ENEE 641

PROGRAMMING ASSIGNMENT 2

Algorithmic Aspects of Vertex Elimination on Graphs-1
(Finding Perfect and Minimal Elimination Order)

Abhay Raina
Date of Submission:
26-11-14

ABSTRACT

The problem is to find the elimination order for a given system of linear equations. We convert the system of linear equations into an undirected graph and then use Lexicographic search for perfect ordering and use Fill-in algorithm to see if the graph has a perfect ordering. If the ordering is not perfect then Lexicographic search for minimal ordering is used to find a minimal order.

1. INTRODUCTION

Solving very large sets of simultaneous sparse linear equations has been one of the most fundamental problems in science and engineering. Gaussian elimination is an algorithm for solving systems of linear equations. We insert the coefficients of various equations in matrices. Gaussian elimination is the sequence of operations performed on these matrices of coefficients to get the correct solution. These operations can be performed in various sequences, but some sequences can be better than others to get the solutions. The objective of this project is to find a perfect sequence and if no such sequence exists then to find sequence with minimal number if fill-ins.

We consider the problem of solving such sets of linear equations using Gaussian elimination method. To take advantage of their sparseness, it is crucial to find a pivoting order that minimizes the number of new non-zero elements, called fill-ins, to be generated in the elimination process. A number of graph-theoretic approaches to finding such an optimal pivoting order have been introduced. Using the model proposed by Rose [1] and implement in C the perfect and optimal vertex elimination ordering algorithms developed by Rose, Tarjan, and Lueker [2]. Using such pivoting orders, we then develop an efficient C code to solve the corresponding set of linear equations using Gaussian elimination method.

We use the LexP followed by fill to check if the matrix has any perfect ordering if not then LexM is used to get a minimal order of elimination followed by fill in to get the corresponding fill in edges.

We observe theoretical analysis of the code w.r.t the space and time as given in the paper and then we use our program to find the time required for the execution of the program and the space required for the program. Later we compare these two, and find that our algorithm is working as desired.

- Section 2 describes the problem definition, Solution and Bound Analysis.
- Section 3 presents the code.
- In Section 4 we discuss the experiment and its analysis with respect to the theoretical results.
- Section 5 contains the discussions about the process of development of the algorithm.
- Section 6 has the references.

2. PROBLEM DEFINITION, SOLUTION & BOUND ANALYSES

2.1 PROBLEM DEFINITION

1. Develop efficient C codes to implement three algorithms LEX P, LEX M, and FILL. as described in the paper by Rose, Tarjan, and Lueker.

2.2 SOLUTION

N vertices are taken in as an input from the Matrix market file. So we take following as the input to the LexP function and generate the ordering this ordering is used to calculate the fill in edges. If the fill in edges are zero then what we have is a perfect ordering if not then LexM algorithm is used to find a minimal ordering and corresponding fill in edges. Finally the results are compared with the fill in edges from the original ordering.

LexP()
For this we use doubly doubly linked list as our main data structure. Vertices are mapped to this doubly doubly linked list and then eliminated one by one in reverse elimination order.

Fillin()
This doesn't require any new data structure, the given graph adjacency list and previously generated order is fed as an input to this, we just traverse the input graph in the given order to find the fill in edges and their number.

LexM()
For this we use doubly singly linked list as our main data structure. Number of distinct label values are mapped onto this structure and then is used to traverse the graph to reach other pre labelled vertices.

PSUEDOCODE:
Same psuedocode as described in the paper by Rose, Tarjan, and Lueker has been used.

Time analysis:
As described in the paper by Rose, Tarjan, and Lueker
For LexP,the algorithm takes $O(n + e)$ time. Fill-in the algorithm takes $O(n + e')$. LexM , takes $O(ne)$ time altogether.

Space Analysis:

For LexP, we map n vertices into the doubly doubly linked list and traverse e edges one by one to get the ordering this tells us that we can safely state that LexP algorithm takes $O(n + e)$ space. Fill-in, takes no additional space. In LexM we need to store n vertices and corresponding e edges hence LEX M requires O(n + e) storage space.

