Chapter 1
Graphs and Matrices

Jeremy Kepner*

Abstract

A linear algebraic approach to graph algorithms that exploits the sparse adjacency matrix representation of graphs can provide a variety of benefits. These benefits include syntactic simplicity, easier implementation, and higher performance. Selected examples are presented illustrating these benefits. These examples are drawn from the remainder of the book in the areas of algorithms, data analysis, and computation.

1.1 Motivation

The duality between the canonical representation of graphs as abstract collections of vertices and edges and a sparse adjacency matrix representation has been a part of graph theory since its inception [Konig 1931, Konig 1936]. Matrix algebra has been recognized as a useful tool in graph theory for nearly as long (see [Harary 1969] and the references therein, in particular [Sabadusi 1960, Weischel 1962, McAndrew 1963, Teh & Yap 1964, McAndrew 1965, Harary & Trauth 1964, Brualdi 1967]). However, matrices have not traditionally been used for practical computing with graphs, in part because a dense 2D array is not an efficient representation of a sparse graph. With the growth of efficient data structures and algorithms for *sparse* arrays and

*MIT Lincoln Laboratory, 244 Wood Street, Lexington, MA 02420 (kepner@ll.mit.edu).

This work is sponsored by the Department of the Air Force under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

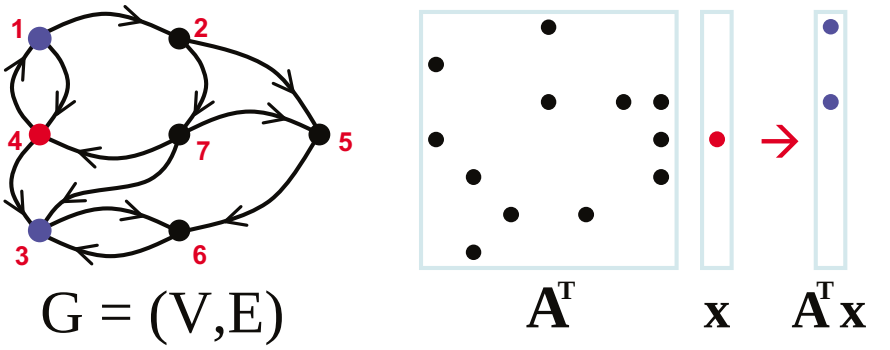

Figure 1.1. Matrix graph duality.
Adjacency matrix **A** is dual with the corresponding graph. In addition, vector matrix multiply is dual with breadth-first search.

matrices, it has become possible to develop a practical array-based approach to computation on large sparse graphs.

There are several benefits to a linear algebraic approach to graph algorithms. These include:

1. *Syntactic complexity.* Many graph algorithms are more compact and are easier to understand in an array-based representation. In addition, these algorithms are accessible to a new community not historically trained in canonical graph algorithms.

2. *Ease of implementation.* Array-based graph algorithms can exploit the existing software infrastructure for parallel computations on sparse matrices.

3. *Performance.* Array-based graph algorithms more clearly highlight the data-access patterns and can be readily optimized.

The rest of this chapter will give a brief survey of some of the more interesting results to be found in the rest of this book, with the hope of motivating the reader to further explore this interesting topic. These results are divided into three parts: (I) Algorithms, (II) Data, and (III) Computation.

1.2 Algorithms

Linear algebraic approaches to fundamental graph algorithms have a variety of interesting properties. These include the basic graph/adjacency matrix duality, correspondence with semiring operations, and extensions to tensors for representing multiple-edge graphs.

1.2.1 Graph adjacency matrix duality

The fundamental concept in an array-based graph algorithm is the duality between a graph and its adjacency representation (see Figure 1.1). To review, for a graph

$G = (V, E)$ with N vertices and M edges, the $N \times N$ adjacency matrix $\mathbf{A}$ has the property $\mathbf{A}(i, j) = 1$ if there is an edge e_{ij} from vertex v_i to vertex v_j and is zero otherwise.

