NANYANG
TECHNOLOGICAL
UNIVERSITY
SINGAPORE

School of Electrical & Electronic Engineering

EE2008 Data Structures and Algorithms

Academic Year 2017-2018

L2008B

Algorithmic Approach to Problem Solving

Electronics II (S2-B4a-04)

Laboratory Manual

1. Objectives

This experiment aims to provide students with hands-on training in order that they are able to gain a better understanding on a range of problem solving strategies (algorithms) and their application in the solution of problems by computer. In particular, given a problem scenario, students are required to write a program to implement an appropriate problem solving strategy to solve the problem.

2. Introduction

Informally, an *algorithm* is a step-by-step method of solving some problem. Such an approach to problem-solving is not new; indeed the word "algorithm" derives from the name of the ninth-century Persian mathematician *al-Khowārizmī*. Today "algorithm" typically refers to a solution that can be executed on a computer.

2.1 Problem Solving using Graph Algorithms

A great variety of problems can be naturally formulated in terms of objects and connections between them. One example is that of electric circuits where interconnections between circuit elements, such as transistors, resistors and capacitors, play a central role. Such circuits can be represented and processed within a computer in order to answer questions such as "Does there exist a connection between a given pair of elements?" or "If a circuit is built, will it work?". Here, the answer to the first question depends only on the properties of the interconnection (wires), while the answer to second question requires detailed information about both the wires and the objects that they connect.

A graph is a mathematical object that accurately models such situations. In particular, a graph *G(V,E)* is a collection of vertices *V* and edges *E*. Vertices are simple objects that can have names and other properties; an edge is a connection between two objects. An example of a graph *G* with its set of vertices *V* and edges *E* is shown in Figure 1.

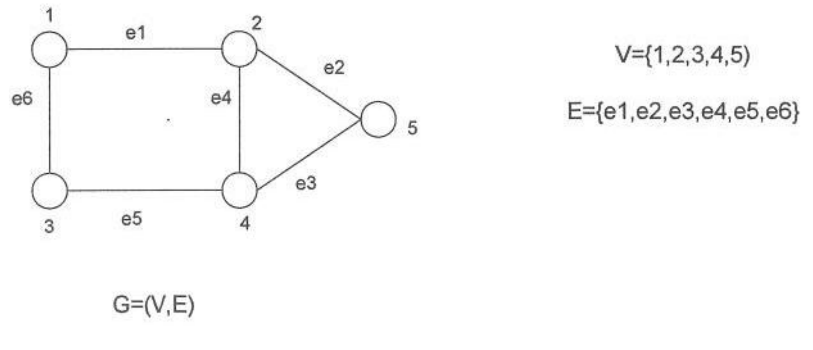


Figure1. An Example of a Graph

2.2 Graph Representation

Given a pair of vertices *u,v* in a graph *G(V,E)*, we say that *u* is *adjacent* to *v* (and vice versa) if there is an edge connecting *u* and *v* in *G*, i.e. $(u,v) \in E$. One way to represent a graph *G* is to use an *adjacency matrix* where an entry in the matrix in row *i*, column *j*, is 1 if *(i,j)* is an edge in *G*, and 0 otherwise. For example, the graph in Figure 1 may be represented using the following adjacency matrix:

$$\begin{array}{c}1\\2\\3\\4\\5\end{array}\begin{pmatrix}0&1&1&0&0\\1&0&0&1&1\\1&0&0&1&0\\0&1&1&0&1\\0&1&0&1&0\end{pmatrix}$$

Figure 2. Adjacency Matrix of Graph in Figure 1

2.3Paths & Cycles

If we think of a graph as modelling streets and cities, a path corresponds to a trip beginning at some city, passing through some cities, and terminating at some city. Formally, a *path* from vertex *x* to vertex *y* in a graph is a list of vertices in which successive vertices are connected by edges in the graph. For example, 1,2,5 is a path from vertex 1 to vertex 5 for the graph in Figure 1. A path that has same starting vertex and ending vertex (and no other repeated vertices) is called a *cycle*. For example, 1,2,4,3,1 is a cycle in the graph shown in Figure 1.

2.4 Connected Graphs & Trees

A graph is *connected* if there is a path from every vertex to every other vertex in the graph. For example, the graph shown in Figure 1 is a connected graph. A connected graph with no cycles is called a *tree*. A *spanning tree* of a graph *G* is a subgraph of *G* which forms a tree and contains all vertices in *G*. For example, the tree shown in Figure 3 is a spanning tree for the graph in Figure 1.