3. CODE:

Here is the code made for implementing the algorithm:

Program

```
#include <stdio.h>
#include"mmio.h"
#include"mmio.c"
#include<time.h>
FILE *f, *ofp;
long long int memcount = 0;

// A structure to represent an adjacency list node
struct AdjacencyListNode
{
        int dest;
        struct AdjacencyListNode* next;
        int weight;
};

// A structure to represent an adjacency liat

struct AdjList
{
        struct AdjacencyListNode* head;  // pointer to head node of list
        struct LexpCellNode* cell;
};


struct Graph
{
        int V;
        struct AdjList* array;
};

struct AdjacencyListNode* newAdjacencyListNode(int dest)
{
        struct AdjacencyListNode* newNode =
                (struct AdjacencyListNode*) malloc(sizeof(struct AdjacencyListNode));
        newNode->dest = dest;
        newNode->next = NULL;
        return newNode;
}

struct Graph* CREATEGRAPH(int V)
{
        struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
```

```
	graph->V = V;

	graph->array = (struct AdjList*) calloc(V + 1, sizeof(struct AdjList));

	int i;
	for (i = 1; i <= V; i++)
		graph->array[i].head = NULL;

	return graph;
}

void EDGEADD(struct Graph* graph, int src, int dest, int weight)
{

	if (src != dest)
	{
		struct AdjacencyListNode* newNode = newAdjacencyListNode(dest);
		newNode->next = graph->array[src].head;
		newNode->weight = weight;
		graph->array[src].head = newNode;

		newNode = newAdjacencyListNode(src);
		newNode->weight = weight;
		newNode->next = graph->array[dest].head;
		graph->array[dest].head = newNode;
	}
}

struct LexpHeadNode
{
	int flag;
	struct LexpHeadNode* head;
	struct LexpCellNode* next;  // pointer to head node of list
	struct LexpHeadNode* back;
};

struct LexpCellNode
{
	int head;
	struct LexpCellNode* next;
	struct LexpCellNode* back;
	struct LexpHeadNode* flag;
};
struct LexpCellNode* newAdjacencyListNode1(int vertex) //diff2
{
	struct LexpCellNode* newNode =
		(struct LexpCellNode*) calloc(1, sizeof(struct LexpCellNode));
```

```
	newNode->head = vertex;
	newNode->next = NULL;
	newNode->back = NULL;
	newNode->flag = NULL;
	return newNode;
}

struct LexpHeadNode* newAdjacencyListNode2()
{
	struct LexpHeadNode* newNode =
		(struct LexpHeadNode*) calloc(1, sizeof(struct LexpHeadNode));
	newNode->head = NULL;
	newNode->next = NULL;
	newNode->back = NULL;
	newNode->flag = 0;
	return newNode;
}

void Lexp(struct Graph* graph, int *alpha, int *alphainv)
{
	int i;
	int y = graph->V;
	struct AdjacencyListNode *w;
	struct LexpHeadNode* RootHeadNode = newAdjacencyListNode2();
	struct LexpHeadNode* FirstHeadNode = newAdjacencyListNode2();
	RootHeadNode->back = NULL;
	RootHeadNode->next = NULL;
	RootHeadNode->head = FirstHeadNode;
	RootHeadNode->flag = 0;
	FirstHeadNode->flag = 0;
	FirstHeadNode->head = NULL;
	FirstHeadNode->back = RootHeadNode;
	struct LexpCellNode* nextptr;
	struct LexpHeadNode* freevar;

	//Making cell nodes and putting adjacent to the first head node.
	for (i = graph->V; i > 0; i--)//diff1
	{
		if (i == graph->V)
		{
			struct LexpCellNode* NewCellNode = newAdjacencyListNode1(i);
			NewCellNode->flag = FirstHeadNode;
			FirstHeadNode->next = NewCellNode;
			graph->array[i].cell = NewCellNode;
		}
		else
		{
```

```
                struct LexpCellNode* NewCellNode = newAdjacencyListNode1(i);
                NewCellNode->flag = FirstHeadNode;
                graph->array[i].cell = NewCellNode;
                NewCellNode->back = graph->array[i + 1].cell;
                nextptr = graph->array[i + 1].cell;
                nextptr->next = NewCellNode;
        }
}


for (i = graph->V; i > 0; i--)
{
        // skipping the vertices
        FirstHeadNode = RootHeadNode->head;
        while (FirstHeadNode->next == NULL)
        {
                RootHeadNode->head = FirstHeadNode->head;
                FirstHeadNode->head->back = RootHeadNode;
                freevar = FirstHeadNode;
                FirstHeadNode = FirstHeadNode->head;
                free(freevar); //freeing variables.
        }

        //Select a new vertex p to delete from the doubly data structure.

        struct LexpCellNode* p;
        struct LexpCellNode* q;
        struct LexpHeadNode* headptr;

        p = FirstHeadNode->next;

        if (p->next != NULL)
        {
                FirstHeadNode->next = p->next;
                p->next->back = NULL;//FirstHeadNode
        }
        else
        {
                FirstHeadNode->next = NULL;
        }

        alpha[i] = p->head;
        alphainv[p->head] = i;

        //update2
        for (w = graph->array[p->head].head; w != NULL; w = w->next)
        {
```

```
                    if (alphainv[w->dest] == 0)  // check if the order has been assigned.
                    {   // delete w from the list .

                            q = graph->array[w->dest].cell;

                            if (q->back == NULL)
                            {
                                    q->flag->next = q->next;
                            }
                            else
                            {
                                    q->back->next = q->next;
                            }

                            if (q->next != NULL)
                            {
                                    q->next->back = q->back;
                            }

                            headptr = q->flag->back;