Perhaps even more important is the duality that exists with the fundamental operation of linear algebra (vector matrix multiply) and a breadth-first search (BFS) step performed on G from a starting vertex s

$$BFS(G, s) \quad \Leftrightarrow \quad \mathbf{A}^T\mathbf{v}, \quad \mathbf{v}(s) = 1$$

This duality allows graph algorithms to be simply recast as a sequence of linear algebraic operations. Many additional relations exist between fundamental linear algebraic operations and fundamental graph operations (see chapters in Part I).

1.2.2 Graph algorithms as semirings

One way to employ linear algebra techniques for graph algorithms is to use a broader definition of matrix and vector multiplication. One such broader definition is that of a semiring (see Chapter 2). In this context, the basic multiply operation becomes (in MATLAB notation)

$$\mathbf{A}\ op_1.op_2\ \mathbf{v}$$

where for a traditional matrix multiply $op_1 = +$ and $op_2 = *$ (i.e., $\mathbf{Av} = \mathbf{A}\ +.*\ \mathbf{v}$). Using such notation, canonical graph algorithms such as the Bellman–Ford shortest path algorithm can be rewritten using the following semiring vector matrix product (see Chapters 3 and 5)

$$\mathbf{d} = \mathbf{d}\ +.\min\ \mathbf{A}$$

where the $N \times 1$ vector $\mathbf{d}$ holds the length of the shortest path from a given starting vertex s to all the other vertices.

More complex algorithms, such as betweenness centrality (see Chapter 6), can also be effectively represented using this notation. In short, betweenness centrality tries to measure the "importance" of a vertex in a graph by determining how many shortest paths the vertex is on and normalizing by the number of paths through the vertex. In this instance, we see that the algorithm effectively reduces to a variety of matrix matrix and matrix vector multiplies.

Another example is subgraph detection (see Chapter 8), which reduces to a series of "selection" operations

Row selection: $\mathbf{A}\ \mathrm{diag}(\mathbf{v})$

Col selection: $\mathrm{diag}(\mathbf{u})\ \mathbf{A}$

Row/Col selection: $\mathrm{diag}(\mathbf{u})\ \mathbf{A}\ \mathrm{diag}(\mathbf{v})$

where $\mathrm{diag}(\mathbf{v})$ is a diagonal matrix with the values of the vector $\mathbf{v}$ along the diagonal.

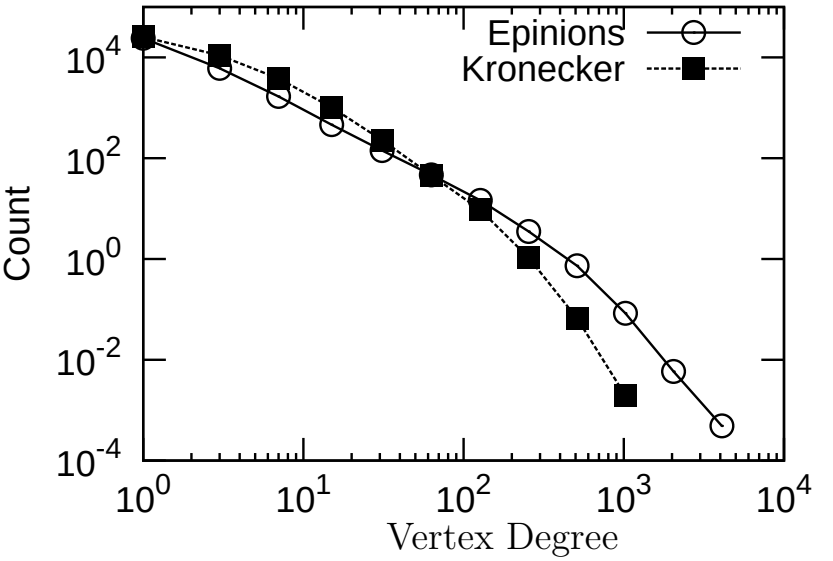

Figure 1.2. Power law graph.
Real and simulated in-degree distribution for the Epinions data set.

