A Prototype for Graph Theory Conjectures

Farial Shahnaz
University of Tennessee - Knoxville

Follow this and additional works at: https://trace.tennessee.edu/utk_chanhonoproj

Recommended Citation
https://trace.tennessee.edu/utk_chanhonoproj/596
UNIVERSITY HONORS PROGRAM

SENIOR PROJECT - APPROVAL

Name: FARIAL SHAHNAZ

College: ARTS & SCIENCES
Department: COMPUTER SCIENCE

Faculty Mentor: DR. MICHAEL LANGSTON

PROJECT TITLE: A PROTOTYPE FOR GRAPH THEORY CONJECTURES

I have reviewed this completed senior honors thesis with this student and certify that it is a project commensurate with honors level undergraduate research in this field.

Signed: _________________________, Faculty Mentor

Date: 10 APR 2023

Comments (Optional):
Abstract

By Farial Shahnaz

This project involves implementation of graph algorithms in the form of computerized tests (programs) that are collected in a graph-testing library. Many real life problems can be represented by graphs, and conducting these tests on a graph results in obtaining either the actual solution, or at least information pertaining to the solution. The graph-testing library is being developed by my faculty mentor's Research group, and my part in the project includes developing and maintaining the library, devising an automated graph generator, and implementing a brute force algorithm to check for four-edge connectivity.
A Prototype for Testing Graph Theory Conjectures

Advisor: Dr. Michael Langston

Graduate Students:
Faisal Abu-Khzam
William Duncan
Melinda Stephenson

Undergraduate:
Farial Shahnaz
Danielle Smith
Ian Watkins
PROJECT OBJECTIVES

Computer theory scientists have been developing algorithms for discerning certain properties of graphs for decades. But as of yet, very few people, or groups of people have actually implemented these algorithms into computer programs. The chief objective of this project is to implement these algorithms not only in theory, but also in practice.

The project was undertaken by Dr. Michael Langston, and is being developed by his research group. My part in the project involves –

A. Helping to develop and maintain the library
B. Writing a generator for graphs
C. Implementing a brute-force algorithm for four-edge connectivity
A. DEVELOPING AND MAINTAINING THE LIBRARY:

I have written the following programs for the library:

The header files:
- g_lib.h

The source code files:
- initialize.c
- enumerate.c
- Connect.c
- k5_subgraph_test.c
- edge_removal.c
- edge_insertion.c
- vert_removal.c
- min_degree.c
- max_degree.c
- printmatrix.c
- fullmatrix.c

The code for the programs are provided in pages 9 – 20.

B. ENUMERATOR:

This is actually a single process in which I take the vertices and edges and decide where to place the edges. We are using adjacency matrices to represent graphs. But how we actually generate the graphs is using an array that represents the upper half of the matrix. Suppose we have a graph of order 5 with 6 edges:

Adjacency matrix:

| 0 1 1 1 0 |
| 1 0 0 1 0 |
| 1 0 0 1 0 |
| 1 0 1 0 1 |
| 0 1 0 0 0 |

Half matrix: [ 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 ]

Formula: vertex*(vertex-1)/2 = 5 * 4/4 = 10
We could have several graphs of order 5 that have 6 edges and the enumerator produces these graphs by using an algorithm devised by faisal.

A brief overview of the algorithm is given below:

We have two arrays vertex and range and we initialize vertex and range to the following:

Vertex [ 1 | 2 | 3 | 4 | 5 | 6 ]

Range [ 5 | 6 | 7 | 8 | 9 | 10 ]

So the half -matrix would go from

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

**B. FOUR-EDGE CONNECTIVITY:**

I have implemented a brute force algorithm to test a graph for 4-edge connectivity. My program removes all possible combination of four edges in the graph and for each case, checks to see if the graph is still connected.

The code for the program is provided in the following page.
#include "g_lib.h"

/*----------------------------------------------------------------*
Function: four_edge_connect
Description: This function takes the matrix and checks to see if it is 4 edge connected.
 *----------------------------------------------------------------*/

int stop;
int v[4];
int range[4], RM[8];
int row, col;

void max_num(int n)
{
    int i, edge=4;

    if(edge<=n) stop=0;
    else stop=1;

    for(i=0;i<edge && stop<1;i++)
    {
        range[i]=n-edge+i+1;
        v[i]=i+1;
    }

    for(i=0;i<4;i++)
    {
        printf("r[%d] = %d ",i,range[i]);
        printf("v[%d] = %d ",i,v[i]);
    }

    return;
}

void next_vector(int n)
{
    int i, j, num, edge=4;

    for( i=edge-1;i>-1;i--){
        if(v[i]<range[i]){
            v[i]=v[i]+1;
            num=v[i];

            for(j=i+1;j<edge;j++){
                num++;
                v[j]=num;
            }
        }
    }
}
if (v[0]==range[0]) stop=1;
else stop=0;
return;

