j=0;j<neighbours[i].length;j++){
            adjacency[i][ neighbours[i][j] ] = 1;
        }
    }

    // Sedgewick's Floyd Algorithm, PP 477
    for(int y=0;y<n;y++){
        for(int x=0;x<n;x++){
            if( adjacency[x][y] != 0 ){
                for(int j=0;j<n;j++){
                    if( adjacency[y][j] > 0 ){
                        if( ( adjacency[x][j] == 0 ) ||
                            ( adjacency[x][y] + adjacency[y][j] < adjacency[x][j] ) ){
                            adjacency[x][j] = adjacency[x][y] + adjacency[y][j];
                        }
                    }
                }
            }
        }
    }
    for(int i=0;i<n;i++){ // fix-up self-distances
        adjacency[i][i] = 0;
    }
    return adjacency;
}

```

Figure 7: D Code for computing all-pairs distances in an adjacency distance matrix using Floyd's algorithm.

```

// generate colour labels for each component:
int [] labelComponents( int [][] neighbours ){
    int nVertices = neighbours.length;
    int [] labels; labels.length = nVertices;

    for(int i=0;i<nVertices;i++){
        labels[i] = i;
    }
    bool changed;
    do{
        changed = false;
        for( int i=0;i<nVertices;i++){
            for(int n=0;n<neighbours[i].length;n++){
                if( labels[i] != labels[ neighbours[i][n] ] ){
                    labels[i] = labels[ neighbours[i][n] ] =
                        min( labels[i], labels[ neighbours[i][n] ] );
                    changed = true;
                }
            }
        }
    } while( changed );
    return labels;
}

```

Figure 8: D Code for Component Labelling the connected-component clusters of a graph.

```

// count the components:
uint countComponents( int [][] neighbours ){
    if( neighbours == null || neighbours.length == 0 ) return 0;
    int [] labels = labelComponents( neighbours );

    int[ int ] count; // associative array giving a vertex tally for each label present
    for(int l=0;l<labels.length;l++){
        count[ labels[l] ]++;
    }
    return count.keys.length; // number of labels used = number of components
}

```

Figure 9: D Code for counting the vertices in each identified cluster component.

```

// split the structure into separate components:
int [][][] splitComponents( int [][] neighbours ){
    int nVertices = neighbours.length;

    int [][][] components;

    int [] labels = labelComponents( neighbours );

    // either do it this way using associative arrays:
    int [ int ] sizes; // sizes of each component
    for( int i=0;i<nVertices;i++)
        sizes[ labels[i] ]++;

    int nComponents = sizes.keys.length; components.length = nComponents;
    int [] clabels; clabels.length = nComponents;
    int c=0;
    foreach( label; sizes.keys ){
        clabels[c] = label;
        components[c++].length = sizes[ label ];
    }

    // we end up with:
    // clabels[0..nComponents-1] gives the vertex-label for each component
    // components[0..nComponents-1] is int array of size nVertices for that component

    int [] translation; translation.length = nVertices;

    for(c=0;c<nComponents;c++){
        int v = 0;
        for(int i=0;i<nVertices;i++){
            if( labels[i] == clabels[c] && v < sizes[ labels[i] ] ){
                components[c][v] = neighbours[i].dup;
                translation[i] = v++; // build translation for relevant vertices in component
            }
        }

        for(v=0;v<components[c].length;v++){ // translate dst list into correct indices:
            for(int j=0;j<components[c][v].length;j++)
                components[c][v][j] = translation[ components[c][v][j] ];
        }
    }

    return components;
}

```

Figure 10: D Code for splitting a neighbours structure for a whole graph into a list of neighbours structures – one for each connected component.

```

// return a structure [j][m] for the j'th cluster ,
// lets us iterate easily over the properties of each cluster
int [][] clusteredVertexIndices( int [][] neighbours ){
    int nVertices = neighbours.length;
    int [][] clusters;

    int [] labels = labelComponents( neighbours );

    // set up sizes[i] to hold size of each separate component (including zero entries):
    int [] sizes;    sizes.length = nVertices;
    sizes [] = 0;
    for( int i=0;i<nVertices;i++){
        sizes[ labels[i] ]++;
    }

    int nClusters = 0;
    for(int i=0;i<nVertices;i++)
        if( sizes[i] != 0 )nClusters++;

    clusters.length = nClusters;
    int j=0;
    for(int i=0;i<nVertices;i++){
        if( sizes[i] != 0 ){
            clusters[j].length = sizes[i];
            int m = 0;
            for(int v=0;v<nVertices;v++){
                if( labels[v] == i )
                    clusters[j][m++] = v;
            }
            j++;
        }
    }
    return clusters;
}

```

Figure 11: D Code for constructing vertex lists for each component.