1.2.3 Tensors

In many domains (e.g., network traffic analysis), it is common to have multiple edges between vertices. Matrix notation can be extended to these graphs using tensors (see Chapter 7). For example, consider a graph with at most N_k edges between any two vertices. This graph can be represented using the $N \times N \times N_k$ tensor $\mathcal{X}$ where $\mathcal{X}(i, j, k)$ is the kth edge going from vertex i to vertex j.

1.3 Data

A matrix-based approach to the analysis of real-world graphs is useful for the simulation and theoretical analysis of these data sets.

1.3.1 Simulating power law graphs

Power law graphs are ubiquitous and arise in the Internet, the web, citation graphs, and online social networks. Power law graphs have the general property that the histograms of their degree distribution $Deg()$ fall off with a power law and are approximately linear in a log-log plot (see Figure 1.2). Mathematically, this observation can be stated as

$$Slope[\log(Count[Deg(g)])] \approx -constant$$

Efficiently generating simulated data sets that satisfy this property is difficult. Interestingly, an array-based approach using Kronecker products naturally produces graphs of this type (see Chapters 9 and 10). The Kronecker product graph generation algorithm can be described as follows. First, let $\mathbf{A} : \mathbb{R}^{M_B M_C \times N_B N_C}$, $\mathbf{B} : \mathbb{R}^{M_B \times N_B}$, and $\mathbf{C} : \mathbb{R}^{M_C \times N_C}$. Then the Kronecker product is defined as follows:

$$\mathbf{A} = \mathbf{B} \otimes \mathbf{C} = \begin{pmatrix} b_{1,1}\mathbf{C} & b_{1,2}\mathbf{C} & \cdots & b_{1,M_B}\mathbf{C} \\ b_{2,1}\mathbf{C} & b_{2,2}\mathbf{C} & \cdots & b_{2,M_B}\mathbf{C} \\ \vdots & \vdots & & \vdots \\ b_{N_B,1}\mathbf{C} & b_{N_B,2}\mathbf{C} & \cdots & b_{N_B,M_B}\mathbf{C} \end{pmatrix}$$

Now let $\mathbf{G} : \mathbb{R}^{N \times N}$ be an adjacency matrix. The Kronecker exponent to the power k is as follows

$$\mathbf{G}^{\otimes k} = \mathbf{G}^{\otimes k-1} \otimes \mathbf{G}$$

which generates an $N^k \times N^k$ adjacency matrix.

1.3.2 Kronecker theory

It would be very useful if it were possible to analytically compute various centrality metrics for power law graphs. This is possible (see Chapter 10), for example, for Kronecker graphs of the form

$$(\mathbf{B}(n, m) + \mathbf{I})^{\otimes k}$$

where $\mathbf{I}$ is the identity matrix and $\mathbf{B}(n, m)$ is the adjacency matrix of a complete bipartite graph with sets of n and m vertices. For example, the degree distribution (i.e., the histogram of the degree centrality) of the above Kronecker graph is

$$Count[Deg = (n+1)^r (m+1)^{k-r}] = \binom{k}{r} n^{k-r} m^r$$

for $r = 0, \ldots, k$.

1.4 Computation

The previous sections have given some interesting examples of the uses of array-based graph algorithms. In many cases, these algorithms reduce to various sparse matrix multiply operations. Thus, the effectiveness of these algorithms depends upon the ability to efficiently run such operations on parallel computers.

1.4.1 Graph analysis metrics

Centrality analysis is an important tool for understanding real-world graphs. Centrality analysis deals with the identification of critical vertices and edges (see Chapter 12). Example centrality metrics include

Degree centrality is the in-degree or out-degree of the vertex. In an array formulation, this is simply the sum of a row or a column of the adjacency matrix.

Closeness centrality measures how close a vertex is to all the vertices. For example, one commonly used measure is the reciprocal of the sum of all the shortest path lengths.

Stress centrality computes how many shortest paths the vertex is on.

Betweenness centrality computes how many shortest paths the vertex is on and normalizes this value by the number of shortest paths to a given vertex.

Many of these metrics are computationally intensive and require parallel implementations to compute them on even modest-sized graphs (see Chapter 12).

