Solving Sparse Linear Equations using efficient pivoting orders in Gaussian Elimination

ENEE 641: Programming Assignment 2 Part 2

Sayantan Sarkar

Contents:

1. Introduction

A sparse matrix has very few non zero elements, hence if storage and manipulation of sparse matrices is done with special sparse structures, it is possible to gain great savings in running time of algorithms. Also sparse matrices are a common occurrence, especially in systems of linear equations. Hence it is crucial to find efficient ways of dealing with sparse matrix equations.

Gaussian Elimination is a tried and tested methods for solving equations, but an inefficient ordering of elimination can give rise to many non zero elements, thus reducing the advantage of a sparse matrix. Therefore it is important to find pivoting orderings that reduce the number of non zero elements (or fillins in graph theoretic terms). In the last part of this assignment, Lexp and Lexm were implemented which were designed to find elimination orders that reduce fillins. In this report we study the implementation of GEM, that follows such an ordering and how that can speed up the process.

2. Problem Definition

In this work, GEM is implemented which follows the lexp/lexm/original ordering and tries to find a solution for a sparse set of equations. The vector b is generated randomly. The number of fillins generated by GEM is compared to the number of filins got from lexp/lexm/original order. Finally a sanity check is also implemented that fills out the graphs using fillin edges from lexp/lexm/original order and runs lexp/lexm on them again. This second run of lexp/lexm should ideally not generate any fillins.

3. Features of current implementation

The gaussian elimination function uses a sparse matrix A, an array of doubles for matrix B and an array of doubles containing the diagonal elements of matrix A as its input. It outputs a list of fillins generated by GEM (that is zero elements turning non zero), the number of fillins generated, the number of rows processed (which is important in case there was a premature termination) and finally the solution (in case the GEM process finishes).

As mentioned above the input matrix A is in a spase form. The matrix A is represented as an array of linked lists. Each linked list node contains 2 numbers, column number and a value. Therefore The ith linked list represents the ith row and if we traverse the linked list, we find the non zero elements of that row. Here is a diagram showing an example of how a matrix is represented in sparse (linked list based) method.

$$\begin{bmatrix} 0 & 1.2 & 2.5 & 0 \\ 1.2 & 0 & 0 & 0 \\ 2.5 & 0 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

Another feature of this sparse representation is that the elements of each linked list is sorted by column when elements are inserted into the matrix. This inserting the elements is an O(a), where a is average number of elements in each row. In the worst case (that is for dense matrices) each insertion is O(n), and as there are n^2 elements the order is $n^2O(n)$ that is $O(n^3)$. But we are dealing with sparse matrices here, so the time complexity is $n^2O(a)$ (that is $O(an^2)$, where 'a' is most likely a sublinear function (eg: a constant or log(n)). Therefore inserting in sorted order does not take much time.

Note for the bonus part, asymmetric matrices are also dealt with. In that case when element i,j is encountered, row i is checked for element j and row j is checked for element i. If not found, the value is inserted at both rows. If they are found, then the smaller (in absolute sense) of the earlier value and the current value is retained.

The Gaussian elimination works as follows. For the current row in the elimination order it checks its diagonal element and then traverses its adjacency list and finds non zero elements. For example for row i, it checks if A(I,i) is non zero. If its zero, GEM terminates here. If not it checks every element in the list, that is A(i,j). Since A(i,j) equals A(j,i) and we want to make the columns i contain zeroes for all the ros that have not been processed yet, we multiply row i with –A(j,i)/A(i,i) and subtract from row j. This makes element A(j,i) zero. We continue this process till we finish. Notice that after we process a whole row, if we consider the unprocessed part of the matrix, it still retains its symmetry.

Then we start back substitution. Now we follow the opposite order, that is, the last equation processed is now the first to be processed. Going in the reverse direction, the solutions for each element is found by back substitution. At the i^{th} back substitution the values for the previous i-1 variables have already been foun, and this we find the value for variable i.

3.1 Pseudocode

Here is a pseudocode for Gaussian elimination:

```
for i = 0 to dim-1
{
	process order[i]th row in current iteration, where order is lexp/lexm/original order
	ithrow = order[i]
	if (abs(diagarray[ithrow]) > 0.000001)  //if diagonal is very small consider it to be an artifact from floating point operations and ignore it
	{
		currrow = matrix[ithrow];
		while (currrow != NULL) //scan the adjacency list
		{
			if (currrow->col != ithrow && processed[currrow->col] == 0) //if its not the diagonal element and if  that row hasn't already been processed
			{
				//attack matrix[currrow->col] row and make matrix[currrow->col, ithrow] element 0
				//factor = M[row2, row1] / M[row1, row1]
				factor = currrow->value / diag_row;
				subtractrows(matrix, ithrow, currrow->col, factor, diagarray, fillincount_gauss, fillintrack);
				b[currrow->col] = b[currrow->col] - factor*b[ithrow];
			}
			currrow = currrow->next;
		}
	}
	else
		premature termination, so break;

	processed[ithrow] = 1;
}
	//back substitution
	if (i == dim) //that is all rows have been processed, (its not premature termination)
	{
		for i = dim-1 to 0
		{
			ithrow = order[i];
			traverse = matrix[ithrow];
			sum = 0
			while (traverse != NULL)
			{
				//In the ith equation, the variable xi is unknown. All other variables we encounter should be known
```

```
                    if (traverse->col != ithrow)
                        sum += soln[traverse->col] * traverse->value;
                    else
                        //take note of the coefficient of the unknown variable
                        coeff = traverse->value;
                    traverse = traverse->next;
                }
                //store solution
                soln[ithrow] = (b[ithrow] - sum) / coeff;
            }
        }
```

Below is the pseudo code for subtractrows() that is used in GEM

```
subtractrows(matrix, r1, r2, factor)
{
    while (row1 != NULL && row2 != NULL) //traverse rows till one of them becomes NULL
    {
        //this case is for a column where row1 is non zero and row2 is 0,
        //So a new value is created in row2 (with value = -A(r1,col)
        if(row1->col < row2->col)
        {
            insert a node in row2.
            if (not a diagonal element)
                Mark as a fillin edge generated from GEM
            else
                update diagonal array
            row1 = row1->next; continue
        }
        //this case is when for a particular column, row2 is non zero, but row 1 is 0
        if (row1->col > row2->col)
            row2 = row2->next; continue;
        //In this case both row1 and row2 have non zero elements for a particular column
        if (row1->col == row2->col)
            if (row2->col == r1)
                delete this node.
            else
                update the value of A(r2, col) = A(r2, col) - factor * A(r1, col)
    }
    if (row2 == NULL) //row 2 finished first
    {
        //we still have some elements of row1 left over
        while(row1 != NULL)
        {
            insert remaining elements in row2
            if they are not diagonal elements
                mark them as fillin edges
            else
                update diagonal array
        }
    }
}
```

Note that we need to find the diagonal element of each row (to check if its non zero, and also to calculate the factor). Another salient feature of this code is that it stores diagonal elements in an array, so we have quick lookup, and do not have to scan the linked list. The diagonal array is kept updated because subtractrows() function updates the diagonal array in case it modifies any diagonal element.

3.2 Optimizations used in Gaussian Elimination

Here are a list of optimizations used in Gaussian elimination:

1. Gaussian elimination works on a sparse matrix, so that saves on space and time
2. The subtractrows() function that subtracts two rows (row2 = row2 – factor * row1), is also implemented in a sparse fashion
3. IN GEM, when row i is being processed, only those rows which are linked to that node in the graph are processed and the rest are ignored. This is because, if a node is not connected to the current node, it means that they is a zero coefficient in between them. For example when processing row i, say A(I,j) is non zero. By symmetry A(j,1) is also 0. We wish to subtract row i from row j, so that A(j,i) becomes zero. But if A(I,j) is zero anyway, we do no need to process row j (as A(j,i) is also zero).
4. Diagonal elements are stored in an array for quick lookup. The array is updated if any diagonal element is modified.
5. No row swappings are performed, something that is usually done in normal GEM.

Thus these optimizations make the GEM code significantly fast.

3.3 Comparing fillins generated by graph and GEM

Given two lists of fillins one from the graph and the other from GEM, compareFillins() function compares them. For each fillin edge in the graphs fillin list (say (i,j), it checks for a fillin edge (i,j) and (j,i) in the fillin list of GEM. If found this fillin edges from both lists are deleted and the count for successful match is incremented. If a match is not found, the counter counting number of fillins in graph not found in GEM fillins is incremented. At the end of the process, we are ideally left with no unmatched fillin edges. Note if the number of processed rows is p, then we do not expect fillin edges of rows beyond p to be present in the GEM fillin list.

3.4 Overview of the code

1. Generate b matrix randomly
2. Read A matrix
3. Lexp
   3.a Run lexp, run fillin on lexp order
   3.b Run GEM on lexp order
   3.c Compare fillins generated by lexp and by GEM
4. Lexm
   4.a Run lexm, run fillin on lexm order
   4.b Run GEM on lexp order
   4.c Compare fillins generated by lexm and by GEM
5. Original
   5.a Run fillin on original order
   5.b Run GEM on original order
   5.c Compare fillins generated by original order and by GEM
6. If COMPARE_FILLINS macro is enabled, check that no fillins are generated from filled in graph
   6.a fill out graph with fillin edges from lexp and run lexp and lexm on it
   6.b fill out graph with fillin edges from lexm and run lexp and lexm on it
   6.c fill out graph with fillin edges from original order and run lexp and lexm on it

Macro ALLOWLEXP set to 0 disables part 3. Macro ALLOWLEXM set to 0 disables part 4. Macro COMPARE_FILLINS set to 0 disables part 6.

4. Time and Space complexity analysis

Normal Gaussian Elimination for n equations takes $O(n^3)$ time and $O(n^2)$ space. Here we take advantage of the sparsity and the graph theoretic analysis that we perform when we run Lexp and Lexm, which allows us to run GEM much faster and using much less space.

4.1 Space Complexity

4.1.1 Space complexity for storing input matrix.

The input matrix is stored in a sparse fashion. Each row is represented by a linked list. The number of elements we need to store as linked lists nodes is given by the number of non zero elements that is number of edges in the original graph (e). Also since we need to store the heads of n linked lists we need an array of size n. Therefore in total, we need O(n+e) space.

4.1.2 Space complexity for running Gaussian Elimination

The GEM process generates fillins and also the solution if it finishes. Therefore the space required by GEM is O(f) when GEM terminates prematurely and O(f + n), when GEM finishes, where f is the number of fillins generated by GEM and n is number of equations

4.2 Time complexity

4.2.1 Time complexity for initializing matrix

We insert the elements in the linked lists in a sorted by column order. Thus if we have a elements per row on an average, for each element we have to do O(a) work (as we have to scan the linked list to find the suitable point of insertion). Thus for e^2 elements, the complexity is $O(ae^2)$ where e is the number of non zero elements. As a can be considered a constant for sparse matrices this quantity is almost $O(e^2)$

4.2.2 Time complexity for Gaussian Elimination

The time complexity analysis for GEM is a bit involved. First consider the case that GEM terminates prematurely. Let p be the number of rows that were processed, e be the number of edges (non zero elements) initially, n be the number of equations and f be the number of fillins generated by GEM. Then e/n denotes the average number of edges per row and f/n denotes the average number of fillins per row.

Now consider the work done in each row. We have to scan the adjacency list of each row (which contains on an average e/n + f/n number of elements), and for each such element we find, we must subtract those rows. Subtraction of rows too consists of scanning e/n + f/n elements. Note that the quantity e/n + f/n denotes the average number of non zero elements present in each row. Therefore each row has e/n + f/n elements and subtraction for each takes around similar time. So each row is processed in $(e/n + f/n)^2$ time. Since p rows were processed before GEM terminated, the complexity is $O(p(e/n + f/n)^2)$. In a simplified notation it is $O(na^2)$, where a is the average number of elements in a row. Notice that in the worst case e and f are both $O(n^2)$ and simplifying the expression above would lead us to $O(n^3)$.

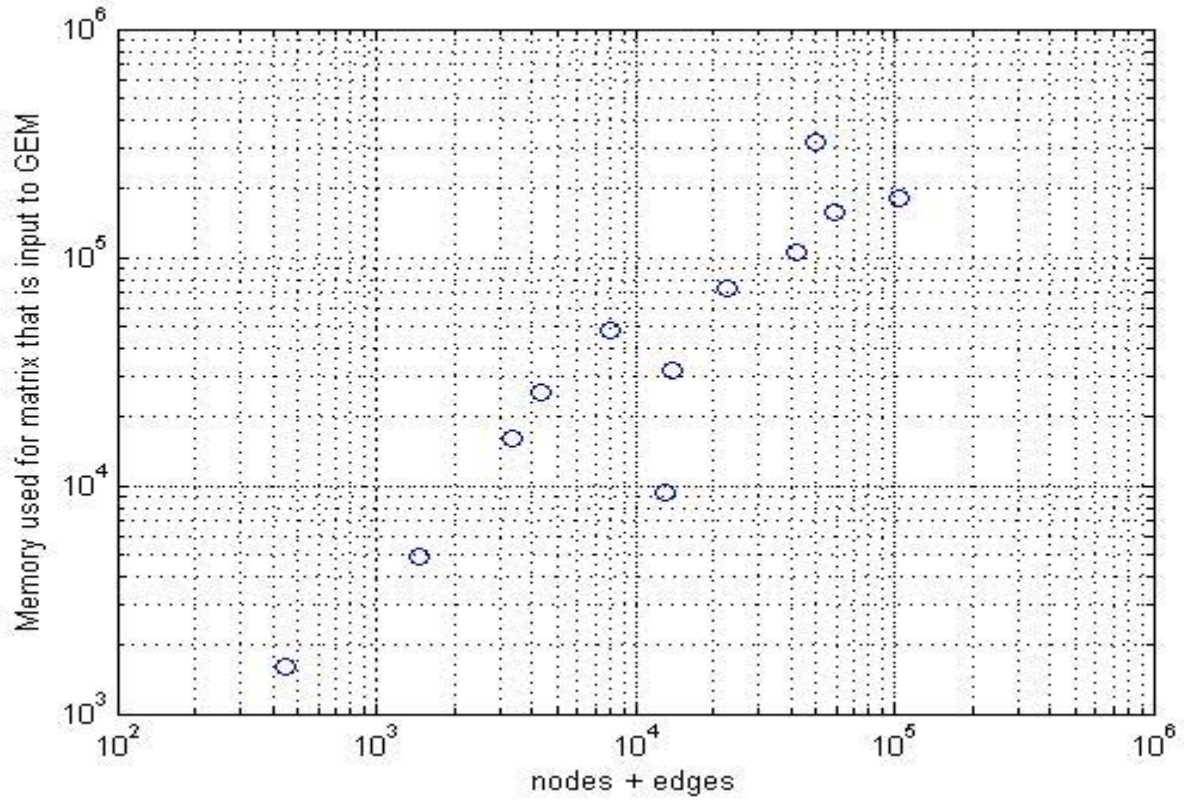
In case GEM terminates all rows are processed. Therefore the number of processed rows p, becomes n. Also to generate each solution, we must scan the adjacency list which has O(e/n + f/n) elements. There are n solutions to be generated so the time for generating solutions by back substitution is O(n(e/n + f/n)) or O(e + f).

