Project 4: Sparse Matrix Algebra

Contents

New concepts: Subclasses and polymorphism, nested Collections classes, exploiting structure in linear matrix algebra

1 Sparse Matrices

As background, we assume you are familiar with matrices and basic matrix algebra (matrix transpose, matrix multiply). In this project, we are going to exploit the structure of large but sparse (many zeroes) matrices to efficiently perform matrix computations that simply are not possible with standard dense representation of matrices as 2-dimensional arrays.

1.1 Matrix Storage

A dense array representation is simply the standard form that you know, for example, the following is a 4 row by 5 column dense matrix (corresponding to file `dmatrix1.txt`):

```
1 2 0 3
4 0 5 6
7 0 0 9
0 11 12 0
1 0 0 2
```

However, in many Operations Research applications and in numerical computing in general, it turns out that matrices have many zeroes and are hence what is called sparse. A typical sparse matrix may have 99.9% of its entries set to zero. In this case, we can be much more efficient with both storage and computation if we avoid representing these zeroes and exploit this during computations like matrix multiplication.

Following is an example of a sparse matrix representation file format (corresponding to file `smatrix1.txt`) that represents the same matrix as the previous dense format example:

```
0 0 1
0 1 2
0 3 3
1 0 4
1 2 5
1 3 6
```

```
2 0 7
2 3 9
3 1 11
3 2 12
4 0 1
4 3 2
```

Here we note that in each row, the first entry is the row, the second entry is the column, and the third entry is the value of the matrix at that row and column. Critically omitted are the zeroes – any entry missing here is assumed to be zero.

While it may not seem that the above sparse matrix representation is more compact than the dense representation for this example, large-dimensional sparse matrices will generally have only a small fraction of non-zero entries, thus leading to large space savings over the dense format. Thus, we could easily represent a 10,000 by 10,000 matrix with only 100 non-zero entries as a sparse matrix, but it would consume an excessive amount of disk or memory as a dense matrix.

1.2 Sparse Matrix Storage in Memory

In this project, we will store sparse matrices in memory using the following **nested HashMap** structure:

```
HashMap<Integer,HashMap<Integer,Double>> _hmRow2Col2Val;
```

The use of this structure is quite simple. If we call

```
int row = 2;
// Note that row_vector may be null if the entire row is zeroes
HashMap<Integer,Double> row_vector = _hmRow2Col2Val.get(row);
```

then row_vector is a simple HashMap representation of a row vector for row index 2 in the matrix (remember that row indices start at 0), where the key in HashMap<Integer,Double> row_vector is a column index. Hence if we want the value of the row_vector at column index 4 (remember that column indices also start at 0), then we simply call:

```
int col = 4;
// Note that value may be null if the value is zero
Double value = row_vector.get(col);
```

Of course if either the row_vector or value are missing in their respective HashMaps then we assume the value is 0.

1.3 Sparse Matrix Computation

The compactness of sparse matrix storage is a huge benefit for very large sparse matrices, but even more importantly, we can exploit this compactness for efficient operations like matrix transposition or matrix multiplication.

For efficient sparse matrix transposition, we need only add entries to the matrix transpose that are non-zero in the original matrix. The complexity of this operation is simply proportional to the number of non-zero elements in the matrix.

For efficient sparse matrix multiplication, we should first consider the special case of sparse vectors. Consider the implementation of the dot product of two row vectors, where $'$ indicates vector or matrix transposition and $\cdot$ indicates a dot (inner) product of two vectors:

$$[0010200]' \cdot [0340056]' = 4$$

Here we note that if we simply iterate through the non-zero indices of the first vector, we need only look up these indices in the second vector for purposes of the running summation of the dot product since zero times any other entries is zero. This suggests a **useful method to define** would be one which takes the

dot product of two vectors stored as `HashMaps` where the keys are vector indices and the values are vector values and all zero entries are omitted. (Note that a very similar operation was done for `VectorUtils.sum(...)` in Project 3.)

We can easily extend this sparsity exploitation idea from vector to matrix multiplication if we consider that to compute $C = A \cdot B$, each entry $C_{i,j}$ is the result of a simple dot product: $C_{i,j} = A'_{i,:} \cdot B_{:,j}$, where the : indicates that we populate the vectors from indices in the respective dimension.

The key to efficient matrix multiplication with the sparse **nested `HashMap`** matrix representation outlined above is to obtain `HashMap` representations of the vectors $A'_{i,:}$ and $B_{:,j}$ needed to compute $C_{i,j}$. Obtaining the row vector $A'_{i,:}$ should be obvious from the previous discussion. Obtaining the column vector $B_{:,j}$ requires some more ingenuity; consider how you can use the Matrix transpose operation for this. (Recall that the sparse matrix transpose is efficient – it does need to compute with all non-zero elements of a matrix, but the matrix multiply may also need to examine these same elements anyways so there is little relative efficiency loss when computing an operand transpose as part of a matrix multiply.)