1.4.2 Sparse matrix storage

An array-based approach to graph algorithms depends upon efficient handling of sparse adjacency matrices (see Chapter 13). The primary goal of a sparse matrix is efficient storage that is a small multiple of the number of nonzero elements in the matrix M. A standard storage format used in many sparse matrix software packages is the Compressed Storage by Columns (CSC) format (see Figure 1.3). The CSC format is essentially a dense collection of sparse column vectors. Likewise, the Compressed Storage by Rows (CSR) format is essentially a dense collection of sparse row vectors. Finally, a less commonly used format is the "tuples" format, which is simply a collection of row, column, and value 3-tuples of the nonzero

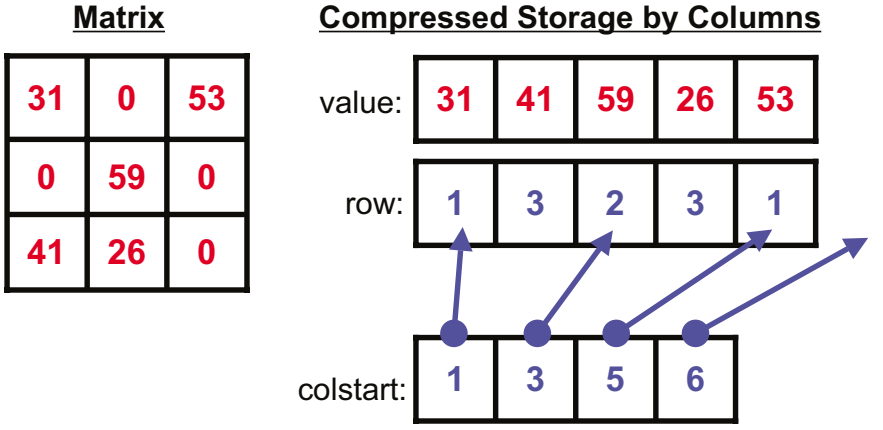

Figure 1.3. Sparse matrix storage.
The CSC format consists of three arrays: `colstart`, `row`, and `value`. `colstart` is an N-element vector that holds a pointer into `row` which holds the row index of each nonzero value in the columns.

elements. Mathematically, the following notation can be used to differentiate these different formats

$\mathbf{A} : \mathbb{R}^{S(N)\times N}$ sparse rows (CSR)

$\mathbf{A} : \mathbb{R}^{N\times S(N)}$ sparse columns (CSC)

$\mathbf{A} : \mathbb{R}^{S(N\times N)}$ sparse rows and columns (tuples)

1.4.3 Sparse matrix multiply

In addition to efficient sparse matrix storage, array-based algorithms depend upon an efficient sparse matrix multiply operation (see Chapter 14). Independent of the underlying storage representation, the amount of useful computation done when two random $N \times N$ matrices with M nonzeros are multiplied together is approximately $2M^2/N$. By using this model, it is possible to quickly estimate the computational complexity of many linear algebraic graph algorithms. A more detailed model of the useful work in multiplying two specific sparse matrices $\mathbf{A}$ and $\mathbf{B}$ is

$$flops(\mathbf{A} \cdot \mathbf{B}) = 2 \sum_{k=1}^{N_v} nnz(\mathbf{A}(:,k)) \cdot nnz(\mathbf{B}(k,:))$$

where $M = nnz()$ is the number of nonzero elements in the matrix. Sparse matrix matrix multiply is a natural primitive operation for graph algorithms but has not been widely studied by the numerical sparse matrix community.

1.4.4 Parallel programming

Partitioned Global Address Space (PGAS) languages and libraries are the natural environment for implementing array-based algorithms. PGAS approaches have been implemented in C, Fortran, C++, and MATLAB (see Chapter 4 and [Kepner 2009]). The essence of PGAS is the ability to specify how an array is decomposed on a parallel processor. This decomposition is usually specified in a structure called a "map" (or layout, distributor, distribution, etc.). Some typical maps are shown in Figure 1.4.