5. Experimental Data and plots

The first table described the memory usage of the input matrix.

Matrix ID	No of rows/columns	Non zero elements/edges	Input size
68	2003	11973	32048
220	100	347	1600
240	1000	2375	16000
344	588	12429	9408
876	306	1162	4896
889	9801	48413	156816
1205	11445	93781	183120
1239	3002	5000	48032
1427	20000	30000	320000
1546	4563	17969	73008
1553	6611	35472	105776
2401	1589	2742	25424

This graph shows the memory used for storing the input matrix in a sparse form vs the number of nodes (variables) + edges (non zero elements)

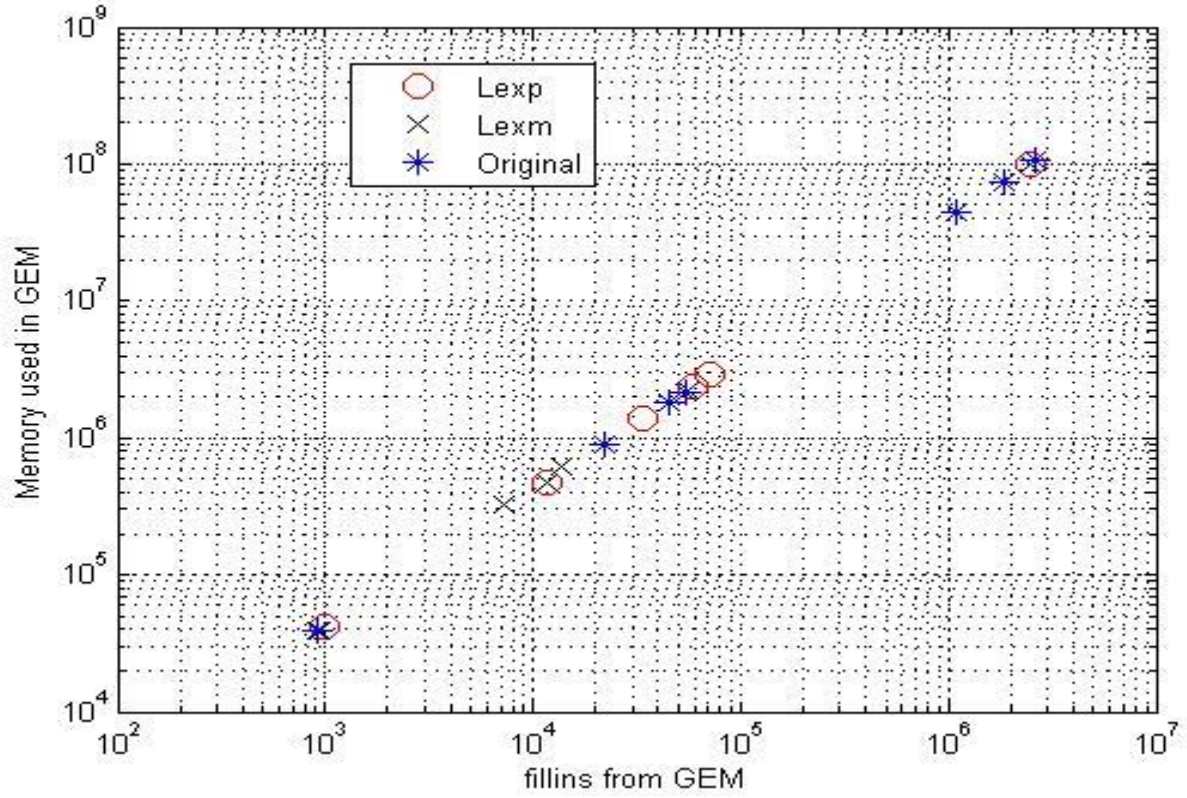


This table tabulates the space requirements of GEM and the number of fillins GEM generates

Matrix ID	Fillins in lexp from GEM	Space for GEM for lexp	Fillins in lexm from GEM	Space for GEM for lexm	Fillins in original order from GEM	Space for GEM for orig

68	0	16080	0	16080	0	16080
220	986	41516	938	39596	916	38716
240	0	8056	0	8056	45220	1816856
344	59700	2399836	55710	2240236	53962	2170316
876	11728	475316	11652	472276	21878	881316
889	2461474	98655056	2461474	98655056	1863176	74723136
1205	0	91616	0	91616	0	91616
1239	0	24096	0	24072	0	24072
1427	0	520076	0	520076	0	520076
1546	33732	1385840	7178	323680	1092580	43794536
1553	70866	2887584	13872	607824	2624446	105110136
2401	0	12768	0	12768	0	12768

The following graph shows the memory used for GEM (for all 3 orderings) vs fillins generated by GEM

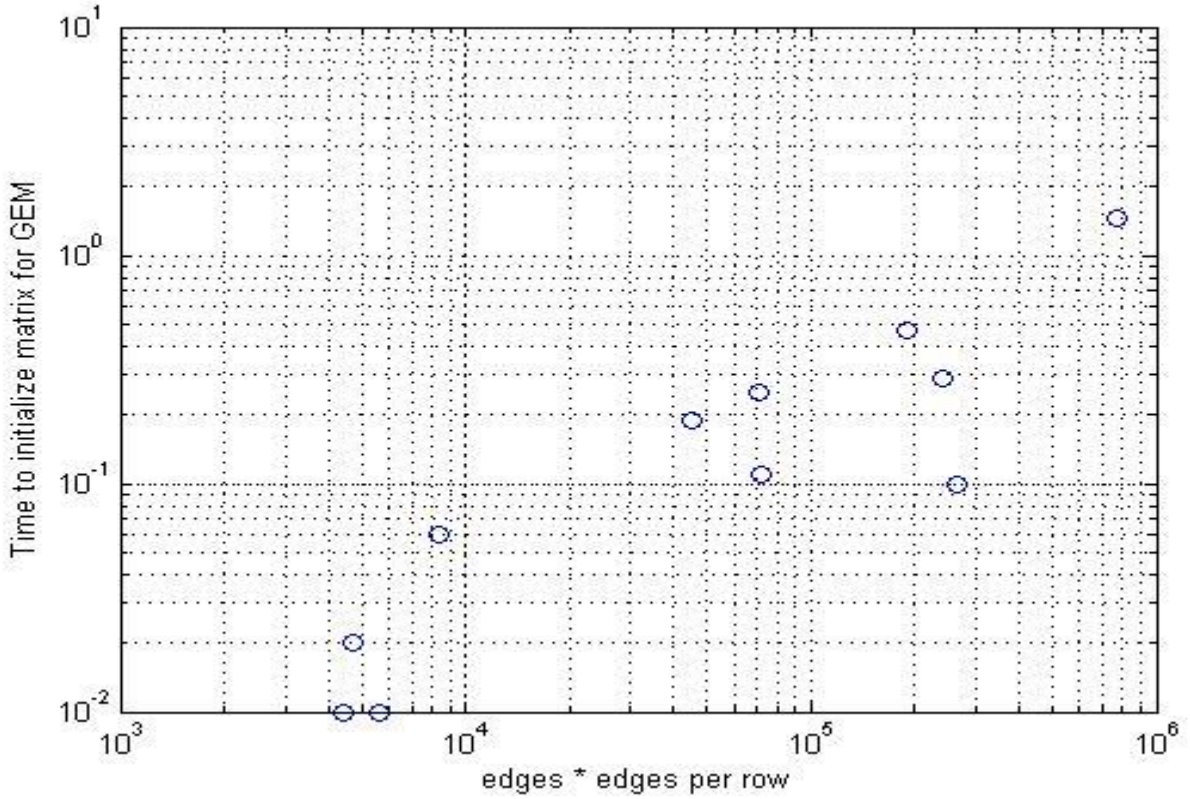


This graph tabulates the time taken to initialize the sparse matrix and the average number of non zero elements per row (which is total number of non zero elements divided by number of rows)

Matrix ID	No of rows/columns	Non zero elements/edges	Edges (non zero elements) per row	Time taken to initialize matrix
68	2003	11973	5.977	0.11
220	100	347	3.470	0
240	1000	2375	2.375	0.01
344	588	12429	21.138	0.1
876	306	1162	3.797	0.01
889	9801	48413	4.940	0.29
1205	11445	93781	8.194	1.44

1239	3002	5000	1.666	0.06
1427	20000	30000	1.500	0.19
1546	4563	17969	3.938	0.25
1553	6611	35472	5.366	0.47
2401	1589	2742	1.726	0.02

The following graph shows how the time taken to initialize the sparse matrix varies according to the product of number of edges and average number of edges per row.



The following table tabulates the data for GEM running on Lexp for different graphs.

Matrix ID	Number of rows processed	Fillins generated by GEM for lexp / #rows	Finished (0 means premature termination)	Estimated solution time	GEM run time
68	3	0	0	0	0
220	100	9.86	1	1000	0.01
240	311	0	0	0	0
344	588	101.53	1	59700	0.76
876	306	38.32	1	11700	0.05
889	9801	251.14	1	2461500	45.24
1205	0	0	0	0	0
1239	1	0	0	0	0
1427	20000	0	1	0	0.04
1546	2313	7.39	0	0	0.23
1553	3151	10.719	0	0	0.37
2401	0	0	0	0	0

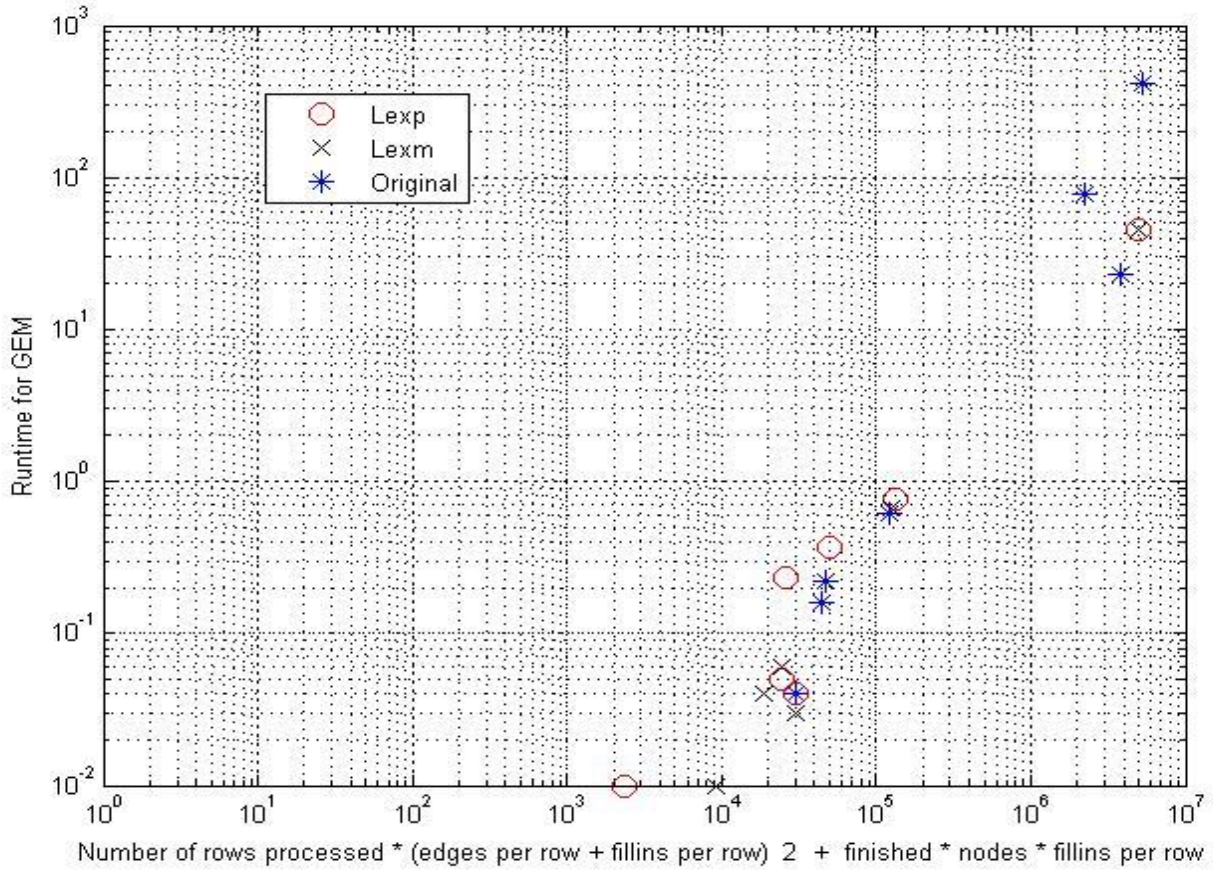
The following table tabulates the data for GEM running on Lexm for different graphs.