                            if (headptr->flag == 0)
                            {
                                    struct LexpHeadNode* NewHeadNode =
newAdjacencyListNode2();
                                    NewHeadNode->head = headptr->head;
                                    headptr->head = NewHeadNode;
                                    NewHeadNode->head->back = NewHeadNode;
                                    NewHeadNode->back = headptr;
                                    NewHeadNode->flag = 1;
                                    NewHeadNode->next = 0;
                                    headptr = NewHeadNode;//??
                                    // NewHeadNode = NewHeadNode->head;//??
                            }

                            // add sselected cell to the new set.

                            q->next = headptr->next;
                            q->flag = headptr;
                            q->back = NULL;


                            if (headptr->next != NULL)
                            {
                                    headptr->next->back = q;

                            }
```

```
                                headptr->next = q;
                        }
                }

                free(p);

                headptr = RootHeadNode;

                while (headptr->head != NULL)
                {
                        headptr->flag = 0;
                        headptr = headptr->head;
                }
        }
        fprintf(ofp, "\nLexP:\n");
        fprintf(ofp, "Alpha Ordering:\n");
        for (i = 1; i <= graph->V; i++)
        {
           fprintf(ofp, "%d->", alpha[i]);
     //fprintf("alphainv: %d\n", alphainv[i]);
        }
        free(RootHeadNode);
        free(FirstHeadNode);
}

int fillin(struct Graph* graph, int *alpha, int *alphainv)
{
        int m = 0;
        int x, j;
        int i = 0;
        int count = 0;
        struct AdjacencyListNode *w;
        struct AdjacencyListNode *z;
        int *test = (int *)calloc((graph->V) + 1, sizeof(int));
        int *y = (int *)calloc((graph->V) + 1, sizeof(int));
        int *elim = (int *)calloc((graph->V) + 1, sizeof(int));

        fprintf(ofp, "\nFill in Edges:\n");

        for (i = 1; i < graph->V; i++)
        {
                int k = graph->V;
                int v = alpha[i];
                //eliminate duplicates in Adj[v] and compute m[v]
                struct AdjacencyListNode *prev = graph->array[v].head;
                for (w = graph->array[v].head; w != NULL; w = w->next)
                {
```

```
                        if (elim[w->dest] == 0)
                        {
                                if (test[alphainv[w->dest]])
                                {
                                        //delete w from adj[v]
                                        y[w->dest] = 0;
                                        count--;
                                        prev->next = w->next;
                                        graph->array[w->dest].head = graph->array[w-
>dest].head->next;

                                }

                                else
                                {
                                        /*if ((y[w->dest]==1))
                                        {
                                        printf("Fill in edges: %d, %d\n", x, w->dest);
                                        }*/
                                        //chk here
                                        //k = min(k, alphainv[w->dest]);//define min function

                                        test[alphainv[w->dest]] = 1;

                                        if (k > alphainv[w->dest])
                                        {
                                                k = alphainv[w->dest];
                                        }
                                        prev = w;
                                }
                        }
                        else { prev = w; }

                }

                for (j = 1; j <= graph->V; j++)
                {
                        if (y[j] != 0)
                        {
                                //printf("Fill in edges: %d, %d\n", x, y[j]);
                                //fprintf(ofp, "Fillin Edges:");

                                if (x < y[j])
                                {
                                        fprintf(ofp, "%d,%d  ", x, y[j]);
                                }
                                else
```

```
                                {
                                        fprintf(ofp, "%d,%d  ", y[j], x);
                                }

                        }
                        y[j] = 0;
                }

                m = alpha[k];


                //add reqn fill in edges and reset test
                for (w = graph->array[v].head; w != NULL; w = w->next)
                {
                        //neighb[w->dest] = 1;
                        test[alphainv[w->dest]] = 0;
                        if (w->dest != m & (elim[w->dest] == 0))
                        {
                                x = m;
                                count = count + 1;
                                //add w to adj[m]
                                EDGEADD(graph, m, w->dest, 1);
                                y[w->dest] = w->dest;