The usefulness of PGAS can be illustrated in the following MATLAB example, which creates two distributed arrays `A` and `B` and then performs a data redistribution via the assignment "=" operation

```
Amap = map([Np 1],{},0:Np-1);   % Row map.
Bmap = map([1 Np],{},0:Np-1);   % Column map.
A = rand(N,N,Amap);             % Distributed array.
B = zeros(N,N,Bmap);            % Distributed array.
B(:,:) = A;                     % Redistribute A to B.
```

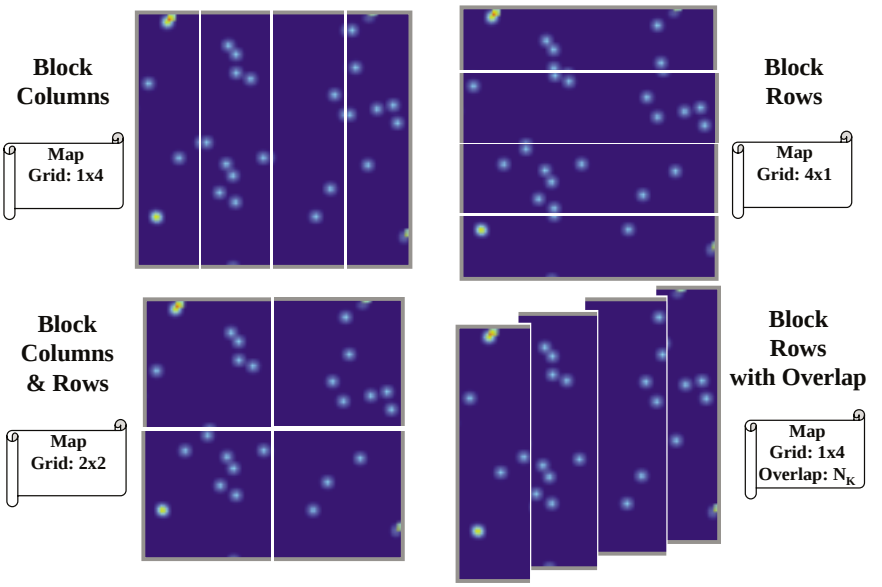

Figure 1.4. Parallel maps.
A selection of maps that are typically supported in PGAS programming environments.

Mathematically, we can write the same algorithm as follows

$$\mathbf{A} : \mathbb{R}^{P(N)\times N}$$

$$\mathbf{B} : \mathbb{R}^{N\times P(N)}$$

$$\mathbf{B} = \mathbf{A}$$

where $P()$ is used to denote the dimension of the array that is being distributed across multiple processors.

1.4.5 Parallel matrix multiply performance

The PGAS notation allows array algorithms to be quickly transformed into graph algorithms. The performance of such algorithms can then be derived from the performance of parallel sparse matrix multiply (see Chapter 14), which can be written as

$$\mathbf{A}, \mathbf{B}, \mathbf{C} : \mathbb{R}^{P(N\times N)}$$

$$\mathbf{A} \leftarrow \mathbf{BC}$$

The computation and communication times of such an algorithm for *random* sparse matrices are