Matrix ID	Number of rows processed	Fillins generated by GEM for lexm / #rows	Finished (0 means premature termination)	Estimated solution time	GEM run time
68	1	0	0	0	0
220	100	9.38	1	900	0
240	305	0	0	0	0
344	588	94.74	1	55700	0.66
876	306	38.08	1	11700	0.06
889	9801	251.14	1	2461500	44.93
1205	0	0	0	0	0
1239	0	0	0	0	0.01
1427	20000	0	1	0	0.03
1546	1658	1.573	0	0	0.01
1553	2549	2.10	0	0	0.04
2401	0	0	0	0	0

The following table tabulates the data for GEM running on original order for different graphs.

Matrix ID	Number of rows processed	Fillins generated by GEM for original / #rows	Finished (0 means premature termination)	Estimated solution time	GEM run time
68	3	0	0	0	0
220	100	9.16	1	916	0
240	988	45.22	0	0	0.22
344	588	91.77	1	53962	0.62
876	306	71.50	1	21878	0.16
889	9801	190.10	1	1863176	22.89
1205	0	0	0	0	0
1239	0	0	0	0	0
1427	20000	0	1	0	0.04
1546	4563	239.44	1	1092580	78.81
1553	6611	396.98	1	2624446	417.25
2401	0	0	0	0	0

The following graph plots the time taken to run GEM for lexp, lexm and original. The plot is versus this quantity: number of processed rows * (average edges per row + average fillins per row)^2 + finished * number of nodes * average fillins per row. The reason why this quantity is chosen is explained in the time complexity analysis section.



Fillins compare: In the output file, the number of fillins from lexp/lexm/original order is compared to number of fillins from GEM. Also code has been implemented to find the intersection of both sets and identify leftover edges if any. The Number of fillins from GEM and lexp/lexm/original ordering match up.

6. Discussions

1. It has been observed that lexp and lexm gave premature terminations for many cases, but original order was able to process all rows and produce a solution. The reason for this is that lexp and lexm tries to minimize fillins. Therefore they have a tendency to select nodes with lower centrality (in complex networks parlance), because fringe nodes are likely to have less neighbours, hence likely to generate less fillins. But as they generate less fillins, they also generate less non zero elements during the Gaussian Elimination process. Therefore in sparse graph, we are likely to encounter zeroes in diagonals (causing GEM to stop). In original ordering it is more likely that the nodes are not fringe nodes, hence they are more likely to generate non zero values, thus allowing GEM to complete.

2. Deviation in linearity in space calculation is due to the fact that diagonal elements give rise to only 1 edge while other elements give rise to 2 edges. So the number of non zero elements is not exactly half of the total number of edges.

3. The GEM method implemented here is extremely fast for sparse perfect graphs, as they generate no fillins and as shown in the analysis, the number of fillins generated play a major role in determining the time complexity. Therefore graph 1427 which is of size 20000x20000 finishes GEM in under 1 second, as lexp is able to find a perfect ordering for it.

4. The graphs have been plotted in a log-log axis, as some values are very large and others very small. So the linear relationship is not quite visible on a normal plot.

5. For GEM to terminate using lexp and lexm ordering, the sparse matrix must have a banded structure, that is whose non zero elements are concentrated around the main diagonal. This is required to get non zero elements in the diagonal without which GEM would terminate.

7. Notes on bonus parts and changes made in past code

Bonus part 1: The initial graph is filled with fillins generated by lexp and then lexp and lexm are run to check for fillins. There were none found. The same was done for the expanded graphs using lexm fillins and original order fillins. The comparision is shown in the output file. The COMPARE_FILLINS macro must be enabled for this.

Bonus part 2: Asymmetric matrices are dealt with. Matrix 83 with is asymmetric has been processed and its output has been included as an example.

Bonus part 3: GEM has been implemented in a sparse format without resorting to $O(n^2)$ storage.

Please note that fillin function has been modified as the earlier function was incorrect. In the earlier fillin function that was submitted, an edge is counted as a fillin in the beginning (and added to fillin list) whenever a new edge forms. Later if a double edge is encountered, the edge is deemed to be not a fillin edge and it is removed from the fillin list. Since this implementation was giving incorrect results, the fillin function has been changed. Now when a potential edge is created, first we check if the edge already exists. If it does, we ignore the edge, but if it doesn't we count it as a fillin edge.