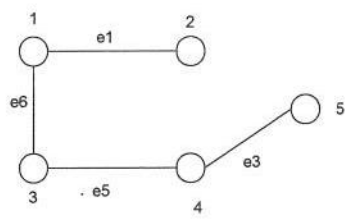


Figure 3. A spanning tree for the graph in Figure 1

A graph which is not connected is made up of *connected components*. For example the graph in Figure 4 has three connected components.

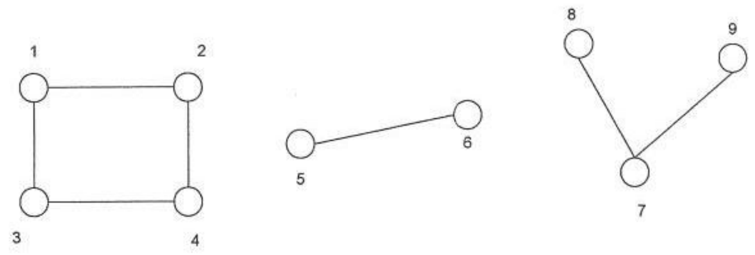


Figure 4. An example of a graph with 3 connected components

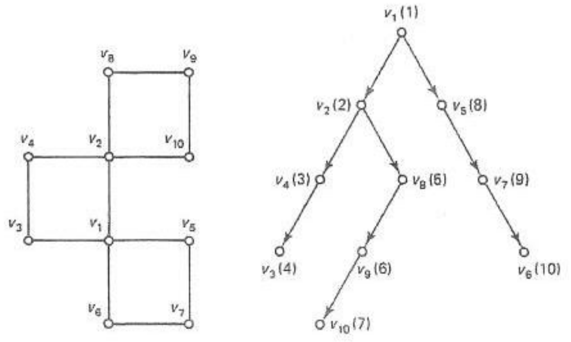
3. Testing for Graph Connectivity

Given a graph *G(V,E)*, one common question that one may asked is whether the graph is connected. The issue of graph connectivity may be addressed by using the *Depth-First Search (DFS) Algorithm*.

3.1 Depth-First Search Algorithm

Depth-first search starts visiting vertices of a graph at an arbitrary vertex *v* by marking it as having been *visited*. Next, if there is a node adjacent to *v* that has not been visited, choose this node as a new starting point and call the depth-first search procedure recursively. On return from the recursive call, if there is another vertex adjacent to *v* that has not been visited, choose this node as the next starting point, call the procedure recursively once again, and so on. When all vertices adjacent to *v* has been marked, the search starting at *v* is finished. By then all vertices in the same connected component as the starting vertex *v* has been visited. If unvisited vertices still remain (it would then imply that the given graph is not connected), the depth-first search procedure may be restarted at any one of them.

An illustration of the DFS algorithm is shown in Figure 5.



(a) Example graph G

(b) DFS tree for graph G (depth-first search label of each vertex is given in parenthesis)

Figure 5. An illustration of Depth-First Search

A pseudocode of the depth-first search algorithm is as follows.

```
Algorithm DFS(G)
Input:   Graph G=(V,E)
Output:  Graph G with its vertices marked with consecutive integers in the
             order they've been encountered by DFS traversal and the number of connected components

count ← 0
mark each vertex with 0 (unvisited)
for each vertex v∈ V do
    if v is marked with 0 then
        component ← component + 1
        dfs(v)

dfs(v)
/* visits recursively all unvisited vertices connected to v and assigns them the numbers in the order they are
encountered via global variable count   */

count ← count + 1
mark v with count
for each vertex w adjacent to v do
   if w is marked with 0 then
      dfs(w)
```

Note: 1. The variable *count* gives the sequence number of which a node is visited
2. The variable *component* gives the number of connected components in the graph. If component = 1, the graph is connected; else there are more than one disconnected components.

3.2 Implementation Requirements

(i) Write a C program to implement an algorithm based on Depth-First Search to test the connectivity of a given network.

Input: Adjacency matrix (refer to sample program on page 6 for reading in an adjacency matrix)

Output:

- The sequence that vertices are being visited
- The vertices of each connected component of a given graph
- The parent of each vertex (a vertex *p* is the parent of vertex *c* if *c* is visited from *p*)

Note:
To mark a vertex to indicate if has been visited or not, a one-dimensional integer array, say *visit*[*maxV*], where *maxV* is the size of the array, could be used. *visit*[*i*] is set to 0 initially for all *i* (unvisited), and is set to 1 if visited.

(ii) Use your program to test the following network.