1.4 0-1 Matrices, Sparse Graphs, and All-pairs n-step Reachability

In this project, you are asked to implement both standard matrix multiplication and 0-1 matrix multiplication, which is defined below in terms of the use case of all-pairs n-step reachability.

Matrices need not always contain values in the full range of the real numbers and variations of matrix multiplication can be very useful in some settings. Consider encoding a directed graph having vertices labelled $\{0, 1, 2, 3\}$ and directed edges $\{1 \to 0, 2 \to 1, 3 \to 2\}$ where we assume all vertices have a (reflexive) cyclic directed edge to themselves. Following is an encoding of this graph as a 0-1 edge matrix E:

```
1 0 0 0
1 1 0 0
0 1 1 0
0 0 1 1
```

Here, a 1 in row i and column j indicates there is a directed edge from vertex i to vertex j, and 0 indicates such an edge does not exist. Note that in many large directed graphs (e.g., the street network for Canada), the 0-1 edge matrix representing one-way roads between two locations (vertices) is extremely sparse and hence exploited by the sparse matrix representation.

Now we might ask, if we start at node 3, can we reach node 1 in two steps? Symmetrically, if we start at node 1, can we reach node 3 in two steps? In general, we might ask for all node pairs i and j, which pairs can reach each other in two steps, or in general, n steps.

We could perform some form of graph search for all pairs of vertices i and j but that can become inefficient since a lot of common path computations among pairs are discarded. A much more efficient way to compute all-pairs reachability is through 0-1 matrix multiplication of the edge matrix. First, let us compute E^2, the above matrix 0-1-multiplied by itself:

```
1 0 0 0
1 1 0 0
1 1 1 0
0 1 1 1
```

The simple modification here to standard matrix multiplication is that if any value in the standard matrix multiply is equal to or larger than 1, we set it to 1, otherwise 0. What we get with the above multiplication is a 1 in any row i and column j if there is path from vertex i to vertex j in two steps or less. What about three steps? Let's 0-1-multiply the above matrix E^2 one more time by the original 0-1 edge matrix E to get E^3:

```
1 0 0 0
1 1 0 0
1 1 1 0
1 1 1 1
```

One can verify that this now encodes three step reachability and so on, i.e., for 0-1 edge matrix E, E^n under 0-1 matrix multiplication encodes all-vertex pairs reachability in n or fewer steps. Critically, such an approach reuses n step computability when it computes $n+1$st step computability and this is much more efficient than repeated searches for each pair of nodes (this is called *dynamic programming* in general and it reduces computational complexity of some tasks from an exponential number of operations to a polynomial number of operations by efficiently caching intermediate computations).

One may ask why this variation of matrix multiplication actually computes all-pairs reachability? The answer is simple if you look at the mathematics. Two step reachability R^2 of vertex j from vertex i can be computed from the 0-1 edge matrix E as follows where $\wedge$ is logical AND and we interpret a 1 as *true* and a 0 as *false*:

$$\begin{aligned} R^2_{i,j} &= \exists_k \, E_{i,k} \wedge E_{k,j} \\ &= \begin{cases} 1: & \sum_k E_{i,k} \cdot E_{k,j} \geq 1 \\ 0: & \text{otherwise} \end{cases} \\ &= \begin{cases} 1: & E_{i,:} \cdot E_{:,j} \geq 1 \\ 0: & \text{otherwise} \end{cases} \end{aligned}$$

Here a two-step path between i and j exists if there is a path from i to k and then from k to j. This is precisely what a single 0-1 matrix multiply is computing as shown in the last step of the derivation.

Inductively, it should be apparent that $R^n_{i,j} = \exists k \, E_{i,k} \wedge R^{n-1}_{k,j}$ and this simply corresponds to additional 0-1 matrix multiplications of the last computed R with E. Hence 0-1 matrix multiplication proves an efficient and simple approach for computing all-pairs reachability in n steps or fewer.

On a final note, consider that computing n step reachability actually takes far fewer 0-1 matrix multiplications than the serial inductive specification above. Consider how you could compute 8-step reachability with only three 0-1 matrix multiplications.