8. C Code

```
#define _CRT_SECURE_NO_WARNINGS
#define COMPARE_FILLINS 1
#define ALLOWLEXM 1
#define ALLOWLEXP 1

#include<stdio.h>
#include<stdlib.h>
#include "mmio.h"
#include<time.h>
#include<limits.h>

//structure for header cell of doubly double linked list
typedef struct headercell
{
	int flag;
	struct headercell *head;
	struct vertexcell *vset;
	struct headercell *back;
}headercell;

//structure for adjacency list
typedef struct vertex
{
	int num; //final ordering
	int varnum;  //variable number
	struct vertexcell* vc;
	struct vertexLL* neighbours;
}vertex;

//structure for adjacency list's linked list that shows neighbours
typedef struct vertexLL
{
	struct vertex* v;
	struct vertexLL *next;
}vertexLL;

//structure for body cell of doubly double linked list
typedef struct vertexcell
{
	struct headercell *flag;
	//struct vertex *vert;
	int vertexLocation; //Location of vertex in the vertex array (adj list array)
	struct vertexcell *next;
```

```
        struct vertexcell *back;
}vertexcell;

//auxiliary structure for implementing fixlist as a linked list
typedef struct newSetLL
{
        struct headercell* hcell;
        struct newSetLL* next;
}newSetLL;  //points to newly created sets, so that we can later set their header->flag to 0

//structure containing pointers to structures required by lexp (doubly double linked list and adjacency list)
typedef struct init
{
        struct headercell* eqgraph;
        struct vertex* adjlist;
}init;

//structure to contain fillin edges
typedef struct fillinLL
{
        double val;
        int varnum;
        struct fillinLL *next;
}fillinLL;

//structure used for keeping track of labels in lexm
typedef struct lexMstruct
{
        int varnum;
        int label;  //even labels are old, odd represent +1/2
        int reached;  //0 shows unreached, 1 shows reached
        struct lexMstruct *next;
}lexMstruct;

//simple linked list for implementing 'reach' in lexm
typedef struct simpleLL
{
        struct vertex* v;
        struct simpleLL *next;
}simpleLL;

typedef struct LLmatrix
{
        int col;
        double value;
        struct LLmatrix *next;
        struct LLmatrix *back;
}LLmatrix;

//function prototypes
init initialize(float* symmatrix, int dim);
void insertIngraph(vertex *adjlist, int i, int j);
void printAdjList(vertex *adjlist, int dim);
void printGraph(headercell *eqgraph);
int* lexP(init *initial, int dim);
void checkcells(init *initial, int dim);
void fillin(init* initial, int *lexpOrder, int dim);
int* lexM(vertex* adjlist, int dim);
fillinLL **fillin2(vertex* initial, int *lexpOrder, int dim, int *fillincount);
void printFillin(fillinLL **fillin, int dim);
init initializelexP(char *filename, int *dim);
vertex* initFromFile(char *filename, int* dim);
void printreached(lexMstruct **labeltrack, int dim);
void printreach(simpleLL** reach, int dim);
void printlabel(lexMstruct **labeltrack, int dim);
void printcountingsort(lexMstruct **countingSort, int dim2);
```

```
void writeFillinToFile(FILE* f, fillinLL **fillin, int dim, int fillincount);
void freeadjlist(vertex* adjlist, int dim);

void insertsortedLL(LLmatrix** row, int x, int y, double temp);
LLmatrix** readFromFile(char *filename, int dim, double* diagarray);
void printMtx(LLmatrix **matrix, int dim, int mode);
double* GaussianElim(LLmatrix **matrix, double* b, int dim, fillinLL **fill, int* order, double* diagarray, int*
fillincount_gauss, fillinLL **fillin_gauss, int* processed);//, int* vertexToOrder);

void subtractrows(LLmatrix **matrix, int row1, int row2, double factor, double* diagarray, int* fillincount_gauss,
fillinLL **fillintrack);
void printMtxRow(LLmatrix* m);
int compareFillins(fillinLL **fill, fillinLL **fillin_gauss, int dim, int* vertexToOrder, int processed);
int* getOrder(vertex* adjlist, int dim);
int deleteFromLL(fillinLL **fill, int row, int varnum, int *delfromhead);

void printToFile(FILE* f, int fillincount_gauss, fillinLL** fillin_graph, fillinLL** fillin_gauss, double *soln, int
successfulmatch, int dim, int processed, int zerodiagonal);
void printFillinsToFile(FILE* f, fillinLL** fillin_gauss, fillinLL** fill, int dim);

void insertsortedLLasym(LLmatrix** mtx, int row, int col, double val);
void insertIngraphasym(vertex *adjlist, int i, int j);
void filloutGraph(vertex* adjlist, fillinLL** fillinlexp, int dim);
fillinLL** makeFillinlistCopy(fillinLL** fillin, int dim);

//global variables to count memory usage
long maxmemcount = 0;
long memcount = 0;

long memcount_GEM_inputs = 0;
long memcount_GEM = 0;

//    memcount_GEM += dim*sizeof(int);
//    maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;

int main(int argc, char *argv[])
{
      int dim;
      int i = 0, j, fillincount = 0;
      char *filename = argv[1];
      char *outfile = (char*)calloc(20, sizeof(char));
      int *originalOrder = NULL;
      clock_t start, stop, temp, stop1;
      int *order;
      fillinLL **fill;
      int fcount = 0;
      char type;
      int *vertexToOrder = NULL;
      int fillincount_gauss = 0;
      int processed = 0;
      fillinLL **fillin_gauss_lexp, **fillin_gauss_lexm, **fillin_gauss_orig;

      int lexmfillincount = INT_MAX, origfillincount = INT_MAX;
      fillinLL **fillinOrigOrder = NULL;
      fillinLL **fillinlexm = NULL;
      int *lexmOrder = NULL;
      int *vertexToOrder_lexm = NULL;
      int *vertexToOrder_orig = NULL;

      int gaussleftover, graphleftover, successfulmatch;
      start = clock();

      float startfloat, stopfloat, timediff;

      srand(time(NULL));
```

```
//generate output filename
while (1)
{
	if (argv[1][i] == '.')
		break;
	outfile[i] = argv[1][i];
	i++;
}
outfile[i++] = '.';
outfile[i++] = 'o'; outfile[i++] = 'u'; outfile[i++] = 't';


FILE* f = fopen(outfile, "w");

//call function to create adjacency list and doubly double linked list for lexp to use
init initial = initializelexP(filename, &dim);

fprintf(f, "File name: %s. Number of variables = %d\n", argv[1], dim);

//generate b matrix
double *matrixB_lexp = (double*)malloc(dim * sizeof(double));
double *matrixB_lexm = (double*)malloc(dim * sizeof(double));
double *matrixB_orig = (double*)malloc(dim * sizeof(double));
memcount_GEM_inputs = memcount_GEM_inputs + dim * sizeof(double);
FILE *fmtx = fopen(filename, "r");
MM_typecode matcode;
if (mm_read_banner(fmtx, &matcode) != 0)
{
	printf("Could not process Matrix Market banner.\n");
	exit(1);
}
fprintf(f, "B matrix: ");
if (mm_is_real(matcode) == 0)  //integers
{
	double r;
	for (i = 0; i < dim; i++)
	{
		r = (rand() >> 8) % 50;
		fprintf(f, "%lf ", r);
		matrixB_lexp[i] = r;
		matrixB_lexm[i] = r;
		matrixB_orig[i] = r;
	}
}
else
{
	double r1, r2;
	for (i = 0; i < dim; i++)
	{
		r1 = (rand() >> 8) % 50;
		r2 = (rand() >> 8) % 50;
		if (r2 != 0)
		{
			matrixB_lexp[i] = r1 / r2;
			matrixB_lexm[i] = r1 / r2;
			matrixB_orig[i] = r1 / r2;
			fprintf(f, "%lf ", r1 / r2);
		}
		else
		{
			matrixB_lexp[i] = r1 / (r2 + 1.0);
			matrixB_lexm[i] = r1 / (r2 + 1.0);
			matrixB_orig[i] = r1 / (r2 + 1.0);
			fprintf(f, "%lf ", r1 / (r2 + 1.0));
		}
	}
```

```
	}
	fprintf(f, "\n\n");
	fclose(fmtx);

	printf("Matrix: %s, Nodes: %d", argv[1], dim);

	temp = clock();
	fillin_gauss_lexp = (fillinLL**)calloc(dim, sizeof(fillinLL*));
	double *diagarray_lexp = (double*)calloc(dim, sizeof(double));
	memcount_GEM_inputs = memcount_GEM_inputs + dim * sizeof(double);

	printf("\nMemory used for inputs to GEM: %d\n", memcount_GEM_inputs);
	LLmatrix **matrixA = readFromFile(filename, dim, diagarray_lexp);  //read A matrix from file
	stop = clock();
	startfloat = temp / 1.0;
	stopfloat = stop / 1.0;
	timediff = (stopfloat - startfloat) / CLOCKS_PER_SEC;
	printf("\n\n Time taken for reading and storing matrix for GEM is: %f seconds  (%lu, %lu)\n\n", timediff, temp,
stop);


#if ALLOWLEXP
	//run lexp
	int *lexpOrder = lexP(&initial, dim);
	int *vertexToOrder_lexp = getOrder(initial.adjlist, dim);

	//calculate fillin edges
	fillinLL **fillinlexp = fillin2(initial.adjlist, lexpOrder, dim, &fillincount);
	int lexpfillincount = fillincount;
	printf("LexP gave %d fillin edges\n", fillincount);
	fprintf(f, "Lexp order: (listed in order of elimination)\n");
	for (i = 0; i < dim; i++)
		fprintf(f, "%d ", lexpOrder[i] + 1);
	fprintf(f, "\nLexP gave %d fillin edges\n", fillincount);

	//Run GEM for lexp ordering
	order = lexpOrder;
	fill = fillinlexp;
	fcount = lexpfillincount;
	vertexToOrder = vertexToOrder_lexp;

	temp = clock();
	memcount_GEM = dim * sizeof(fillinLL*); //accounts for size of fillin array which contains fill in edges generated by GEM
	double *soln_lexp = GaussianElim(matrixA, matrixB_lexp, dim, fill, order, diagarray_lexp, &fillincount_gauss,
fillin_gauss_lexp, &processed);//, vertexToOrder);
	printf("Memory used for Gaussian Elimination in lexp ordering = %d", memcount_GEM);
	stop = clock();
	startfloat = temp / 1.0;
	stopfloat = stop / 1.0;
	timediff = (stopfloat - startfloat) / CLOCKS_PER_SEC;
	printf("\nTime taken for running GEM on lexp ordering: %f seconds  (%lu, %lu)", timediff, temp, stop);
	printFillinsToFile(f, fillin_gauss_lexp, fill, dim);

#if COMPARE_FILLINS
	fillinLL** fillinlexp_copy = makeFillinlistCopy(fillinlexp, dim);


	temp = clock();
	successfulmatch = compareFillins(fill, fillin_gauss_lexp, dim, vertexToOrder, processed);  //compare fillins
generated by graph and by lexp
	stop = clock();
	startfloat = temp / 1.0;
	stopfloat = stop / 1.0;
	timediff = (stopfloat - startfloat) / CLOCKS_PER_SEC;
	printf("\nTime taken for comparing fillins of GEM and fillins from running Lexp is: %f seconds  (%lu, %lu)\n\n",
timediff, temp, stop);
```

```
	printToFile(f, fillincount_gauss, fill, fillin_gauss_lexp, soln_lexp, successfulmatch, dim, processed,
order[processed] + 1);
#endif
	if (soln_lexp != NULL)
		free(soln_lexp);
	if (diagarray_lexp != NULL)
		free(diagarray_lexp);
	if (lexpOrder != NULL)
		free(lexpOrder);
	if (fillin_gauss_lexp != NULL)
		free(fillin_gauss_lexp);
	//free structures used in lexp
	freeadjlist(initial.adjlist, dim);
#endif


#if ALLOWLEXM
	//create new adjacency list
	vertex* adjlist1 = initFromFile(filename, &dim);
	//run lexm
	lexmOrder = lexM(adjlist1, dim);
	vertexToOrder_lexm = getOrder(adjlist1, dim);

	fillincount = 0;
	//calculate fillin for lexm
	fillinlexm = fillin2(adjlist1, lexmOrder, dim, &fillincount);

#if COMPARE_FILLINS
	fillinLL** fillinlexm_copy = makeFillinlistCopy(fillinlexm, dim);
#endif

	lexmfillincount = fillincount;

	printf("LexM gave %d fillin edges\n", fillincount);
	fprintf(f, "Lexm order: (listed in order of elimination)\n");
	for (i = 0; i < dim; i++)
		fprintf(f, "%d ", lexmOrder[i] + 1);
	fprintf(f, "\nLexM gave %d fillin edges", fillincount);

	fprintf(f, "\n");

	//Run GEM on lexm ordering
	order = lexmOrder;
	fill = fillinlexm;
	fcount = lexmfillincount;
	vertexToOrder = vertexToOrder_lexm;
	fillincount_gauss = 0;

	double *diagarray_lexm = (double*)calloc(dim, sizeof(double));
	matrixA = readFromFile(filename, dim, diagarray_lexm);
	fillin_gauss_lexm = (fillinLL**)calloc(dim, sizeof(fillinLL*));

	temp = clock();
	memcount_GEM = dim * sizeof(fillinLL*); //accounts for size of fillin array which contains fill in edges generated by GEM
	double *soln_lexm = GaussianElim(matrixA, matrixB_lexm, dim, fill, order, diagarray_lexm, &fillincount_gauss,
fillin_gauss_lexm, &processed);//, vertexToOrder);
	printf("Memory used for Gaussian Elimination in lexm ordering = %d\n", memcount_GEM);
	stop = clock();
	startfloat = temp / 1.0;
	stopfloat = stop / 1.0;
	timediff = (stopfloat - startfloat) / CLOCKS_PER_SEC;
	printf("Time taken for running GEM on lexm ordering: %f seconds  (%lu, %lu)\n", timediff, temp, stop);

	printFillinsToFile(f, fillin_gauss_lexm, fill, dim);
```

```
#if COMPARE_FILLINS
	temp = clock();
	successfulmatch = compareFillins(fill, fillin_gauss_lexm, dim, vertexToOrder, processed); //compare fillins generated by graph and by lexm
	stop = clock();
	startfloat = temp / 1.0;
	stopfloat = stop / 1.0;
	timediff = (stopfloat - startfloat) / CLOCKS_PER_SEC;
	printf("Time taken for comparing fillins of GEM and fillins from running Lexm is: %f seconds  (%lu, %lu)\n", timediff, temp, stop);

	printToFile(f, fillincount_gauss, fill, fillin_gauss_lexm, soln_lexm, successfulmatch, dim, processed, order[processed] + 1);
#endif
	if (soln_lexm != NULL)
		free(soln_lexm);
	if (diagarray_lexm != NULL)
		free(diagarray_lexm);
	if (lexmOrder != NULL)
		free(lexmOrder);
	if (fillin_gauss_lexm != NULL)
		free(fillin_gauss_lexm);

	freeadjlist(adjlist1, dim);

#endif


	//original order
	vertex* adjlist2 = initFromFile(filename, &dim);
	originalOrder = (int*)calloc(dim, sizeof(int));
	memcount += dim*sizeof(int);
	maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
	printf("init finished\n");
	for (i = 0; i < dim; i++)
	{
		originalOrder[i] = i;
		adjlist2[i].num = i;
	}
	fillincount = 0;

	vertexToOrder_orig = (int*)malloc(dim * sizeof(int));
	printf("init arrays\n");
	for (i = 0; i < dim; i++)
		vertexToOrder_orig[i] = i;
	printf("start fillin\n");
	fillinOrigOrder = fillin2(adjlist2, originalOrder, dim, &fillincount);

#if COMPARE_FILLINS
	fillinLL** fillinorig_copy = makeFillinlistCopy(fillinOrigOrder, dim);
#endif

	origfillincount = fillincount;
	printf("\nOriginal order gave %d fillin edges\n", fillincount);
	fprintf(f, "Original order: ");
	for (i = 0; i < dim; i++)
		fprintf(f, "%d ", i + 1);
	fprintf(f, "\nOriginal Order gave %d fillin edges\n", fillincount);

	//Run GEM on original ordering
	order = originalOrder;
	fill = fillinOrigOrder;
	fcount = origfillincount;
	vertexToOrder = vertexToOrder_orig;
	fillincount_gauss = 0;
```

```
	double *diagarray_orig = (double*)calloc(dim, sizeof(double));
	matrixA = readFromFile(filename, dim, diagarray_orig);
	fillin_gauss_orig = (fillinLL**)calloc(dim, sizeof(fillinLL*));

	temp = clock();
	memcount_GEM = dim * sizeof(fillinLL*); //accounts for size of fillin array which contains fill in edges generated by GEM
	double *soln_orig = GaussianElim(matrixA, matrixB_orig, dim, fill, order, diagarray_orig, &fillincount_gauss,
fillin_gauss_orig, &processed);//, vertexToOrder);
	printf("Memory used for Gaussian Elimination in original ordering = %d\n", memcount_GEM);
	stop = clock();
	startfloat = temp / 1.0;
	stopfloat = stop / 1.0;
	timediff = (stopfloat - startfloat) / CLOCKS_PER_SEC;
	printf("Time taken for running GEM on original ordering: %f seconds  (%lu, %lu)\n", timediff, temp, stop);

	printFillinsToFile(f, fillin_gauss_orig, fill, dim);
#if COMPARE_FILLINS
	temp = clock();
	successfulmatch = compareFillins(fill, fillin_gauss_orig, dim, vertexToOrder, processed); //compare fillins generated
by graph and by original order
	stop = clock();
	startfloat = temp / 1.0;
	stopfloat = stop / 1.0;
	timediff = (stopfloat - startfloat) / CLOCKS_PER_SEC;
	printf("Time taken for comparing fillins of GEM and fillins from running original order is: %f seconds  (%lu,
%lu)\n", timediff, temp, stop);

	printToFile(f, fillincount_gauss, fill, fillin_gauss_orig, soln_orig, successfulmatch, dim, processed,
order[processed] + 1);
#endif


#if COMPARE_FILLINS
	init initial_filled ;
#if ALLOWLEXP
	//this section fills out the graph with fillins generated by lexp and runs lexp on it again
	initial_filled = initializelexP(filename, &dim);
	filloutGraph(initial_filled.adjlist, fillinlexp_copy, dim);
	int *lexpOrderfilled = lexP(&initial_filled, dim);
	fprintf(f, "LexP ordering (when run on graph filled out by lexp fillin edges): ");
	for (i = 0; i < dim; i++)
		fprintf(f, "%d ", lexpOrderfilled[i] + 1);
	fillincount = 0;
	fillin2(initial_filled.adjlist, lexpOrderfilled, dim, &fillincount);
	fprintf(f, "\nNo of fillin edges when lexp is run on a graph filled out by lexp fillin edges = %d\n", fillincount);
	if (lexpOrderfilled != NULL)
		free(lexpOrderfilled);

	//this section fills out the graph with fillins generated by lexp and runs lexm on it again
	{
		vertex* adj_filled = initFromFile(filename, &dim);
		filloutGraph(adj_filled, fillinlexp_copy, dim);
		lexpOrderfilled = lexM(adj_filled, dim);
		fprintf(f, "LexM ordering (when run on graph filled out by lexp fillin edges): ");
		for (i = 0; i < dim; i++)
			fprintf(f, "%d ", lexpOrderfilled[i] + 1);
		fillincount = 0;
		fillin2(adj_filled, lexpOrderfilled, dim, &fillincount);
		fprintf(f, "\nNo of fillin edges when lexm is run on a graph filled out by lexp fillin edges = %d\n",
fillincount);
		if (lexpOrderfilled != NULL)
			free(lexpOrderfilled);
	}

#endif
```

```
#if ALLOWLEXM
	//this section fills out the graph with fillins generated by lexm and runs lexp on it again
	initial_filled = initializelexP(filename, &dim);
	filloutGraph(initial_filled.adjlist, fillinlexm_copy, dim);
	int *lexmOrderfilled = lexP(&initial_filled, dim);
	fprintf(f, "LexP ordering (when run on graph filled out by lexm fillin edges): ");
	for (i = 0; i < dim; i++)
		fprintf(f, "%d ", lexmOrderfilled[i] + 1);
	fillincount = 0;
	fillin2(initial_filled.adjlist, lexmOrderfilled, dim, &fillincount);
	fprintf(f, "\nNo of fillin edges when lexp is run on a graph filled out by lexm fillin edges = %d\n", fillincount);
	if (lexmOrderfilled != NULL)
		free(lexmOrderfilled);

	//this section fills out the graph with fillins generated by lexm and runs lexm on it again
	{
		vertex* adj_filled = initFromFile(filename, &dim);
		filloutGraph(adj_filled, fillinlexm_copy, dim);
		lexmOrderfilled = lexM(adj_filled, dim);
		fprintf(f, "LexM ordering (when run on graph filled out by lexm fillin edges):");
		for (i = 0; i < dim; i++)
			fprintf(f, "%d ", lexmOrderfilled[i] + 1);
		fillincount = 0;
		fillin2(adj_filled, lexmOrderfilled, dim, &fillincount);
		fprintf(f, "\nNo of fillin edges when lexm is run on a graph filled out by lexm fillin edges = %d\n",
fillincount);
		if (lexmOrderfilled != NULL)
			free(lexmOrderfilled);
	}
#endif
	//this section fills out the graph with fillins generated by original order and runs lexp on it again
	initial_filled = initializelexP(filename, &dim);
	filloutGraph(initial_filled.adjlist, fillinorig_copy, dim);
	int *origOrderfilled = lexP(&initial_filled, dim);
	fprintf(f, "LexP ordering (when run on graph filled out by original order fillin edges): ");
	for (i = 0; i < dim; i++)
		fprintf(f, "%d ", origOrderfilled[i] + 1);
	fillincount = 0;
	fillin2(initial_filled.adjlist, origOrderfilled, dim, &fillincount);
	fprintf(f, "\nNo of fillin edges when lexp is run on a graph filled out by original order fillin edges = %d\n",
fillincount);
	if (origOrderfilled != NULL)
		free(origOrderfilled);

	//this section fills out the graph with fillins generated by original order and runs lexm on it again
	{
		vertex* adj_filled = initFromFile(filename, &dim);
		filloutGraph(adj_filled, fillinorig_copy, dim);
		origOrderfilled = lexM(adj_filled, dim);
		fprintf(f, "LexM ordering (when run on graph filled out by lexm fillin edges):");
		for (i = 0; i < dim; i++)
			fprintf(f, "%d ", origOrderfilled[i] + 1);
		fillincount = 0;
		fillin2(adj_filled, origOrderfilled, dim, &fillincount);
		fprintf(f, "\nNo of fillin edges when lexm is run on a graph filled out by lexm fillin edges = %d\n",
fillincount);
	}
#endif

	stop1 = clock();
	startfloat = start / 1.0;
	stopfloat = stop1 / 1.0;
	timediff = (stopfloat - startfloat) / CLOCKS_PER_SEC;
	printf("\n\n Total time taken is: %f seconds (%lu, %lu)\n", timediff, start, stop1);
```

```
	fclose(f);
	printf("bye!");
	return 0;
}

//this function takes an adjacency list and a list of fill ins and creates a 'perfect' graph by adding the fillin edges
void filloutGraph(vertex* adjlist, fillinLL** fillin, int dim)
{
	fillinLL* temp = NULL;
	int i;

	for (i = 0; i < dim; i++)
	{
		temp = fillin[i];
		while (temp != NULL)
		{
			insertIngraph(adjlist, i, temp->varnum);
			temp = temp->next;
		}
	}
}

//makes a copy of fillin list
fillinLL** makeFillinlistCopy(fillinLL** fillin, int dim)
{
	int i;
	fillinLL* temp;
	fillinLL** copy = (fillinLL**)calloc(dim, sizeof(fillinLL*));
	for (i = 0; i < dim; i++)
	{
		temp = fillin[i];
		while (temp != NULL)
		{
			fillinLL* newnode = (fillinLL*)calloc(1, sizeof(fillinLL));
			newnode->varnum = temp->varnum;