}
}
stop=1;
return;

}

int get_Edge(node *h_matrix)
{

int v1,n,i,j,k,r,c,brk;
n=h_matrix->n;
v1=h_matrix->vertex;
j=0;
k=0;
i=0;
brk=1;
for(r=v1-2;r>=0;r--){
    for(c=r+1;c<v1;c++){
        if(v[j]==k){
            if(h_matrix->matrix[k]==1){
                RM[i]=r;
                RM[i+1]=c;
                i+=2;
                j++;
            }
            else return(-1);
        }
        k++;
    }
}
return(1);
}

void four_edge_connect(node *h_matrix)
{

int **F,n,r,c,i,j,k,v1,v2,temp,check,check1;
node *matrix;
int count;
n=h_matrix->n;
F=h_matrix->FM;
check=0;
k=0;
count=0;

for(i=0;i<8;i++) RM[i]=-1;
max_num(n);

while(stop<1){
    next_vector(n);
    printf("edges %d %d %d %d\n", v[0], v[1], v[2], v[3]);
    count=0;
    check1=get_Edge(h_matrix);
    printf("check1 = %d\n", check1);
    if(check1>0){
        printf("edges %d %d %d %d\n", v[0], v[1], v[2], v[3]);
        for(i=0;i<8;i+=2){
            v1=RM[i];
            v2=RM[i+1];
            printf("removing edge %d %d\n", v1, v2);
            remove_edge(h_matrix, v1, v2);
            count++;
            check=connectivity(h_matrix);
            if(check<0) break;
        }
        for(i=0;i<8;i+=2){
            v1=RM[i];
            v2=RM[i+1];
            add_edge(h_matrix, v1, v2);
        }
        if(check<0 && count==4) {
            printf("Four Edge Connected\n");
            break;
        }
    }
}

if(!(check<0 && count==4)) {
    printf("Not Four Edge Connected\n");
}
}
/* structure for an individual graph containing an array of integers called matrix that represents the upper triangular part of the adjacency matrix, an integer called vertex that represents the number of vertices, and two other integers that are required for manipulation of the UPadjacency matrix */

typedef struct {
    int *matrix;
    int vertex;
    int edge;
    int n;
    int count;
    int **FM;
} node;

FILE *fp;
Function: initialize_node  
Description: This function initializes the structure for  
the individual graphs.

```c
void initialize_node(int ii, node *Matrix)
{
    int n, i, j, k, vertx, num, row, col;

    n=(ii*(ii-1))/2;
    Matrix->n=n;
    Matrix->edge=0;
    Matrix->matrix=(int*)malloc(sizeof(int)*n);
    for(j=0; j<Matrix->n; j++) {
        Matrix->matrix[j]=0;
    }
    Matrix->count=0;
    Matrix->vertex=ii;

    Matrix->FM=(int**)malloc(sizeof(int*)*ii);
    for(i=0; i<ii; i++)
        Matrix->FM[i]=(int*)malloc(sizeof(int)*ii);

    return;
}
```
#include "g_lib.h"

Function: graph_enumerator

Description: This function generates graphs of a given order. The enumeration method used follows the following steps:

a) takes the h_matrix.count variable (which represents how many graphs have so far been generated) and

b) puts the binary representation of this variable into the matrix field of h_matrix.

int graph_enumerator(node *h_matrix, int edge, int v[edge])
{
    //int i, j, k, count;
    int count, i, j, k, vertex, num, row, col;

    h_matrix->edge=edge;
    for(i=0; i<edge; i++) {
        j=v[i]-1;
        h_matrix->matrix[j]=1;
    }

    vertex=h_matrix->vertex;
    num=h_matrix->n;
    h_matrix->FM[0][0]=0;
    i=num-1;
    /* converts the one dimensional UP to a 2 dimensionaal UP */
    for(row=0; row<vertex; row++) {
        for(col = vertex-1; col>0; col--){
            if(col > row){
                k=h_matrix->matrix[i];
                //full_matrix[row][col]=k;
                h_matrix->FM[row][col]=k;
                i--;
                if(i<0) {
                    printf("ERROR: Invalid index.\n");
                    exit(1);
                }
            }
            else full_matrix[row][col]=0;
            else h_matrix->FM[row][col]=0;
        }
    }
    /* completes the matrix by transposing the rows and columns */
    for(row=0; row<vertex; row++) {
        for(col=0; col<vertex; col++) {
/full_matrix[col][row]=full_matrix[row][col];
h_matrix->FM[col][row]=h_matrix->FM[row][col];
}
}