2 Mini-Matlab

Matrix computations are such an important part of scientific computation that entire programming scripting languages are built around them, e.g., consider Matlab. In this project, file `Pro4_utorid.java` implements a small subset of Matlab. Following is an example script provided in file `input1.txt` demonstrating an interaction with the Mini-Matlab interface:

```
% This input should work with a dense matrix
% Note that the interpreter is not case-sensitive
HELP
SHOW_RESULTS on
A = read_file dmatrix1.txt
B = read_file smatrix1.txt
C = A'
% The following command should not work
D = A * A
% The following commands should work
D = A * C
E = C * A
F = D * D
G = D *01 D
SHOW_VARS
F[0,0]
F[0,1]
F[1,1]
QUIT
```

Try running the above script when you load the project in Eclipse (or via the solution jar). Typing HELP from the command line explains what each command does if it is not obvious from the output.

We'll write a Mini-Matlab script such as the one above for testing the functionality implemented as part of this project.

3 Project Description

The project has two implementation parts:

1. **Reading the sparse file format:** Implement sparse file reading in method `readSparseMatrixFormat` in `Matrix.java`. The file format should be self-explanatory from previous discussion. Parts of the file reading have already been done for you – critically look at the method `readDenseMatrixFormat` for general pointers on code interaction with the constructor that calls it and for code snippets you can reuse (e.g., how to split a String on whitespace).

2. **Implementing `SparseMatrix`:** Create a new class `SparseMatrix` that extends `Matrix` and provides the same functionality as `DenseMatrix` except by exploiting sparse Matrix representations as discussed above.

   One you complete `SparseMatrix` you should see the comments of `Pro4_utorid.java` for how to change the Mini-Matlab implementation from `DenseMatrix` to `SparseMatrix`. **It is critical to make this change before submitting your project so that we can test your `SparseMatrix` functionality.** An indicator that your `SparseMatrix` library is implemented efficiently is whether it runs quickly (much less than one second) on the Mini-Matlab input script `input2.txt`.

4 Helpful Hints

- The first difficulty with this project will come in fully understanding what has been given to you – especially tracing the Polymorphic method calls that occur in the Matrix class (note that methods in Matrix call many methods that are abstract in Matrix). `DenseMatrix.java` has been given to you as a correct and complete example of how to extend the Matrix subclass and understanding every detail of how that code works is critical for implementing `SparseMatrix.java` in this project.

- A second and more minor difficulty will come in understanding the file formats and that they are distinct from `DenseMatrix` and `SparseMatrix`. Consider that `dmatrix1.txt` and `smatrix1.txt` in the provided starter code encode the same matrix – in addition to the information in this project handout below, this should be sufficient for you to figure out how each matrix is encoded. Also critically note that the file formats on disk (dense and sparse) **are different** than the way the matrices are stored in memory in Java (also dense and sparse); i.e., it is possible to read a `DenseMatrix` from either a dense or sparse file format and similarly for `SparseMatrix` – this is why the file reading code is implemented in the `Matrix` superclass.

- The third and major difficulty will come in understanding how to manipulate nested `HashMaps` to achieve the sparse encoding in your `SparseMatrix` class. You should start with the `get` and `set` methods of `SparseMatrix` – **note that zeroes should never be stored**. A related difficulty will involve implementing `multiply` in `SparseMatrix` – while the solution can be specified in 10 lines of Java – this single method will take a lot of your project time and some ingenuity in implementing it efficiently.

- For debugging `SparseMatrix`, note that `DenseMatrix` has been given to you and by definition of being a correct implementation of matrix storage and operations, its output should be identical to `SparseMatrix` when the same files are loaded and same operations computed. You should use a comparison of `SparseMatrix` and `DenseMatrix` for testing purposes.

- The mini-Matlab interface implemented in `Pro4_utorid.java` provides a convenient interface for testing out functionality of this Java matrix library. `input1.txt` and `input2.txt` are provided as sample inputs. **If you do not know how (a) to redirect these inputs to your program from a file so that you**

do not have to type everything in manually or (b) you do not know how to run the solution jar with redirected input, see previous Project posts on Blackboard, and if you're still unsure, ask a lab TA. Note that except for one line that you need to change once your `SparseMatrix` is complete (see the comments in `Pro4_utorid.java` for more details), this mini-Matlab interface only references the `Matrix` class and relies on Polymorphism to invoke the appropriate method calls for the `Matrix` objects. While you do not need to change `Pro4_utorid.java`, it is worthwhile understanding how this code works – the actual Matlab implementation is not so much different.

- **Error Checking:** As in Project 3, grading emphasis is not on error-checking and will mainly focus on well-formed test cases. However, you should check for and report errors in `SparseMatrix` for any cases that `DenseMatrix` reports errors (e.g., out of bounds matrix indices or improper dimensions of the operands on matrix multiplication). You do not need to report errors in `SparseMatrix` if they are not reported in `DenseMatrix`.