			newnode->next = copy[i];
			copy[i] = newnode;
			temp = temp->next;
		}
	}
	return copy;
}

void printFillinsToFile(FILE* f, fillinLL** fillin_gauss, fillinLL** fill, int dim)
{
	int i;
	fillinLL* temp = NULL;
	fprintf(f, "Fillins generated by graph\n");
	for (i = 0; i < dim; i++)
	{
		temp = fill[i];
		while (temp != NULL)
		{
			if (i < temp->varnum)
				fprintf(f, "%d,%d  ", i + 1, temp->varnum + 1);
			else
				fprintf(f, "%d,%d  ", temp->varnum + 1, i + 1);
			temp = temp->next;
		}
	}
	fprintf(f, "\n");
	fprintf(f, "Fillins generated by GEM\n");
	for (i = 0; i < dim; i++)
	{
		temp = fillin_gauss[i];
```

```
            while (temp != NULL)
            {
                  fprintf(f, "%d %d %lf, ", i + 1, temp->varnum + 1, temp->val);
                  temp = temp->next;
            }
      }
      fprintf(f, "\n");
}

void printToFile(FILE* f, int fillincount_gauss, fillinLL** fillin_graph, fillinLL** fillin_gauss, double *soln, int
successfulmatch, int dim, int processed, int zerodiagonal)
{
      int i;
      fillinLL* temp;
      int gaussleftover = 0, graphleftover = 0;

      fprintf(f, "Number of non zero elements generated by GEM = %d\n", fillincount_gauss);

      if (soln != NULL)
      {
            fprintf(f, "SOLUTION:  ");
            for (i = 0; i < dim; i++)
            {
                  fprintf(f, "x%d = %lf ", i + 1, soln[i]);
            }
            fprintf(f, "\n");
      }
      else
      {
            fprintf(f, "Immature Termination:  ");
            fprintf(f, "Number of rows processed = %d, ", processed);
            fprintf(f, "Zero encountered at row number %d \n", zerodiagonal);
      }

      fprintf(f, "Number of successful matches between fillins in graphs and non zero elements Generated by GEM:
%d\n", successfulmatch);

      //left over edges
      fprintf(f, "fillin edges in graph not found in GEM: ");
      for (i = 0; i < dim; i++)
      {
            temp = fillin_graph[i];
            while (temp != NULL)
            {
                  fprintf(f, "%d,%d, ", i + 1, temp->varnum + 1);
                  graphleftover++;
                  temp = temp->next;
            }
      }
      fprintf(f, "\n");
      fprintf(f, "Number of fillin edges from graph not found in fillins from GEM: %d\n", graphleftover);

      //left over edges
      fprintf(f, "fillin edges in GEM not found in graph: ");
      for (i = 0; i < dim; i++)
      {
            temp = fillin_gauss[i];
            while (temp != NULL)
            {
                  fprintf(f, "%d %d %lf, ", i + 1, temp->varnum + 1, temp->val);
                  gaussleftover++;
                  temp = temp->next;
            }
      }
      fprintf(f, "\n");
```

```
	fprintf(f, "Number of fillin edges from graph not found in fillins from GEM: %d\n\n", gaussleftover);
}

//compares fillins generated by graph an fillins generated by GEM
int compareFillins(fillinLL **fill, fillinLL **fillin_gauss, int dim, int *vertexToOrder, int processed)
{
	int i, flag = 0;
	fillinLL *temp, *tempgauss, *temppp, *temp1;

	int delfromhead = 0;
	int successfulmatch = 0;
	for (i = 0; i < dim; i++)
	{
		temppp = fill[i];
		temp = fill[i];
		while (temp != NULL)
		{
			flag = 0;
			delfromhead = 0;
			temp1 = temp->next;
			if (vertexToOrder[temp->varnum] <= processed)
			{
				tempgauss = fillin_gauss[i];
				if (tempgauss != NULL)
				{
					flag = deleteFromLL(fillin_gauss, i, temp->varnum, &delfromhead);
					successfulmatch = successfulmatch + flag;
					if (flag == 0)
						printf("error1");
				}
				else
				{
					printf("error2\n");
				}
				flag = 0;


				tempgauss = fillin_gauss[temp->varnum];
				if (tempgauss != NULL)
				{
					flag = deleteFromLL(fillin_gauss, temp->varnum, i, &delfromhead);
					successfulmatch = successfulmatch + flag;
					if (flag == 0)
						printf("error3");
				}
				else
				{
					printf("error4\n");
				}

				deleteFromLL(fill, i, temp->varnum, &delfromhead);
			}
			if (delfromhead == 1)
				temp = fill[i];
			else
				temp = temp1;
		}
	}
	return successfulmatch;
}

//a helper function used by compareFillins. deletes elements from linked lists
int deleteFromLL(fillinLL **fill, int row, int varnum, int *delfromhead)
{
	//delete varnum from row
	fillinLL* temp = fill[row];
```

```
	fillinLL* temp1 = NULL;
	//delete from head
	if (temp == NULL)
	{
		*delfromhead = 0;
		return 0;
	}
	if (temp->varnum == varnum)
	{
		fill[row] = temp->next;
		free(temp);
		*delfromhead = 1;
		return 1;
	}
	//delete from body
	while (temp->next != NULL)
	{
		if (temp->next->varnum == varnum)
		{
			temp1 = temp->next;
			temp->next = temp->next->next;
			free(temp1);
			*delfromhead = 0;
			return 1;
		}
		temp = temp->next;
	}
	*delfromhead = 0;
	return 0;
}

//performs gaussian elimination
double* GaussianElim(LLmatrix **matrix, double* b, int dim, fillinLL **fill, int* order, double* diagarray, int*
fillincount_gauss, fillinLL **fillintrack, int* proc)//, int* vertexToOrder) //check later: delete fillin
{
	int ithrow, i, j, index = 0;
	int* processed = (int*)calloc(dim, sizeof(int));
	LLmatrix* currrow = NULL;
	double factor, diag_row = 0.0;
	fillinLL *fillinlist = NULL;
	double *soln = NULL, tempabs = 0.0;
	*proc = dim - 1;

	memcount_GEM = memcount_GEM + (dim + 4) * sizeof(int)+4 *
sizeof(double)+sizeof(fillinLL*)+sizeof(LLmatrix*);

	for (i = 0; i < dim; i++)
	{
		ithrow = order[i];
		diag_row = 0.0;
		currrow = matrix[ithrow];
		//get diagonal element of this current row
		diag_row = diagarray[ithrow];
		tempabs = diag_row > 0 ? diag_row : (-1.0 * diag_row);

		if (tempabs > 0.000001) //very small numbers are artifacts of floating point numbers, so treating them as zero
		{
			//traverse adjacency list
			currrow = matrix[ithrow];
			while (currrow != NULL)
			{
				if (currrow->col != ithrow && processed[currrow->col] == 0)  //actually we dont need
processed[currrow->col] == 0... check later
				{
					//attack matrix[currrow->col] row and make matrix[currrow->col, ithrow] element 0
					//factor = M[row2, row1] / M[row1, row1]
```

```
                        factor = currrow->value / diag_row;
                        subtractrows(matrix, ithrow, currrow->col, factor, diagarray, fillincount_gauss, fillintrack);
                        b[currrow->col] = b[currrow->col] - factor*b[ithrow];
                    }
                    currrow = currrow->next;
                }
            }
            else //diagonal element is 0, so terminate
            {
                printf("immature termination\n");
                *proc = i;  //number of rows processed
                break;
            }
            processed[ithrow] = 1;
        }

        //if all rows have been procesed, attempt to find solutions
        *proc = i;
        if (i == dim)
        {
            int *found = (int*)calloc(dim, sizeof(int)); //initially no variables are solved for
            LLmatrix* backsubstraverse = NULL;
            double sum, coeff = 0;
            soln = (double*)malloc(dim*sizeof(double));
            memcount_GEM = memcount_GEM + dim*sizeof(int)+(dim + 2)*sizeof(double)+sizeof(LLmatrix*);
            for (i = dim - 1; i >= 0; i--)
            {
                ithrow = order[i];
                backsubstraverse = matrix[ithrow];
                sum = 0.0;
                while (backsubstraverse != NULL)
                {
                    if (backsubstraverse->col != ithrow)
                    {
                        sum += soln[backsubstraverse->col] * backsubstraverse->value;
                    }
                    else
                    {
                        coeff = backsubstraverse->value;
                    }
                    backsubstraverse = backsubstraverse->next;
                }
                soln[ithrow] = (b[ithrow] - sum) / coeff;
            }
        }
        return soln;
}

//row2 = row2 - factor*row1. This function subtracts two rows when they are represented sparsely (linked list form)
void subtractrows(LLmatrix **matrix, int row1, int row2, double factor, double* diagarray, int* fillincount_gauss,
fillinLL **fillintrack)//, int* vertexToOrder)
{
    LLmatrix* rowLL1 = matrix[row1];
    LLmatrix* rowLL2 = matrix[row2];
    LLmatrix* temp = NULL, *lastrow2 = NULL;
    int freeflag;
    double tempabs = 0.0;
    fillinLL *tempfillin;

    while (rowLL1 != NULL && rowLL2 != NULL)
    {
        freeflag = 0;
        if (rowLL1->col < rowLL2->col)
        {
            //insert temp before rowLL2
```

```
        LLmatrix* temp = (LLmatrix*)malloc(sizeof(LLmatrix));
        memcount_GEM = memcount_GEM + sizeof(LLmatrix);
        temp->col = rowLL1->col;
        temp->value = -factor * rowLL1->value;
        temp->next = rowLL2;
        temp->back = rowLL2->back;
        if (temp->back != NULL)
                temp->back->next = temp;
        else
                matrix[row2] = temp;
        rowLL2->back = temp;
        tempabs = temp->value > 0 ? temp->value : (-1.0 * temp->value);

        //diagonal elements that turn non zero are not fillins (in the graph theoritic sense)
        if (row2 != rowLL1->col)
        {
                *fillincount_gauss = *fillincount_gauss + 1;
                tempfillin = (fillinLL*)malloc(sizeof(fillinLL));
                memcount_GEM = memcount_GEM + sizeof(fillinLL);
                tempfillin->varnum = rowLL1->col;
                tempfillin->next = fillintrack[row2];
                tempfillin->val = temp->value;
                fillintrack[row2] = tempfillin;
        }
        if (rowLL1->col == row2)
                diagarray[row2] = temp->value;

        rowLL1 = rowLL1->next;

        continue;
}
if (rowLL1->col > rowLL2->col)
{
        lastrow2 = rowLL2;
        rowLL2 = rowLL2->next;
        continue;
}
if (rowLL1->col == rowLL2->col)
{
        //the 2 values cancel out so we need to free a LL member as value is 0
        tempabs = rowLL1->value*factor - rowLL2->value;
        tempabs = tempabs > 0 ? tempabs : (-1.0 * tempabs);
        //if(tempabs < 0.000001)
        if (rowLL2->col == row1)
        {
                temp = rowLL2;
                if (rowLL2->back != NULL)
                        rowLL2->back->next = rowLL2->next;
                else
                        matrix[row2] = rowLL2->next;

                if (rowLL2->next != NULL)
                        rowLL2->next->back = rowLL2->back;
                freeflag = 1;

                if (rowLL2->col == row2)
                        diagarray[row2] = 0;
        }
        else
        {
                rowLL2->value = rowLL2->value - factor * rowLL1->value;
                if (rowLL2->col == row2)
                        diagarray[row2] = rowLL2->value;
        }

        lastrow2 = rowLL2;
```

```
                rowLL1 = rowLL1->next;
                rowLL2 = rowLL2->next;
                if (freeflag == 1)
                {
                        if (lastrow2->back != NULL)
                                lastrow2 = lastrow2->back;
                        else
                                lastrow2 = NULL;
                        free(temp);
                }
        }
    }
    //append rest of row1 to row2
    if (rowLL2 == NULL)
    {
        while (rowLL1 != NULL)
        {
                LLmatrix* temp = (LLmatrix*)malloc(sizeof(LLmatrix));
                memcount_GEM = memcount_GEM + sizeof(LLmatrix);

                temp->col = rowLL1->col;
                temp->value = -factor * rowLL1->value;
                tempabs = temp->value > 0 ? temp->value : (-1.0 * temp->value);

                //diagonal elements that turn non zero are not fillins (in the graph theoritic sense)
                if (row2 != rowLL1->col)
                {
                        *fillincount_gauss = *fillincount_gauss + 1;
                        tempfillin = (fillinLL*)malloc(sizeof(fillinLL));
                        memcount_GEM = memcount_GEM + sizeof(fillinLL);
                        tempfillin->varnum = rowLL1->col;
                        tempfillin->next = fillintrack[row2];
                        tempfillin->val = temp->value;
                        fillintrack[row2] = tempfillin;
                }

                temp->back = lastrow2;

                if (rowLL1->col == row2)
                        diagarray[row2] = temp->value;

                if (lastrow2 != NULL)
                        lastrow2->next = temp;
                else
                        matrix[row2] = temp;
                temp->next = NULL;
                lastrow2 = temp;
                rowLL1 = rowLL1->next;
        }
    }
}

int* getOrder(vertex* adjlist, int dim)
{
    int *vertexToOrder = (int*)malloc(dim * sizeof(int));
    int i;
    for (i = 0; i < dim; i++)
    {
        vertexToOrder[i] = (adjlist + i)->num;
    }
    return vertexToOrder;
}

//create matrix from file
LLmatrix** readFromFile(char *filename, int dim, double* diagarray)
{
```

```
	LLmatrix** mtx = (LLmatrix**)calloc(dim, sizeof(LLmatrix*));
	LLmatrix *temp = NULL;
	int ret_code, M, N, nonzero, i, x, y;
	double val;
	FILE *fmtx = fopen(filename, "r");

	memcount_GEM_inputs = memcount_GEM_inputs + dim * sizeof(LLmatrix*)+sizeof(LLmatrix*)+7 *
sizeof(int)+sizeof(double);
	if (fmtx == NULL)
	{
		printf("file not found\n");
		exit(1);
	}
	MM_typecode matcode;

	if (mm_read_banner(fmtx, &matcode) != 0)
	{
		printf("Could not process Matrix Market banner.\n");
		exit(1);
	}
	if (mm_is_complex(matcode) || mm_is_pattern(matcode))
	{
		printf("This application does not support ");
		printf("Market Market type: [%s]\n", mm_typecode_to_str(matcode));
		exit(1);
	}
	if ((ret_code = mm_read_mtx_crd_size(fmtx, &M, &N, &nonzero)) != 0)
	{
		printf("Unable to read size: exiting\n");
		exit(1);
	}

	if (M != N)
	{
		printf("Matrix not symmetric. Exiting!");
		exit(1);
	}

	for (i = 0; i < nonzero; i++)
	{
		fscanf(fmtx, "%d %d %lf\n", &x, &y, &val);

		if (mm_is_symmetric(matcode))
		{
			insertsortedLL(mtx, x - 1, y - 1, val);
			if (x != y)
				insertsortedLL(mtx, y - 1, x - 1, val);
			else
				diagarray[x - 1] = val;
		}
		else
		{
			insertsortedLLasym(mtx, x - 1, y - 1, val);
			if (x != y)
				insertsortedLLasym(mtx, y - 1, x - 1, val);
			else
				diagarray[x - 1] = val;
		}
	}
	return mtx;
}

//function for inserting into asymmetric matrices
void insertsortedLLasym(LLmatrix** mtx, int row, int col, double val)
{
```

```
	LLmatrix *traverse, *last = NULL;
	LLmatrix* temp;
	double abstemp1 = 0.0, abstemp2 = 0.0;
	int flag = 0, looped = 0;

	temp = (LLmatrix*)malloc(sizeof(LLmatrix));
	memcount_GEM_inputs = memcount_GEM_inputs + sizeof(LLmatrix);
	temp->value = val;
	temp->col = col;

	traverse = mtx[row];

	if (traverse == NULL)
	{
		mtx[row] = temp;
		temp->next = NULL;
		temp->back = NULL;
		flag = 1;
	}
	else
	{
		while (traverse != NULL)
		{
			if (traverse->col == col) //element already present, so check values and modify
			{
				abstemp1 = traverse->value > 0 ? traverse->value : (-1.0 * traverse->value);
				abstemp2 = val > 0 ? val : (-1.0 * val);
				if (abstemp1 > abstemp2)  //current value is greater than the value that was already present, so replace
				{
					traverse->value = val;
				}
				return;
			}
			if (traverse->col > col)
			{
				temp->back = traverse->back;
				if (temp->back != NULL)
					temp->back->next = temp;
				else
					mtx[row] = temp; //insert at head;
				temp->next = traverse;
				traverse->back = temp;
				flag = 1;
				break;
			}
			last = traverse;
			traverse = traverse->next;
		}
	}

	if (flag == 0) //insert at the end
	{
		last->next = temp;
		temp->back = last;
		temp->next = NULL;
	}
}

//insert into matrix structure in a sorted fashion
void insertsortedLL(LLmatrix** mtx, int row, int col, double val)
{
	LLmatrix *traverse, *last = NULL;
	LLmatrix* temp;
	int flag = 0, looped = 0;
```

```
	temp = (LLmatrix*)malloc(sizeof(LLmatrix));
	memcount_GEM_inputs = memcount_GEM_inputs + sizeof(LLmatrix);
	temp->value = val;
	temp->col = col;

	traverse = mtx[row];

	if (traverse == NULL)
	{
		mtx[row] = temp;
		temp->next = NULL;
		temp->back = NULL;
		flag = 1;
	}
	else
	{
		while (traverse != NULL)
		{
			if (traverse->col > col)
			{
				temp->back = traverse->back;
				if (temp->back != NULL)
					temp->back->next = temp;
				else
					mtx[row] = temp; //insert at head;
				temp->next = traverse;
				traverse->back = temp;
				flag = 1;
				break;
			}
			last = traverse;
			traverse = traverse->next;
		}
	}

	if (flag == 0) //insert at the end
	{
		last->next = temp;
		temp->back = last;
		temp->next = NULL;
	}
}

//print a row of the matrix
void printMtxRow(LLmatrix* m)
{
	printf("printing row:\n");
	while (m != NULL)
	{
		printf(" %d:%lf ", m->col, m->value);
		m = m->next;
	}
	printf("\n");
}

void printMtx(LLmatrix **matrix, int dim, int mode)
{
	int i, j, lastcol, flag;
	printf("printing matrix\n");
	LLmatrix *currrow;
	for (i = 0; i < dim; i++)
	{
		currrow = matrix[i];
		printf("%d:: ", i);
		lastcol = 0;
```

```
            flag = 0;
            while (currrow != NULL)
            {
                  if (mode == 1)
                  {
                        if (flag == 0)
                        {
                              for (j = 0; j < currrow->col - lastcol; j++)
                                    printf(" xx%lf", 0);
                        }
                        else
                        {
                              for (j = 0; j < currrow->col - lastcol - 1; j++)
                                    printf(" xx%lf", 0);
                        }
                  }
                  flag = 1;
                  printf(" %lf:%d", currrow->value, currrow->col);
                  lastcol = currrow->col;
                  currrow = currrow->next;
            }
            if (mode == 1)
            {
                  for (j = 0; j < dim - lastcol - 1; j++)
                        printf(" xx%lf", 0);
            }
            printf("\n");
      }
}


//frees adjacency list structure's memory
void freeadjlist(vertex* adjlist, int dim)
{
      int i;
      for (i = 0; i < dim; i++)
      {
            vertexLL* temp = adjlist[i].neighbours;
            while (temp != NULL)
            {
                  adjlist[i].neighbours = adjlist[i].neighbours->next;
                  free(temp);
                  temp = adjlist[i].neighbours;
                  memcount -= sizeof(vertexLL);
                  maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
            }
      }
      free(adjlist);
      memcount -= dim*sizeof(vertexLL);
      maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
}

//write fillin edges to file
void writeFillinToFile(FILE* f, fillinLL **fillin, int dim, int fillincount)
{
      fillinLL* temp;
      int i = 0;

      for (i = 0; i < dim; i++)
      {
            temp = fillin[i];
            while (temp != NULL)
            {
                  if (i < temp->varnum)
                        fprintf(f, "%d,%d ", i + 1, temp->varnum + 1);
```

```
                else
                        fprintf(f, "%d,%d ", temp->varnum + 1, i + 1);
                temp = temp->next;
        }
    }
    fprintf(f, "\n");
}

//read .mtx file using mmio.c and mmio.h APIs and create adjacency list
vertex* initFromFile(char *filename, int* dim)
{
    init initial;
    int ret_code, M, N, nonzero, i, x, y;
    double val;
    memcount = memcount + sizeof(init)+(7 * sizeof(int)) + sizeof(double);
    maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
    FILE *fmtx = fopen(filename, "r");
    if (fmtx == NULL)
    {
        printf("file not found\n");
        exit(1);
    }
    MM_typecode matcode;

    if (mm_read_banner(fmtx, &matcode) != 0)
    {
        printf("Could not process Matrix Market banner.\n");
        exit(1);
    }
    if (mm_is_complex(matcode) || mm_is_pattern(matcode))
    {
        printf("This application does not support ");
        printf("Market Market type: [%s]\n", mm_typecode_to_str(matcode));
        exit(1);
    }
    if ((ret_code = mm_read_mtx_crd_size(fmtx, &M, &N, &nonzero)) != 0)
    {
        printf("Unable to read size: exiting\n");