                //printf("New Fillin edge: <%d %d> \n \n \n ", m, w->dest);//print the edge
                        }
                }
                //printGraph(graph);
                //printf("count is %d\n", count);
                elim[v] = 1;
        }
        printf("count is %d\n", count);
        fprintf(ofp, "\nNumber of Fill-ins: %d\n\n", count);
        free(test);
        free(elim);
        free(y);
        free(graph);
        return count;

}



struct Lexmcell
{
        int dest;
        struct Lexmcell* next;
};
```

```
// A structure to represent an adjacency list in the LexM

struct Lexmhead
{
	struct Lexmcell* head;  // pointer of head node in LexM
};



void LexM(struct Graph* graph, int *alpha, int *alphainv)
{

	int i, k, j, p, sel_vertex,m,temp,templ,count;
	float *L = (float *)calloc((graph->V) + 1, sizeof(float));
	int *vmap = (int *)calloc((graph->V) + 1, sizeof(int));
	int *Lint = (int *)calloc((graph->V) + 1, sizeof(int));
	int *vreach = (int *)calloc((graph->V) + 1, sizeof(int));
	struct Lexmhead* reach;
	struct AdjacencyListNode *w;
	struct AdjacencyListNode *z;
	struct Lexmcell* d;


	for (i = 1; i <= graph->V; i++)
	{
		vmap[i] = i;

	}

	for (i = 1; i <= graph->V; i++)
	{
		L[i] = 1;
		alphainv[i] = 0;
	}

	k = 1;

	//loop

	for (i = graph->V; i > 0; i--)
	{
		//select

		sel_vertex = vmap[graph->V];
		vreach[sel_vertex] = 1;
		L[sel_vertex] = 0;
```

```
alphainv[sel_vertex] = i;
alpha[i] = sel_vertex;

reach = (struct Lexmhead*)calloc(k + 1, sizeof(struct Lexmhead));

for (j = 1; j <= k; j++)
{
        reach[j].head = NULL;
}

//all vertices unnumbered

for (j = 1; j <= graph->V; j++)
{
        if (alphainv[j] == 0)
        {
                vreach[j] = 0;
        }
}

for (w = graph->array[sel_vertex].head; w != NULL; w = w->next)
{

        // add w to reach(l(w))
if (alphainv[w->dest] == 0)
        {
struct Lexmcell* node = (struct Lexmcell*) malloc(sizeof(struct Lexmcell));
                node->dest = w->dest;
                node->next = reach[(int)(L[w->dest])].head;
                reach[(int)(L[w->dest])].head = node;
                vreach[w->dest] = 1;
                L[w->dest] = L[w->dest] + 0.5;
                // Mark {v,w} as an edge of G
        }
}



for (j = 1; j <= k; j++)
{
        while (reach[j].head != NULL)
        {
                //delete a vertex w from reach(j)??
                d = reach[j].head;
                reach[j].head = reach[j].head->next;

                for (z = graph->array[d->dest].head; z != NULL; z = z->next)
```

```
                    {

        if (vreach[z->dest] != 1)
                {
                        vreach[z->dest] = 1;

                if (L[z->dest] > j)
                {
        struct Lexmcell* nodez = (struct Lexmcell*) malloc(sizeof(struct Lexmcell));
                                                nodez->dest = z->dest;
                                                nodez->next = reach[(int)(L[z->dest])].head;
                                                reach[(int)(L[z->dest])].head = nodez;
                                                L[z->dest] = L[z->dest] + (0.5);
                                                // mark {v,z} as an edge of G
        }
        else
        {
        struct Lexmcell* nodez = (struct Lexmcell*) malloc(sizeof(struct Lexmcell));
                                                nodez->dest = z->dest;
                                                nodez->next = reach[j].head;
                                                reach[j].head = nodez;
                                        }
                                }
                        }
                        free(d);
                }
        }

        for (p = 1; p <= graph->V; p++)
        {
                if (L[p] != 0)
                {
                        Lint[p] = ((10*L[p]));
                }
        }


        for ( m = 1; m <= graph->V; m++)
        {
                vmap[m] = m;
        }

        free(reach);
        int size = (graph->V) + 1;
        int large = (20 * k) + 1;
        int *semiSorted, *bsort;
        semiSorted = (int*)calloc(size, sizeof(int));
```

```
bsort = (int*)calloc(size, sizeof(int));
int significantDigit = 1;
int largestNum = large;

// Loop until we reach the largest significant digit
while (largestNum / significantDigit > 0)
{

	int bucket[10] = { 0 };

	// Counts the number of "keys" or digits that will go into each bucket
	for (p = 1; p < size; p++)
		bucket[(Lint[p] / significantDigit) % 10]++;

	/*
	* Add the count of the previous buckets,
	* Acquires the indexes after the end of each bucket location in the array
	* Works similar to the count sort algorithm
	*/
	for (p = 1; p < 10; p++)
		bucket[p] += bucket[p - 1];

	// Use the bucket to fill a "semiSorted" array
	for (p = size - 1; p > 0; p--)
	{
		int l = --bucket[(Lint[p] / significantDigit) % 10];
		semiSorted[l] = Lint[p];
		bsort[l] = vmap[p];

	}

	for (p = 0; p < size - 1; p++)
		Lint[p + 1] = semiSorted[p];

	for (p = 0; p < size - 1; p++)
		vmap[p + 1] = bsort[p];

	// Move to next significant digit
	significantDigit *= 10;


}

m = 1;

while (Lint[m] == 0)
{
```

```
                        m = m + 1;
                }

                temp = 0;
                templ = 0;
                count = 0;

                for (m; m <= graph->V; m++)
                {