```

// merge set of individual separate components into single unified number scheme:
int [][] mergeComponents( int [][][] components ){
    int nVertices = 0;
    for(int c=0;c<components.length;c++){
        nVertices += components[c].length;

    int [][] neighbours; neighbours.length=nVertices;

    int v=0;
    int vc=0;
    for(int c=0;c<components.length;c++){
        for(int i=0;i<components[c].length;i++){
            neighbours[v] = components[c][i].dup;
            for(int j=0;j<components[c][i].length;j++){
                neighbours[v][j] += vc;
            }
            v++;
            vc += components[c].length;
        }
    }
    return neighbours;
}

```

Figure 12: D Code for merging together disparate neighbour lists from independently numbered component subgraphs.

```

// return index if test is in the list of ints:
int iPosition( int test, int list[] ){
    for(int i=0;i<list.length;i++){
        if( test == list[i] ){
            return i;
        }
    }
    return -1;
}

// remove the nodes listed from the structure:
int [][] disconnectNeighbours( int [][] neighbours, int list[] ){
    if( list == null || list.length == 0 ) return neighbours;
    // assume no duplicate entries in the node removal list

    int [][] retval; retval.length = neighbours.length - list.length;
    int translate[]; translate.length = neighbours.length;
    int backtranslate[]; backtranslate.length = retval.length;

    int count = 0;
    // check the number of nodes in the new structure:
    for( int i=0;i<neighbours.length;i++){
        if( iPosition( i, list ) >= 0 ) continue;
        backtranslate[count] = i;
        translate[i] = count++;
    }
    assert( count == retval.length );

    // go through new structure fixing up its translated neighbours:
    for( int i=0;i<retval.length;i++){
        int row = backtranslate[i];
        int nlist[] = neighbours[ row ].dup; // make a working copy

        // count number of neighbours that are still valid for node i:
        int ncount = 0;
        for(int l=0;l<nlist.length;l++){
            if( iPosition( nlist[l], list ) >= 0 ){
                nlist[l] = -1;
                continue;
            }
            ncount++;
        }
        retval[i].length = ncount;

        // copy them iff they are valid:
        int k = 0;
        for(int l=0;l<nlist.length;l++){
            if( nlist[l] >= 0 ) retval[i][k++] = nlist[l];
        }

        // translate from their old index values in the old structure:
        for( int j=0;j<retval[i].length;j++){
            retval[i][j] = translate[ retval[i][j] ];
        }
    }

    return retval;
}

```

Figure 13: D Code for removing a list of vertices from a neighbours structure, while maintaining consecutive node numbering without gaps.

```

// computed by neighbours\_analysis
writefln("nVertices____\t%d", nVertices );

writefln("nArcs_____\t%d", nArcs );
writefln("nDiArcs_____\t%d", nDiArcs );
writefln("nSelf_____\t%d", nSelf );
writefln("nMultiple____\t%d", nMultiple );

writefln("minOutputs____\t%d", minOutputs );
writefln("maxOutputs____\t%d", maxOutputs );
writefln("meanOutputs__\t%f", meanOutputs );

writefln("minInputs____\t%d", minInputs );
writefln("maxInputs____\t%d", maxInputs );
writefln("meanInputs____\t%f", meanInputs );

writefln("nClusters____\t%d", nClusters );
writefln("nMonomers____\t%d", nMonomers );
writefln("nDimers_____\t%d", nDimers );
writefln("nTrimers_____\t%d", nTrimers );
writefln("nIncMax_____\t%d", nIncMax );
writefln("fracGiant____\t%f", fracGiant );
writefln("clusterCoeff_\t%f", meanClusteringCoefficient );

```

Figure 14: D Code for reporting graph properties using the type-checked writef routine.