$$T_{comp}(N_P) \propto (M/N)M/N_P$$

$$T_{comm}(N_P) \propto MN_P^{1/2}$$

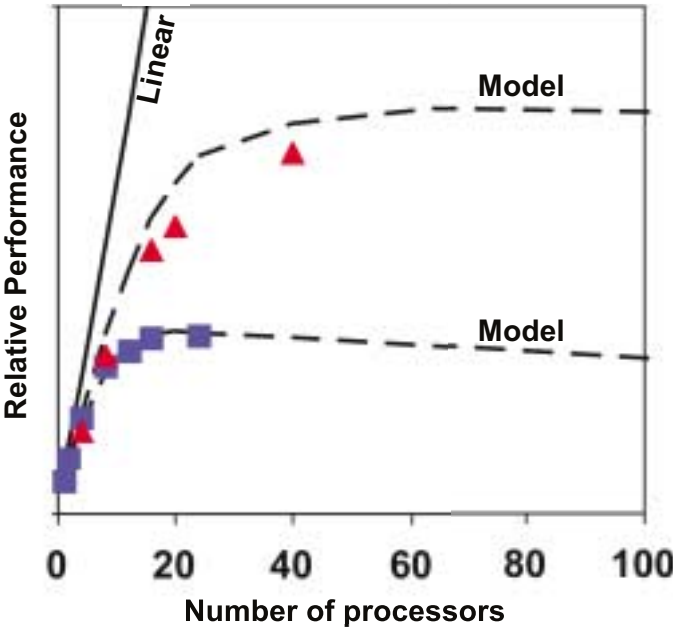

Figure 1.5. Sparse parallel performance.
Triangles and squares show the measured performance of a parallel betweenness centrality code on two different computers. Dashed lines show the performance predicted from the parallel sparse matrix multiply model showing the implementation achieved near the theoretical maximum performance the computer hardware can deliver.

The resulting performance speedup (see Figure 1.5) on a typical parallel computing architecture then shows the characteristic scaling behavior empirically observed (see Chapter 11).

Finally, it is worth mentioning that the above performance is for *random* sparse matrices. However, the adjacency matrices of power law graphs are far from random, and the parallel performance is dominated by the large load imbalance that occurs because certain processors hold many more nonzero values than others. This has been a historically difficult problem to address in parallel graph algorithms. Fortunately, array-based algorithms combined with PGAS provide a mechanism to address this issue by remapping the matrix. One such remapping is the two-dimensional cyclic distribution that is commonly used to address load balancing in parallel linear algebra. Using $P_c()$ to denote this distribution, we have the following algorithm

$$\mathbf{A}, \mathbf{B}, \mathbf{C} : \mathbb{R}^{P_c(N \times N)}$$

$$\mathbf{A} = \mathbf{BC}$$

Thus, with a very minor algorithmic change: $P() \rightarrow P_c()$, the distribution of nonzero values can be made more uniform across processors.

More optimal distributions for sparse matrices can be discovered using automated parallel mapping techniques (see Chapter 15) that exploit the specific distribution of non-zeros in a sparse matrix.

1.5 Summary

This chapter has given a brief survey of some of the more interesting results to be found in the rest of this book, with the hope of motivating the reader to further explore this fertile area of graph algorithms. The book concludes with a final chapter discussing some of the outstanding issues in this field as it relates to the analysis of large graph problems.

References

[Brualdi 1967] R.A. Brualdi. Kronecker products of fully indecomposable matrices and of ultrastrong digraphs. *Journal of Combinatorial Theory,* 2:135–139, 1967.

[Harary & Trauth 1964] F. Harary and C.A. Trauth Jr. Connectedness of products of two directed graphs. *SIAM Journal on Applied Mathematics,* 14:250–254, 1966.

[Harary 1969] F. Harary. *Graph Theory.* Reading: Addison–Wesley. 1969.

[Kepner 2009] J. Kepner. *Parallel MATLAB for Multicore and Multinode Computers.* Philadelphia: SIAM. 2009.

[Konig 1931] D. Konig. Graphen und Matrizen (Graphs and matrices). *Matematikai Lapok,* 38:116–119, 1931.

[Konig 1936] D. Konig. *Theorie der endlichen und unendlichen graphen* (Theory of Finite and Infinite Graphs). Leipzig: Akademie Verlag M.B.H. 1936. See Richard McCourt (Birkhauser 1990) for an English translation of this classic work.

[McAndrew 1963] M.H. McAndrew. On the product of directed graphs. *Proceedings of the American Mathematical Society,* 14:600–606, 1963.

[McAndrew 1965] M.H. McAndrew. On the polynomial of a directed graph. *Proceedings of the American Mathematical Society,* 16:303–309, 1965.

[Sabadusi 1960] G. Sabadusi. Graph multiplication. *Mathematische Zeitschrift,* 72:446–457, 1960.

[Teh & Yap 1964] H.H. Teh and H.D. Yap. Some construction problems of homogeneous graphs. *Bulletin of the Mathematical Society of Nanying University,* 1964:164–196, 1964.

[Weischel 1962] P.M. Weischel. The Kronecker product of graphs. *Proceedings of the American Mathematical Society,* 13:47–52, 1962.