Lecture 17 — Sparse Matrices

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 25, 2011

Today:

- Vector and Matrix Review
- Sparse Matrices and Graphs
- Pagerank

1 Vectors and Matrices

Vectors and matrices are fundamental data types in linear algebra. They have a long and rich history of applications. The first example, which dates back to between 300 BC and AD 200, appears in the Chinese text *Jiu Zhang Suan Shu*, which discusses the use of matrix methods to solve a system of linear equations. Indeed, many believe matrices and vectors were initially invented to express complex sets of equations and operations in a compact form. In Physics, vectors are often used to represent displacements (or velocities) and matrices to represent transformations on these displacements, such as rotations. More generally, however, vectors and matrices can be used to represent a wide variety of physical and virtual data. As we will see, matrices can be used to represent the web graph, social networks, or large scale physical simulations. In such applications, the number of entries in the vectors and matrices can easily be thousands, millions, billions, or more.

In this class, we are most interested in such large vectors and matrices, and not the small 3×3 matrices that are involved in transformations in 3-d space. In such usage, it is typical that most of the entries in a matrix are zero. These matrices with a large number of zero entries are referred to as *sparse matrices*. Sparse matrices are important because with the right representations, they can be much more efficient to manipulate than the equivalently-sized dense matrices. Sparse matrices are also interesting because they are closely related to graphs.

Let's consider the data type specifications for vectors and matrices. Here, we only consider a minimal interface. A full library would include many more functions, but the ones given are sufficient for our purposes. Vectors and matrices are made up of elements that support addition and multiplication. For now, we can assume elements are real numbers and will use $\cdot$ to indicate multiplication, but we will extend this later to allow elements from an arbitrary ring or field. We use $\mathcal{V}$ to indicate the type of a vector, $\mathcal{A}$ the type of a matrix and $\mathcal{E}$ for the type of the elements. By convention, we use bold face upper and lower case letters to denote matrices and vectors, respectively, and lowercase letters without bold face for element. A vector is a length-n sequence of elements

$$\mathbf{x} = \langle x_1, x_2, \ldots, x_n \rangle, \text{ where } x_i \text{ belongs to an arbitrary ring or field } \mathbb{F}.$$

The vectors support the following functions:[1]

[1] In some contexts it can be important to distinguish between row and column vectors, but we won't make that distinction here.

function	**type**	**semantics**	**requires**
$\texttt{sub}(\mathbf{x}, i)$	$\mathcal{V} \times \texttt{int} \to \mathcal{E}$	$\mathbf{x}_i$	$1 \leq i \leq \|\mathbf{x}\|$
$\texttt{dotProd}(\mathbf{x}, \mathbf{y})$	$\mathcal{V} \times \mathcal{V} \to \mathcal{E}$	$\sum_{i=1}^{\|\mathbf{x}\|} A_i \cdot B_i$	$\|A\| = \|B\|$
$\texttt{scale}(c, \mathbf{x})$	$\mathcal{E} \times \mathcal{V} \to \mathcal{V}$	$\langle\, c \cdot x : x \in \mathbf{x} \,\rangle$	
$\texttt{add}(\mathbf{x}, \mathbf{y})$	$\mathcal{V} \times \mathcal{V} \to \mathcal{V}$	$\langle\, x + y : x \in \mathbf{x}, y \in \mathbf{y} \,\rangle$	$\|\mathbf{x}\| = \|\mathbf{y}\|$

A matrix is an $n \times m$ array of elements

$$\mathbf{A} = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix}.$$

Thus, a matrix can be viewed as a collection of n length-m row vectors or a collection of m length-n column vectors. Our matrices support the following functions:

function	**type**	**semantics**	**where**
$\texttt{sub}(\mathbf{A}, i, j)$	$\mathcal{A} \times \mathcal{I} \times \mathcal{I} \to \mathcal{E}$	a_{ij}	$1 \leq i \leq \|\mathbf{A}\|$
$\texttt{mmmult}(\mathbf{A}, \mathbf{B})$	$\mathcal{A} \times \mathcal{A} \to \mathcal{A}$	$\{(i,j) \mapsto \sum_{k=1}^{l} a_{ik} \cdot b_{kj} : (i,j) \in \texttt{Idx}(n,m)\}$	$\|\mathbf{A}\| = n \times l$, $\|\mathbf{B}\| = l \times m$
$\texttt{mvmult}(\mathbf{A}, \mathbf{x})$	$\mathcal{A} \times \mathcal{V} \to \mathcal{V}$	$\left\{i \mapsto \sum_{j=1}^{m} a_{ij} \cdot x_j : i \in \{1, \ldots, n\}\right\}$	$\|\mathbf{A}\| = n \times m$, $\|\mathbf{x}\| = m$
$\texttt{scale}(c, \mathbf{A})$	$\mathcal{E} \times \mathcal{A} \to \mathcal{A}$	$\{(i,j) \mapsto c \cdot a_{ij} : (i,j) \in \texttt{Dom}(\mathbf{A})\}$	
$\texttt{add}(\mathbf{A}, \mathbf{B})$	$\mathcal{A} \times \mathcal{A} \to \mathcal{A}$	$\{(i,j) \mapsto a_{ij} + b_{ij} : (i,j) \in \texttt{Dom}(\mathbf{A})\}$	$\|\mathbf{A}\| = \|\mathbf{B}\|$
$\texttt{transpose}(\mathbf{A})$	$\mathcal{A} \to \mathcal{A}$	$\{(j,i) \mapsto a_{ij} : (i,j) \in \texttt{Dom}(\mathbf{A})\}$	

where $\texttt{Idx}(n, m)$ means the index set $\{(i,j) : i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}\}$, and $\texttt{Dom}(A)$ for an $n \times m$ matrix means $\texttt{Idx}(n, m)$.

It can sometimes be hard to remember how the indices are combined in matrix multiply. After all, the way it works is completely by convention and therefore arbitrary. One way to remember it is to note that when multiplying $\mathbf{A} \times \mathbf{B}$ ($\texttt{mult}(\mathbf{A}, \mathbf{B})$), one takes a dot-product of every row of $\mathbf{A}$ with every column of $\mathbf{B}$. Think of it as a T on its side, as in $\dashv$, where the the horizontal part represents a row on the left, and the vertical a column on the right. If you can't remember which way the T faces, then think of driving from left to right until you hit the T intersection. Also note that for two matrices with dimension $n \times \ell$ and $\ell \times m$, the output dimension is $n \times m$, so that the ℓ's cancel. This allows one to figure out how the dot products are placed in the result matrices.

The only operations used on the elements are addition and multiplication. In certain situations, it is useful to generalize these to other operations. For the properties of matrices to be maintained, what we need is that the elements $\mathbb{E}$ and functions $+$ and $\cdot$ with certain properties. Consider the tuple $(\mathbb{E}, +, \cdot)$ with the following properties:

- Closure of $\mathbb{E}$ under addition and multiplication: if $a, b \in \mathbb{E}$, then $a + b \in \mathbb{E}$ and $a \cdot b \in \mathbb{E}$.

- Associativity of addition and multiplication: if $a, b, c \in \mathbb{E}$, then $(a + b) + c = a + (b + c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
- Identity for addition and multiplication: there exist an additive identity $I_+ \in \mathbb{E}$ and a multiplicative identity $I.$ such that for all $a \in \mathbb{E}$, $I_+ + a = a = a + I_+$, $I. \cdot a = a = a \cdot I.$.
- Commutativity of addition: $a + b = b + a$
- Inverse for addition: $-a$
- Distributivity of multiplication over addition $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

These rules together define an a *ring* $(\mathbb{E}, +, \cdot)$. Indeed, a ring can be thought of as another abstract data type.

In algorithms and applications, there are many uses of rings other than addition and multiplication over numbers. One that often comes up is having $+$ be minimum, and $\cdot$ be addition. As will be discussed in recitation, this can be used to implement the Bellman-Ford algorithm.

2 Sparse Matrices

As mentioned before, most often in practice when dealing with large matrices, the matrices are sparse, i.e., only a small fraction of the matrix elements are nonzero (for general rings, zero is I_+). When implementing matrix routines that will operate on sparse matrices, representing all the zero elements is both a waste of time and space. Instead, we can store just the nonzero elements. Such a representation of matrices is called a *sparse matrix representation*. Advances in sparse matrix routines over the past ten to twenty years is possibly the most important advance with regards to efficiency in large scale simulations in scientific, economic, and business applications.

We often use $nz(\mathbf{A})$ to denote the number of nonzero elements in a matrix. For an $n \times m$ matrix, clearly this cannot be larger than $n \times m$.