                        if (Lint[m] == temp)
                        {
                                Lint[m] = count;
                        }
                        else
                        {
                                count = count + 1;
                                templ = Lint[m];
                                Lint[m] = count;
                                temp = templ;
                        }

                }

                for (m = 1; m <= graph->V; m++)
                {
                        L[vmap[m]] = Lint[m];
                }

                k = Lint[graph->V];
        }

        printf("\n");
        fprintf(ofp,"LexM:\n");
        fprintf(ofp, "Alpha Ordering:\n");
        for (i = 1; i <= graph->V; i++)
        {

                fprintf(ofp,"%d->", alpha[i]);
                //fprintf(ofp, "%d->", alpha1[i]);
        }
        printf("\n");
        free(vmap);
        free(Lint);
        free(vreach);
        free(L);

}
```

```
int main(int argc, char *argv[])
{
	clock_t t1, t2;
	t1 = clock();
	int ret_code;
	char *inFileString, *outFileString;
	MM_typecode matcode;
	inFileString = argv[1];
	outFileString = argv[2];
	//FILE *f, *ofp;
	int M, N, nz;
	int i, *I, *J;
	double *val;
	/*if (argc < 2)
	{
	fprintf(stderr, "Usage: %s [martix-market-filename]\n", argv[0]);
	exit(1);
	}
	else
	{
	if ((f = fopen(argv[1], "r")) == NULL)
	exit(1);
	}*/

	f = fopen(inFileString, "r");
	if (f == NULL)
	{
		printf("File null\n");
		exit(1);
	}
	if (f != NULL)
	{
		printf("FIle is not null\n");
	}


	if (mm_read_banner(f, &matcode) != 0)
	{
		printf("Could not process Matrix Market banner.\n");
		exit(1);
	}

	/*  This is how one can screen matrix types if their application */
	/*  only supports a subset of the Matrix Market data types.     */
```

```
if (mm_is_complex(matcode) && mm_is_matrix(matcode) &&
        mm_is_sparse(matcode))
{
        printf("Sorry, this application does not support ");
        printf("Market Market type: [%s]\n", mm_typecode_to_str(matcode));
        exit(1);
}

/* find out size of sparse matrix .... */

if ((ret_code = mm_read_mtx_crd_size(f, &M, &N, &nz)) != 0)
        exit(1);


/* reseve memory for matrices */

I = (int *)malloc(nz * sizeof(int));
memcount = memcount + (nz * sizeof(int));
J = (int *)malloc(nz * sizeof(int));
memcount = memcount + (nz * sizeof(int));
val = (double *)malloc(nz * sizeof(double));
memcount = memcount + (nz * sizeof(double));


/* NOTE: when reading in doubles, ANSI C requires the use of the "l"  */
/*   specifier as in "%lg", "%lf", "%le", otherwise errors will occur */
/*  (ANSI C X3.159-1989, Sec. 4.9.6.2, p. 136 lines 13-15)          */

for (i = 0; i<nz; i++)
{
        fscanf(f, "%d %d %lg\n", &I[i], &J[i], &val[i]);
        I[i]--;  /* adjust from 1-based to 0-based */
        J[i]--;
}

if (f != stdin) fclose(f);

/************************/
/* now write out matrix */
/************************/

mm_write_banner(stdout, matcode);
mm_write_mtx_crd_size(stdout, M, N, nz);
struct Graph* graph = CREATEGRAPH(M);
memcount = memcount + ((N + 1)*sizeof(struct AdjacencyListNode));
struct Graph* graph2 = CREATEGRAPH(M);
memcount = memcount + ((N + 1)*sizeof(struct AdjacencyListNode));
```

```
        struct Graph* graph3 = CREATEGRAPH(M);
        memcount = memcount + ((N + 1)*sizeof(struct AdjacencyListNode));
        for (i = 0; i<nz; i++)
        {
                EDGEADD(graph, I[i] + 1, J[i] + 1, val[i]);
                EDGEADD(graph2, I[i] + 1, J[i] + 1, val[i]);
                EDGEADD(graph3, I[i] + 1, J[i] + 1, val[i]);
                //fprintf(stdout, "%d %d %20.19g\n", I[i] + 1, J[i] + 1, val[i]);
        }
        //fprintf(stdout, "%d %d %20.19g\n", I[i] + 1, J[i] + 1, val[i]);
        //printGraph(graph);
        int *array1;
        int     *alpha1;
        int     *alpha2;
        alpha1 = (int *)calloc(M + 1, sizeof(int));
        memcount = memcount + ((N + 1)*sizeof(int));
        alpha2 = (int *)calloc(M + 1, sizeof(int));
        memcount = memcount + ((N + 1)*sizeof(int));
        char outputFilename[] = "major.txt";
        ofp = fopen(outFileString, "w");
        if (ofp == NULL)
        {
                printf("file is not read properly\n");
        }
        fprintf(ofp, "Name of the file is %s\n",inFileString);
        fprintf(ofp, "Size: %d * %d\nNonZero Elements: %d\n", M, N, nz);
   int count=0;
        // ordering of elimination
        memcount = memcount + (3 * (N + 1)*sizeof(int));
        Lexp(graph, alpha1, alpha2);
        memcount = memcount + ((N + 1)*sizeof(struct LexpCellNode));
        printf("\n LexP number of Fillin ");
        count=fillin(graph, alpha1, alpha2);
        if(count!=0)
        {
        LexM(graph2, alpha1, alpha2);
        memcount = memcount + ((N + 1)*sizeof(struct Lexmcell));
        printf("\n LexM number of Fillin ");
   count=fillin(graph2, alpha1, alpha2);
        }
        fprintf(ofp, "Original Ordering:\n");
        for (i = 1; i <= graph3->V; i++)
{
alpha1[i] = i;
alpha2[i] = i;
fprintf(ofp, "%d->", alpha1[i]);
}
```

```
printf("\n Original ordering number of Fillin ");
count=fillin(graph3, alpha1, alpha2);
fprintf(ofp, "\n");
fclose(ofp);
t2 = clock();
float diff = (((float)t2 - (float)t1)*0.000001);
printf("\n \n Total Execution time %f seconds \n", diff);
printf("\n \n Total Memory used is %llu Kilobytes \n", memcount/1024);
return 0;
}
```