        exit(1);
    }

    if (M != N)
    {
        printf("Matrix not symmetric. Exiting!");
        exit(1);
    }

    vertex* adjlist = (vertex*)calloc(N, sizeof(vertex)); //adjacency list.
    memcount = memcount + N*sizeof(vertex*);
    maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
    for (i = 0; i < N; i++)
    {
        adjlist[i].num = 0;
        adjlist[i].varnum = i;
    }

    for (i = 0; i < nonzero; i++)
    {
        fscanf(fmtx, "%d %d %f\n", &x, &y, &val);
        //assuming val is always non-zero
        if (x != y) //ignore diagonal elements
        {
            if (mm_is_symmetric(matcode))
                insertIngraph(adjlist, x - 1, y - 1); //mtx market format starts from 1 not 0
            else
```

```
                        insertIngraphasym(adjlist, x - 1, y - 1);
                }
        }
        *dim = N;
        fclose(fmtx);
        return adjlist;
}

//create adjacency list using initFromFile and create doubly double linked list and connect the 2 structures
init initializelexP(char *filename, int *dim)
{
        headercell *eqgraph, *labelphi;
        int i, j;
        vertexcell *tempvcell;
        init initial;

        vertex* adjlist = initFromFile(filename, dim);
        eqgraph = (headercell*)calloc(1, sizeof(headercell));
        labelphi = (headercell*)calloc(1, sizeof(headercell));
        memcount = memcount + sizeof(vertex*)+2 * sizeof(int)+2 * sizeof(headercell*)+sizeof(init)+2 *
sizeof(headercell);
        maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
        eqgraph->flag = 0;
        eqgraph->back = NULL;
        eqgraph->head = labelphi;
        eqgraph->vset = NULL;
        labelphi->back = eqgraph;
        labelphi->flag = 0;
        labelphi->head = NULL;
        labelphi->vset = NULL;
        tempvcell = labelphi->vset;

        for (i = 0; i < *dim; i++) //add all vertexcells to philabel
        {
                vertexcell* v = (vertexcell*)calloc(1, sizeof(vertexcell));
                memcount = memcount + sizeof(vertexcell);
                maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
                v->vertexLocation = i;
                v->flag = labelphi;
                v->next = tempvcell;
                v->back = NULL;   //Note in paper implementation it points to headercell. But we can get to headercell using flag.
                if (tempvcell != NULL)
                        tempvcell->back = v;
                labelphi->vset = v;
                tempvcell = labelphi->vset;
        }

        //make pointer assignments from adlist's vertices to corresponding cells
        tempvcell = eqgraph->head->vset;
        for (i = *dim - 1; i >= 0; i--)
        {
                adjlist[i].vc = tempvcell;
                tempvcell = tempvcell->next;
        }

        initial.adjlist = adjlist;
        initial.eqgraph = eqgraph;
        return initial;
}

int* lexM(vertex* adjlist, int dim)
{
        int *lexmOrder = (int*)calloc(dim, sizeof(int)); //output of this function
        lexMstruct** labeltrack = (lexMstruct**)calloc(dim, sizeof(lexMstruct*));
        lexMstruct** vertexToLabel = (lexMstruct**)calloc(dim, sizeof(lexMstruct*));
```

```
	lexMstruct *temp, *tempCS;
	simpleLL* tempSimpleLL;
	int i, k, m, n;
	simpleLL** reach = (simpleLL**)calloc(dim, sizeof(simpleLL*));
	lexMstruct** countingSort = (lexMstruct**)calloc(2 * dim, sizeof(lexMstruct*));

	memcount = memcount + dim * sizeof(int)+2 * dim*sizeof(lexMstruct*)+4 *
sizeof(int)+dim*sizeof(simpleLL*)+2 * dim*sizeof(lexMstruct*);
	maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
	//initialize
	for (i = 0; i < dim; i++)
	{
		temp = (lexMstruct*)calloc(1, sizeof(lexMstruct));
		memcount = memcount + sizeof(lexMstruct);
		maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
		temp->varnum = i;
		temp->label = 0; //check
		temp->next = NULL;
		temp->reached = 0;
		labeltrack[i] = temp;
		vertexToLabel[i] = temp;
	}
	k = 0;
	for (i = dim - 1; i >= 0; i--)
	{
		//at iteration i, ith position of labeltrack contains the largest label
		//assign alpha and alpha inverse mappings
		adjlist[labeltrack[i]->varnum].num = i;
		lexmOrder[i] = labeltrack[i]->varnum;
		labeltrack[i]->reached = 1;

		//traverse adj list of the vertex.
		vertexLL* neighbours = adjlist[labeltrack[i]->varnum].neighbours;
		//update labels of immediate neighbours
		while (neighbours != NULL)
		{
			//consider only unreached vertices
			if (vertexToLabel[neighbours->v->varnum]->reached == 0)
			{
				tempSimpleLL = (simpleLL*)calloc(1, sizeof(simpleLL));
				memcount = memcount + sizeof(simpleLL);
				maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
				tempSimpleLL->v = neighbours->v;

				//add w to reach(l(w))
				//l(w) label of w is vertexToLabel[neighbours->v->varnum]->label, which is going to be even for sure
				vertexToLabel[neighbours->v->varnum]->reached = 1;

				tempSimpleLL->next = reach[(vertexToLabel[neighbours->v->varnum]->label) >> 1];
				reach[(vertexToLabel[neighbours->v->varnum]->label) >> 1] = tempSimpleLL;
				vertexToLabel[neighbours->v->varnum]->label += 1;
			}
			neighbours = neighbours->next;
		}
		simpleLL* tempreach;
		//now try to update labels of nodes that are not immediate neighbours
		for (m = 0; m <= k >> 1; m++)
		{
			tempreach = reach[m];
			//traverse through mth linked list of reach
			while (tempreach != NULL)
			{
				neighbours = tempreach->v->neighbours;
				reach[m] = reach[m]->next;

				while (neighbours != NULL)
```

```
                {
                        lexMstruct *tempLM = vertexToLabel[neighbours->v->varnum];
                        if (tempLM->reached == 0) //find an unreached node
                        {
                                tempLM->reached = 1; //mark it reached

                                tempSimpleLL = (simpleLL*)calloc(1, sizeof(simpleLL));
                                memcount = memcount + sizeof(simpleLL);
                                maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
                                tempSimpleLL->v = neighbours->v;
                                if (tempLM->label > 2 * m)
                                {
                                        //add to reach
                                        tempSimpleLL->next = reach[(vertexToLabel[neighbours->v->varnum]-
>label) >> 1];

                                        reach[(vertexToLabel[neighbours->v->varnum]->label) >> 1] =
tempSimpleLL;

                                        vertexToLabel[neighbours->v->varnum]->label += 1;
                                }
                                else
                                {
                                        //add to reach
                                        tempSimpleLL->next = reach[m];
                                        reach[m] = tempSimpleLL;
                                }
                        }

                        neighbours = neighbours->next;
                }

                free(tempreach);
                memcount -= sizeof(simpleLL);
                maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
                tempreach = reach[m];
        }
}

//sort by label and readjust labels
//place in counting sort array
for (m = 0; m <= i - 1; m++)
{
        labeltrack[m]->next = countingSort[labeltrack[m]->label];
        countingSort[labeltrack[m]->label] = labeltrack[m];
        labeltrack[m] = NULL;
}

//retrieve from counting sort array
m = 0;
n = 0;
int lb = 0, flag = 0;;
while (n <= i - 1)
{
        tempCS = countingSort[m];

        while (tempCS != NULL)
        {
                tempCS->label = 2 * lb;
                labeltrack[n] = tempCS;
                countingSort[m] = tempCS->next;
                labeltrack[n]->next = NULL;
                n++;
                tempCS = countingSort[m];
                flag = 1;
        }
        if (flag == 1)
                lb++;
```

```
                flag = 0;
                m++;
            }
            k = 2 * (lb - 1);
            for (n = 0; n < i - 1; n++)
                labeltrack[n]->reached = 0;
        }

        return lexmOrder;
}


fillinLL** fillin2(vertex* adjlist, int *lexpOrder, int dim, int *fillincount)
{
        int flg = 1;
        int *test = (int*)calloc(dim, sizeof(int)); //calloc sets it to 0 (ie false)
        int i, k, mv;
        vertex *vert, *nextElimination, *tempvert;
        vertexLL *vLL, *temp, *temp1;
        fillinLL **fillin = (fillinLL**)calloc(dim, sizeof(fillinLL**));
        fillinLL *tempfillinLL, *temp1fillinLL;
        memcount = memcount + (dim + 3)*sizeof(int)+3 * sizeof(vertex*);
        maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
        int present = 1;
        vertexLL* check = NULL;
        for (i = 0; i < dim - 1; i++)
        {
                k = dim - 1;
                vert = adjlist + lexpOrder[i];

                vLL = vert->neighbours;
                if (vLL == NULL)
                        continue;

                //for first node (wich cant be a duplicate as its only 1 node)
                if (vLL->v->num > i)
                {
                        test[vLL->v->num] = 1;
                        k = vLL->v->num;
                }
                else
                        k = dim - 1;

                while (vLL->next != NULL) //eliminate duplicates
                {
                        if (test[vLL->next->v->num] == 1) //delete duplicate
                        {
                                tempfillinLL = fillin[lexpOrder[i]];
                                temp = vLL->next;
                                vLL->next = temp->next;
                                free(temp);
                                memcount = memcount - sizeof(vertexLL);
                                maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
                                continue;
                        }
                        else
                        {
                                test[vLL->next->v->num] = 1;
                                if (vLL->next->v->num > i)
                                        k = k < vLL->next->v->num ? k : vLL->next->v->num;
                        }
                        vLL = vLL->next;
                        printf("");
                }
                if (vLL->v->num > i)
```

```
                k = k < vLL->v->num ? k : vLL->v->num;

            tempvert = adjlist + lexpOrder[k];

            mv = tempvert->varnum;
            nextElimination = adjlist + lexpOrder[k];
            vLL = vert->neighbours;
            while (vLL != NULL)
            {
                present = 0;
                temp1 = vLL->next;
                test[vLL->v->num] = 0;  //reset test[]
                //add to the neighbour that is going to be eliminated next
                if (vLL->v->varnum != mv && vLL->v->num > i)
                {
                    check = (adjlist + lexpOrder[k])->neighbours;
                    while (check != NULL)
                    {
                        if (vLL->v->varnum == check->v->varnum)
                        {
                            present = 1;
                            break;
                        }
                        check = check->next;
                    }
                    tempvert = adjlist + lexpOrder[k];
                    temp = tempvert->neighbours;
                    tempvert->neighbours = vLL;
                    vLL->next = temp;

                    if (present == 0)
                    {
                        fillinLL* newnode = (fillinLL*)calloc(1, sizeof(fillinLL));
                        memcount = memcount + sizeof(fillinLL);
                        maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
                        newnode->varnum = vLL->v->varnum;
                        newnode->next = fillin[mv];
                        fillin[mv] = newnode;
                        *fillincount += 1;
                    }
                }
                vert->neighbours = temp1;
                vLL = temp1;
            }
        }
        return fillin;
}

int* lexP(init *initial, int dim)
{
        int *lexpOrder = (int*)calloc(dim, sizeof(int));
        int i;
        headercell *eqgraph = initial->eqgraph;
        headercell *eqgraphtemp = eqgraph;
        headercell *tempheadercell;
        vertexcell *vcell, *tempcell;
        vertexLL *neighbours;
        vertexcell *tempvset;
        headercell *newlabel;
        headercell *tempheader;
        newSetLL *fixlist, *tempFL;
        memcount = memcount + 5 * sizeof(headercell*)+sizeof(vertexLL*)+sizeof(vertexcell*)+2 *
sizeof(vertexcell*)+2 * sizeof(newSetLL*);
        maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
        for (i = dim - 1; i >= 0; i--)
```

```
    {
        eqgraphtemp = eqgraph;
        while ((eqgraphtemp->head != NULL) && (eqgraphtemp->head->vset == NULL)) //this while loop deletes empty labels
        {
            tempheadercell = eqgraphtemp->head;
            if (tempheadercell->head != NULL)
            {
                eqgraphtemp->head = tempheadercell->head;
                eqgraphtemp->head->back = eqgraphtemp;
            }
            else
            {
                eqgraphtemp->head = NULL;
            }
            free(tempheadercell);
            memcount = memcount - sizeof(headercell);
            maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
        }

        vcell = eqgraph->head->vset;

        eqgraph->head->vset = vcell->next;
        if (eqgraph->head->vset != NULL)
        {
            eqgraph->head->vset->back = NULL;
        }
        initial->adjlist[vcell->vertexLocation].num = i;
        initial->adjlist[vcell->vertexLocation].vc = NULL;
        lexpOrder[i] = vcell->vertexLocation;

        fixlist = NULL;
        vertexLL *neighbours = initial->adjlist[vcell->vertexLocation].neighbours;
        while (neighbours != NULL)
        {
            tempcell = neighbours->v->vc; //locate cell corresponding to this neighbour

            if (tempcell != NULL)
            {
                //delete cell from current set:
                //delete only if it is unnumbered (but its present in double double LL oly if its unnumbered so looks safe)
                if (tempcell->back != NULL)
                {
                    tempcell->back->next = tempcell->next;
                }
                else//tempcell is the first cell in this list
                {
                    tempcell->flag->vset = tempcell->next;
                }
                if (tempcell->next != NULL)
                {
                    tempcell->next->back = tempcell->back;
                }

                //add cell to new set
                if (tempcell->flag->back->flag == 0)  //create a new label set if needed
                {
                    newlabel = (headercell*)calloc(1, sizeof(headercell));
                    memcount = memcount + sizeof(headercell);
                    maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
                    newlabel->flag = 1;
                    tempheader = tempcell->flag->back;
                    newlabel->head = tempcell->flag;
                    newlabel->back = tempheader;
                    tempheader->head = newlabel;
                    tempcell->flag->back = newlabel;
```

```
                        //add to fixlist
                        newSetLL *tempFixlistElement = (newSetLL*)calloc(1, sizeof(newSetLL));
                        memcount = memcount + sizeof(newSetLL);
                        maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
                        tempFixlistElement->hcell = newlabel;
                        tempFixlistElement->next = fixlist;
                        fixlist = tempFixlistElement;
                    }
                    tempvset = tempcell->flag->back->vset;
                    tempcell->flag->back->vset = tempcell;
                    tempcell->next = tempvset;
                    tempcell->flag = tempcell->flag->back;
                    tempcell->back = NULL;
                    if (tempcell->next != NULL)
                        tempcell->next->back = tempcell;
                }

                neighbours = neighbours->next;
            }
            free(vcell);
            memcount -= sizeof(vertexcell);
            maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;

            //update fixlist elements's flags
            tempFL = fixlist;
            while (fixlist != NULL)
            {
                tempFL = fixlist->next;
                fixlist->hcell->flag = 0;
                free(fixlist);
                fixlist = tempFL;
                memcount -= sizeof(newSetLL);
                maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;
            }
        }
        return lexpOrder;
}

void insertIngraphasym(vertex *adjlist, int i, int j)
{
        int flag = 0;
        vertexLL* temp;
        temp = adjlist[i].neighbours;
        while (temp != NULL)
        {
            if (temp->v->varnum == j) //edge has already been added, so dont reinsert
                return;
            temp = temp->next;
        }
        insertIngraph(adjlist, i, j);
}

void insertIngraph(vertex *adjlist, int i, int j)
{
        //insert at adjlist[i] and adjlist[j]
        vertexLL* temp;
        vertexLL* v1 = (vertexLL*)calloc(1, sizeof(vertexLL));
        v1->v = (adjlist + j);
        temp = adjlist[i].neighbours;
        adjlist[i].neighbours = v1;
        v1->next = temp;

        vertexLL* v2 = (vertexLL*)calloc(1, sizeof(vertexLL));
        v2->v = (adjlist + i);
        temp = adjlist[j].neighbours;
```

```
	adjlist[j].neighbours = v2;
	v2->next = temp;

	memcount += 2 * sizeof(vertexLL);
	maxmemcount = maxmemcount > memcount ? maxmemcount : memcount;

	return;
}

////////////////////////////////////////////////////////////////////////////////////////////////
//auxiliary functions (not used in main code)
void printreach(simpleLL** reach, int dim)
{
	int i;
	simpleLL *temp;
	printf("in printreach\n");
	for (i = 0; i < dim; i++)
	{
		printf("%d:  ", i);
		temp = reach[i];
		while (temp != NULL)
		{
			printf("%d ", temp->v->varnum);
			temp = temp->next;
		}
		printf("\n");
	}
	printf("exit printreach\n");
}

void printcountingsort(lexMstruct **countingSort, int dim2)
{
	int i;
	lexMstruct *temp;
	printf("printcountingsort\n");
	for (i = 0; i < dim2; i++)
	{
		temp = countingSort[i];
		printf("\n%d: ", i);
		while (temp != NULL)
		{
			printf("%d ", temp->varnum);
			temp = temp->next;
		}
	}
}
void printlabel(lexMstruct **labeltrack, int dim)
{
	int i;
	printf("printlabels\n");
	for (i = 0; i < dim; i++)
	{
		printf("%d: %d\n", labeltrack[i]->varnum, labeltrack[i]->label);
	}
	printf("\n");
}

void printreached(lexMstruct **labeltrack, int dim)
{
	int i;
	printf("printreached\n");
	for (i = 0; i < dim; i++)
	{
		printf("%d: %d %d\n", i, labeltrack[i]->reached, labeltrack[i]->label);
	}
```

```
	printf("\n");
}


void printFillin(fillinLL **fillin, int dim)
{
	fillinLL* temp;
	int i = 0;
	printf("\nenter printFillin\n");
	int count = 0;

	for (i = 0; i < dim; i++)
	{
		temp = fillin[i];
		printf("%d: ", i);
		while (temp != NULL)
		{
			printf(" %d ", temp->varnum);
			temp = temp->next;
			count++;
		}
		printf("\n");
	}
	printf("\nexit printFillin...count = %d\n", count);
}


void printGraph(headercell *eqgraph)
{
	headercell *temp = eqgraph;
	vertexcell *vtemp;
	printf("printgraph\n");
	temp = temp->head;
	printf(" wqwqwqwqwhead \n");
	while (temp != NULL)
	{
		vtemp = temp->vset;
		printf(" wqwqwqwqw ");
		while (vtemp != NULL)
		{
			printf("%d (%d)  ", vtemp->vertexLocation, vtemp->flag->flag);
			vtemp = vtemp->next;
		}
		printf("\n");
		temp = temp->head;
	}
}

void printAdjList(vertex *adjlist, int dim)
{
	int i;
	vertexLL* temp;
	printf("printAdjList\n");
	for (i = 0; i < dim; i++)
	{
		temp = adjlist[i].neighbours;
		printf("%d: ", i);
		while (temp != NULL)
		{
			printf("%d ", temp->v->varnum);
			temp = temp->next;
		}
		printf("\n");
	}
}
```