How do we represent a sparse matrix or sparse vector? One of the most standard ways is the so-called compressed sparse row (CSR) representation. We will consider two forms of this representation, one using sequences and one sets. Consider the matrix

$$\mathbf{A} = \begin{pmatrix} a & 0 & 0 & b & 0 \\ 0 & c & d & 0 & 0 \\ 0 & 0 & e & 0 & 0 \\ f & 0 & g & 0 & h \\ 0 & i & 0 & 0 & j \end{pmatrix}$$

It can be presented as a sequence of sequences, as in:

$$\langle \langle (0, a), (3, b) \rangle, \langle (1, c), (2, d) \rangle, \langle (2, e) \rangle, \langle (0, f), (2, g), (4, h) \rangle, \langle (1, i), (4, j) \rangle \rangle$$

In this representation, each subsequence represents a row of the matrix. Each of these rows contains only the non-zero elements each of which is stored as the non-zero value along with the column index in

which it belongs. Hence the name compressed sparse row. Now lets consider how we do a vector matrix multiply using this representation.

$$1 \quad \textbf{fun } \texttt{mvmult}(\mathbf{A}, \mathbf{x}) = \quad \left\langle \sum_{(j,v)\in\mathbf{r}} \mathbf{x}[j] * v : \mathbf{r} \in \mathbf{A} \right\rangle$$

For each non-zero element within each row, the algorithm reads the appropriate value from the vector and multiplies it the value of the element. These are then summed up across the rows. For example, let's consider the following matrix-vector product:

$$\underbrace{\begin{pmatrix} 3 & 0 & 0 & -1 \\ 0 & -2 & 2 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 0 & -3 & 1 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} 2 \\ 1 \\ 3 \\ 1 \end{pmatrix}}_{\mathbf{x}}$$

which is represented as

```
A = <<(0,3),(3,-1)>,<(1,-2),(2,2)>,<(2,1)>,<(0,2),(2,-3),(3,1)>

x = < 2, 1, 3, 1 >
```

This is perhaps the most common way to implement sparse matrix vector multiplication in practice. If we assume an array implementation of sequences then this routine will take $O(nz(\mathbf{A}))$ work and $O(\log n)$ span. To calculate the work we note that we only multiply each non-zero element once and each non-zero element is only involved in one reduction. For the span we note that the sum requires a reduction but otherwise everything

It is however not well suited when the vector is sparse as well. Consider the two vectors:

$$\mathbf{x} = \langle\, 0, 1, 0, 0, -2, 0, 0, 0, -1, 0 \,\rangle$$

$$\mathbf{y} = \langle\, -3, 0, 1, 0, 1, 0, 0, 2, 0, 0 \,\rangle$$

Which we would represent as a compressed "row" as

$$\mathbf{x} = \langle (1, 1), (4, -2), (8, -1) \rangle$$

$$\mathbf{y} = \langle (0, -3), (2, 1), (4, 1), (7, 2) \rangle$$

Now lets say we want to implement the vector operations, such as `dotp` and `add`. What do we get? How do we do this.

One easy way to do this is to present sparse vectors as $\mathcal{E}$ tables. We then have, e.g.

$$\mathbf{x} = \{1 \mapsto 1, 4 \mapsto -2, 8 \mapsto -1\}$$

Now we can easily write vector addition and and dot product

```
1  fun add(x, y) = Table.merge (fn (x, y) ⇒ x + y) (x, y)

2  fun dotp(x, y) =
3    reduce + 0 (Table.merge (fn (x, y) ⇒ x · y) (x, y))
```

3 Relation to Graphs

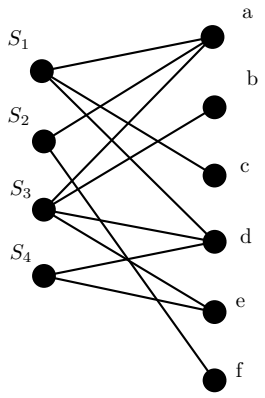
Sparse matrices are closely related to graphs. In particular, an $n \times m$ matrix M can be viewed as a bipartite graph with n vertices V on the left and m on vertices U on the right—i.e. the rows correspond to the vertices U and the columns to the vertices V. We place a weighted edge from vertex v_i on the the left to u_i on the right if $M_{ij} \neq 0$. If $n = m$ (i.e., the matrix is square), then we can view the matrix as a graph from vertices back to themselves—i.e., the rows and columns correspond to the same set of vertices V.