- A graph *G(V,E)* with the following adjacency matrix:

$$\begin{array}{c|ccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
3 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
6 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
7 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}$$

4. Additional Implementation Requirements

One of the following parts will be selected for the student to implement during their lab session.

a) Given a depth-first search tree T, the set of edges in T are referred to as "tree edges" while those not in T are referred to as "back edges". Modify the implementation of the Depth-First Search algorithm to print out the set of tree edges and the set of back edges for the following graph.

$$\begin{array}{c|ccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
3 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
6 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
7 & 0 & 0 & 1 & 0 & 1 & 1 & 0
\end{array}$$

b) One application of the depth-first search is for checking acyclicity of a graph. If there is a back edge (an edge not in the DFS tree T) from some vertex u to its ancestor v in T, the graph has a cycle that comprises the path from v to u via a sequence of tree edges in T followed by the back edge from u to v. Modify your program to determine if there is a cycle in each connected components of a given graph. Test your program with the graph given in part(a) above.

A Sample Program

```
#include <stdio.h>                /* Include header file for printf */
#define maxV 100                  /* Define a constant */

int V, E, x, y;                   /* Declare variables */
int a[maxV][maxV];                /* Declare a two dimensional array */
int count = 0;                        /* Declare and initialize variable */

void read_graph(void);            /* Function prototype */
void print_graph(void);

void main()                       /* Main program. */
{
   read_graph();                  /* call function to input graph adjacency matrix */
   print_graph();                 /* call function to output graph adjacency matrix */

}


 void read_graph( void )          /* Function to read graph adjacency matrix */
   {
        int edge, x;
        printf("\nInput number of vertices :");
        scanf("%d", &V);
        if (V > maxV)
               printf("Exceed the maximum number of vertices permitted");
        else
        {
          for (x=1; x <= V; x++)
              for (y=1; y <= V; y++)
                 a[x][y] = 0;

          for (x=1; x <= V; x++)
              for (y=x; y <= V; y++)
              {
                     printf("\na[ %d ][ %d ]=", x, y);
                     scanf("%d", &edge);
                     a[x][y] = edge;
                     a[y][x] = edge;
              }
          }

        }


        void print_graph(void)            /* Function to print graph adjacency matrix */
          {
            int x,y;
            for (x=1; x <= V; x++)
               for (y=1; y <= V; y++)
                    printf("a[ %d ][  %d ]= %d", x, y, a[x][y]);
          }
```

Some Explanatory Notes on the Sample Program

- Commands intended for the C preprocessor, instead of the C compiler itself, start with a "#" and are known as "preprocessor directives" or "metacommands". The sample program has two such metacommands:

  ```
  #include <stdio.h>
  #define maxV 100
  ```

  The first statement, "#include <stdio.h>", simply merges the contents of the file "stdio.h" into the current program file before compilation. The "stdio.h" file contains declarations required for use of the standard-I/O library, which provides the "printf()" and scanf() functions.

- Any part of the program that starts with /* and ends with */ is called a *comment*. The purpose of using comments is to provide supplementary information to make it easier to understand the program.

- A C program is built up of one or more functions. The program above contains three user-defined functions, "main()", "read_graph()" and "print_graph()".

- The "main()" function is mandatory when writing a self-contained program. It defines the function that is automatically executed when the program is run. All other functions will be directly or indirectly called by "main()".

- You call a C function simply by specifying its name, with any arguments enclosed in following parentheses, with commas separating the arguments.

- All variables in a C program must be "declared" by specifying their name and type. The sample program declares two variables for the "read_graph()" routine:

  ```
  int edge, x;
  ```

  The above statement declares "edge" and "x" as integer variables.

- You'll notice that besides the variable declarations, there is also a function declaration, or "function prototype", that allows the C compiler to perform checks on calls to the function in the program and flag an error if they are not correct. The sample program declares the following two function prototypes:

  ```
  void read_graph(void);
   void print_graph(void);
  ```

  The function prototypes declare the type of value the function returns (the type will be "void" if it does not return a value), and the arguments that are to be provided with the function.

- The "printf()" library function provides text output capabilities for the program. For example, the statement

  ```
  printf("\nInput number of vertices :")
  ```

  display the text : Input number of vertices:

  The printf() statement doesn't automatically add a "newline" to allow the following printf() statement to print on the next display line. You must add a newline character ("\n") to force a newline.

- You can also include "format codes" in the string and then follow the string with one or more variables to print the values they contain. For example, the statement

  ```
  printf("\na[ %d ][ %d ]=", x, y)
  ```

  would print something like a[1][2] (i.e. assume that x=1 and y=2). The "%d" is the format code that tells "printf" to print a decimal integer (i.e. we assume that both x and y have been declared as integer variables).

- The scanf() function reads data from the keyboard according to a specified format and assigns the input data to one or more program variables. Like printf(), scanf() use a format string to describe the format of the input. For example, the statement

  ```
  scanf("%d", &V)
  ```

  reads a decimal integer from the keyboard and assigns it to the integer variable V.