9. Screenshots showing runtimes and memory usage

```
y:~/pa2_2: ./a.out 68.mtx
Matrix: 68.mtx, Nodes: 2003
Memory used for inputs to GEM: 32048


 Time taken for reading and storing matrix for GEM is: 0.110000 seconds  (90000, 200000)

LexP gave 51926 fillin edges
immature termination
Memory used for Gaussian Elimination in lexp ordering = 16080
Time taken for running GEM on lexp ordering: 0.000000 seconds  (470000, 470000)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.010000 seconds  (570000, 580000)

LexM gave 49825 fillin edges
immature termination
Memory used for Gaussian Elimination in lexm ordering = 16080
Time taken for running GEM on lexm ordering: 0.000000 seconds  (2170000, 2170000)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.000000 seconds  (2270000, 2270000)

Original order gave 46393 fillin edges
immature termination
Memory used for Gaussian Elimination in original ordering = 16080
Time taken for running GEM on original ordering: 0.000000 seconds  (2790000, 2790000)
Time taken for comparing fillins of GEM and fillins from running original order is: 0.010000 seconds  (2880000, 2890000)


 Total time taken is: 2.980000 seconds (0, 2980000)
```

68

```
y:~/pa2_2: ./a.out 83.mtx
Matrix: 83.mtx, Nodes: 822
Memory used for inputs to GEM: 13152


 Time taken for reading and storing matrix for GEM is: 0.050000 seconds  (30000,
 80000)

LexP gave 144220 fillin edges
immature termination
Memory used for Gaussian Elimination in lexp ordering = 6632
Time taken for running GEM on lexp ordering: 0.000000 seconds  (1580000, 1580000
)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.0200
00 seconds  (1850000, 1870000)

LexM gave 139282 fillin edges
immature termination
Memory used for Gaussian Elimination in lexm ordering = 6632
Time taken for running GEM on lexm ordering: 0.000000 seconds  (4340000, 4340000
)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.0200
00 seconds  (4610000, 4630000)

Original order gave 192403 fillin edges
immature termination
Memory used for Gaussian Elimination in original ordering = 3718728
Time taken for running GEM on original ordering: 0.110000 seconds  (7340000, 7450000)
Time taken for comparing fillins of GEM and fillins from running original order is: 0.0
30000 seconds  (8270000, 8300000)


 Total time taken is: 9.150000 seconds (0, 9150000)
```