Data Structure & Time analysis:

For LexP, we map n vertices into the doubly doubly linked list and traverse e edges one by one to get the ordering this tells us that we can safely state that LexP algorithm takes $O(n + e)$ time.

In fill-in, Whenever statement add is executed, A(v) is free of redundancies as dup eliminates such redundancies. The number of entries made to adjacency lists by one execution of add is thus bounded by A (v) (defined in G),* and the total number of additions to adjacency lists made by add over all iterations of loop is bounded by e'. The total time spent in dup over all Iterations of loop is bounded by the total number of additions made to adjacency lists and is thus $O(n + e')$ Hence total running time of add over all iterations of loop is also bounded by the total number of additions to adjacency lists and is $O(n + e')$.

In LexM , the time required per execution of statement search is O(e) since each vertex can only be marked "reached" once and thus each edge can only be examined once. Statement sort requires $O(n)$ time when implemented as a radix sort. The running time of the program is thus $O(e)$ per execution of statement loop, or $O(ne)$ time altogether. LEX M requires O(n + e) storage space.

Data Structure & Space Analysis:

For LexP, we map n vertices into the doubly doubly linked list and traverse e edges one by one to get the ordering this tells us that we can safely state that LexP algorithm takes $O(n + e)$ space. Fill-in, takes no additional space. In LexM we need to store n vertices and corresponding e edges hence LEX M requires O(n + e) storage space.

4. EXPERIMENTS AND ANALYSIS:

The experiment here was to calculate the time and space requirements theoretically and then to compare it with the practical values that we get from our program.

Table 1: File Id, File Name, Size, Non Zero Elements, Time taken and Memory used in KB.

File ID	File Name	Size	Non-zero elements	Time taken (in sec)	Memory Used (in Kbytes)
220	nos4.mtx	100 * 100	594	0.03	13
876	mesh2em5.mtx	306 * 306	2,018	0.46	42
344	bcsstm34.mtx	588 * 588	21,418	2.31	240
240	saylr3.mtx	1000 * 1000	3,750	2.78	115
2401	netscience.mtx	1589 * 1589	5,484	6.17	167
68	bcsstm13.mtx	2003 * 2003	21,181	10.16	343
1239	laser.mtx	3002 * 3002	9,000	22.7	312
1546	c-27.mtx	4563 * 4563	30,927	59.41	637
889	fv3.mtx	9801 * 9801	87,025	356.28	1522
1427	qpband.mtx	20000 * 20000	45,000	40.31	1875

The above graph contains the experimental values obtained when the c program was run first column contains the file id, second has the file name, third has the size and fourth has the memory requirements of the code in Kilobytes.