return 1;
/*------------------THE END -------------------*/
/*
  i=h_matrix->n;
h_matrix->edge=0;
count=h_matrix->count;
if(i<=2){
  if(count==0)
    h_matrix->matrix[0]=0;
  else if(count==1){
    h_matrix->matrix[0]=1;
    h_matrix->edge=1;
  }
}
else{
  while(1){
    if(count==0 & i>0){
      h_matrix->matrix[i-1]=count%2;
      if(count%2==1) {
        h_matrix->edge+=1;
      }
      i--;
      count=count/2;
    }
    else break;
  }
  if(h_matrix->edge>edge || h_matrix->edge<edge)
    return -1;
  else return 1;
*/
Function: connectivity

Description: This function takes the matrix and checks to see if the graph is connected

```c
int connectivity(node *h_matrix) {
    int **F;
    node matrix;
    int n,v,r,c,*d,i,j,k,col,count,check;
    memcpy(&matrix,h_matrix,sizeof(node));
    n=h_matrix->vertex;
    F=h_matrix->FM;
    check=0;
    count=0;
    j=0;
    //printf("in connect v = %d\n",n);
    d=(int*)malloc(sizeof(int)*n);
    for(i=0;i<n;i++) d[i]=-1;
    r=0;
    col=0;

    while(count<n) {
        for(c=0;c<n;c++) {
            if(F[r][c]==1) {
                printf(" r = %d , c = %d \n",r,c);
                if(d[c]<0) {
                    col=c;
                    d[c]=0;
                    count++;
                }
            }
        }
        if(count<1 || count==n-1) break;
        else {
            if(r!=col) {
                d[r]=1;
                r=col;
            }
            else {
                for(i=0;i<n;i++) {
                    if(d[i]==0) break;
                }
                if(i==n) break;
            }
            d[r]=1;
            r=1;
        }
    }
    //printf("connectivity finished \n");
    free(d);
    return count;
}
```
if the graph is not connected, return -1 */
for(i=0;i<n;i++){
    //printf("d[%d] = %d\n",i,d[i]);
    if(d[i]<0) {
        printf("graph is not connected\n");
        return(-1);
    }
}
printf("graph is connected\n");
return(1);
#include "g_lib.h"

Function: K5_subgraph_test

Description: This function takes the graph and checks to see if it has a k5 as a sub graph.

*/

void K5_subgraph(node *h_matrix)
{

int **F,n,v,a,b,c,d,e,i,j,k,temp;

n=h_matrix->vertex;
F=h_matrix->FM;

a=0;
b=1;
c=2;
d=3;
e=4;

while(1){


if(v==10){
   printf("K5 subgraph %d %d %d %d %d
",a,b,c,d,e);
   break;
}

if(e<n-1) e++;
else {
   if(d<n-2) d++;
   else {
      if(c<n-3) c++;
      else {
         if(b<n-4) b++;
         else {
            if(a<n-5) a++;
            else {
               printf("K5 not found\n");
               break;
            }
         }
      }
   }
}
}

}

#include "g_lib.h"

/*----------------------------------------------------------------
Function: remove_edge
Description: This function takes the matrix and two vertices
and deletes the edge between them.
*/
void remove_edge(node *h_matrix, int vert1, int vert2) {

    int i, j, k, n, v, r, c, num, row, col;
    if (vert2 > vert1) {
        r = vert1 - 1;
        c = vert2 - 1;
    } else {
        r = vert2 - 1;
        c = vert1 - 1;
    }
    v = h_matrix->vertex - 1;
    h_matrix->FM[vert1 - 1][vert2 - 1] = 0;
    h_matrix->FM[vert2 - 1][vert1 - 1] = 0;
    if (r % 2 == 0) k = (r / 2) * (r - 1);
    else k = r * ((r - 1) / 2);
    i = r * v - k + v - c;
    k = h_matrix->n - i - 1;
    h_matrix->matrix[k] = 0;
    return;
}
#include "g_lib.h"

/*----------------------------------------------------------------*
Function:     insert_edge
Description: This function takes the matrix and two vertices
               and inserts an edge between them.
*/

void insert_edge(node *h_matrix, int vert1, int vert2)
{
    int i, j, k, n, v, r, c, num, row, col;

    if (vert2 > vert1) {
        r = vert1 - 1;
        c = vert2 - 1;
    } else {
        r = vert2 - 1;
        c = vert1 - 1;
    }

    v = h_matrix->vertex - 1;

    h_matrix->FM[vert1 - 1][vert2 - 1] = 1;
    h_matrix->FM[vert2 - 1][vert1 - 1] = 1;

    if (r % 2 == 0) k = (r / 2) * (r - 1);
    else k = r * ((r - 1) / 2);

    i = r * v - k + v - c;
    k = h_matrix->n - i - 1;

    h_matrix->matrix[k] = 1;
    return;
}
#include "g_lib.h"