For example, in the last lecture we discussed error correcting codes and how they can be represented as a bipartite graph with the code bits on the left, the check constraints on the right, and edges between the two. Such an error correcting code could equally easily be represented as a matrix. For example here is a matrix for a code corresponding to a codeword of length 6 with 4 parity constraints.

$$\mathbf{B} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Similarly we could represent the set cover problem as a matrix where the rows correspond to sets and the columns to elements. For example, the sets

{a, c, d}
{a, f}
{a, b, d, e}
{b, e, f}
{d, e}



correspond to the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

where the rows correspond to the five sets, and the columns correspond to the elements (a, b, c, d, e, f). We can now describe the set cover problem as a matrix problem. In particular for an $n \times m$ binary matrix A, the problem is to

minimize $x \cdot \mathbf{1}$
such that $xA \geq \mathbf{1}$ and $x_i \in \{0, 1\}$

where $\mathbf{1}$ is a vector of all ones. Here the vector x is a binary vector representing which sets are selected. Minimizing the dot product $x \cdot \mathbf{1}$ therefore minimizes the number of sets that are selected. The constraint $xA \geq \mathbf{1}$ makes sure that every element is covered at least once.

The connection between graphs and matrices has been exploited heavily over the past decade in both directions—graph algorithms have been applied to improve matrix computations especially with regards to sparse matrices, and matrix algebra has been used to analyze graphs. For example, the best algorithms for doing Gaussian elimination on sparse matrices are based on graph properties and algorithms. In the other direction, we will see that analyzing random walks on graphs is best dealt with using matrix operations.

We note, however, there are some differences between matrices and graphs. Firstly, the vertices of a graph are not necessarily indexed from 1 to n and 1 to m, they are unordered sets. This means that certain techniques such as graph contractions, are not very naturally described as matrix routines. Secondly, matrices assume specific operations while graphs are really only defined in terms of the vertices and edges.

4 PageRank and Iterative Linear System Solving

We now consider a simple application of matrix-vector multiply for the so-called PageRank problem. PageRank is used by Google to give a score to all pages on the web based on a method for predicting how important the pages are. The PageRank problem was introduced in a paper by Brin, Page, Motwani, and Winograd. Brin and Page were graduate students in CS at Stanford and went on to form Google. Motwani and Winograd were professors at Stanford. Motwani is an author of one of probably the most widely used textbook on Randomized Algorithms (do a search on "randomized algorithms" on google and see how well it is ranked). When Google was started, Stanford received 1.8 million shares of google stock in exchange for patent PageRank patents. It later sold these for $336 million.

To understand PageRank, consider the following process for visiting pages on the web, which we will refer to as WebSurf.

0. Start at some arbitrary page
1. For a very long time,

a. with probability ε jump to a random page
b. otherwise with equal probability visit one of the outlinks of the current page

The question is after a very long time, what is the probability that the process is at a given page. The idea of PageRank is that this probability is a reasonable measure of the importance of a page. In particular, if a page p has many pointers to it, then it will end up with a high rank. Furthermore, if the pages that point to p also have high ranks, then they will effectively improve the rank of p.

It turns out that the probability of being at a page can be calculated reasonably easily and efficiently using matrix-vector multiplication. In particular, we can use a matrix to represent the transition probabilities described by the WebSurf process. In particular, consider a directed web graph G with n vertices (pages), and denote the out degree of page i as d_i. We define an $n \times n$ adjacency matrix $\mathbf{A}$ such that $\mathbf{A}_{ij} = 1/d_i$ if there is an edge from i to j and 0 otherwise. Also let $\mathbf{J}$ be an $n \times n$ matrix of all ones. Now consider the matrix:

$$\mathbf{T} = \frac{\varepsilon}{n}\mathbf{J} + (1 - \varepsilon)\, vA.$$

The claim is that entry $\mathbf{T}_{ij}$ is exactly the probability of going from vertex i to vertex j in process WebSurf. Now to calculate the probability distribution, we use the following simple algorithm.

```
1  fun pageRank(T) =
2  let
3    fun pageRank'(p) =
4      let val p' = Tp
5      in
6        if ||p − p'|| < some threshold
7        then p'
8        else pageRank'(p')
9      end
10 in
11   pageRank'(1/n [1, ..., 1])
12 end
```

In the code, the vector norm $\|x\|$ stands for the Euclidean norm (or ℓ_2 norm) $\sqrt{x^\top x}$. The vector $\frac{1}{n}[1, \ldots, 1]$ is an initial "guess" at the probabilities. This routine converges on the actual probability distribution we desire and in practice does not take very many rounds (e.g. 10s or 100s). Note that the only real computations are the matrix vector multiply on line 4 and the calculation of $\|p - p'\|$. The cost is dominated by the matrix vector multiply.

Exercise 1. *Note that the matrix T is dense since U is dense. Explain how to computer Tp in $O(|E|)$ work, where E is the number of edges in the web graph.*