4.1 TIME CALCULATION EXPERIMENT:

For the time calculation a time counter was placed at the beginning and the end of the program to find the time required to go through the program. The same was repeated for the 10 files given and corresponding execution times were noted down. Table 1 contains the values of the same.

Also we plot the following graph using the values from Table 1.

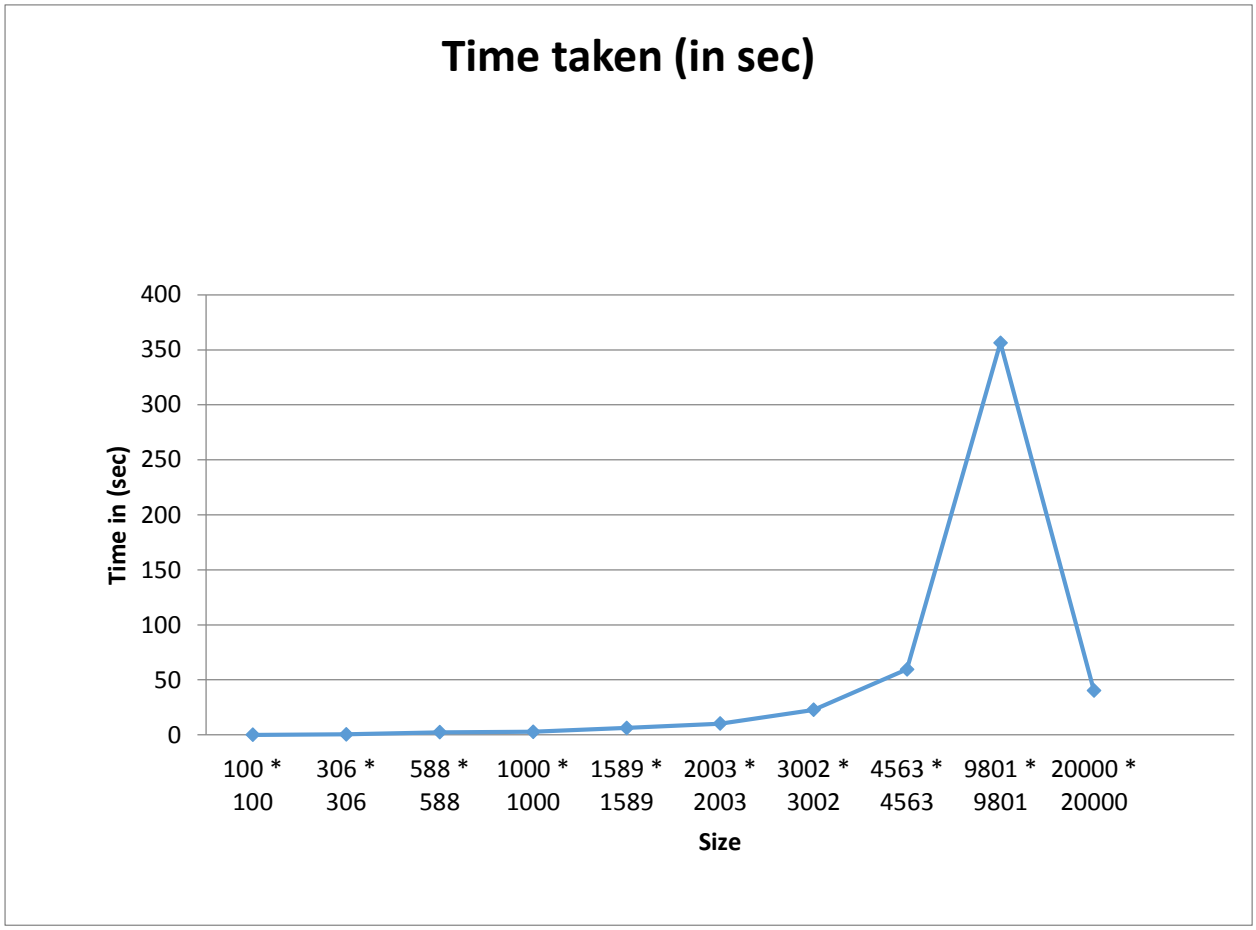

Figure 1: Graph of Execution Time vs. Input File Size

My theoretical analysis of the code showed that the runtime of the code increases as the number of vertices and the number of edges increase. When the experimental values were plotted in the graph above result was obtained. As the 9801*9801 file has 87025 edges hence this increases the runtime of the algorithm substantially.

As we can see the timing goes down for 20000*20000 file as it has a perfect order and the program exits without going to LexM. Hence from the above graph we can safely conclude that our code is working as desired from it.