Function: remove_vertex

Description: This function takes the nxn matrix (old) and a vertex and deletes the vertex to create a new (n-1)x(n-1) matrix.

void remove_vertex(node *h_matrix, int vert, node *new)
{
    int i, j, k, n, vertx, r, c, num, row, col;
    vertx = h_matrix->vertex;
    i = num - 1;
    initialize_node(vertx - 1, new);

    //creating the n-1xn-1 matrix
    for (row = 0, r = 0; row < vertx; row++, r++) {
        num = 0;
        if (row != vert) {
            for (col = 0, c = 0; col < vertx; col++, c++) {
                if (col != vert && num < 1)
                    new->FM[r][c] = h_matrix->FM[row][col];
                else if (col == vert && num < 1) {
                    num = 1;
                    c--;
                } else if (col != vert && num > 0)
                    new->FM[r][c] = h_matrix->FM[row][col];
            }
        } else r--;
    }

    //initializing the half_matrix array.
    i = new->n - 1;
    num = new->vertex - 1;
    for (r = 0; r < num; r++) {
        for (c = num; c > r; c--) {
            new->matrix[i] = new->FM[r][c];
            i--;
        }
    }
    return;
}
# include "g_lib.h"

/*----------------------------------*--------------------------------------*/

Function:   min_deg

Description: This function takes the matrix and finds the min degree

*----------------------------------*--------------------------------------*/

int min_deg(node *h_matrix)
{
    int i,j,k,n,v,r,c,temp,min;
    min=h_matrix->vertex;
    for (r=0;r<h_matrix->vertex;r++) {
        temp=0;
        for (c=0;c<h_matrix->vertex;c++) {
            temp+=h_matrix->FM[r][c];
            if (temp<min) min=temp;
        }
    return(min);
    }

# include "g_lib.h"

/*----------------------------------------------------------------*/

Function:   max_deg

Description: This function takes the matrix and finds the max degree.

/*----------------------------------------------------------------*/

int max_deg(node *h_matrix)
{
    int i,j,k,n,v,r,c,temp,max;
    max=0;
    for (r=0;r<h_matrix->vertex;r++) {
        temp=0;
        for (c=0;c<h_matrix->vertex;c++) {
            temp+=h_matrix->FM[r][c];
            if (temp>max) max=temp;
        }
    return(max);
    }
#include "g_lib.h"

/*----------------------------------------------------------------*
Function: print_matrix
Description: This function prints the given matrix to screen.
*/

void print_matrix(node *h_matrix)
{
    int i, j, k, row, col, vertx;
    int full_matrix[h_matrix->vertex][h_matrix->vertex];
    vertx = h_matrix->vertex;

    for (row = 0; row < vertx; row++) {
        printf("[ ");
        for (col = 0; col < vertx; col++) {
            printf("%d ", h_matrix->FM[row][col]);
        }
        printf("] 
");
    }
    printf("\n");
    return;
}
#include "g_lib.h"

**Function: initialize_fullmatrix**

**Description:** This function takes the matrix field of h_matrix containing UPadjacency elements and manipulates it to produce the completed nxn matrix.

```c
void initialize_fullmatrix(node *h_matrix)
{
    int i, j, k, vertx, num, row, col;
    vertx=h_matrix->vertex;
    num=h_matrix->n;
    h_matrix->FM=(int**)malloc(sizeof(int*)*vertx);
    for(i=0;i<vertx;i++)
        h_matrix->FM[i]=(int*)malloc(sizeof(int)*vertx);
    h_matrix->FM[0][0]=0;
    i=num-1;
    /* converts the one dimensional UP to a 2 dimensionaal UP */
    for(row=0; row<vertx; row++)
    {
        for(col = vertx-1; col>0; col--){
            if(col > row){
                k=h_matrix->matrix[i];
                h_matrix->FM[row][col]=k;
                i--;
                if(i<1) {
                    printf("ERROR: Invalid index. \n");
                    exit(1);
                }
            }
        }
    }
    h_matrix->FM[row][col]=0;
}
/* completes the matrix by transposing the rows and columns */
for(row=0; row<vertx; row++)
{
    for(col=0; col<vertx ; col++){
        h_matrix->FM[col][row]=h_matrix->FM[row][col];
    }
}
return;
}```