83

```
y:~/pa2_2: ./a.out 220.mtx
Matrix: 220.mtx, Nodes: 100
Memory used for inputs to GEM: 1600


 Time taken for reading and storing matrix for GEM is: 0.000000 seconds  (0, 0)

LexP gave 493 fillin edges
Memory used for Gaussian Elimination in lexp ordering = 41516
Time taken for running GEM on lexp ordering: 0.010000 seconds  (0, 10000)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.000000 seconds  (10000, 10000)

LexM gave 469 fillin edges
Memory used for Gaussian Elimination in lexm ordering = 39596
Time taken for running GEM on lexm ordering: 0.000000 seconds  (20000, 20000)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.000000 seconds  (30000, 30000)

Original order gave 458 fillin edges
Memory used for Gaussian Elimination in original ordering = 38716
Time taken for running GEM on original ordering: 0.000000 seconds  (30000, 30000)
Time taken for comparing fillins of GEM and fillins from running original order is: 0.000000 seconds  (50000, 50000)


 Total time taken is: 0.050000 seconds (0, 50000)
bye!y:~/pa2_2:
```

220

```
y:~/pa2_2: ./a.out 240.mtx
Matrix: 240.mtx, Nodes: 1000
Memory used for inputs to GEM: 16000


 Time taken for reading and storing matrix for GEM is: 0.010000 seconds  (20000, 30000)

LexP gave 21878 fillin edges
immature termination
Memory used for Gaussian Elimination in lexp ordering = 8056
Time taken for running GEM on lexp ordering: 0.000000 seconds  (80000, 80000)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.000000 seconds  (120000, 120000)

LexM gave 21783 fillin edges
immature termination
Memory used for Gaussian Elimination in lexm ordering = 8056
Time taken for running GEM on lexm ordering: 0.000000 seconds  (590000, 590000)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.000000 seconds  (630000, 630000)

Original order gave 22610 fillin edges
immature termination
Memory used for Gaussian Elimination in original ordering = 1816856
Time taken for running GEM on original ordering: 0.220000 seconds  (730000, 950000)
Time taken for comparing fillins of GEM and fillins from running original order is: 0.100000 seconds  (1320000, 1420000)


 Total time taken is: 1.430000 seconds (0, 1430000)
bye!y:~/pa2_2:
```

240

```
y:~/pa2_2: ./a.out 344.mtx
Matrix: 344.mtx, Nodes: 588
Memory used for inputs to GEM: 9408


 Time taken for reading and storing matrix for GEM is: 0.100000 seconds  (90000, 190000)

LexP gave 29850 fillin edges
Memory used for Gaussian Elimination in lexp ordering = 2399836
Time taken for running GEM on lexp ordering: 0.760000 seconds  (350000, 1110000)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.190000 seconds  (1410000, 1600000)

LexM gave 27855 fillin edges
Memory used for Gaussian Elimination in lexm ordering = 2240236
Time taken for running GEM on lexm ordering: 0.660000 seconds  (2620000, 3280000)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.170000 seconds  (3570000, 3740000)

Original order gave 26981 fillin edges
Memory used for Gaussian Elimination in original ordering = 2170316
Time taken for running GEM on original ordering: 0.620000 seconds  (4050000, 4670000)
Time taken for comparing fillins of GEM and fillins from running original order is: 0.170000 seconds  (4960000, 5130000)


 Total time taken is: 5.130000 seconds (0, 5130000)
```

344

```
y:~/pa2_2: ./a.out 876.mtx
Matrix: 876.mtx, Nodes: 306
Memory used for inputs to GEM: 4896


 Time taken for reading and storing matrix for GEM is: 0.010000 seconds  (0, 10000)

LexP gave 5864 fillin edges
Memory used for Gaussian Elimination in lexp ordering = 475316
Time taken for running GEM on lexp ordering: 0.050000 seconds  (20000, 70000)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.020000 seconds  (190000, 210000)

LexM gave 5826 fillin edges
Memory used for Gaussian Elimination in lexm ordering = 472276
Time taken for running GEM on lexm ordering: 0.060000 seconds  (310000, 370000)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.030000 seconds  (510000, 540000)

Original order gave 10939 fillin edges
Memory used for Gaussian Elimination in original ordering = 881316
Time taken for running GEM on original ordering: 0.160000 seconds  (590000, 750000)
Time taken for comparing fillins of GEM and fillins from running original order is: 0.080000 seconds  (1020000, 1100000)


 Total time taken is: 1.100000 seconds (0, 1100000)
```

876

```
y:~/pa2_2: ./a.out 889.mtx
Matrix: 889.mtx, Nodes: 9801
Memory used for inputs to GEM: 156816


 Time taken for reading and storing matrix for GEM is: 0.290000 seconds  (300000, 590000)

LexP gave 1230737 fillin edges
Memory used for Gaussian Elimination in lexp ordering = 98655056
Time taken for running GEM on lexp ordering: 45.240002 seconds  (5960000, 51200000)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 17.320000 seconds  (72320000, 89640000)

LexM gave 1230737 fillin edges
Memory used for Gaussian Elimination in lexm ordering = 98655056
Time taken for running GEM on lexm ordering: 44.930000 seconds  (203490000, 248420000)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 20.860001 seconds  (270960000, 291820000)

Original order gave 931588 fillin edges
Memory used for Gaussian Elimination in original ordering = 74723136
Time taken for running GEM on original ordering: 22.890017 seconds  (295460000, 318350000)
Time taken for comparing fillins of GEM and fillins from running original order is: 10.300000 seconds  (331340000, 341640000)


 Total time taken is: 341.679993 seconds (0, 341680000)
```

889

```
z:~/pa2_2: ./a.out 1205.mtx
Matrix: 1205.mtx, Nodes: 11445
Memory used for inputs to GEM: 183120


 Time taken for reading and storing matrix for GEM is: 1.440000 seconds  (880000, 2320000)

LexP gave 909164 fillin edges
immature termination
Memory used for Gaussian Elimination in lexp ordering = 91616
Time taken for running GEM on lexp ordering: 0.000000 seconds  (5720000, 5720000)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.100000 seconds  (7630000, 7730000)

LexM gave 909164 fillin edges
immature termination
Memory used for Gaussian Elimination in lexm ordering = 91616
Time taken for running GEM on lexm ordering: 0.010000 seconds  (242400000, 242410000)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.110000 seconds  (244310000, 244420000)

Original order gave 1308291 fillin edges
immature termination
Memory used for Gaussian Elimination in original ordering = 91616
Time taken for running GEM on original ordering: 0.000000 seconds  (254050000, 254050000)
Time taken for comparing fillins of GEM and fillins from running original order is: 0.150000 seconds  (256910000, 257060000)


 Total time taken is: 259.880005 seconds (0, 259880000)
```

1205

```
z:~/pa2_2: orig.out 1210.mtx
Matrix: 1210.mtx, Nodes: 20360
Memory used for inputs to GEM: 325760


 Time taken for reading and storing matrix for GEM is: 4.390000 seconds  (288000
0, 7270000)


Original order gave 14473485 fillin edges
immature termination
Memory used for Gaussian Elimination in original ordering = 1158041736
Time taken for running GEM on original ordering: -635.897278 seconds  (412450000
, 4071520000)


 Total time taken is: 202.622711 seconds (0, 202622704)
bye!z:~/pa2_2:
```

1210

```
z:~/pa2_2: ./a.out 1239.mtx
Matrix: 1239.mtx, Nodes: 3002
Memory used for inputs to GEM: 48032


 Time taken for reading and storing matrix for GEM is: 0.060000 seconds  (80000, 140000)

LexP gave 999 fillin edges
immature termination
Memory used for Gaussian Elimination in lexp ordering = 24096
Time taken for running GEM on lexp ordering: 0.000000 seconds  (150000, 150000)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.000000 seconds  (160000, 160000)

LexM gave 999 fillin edges
immature termination
Memory used for Gaussian Elimination in lexm ordering = 24072
Time taken for running GEM on lexm ordering: 0.010000 seconds  (7030000, 7040000)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.000000 seconds  (7050000, 7050000)

Original order gave 1997 fillin edges
immature termination
Memory used for Gaussian Elimination in original ordering = 24072
Time taken for running GEM on original ordering: 0.000000 seconds  (7150000, 7150000)
Time taken for comparing fillins of GEM and fillins from running original order is: 0.000000 seconds  (7160000, 7160000)


 Total time taken is: 7.160000 seconds (0, 7160000)
```

1239

```
z:~/pa2_2: ./a.out 1427.mtx
Matrix: 1427.mtx, Nodes: 20000
Memory used for inputs to GEM: 320000


 Time taken for reading and storing matrix for GEM is: 0.190000 seconds  (240000, 430000)

LexP gave 0 fillin edges
Memory used for Gaussian Elimination in lexp ordering = 520076
Time taken for running GEM on lexp ordering: 0.040000 seconds  (480000, 520000)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.000000 seconds  (520000, 520000)

LexM gave 0 fillin edges
Memory used for Gaussian Elimination in lexm ordering = 520076
Time taken for running GEM on lexm ordering: 0.030000 seconds  (45390000, 45420000)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.010000 seconds  (45420000, 45430000)

Original order gave 0 fillin edges
Memory used for Gaussian Elimination in original ordering = 520076
Time taken for running GEM on original ordering: 0.040000 seconds  (45900000, 45940000)
Time taken for comparing fillins of GEM and fillins from running original order is: 0.000000 seconds  (45950000, 45950000)


 Total time taken is: 46.029999 seconds (0, 46030000)
bye!z:~/pa2_2:
```

1427

```
z:~/pa2_2: ./a.out 1546.mtx
Matrix: 1546.mtx, Nodes: 4563
Memory used for inputs to GEM: 73008


 Time taken for reading and storing matrix for GEM is: 0.250000 seconds  (270000, 520000)

LexP gave 126509 fillin edges
immature termination
Memory used for Gaussian Elimination in lexp ordering = 1385840
Time taken for running GEM on lexp ordering: 0.230000 seconds  (1700000, 1930000)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.040000 seconds  (2680000, 2720000)

LexM gave 119543 fillin edges
immature termination
Memory used for Gaussian Elimination in lexm ordering = 323680
Time taken for running GEM on lexm ordering: 0.010000 seconds  (11660000, 11670000)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.010000 seconds  (12070000, 12080000)

Original order gave 546290 fillin edges
Memory used for Gaussian Elimination in original ordering = 43794536
Time taken for running GEM on original ordering: 78.809998 seconds  (21140000, 99950000)
Time taken for comparing fillins of GEM and fillins from running original order is: 23.670000 seconds  (110090000, 133760000)


 Total time taken is: 133.779999 seconds (0, 133780000)
bye!z:~/pa2_2:
```

1546

```
z:~/pa2_2: ./a.out 1553.mtx
Matrix: 1553.mtx, Nodes: 6611
Memory used for inputs to GEM: 105776


 Time taken for reading and storing matrix for GEM is: 0.470000 seconds  (420000
, 890000)

LexP gave 400862 fillin edges
immature termination
Memory used for Gaussian Elimination in lexp ordering = 2887584
Time taken for running GEM on lexp ordering: 0.370000 seconds  (6590000, 6960000
)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.0600
00 seconds  (8290000, 8350000)

LexM gave 400319 fillin edges
immature termination
Memory used for Gaussian Elimination in lexm ordering = 607824
Time taken for running GEM on lexm ordering: 0.040000 seconds  (31170000, 312100
00)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.0500
00 seconds  (32200000, 32250000)

Original order gave 1312223 fillin edges
Memory used for Gaussian Elimination in original ordering = 105110136
Time taken for running GEM on original ordering: 417.249969 seconds  (76390000,
493640000)
Time taken for comparing fillins of GEM and fillins from running original order is: 126.76998
1 seconds  (519390000, 646160000)


 Total time taken is: 646.200012 seconds (0, 646200000)
```

1553

```
z:~/pa2_2: ./a.out 2401.mtx
Matrix: 2401.mtx, Nodes: 1589
Memory used for inputs to GEM: 25424


 Time taken for reading and storing matrix for GEM is: 0.020000 seconds  (20000, 40000)

LexP gave 152 fillin edges
immature termination
Memory used for Gaussian Elimination in lexp ordering = 12768
Time taken for running GEM on lexp ordering: 0.000000 seconds  (50000, 50000)
Time taken for comparing fillins of GEM and fillins from running Lexp is: 0.000000 seconds  (50000, 50000)

LexM gave 132 fillin edges
immature termination
Memory used for Gaussian Elimination in lexm ordering = 12768
Time taken for running GEM on lexm ordering: 0.000000 seconds  (380000, 380000)
Time taken for comparing fillins of GEM and fillins from running Lexm is: 0.000000 seconds  (380000, 380000)

Original order gave 30443 fillin edges
immature termination
Memory used for Gaussian Elimination in original ordering = 12768
Time taken for running GEM on original ordering: 0.000000 seconds  (530000, 530000)
Time taken for comparing fillins of GEM and fillins from running original order is: 0.000000 seconds  (590000, 590000)


 Total time taken is: 0.650000 seconds (0, 650000)
bye!z:~/pa2_2:
```

2401