4.2 SPACE CALCULATION EXPERIMENT:

The space calculation was done using counting the number of calloc() assignments in the program and finding the amount of memory that the reserve in each case and then adding them all up to get an idea of how much memory is required in general. The same was repeated for the files given and corresponding space requirements were noted. Following table contains the values of the same. As at a time we are using only 3 arrays in general, that amount has been added to the normal space requirements.

Also we plot the following graph using the above values from Table 1.

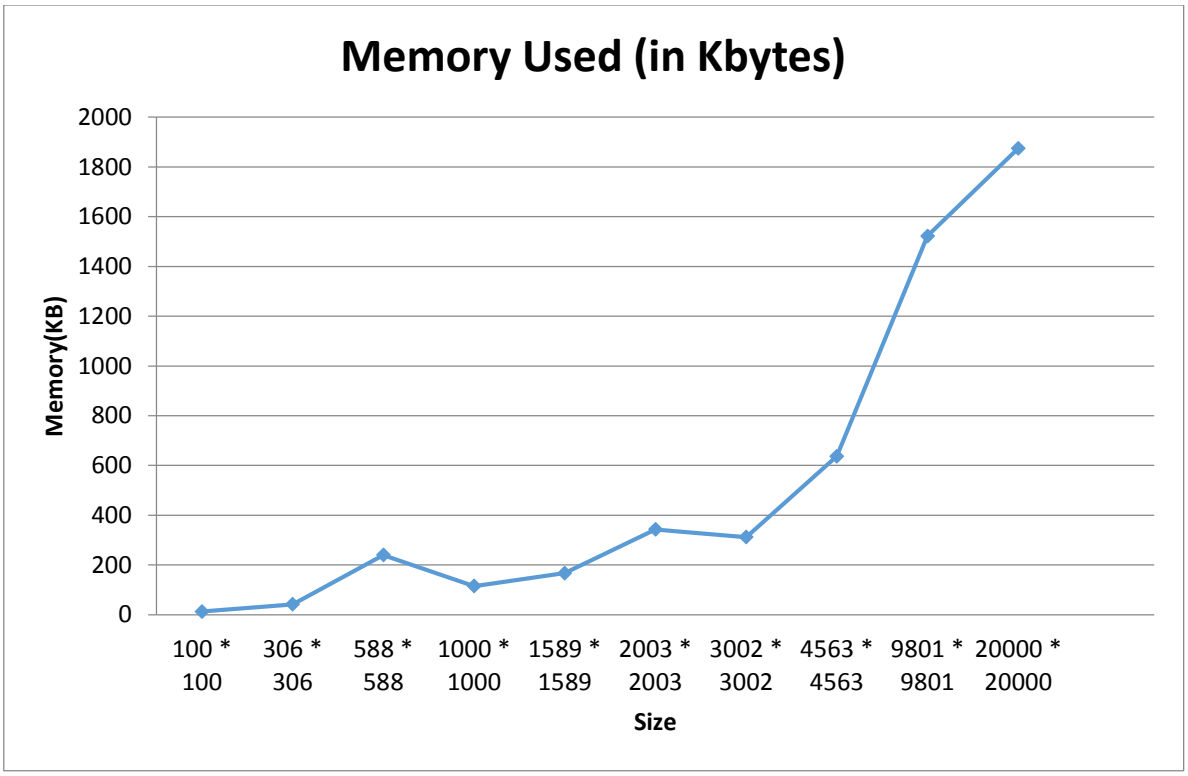

Figure 2: Graph of Memory Required Vs. Input File Size

Therefore from the above graph we can safely conclude that our code is working as desired from it i.e. the space requirements are as expected. As the input size becomes large the space requirement might serve as a constraint for the program to run and the program may need access to secondary memory.

5. DISSCUSSIONS

While designing the algorithm following problems were faced:

1. Mapping of vertices in adjacency list to doubly doubly linked list nodes.
2. Not printing the fill in edges that are duplicate.
3. To reassign the vertices correct and distinct ‘K’ values (label numbers).

The strengths of my approach would be as follows:

1. Same fill in function is used for both LexP and LexM hence no extra memory required.
2. LexM uses radix sort which is linear time, hence no extra timing burden.

Weaknesses of my approach would be as follows:

1. As the number of inputs increase the amount of memory required at that moment increases and sometimes the system stack won’t be having the memory to hold such huge amount of numbers and tries to access the secondary memory this leads to slowing of the program.

Future improvements to my approach could be to improve the memory consumption of the program.

6. REFERENCES:

1. Paper on Algorithmic aspects of vertex elimination on graphs by Donald J. Rose, R. Endre Tarjan and George s. Lueker.
2. http://www.tutorialspoint.com/c_standard_library/
3. http://en.wikipedia.org/wiki/Gaussian_elimination
4. www.wikipedia.org/wiki/**Radix_sort**
5. http://www.geeksforgeeks.org/