Algorithm Lab With C

in Linux

Graph Traversal: BFS, DFS

Anupam Pattanayak[1]

M.C.A., M. Tech.

Assistant Professor,

Department of Computer Science,

Raja N. L. Khan Women's College (Autonomous),

Midnapore, West Bengal - 721102

April 15, 2020

[1]anupam.pk@gmail.com

Contents

1

Graph Traversal Algorithms

Hopefully, you have implemeted the previously given programs on quick sort, randomized quick sort, and heap sort without any difficulty. Now, we will see implementation of graph traversal algorithms.

Graph is a very important data structure. Many real life scenarios are best described by graph data structure. You must have studied graph in data structure course. Remember, a graph G is defined by $G\,(V, E)$ where V is the set of vertices and E is the set of edges. Consider the following graph in figure 1.1. Here, $V = \{A, B, C, D, E, F, G, H\}$, and $E =$

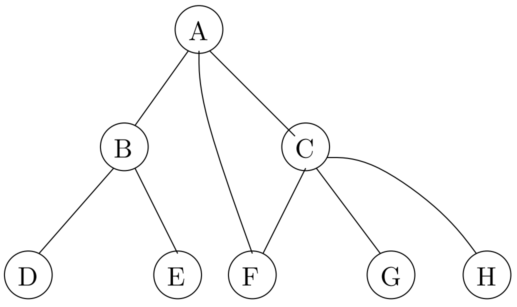

Figure 1.1: A Graph

$\{AB, AC, AF, BD, BE, CF, CG, CH\}$. Here the edges are bidirectional. Two vertices X and Y are said to be *adjacent* if there is an edge between vertices X and Y.

1.1 Graph Representation

For programming with graphs, we need to use standard representations of graph. In general, a graph can be represented in two ways:

I. Adjacency Matrix representation

II. Adjacency List representation

To represent, $G(V, E)$ using adjacency matrix representation, a binary matrix M of order $|V| \times |V|$ is used.

$$M_{i,j} = \begin{cases} 1, & \text{if there is an edge between vertx } i \text{ and vertex } j \\ 0, & \text{otherwise} \end{cases}$$

Adjacency list representation uses linked lists to represent a graph. Each of these representations has it's own advatages and disadvatages. In general, if the graph is *sparse*, then adjacency list is prefered. Also, if the graph is moderately huge, then adjacency list is preferable. Whereas adjacency matrix is better choice if the graph is *dense*. Please refer any standard text book on data structure for more on this important topic: graph representation. Choice of proper graph representation has a huge impact on the performance of implementation of graph based applications.

We will use adjacency matrix representation as it is simple to implement. For the graph 1.1, we have 8 vertices, and the corresponding adjacency matrix $M_{8\times 8}$ is shown below.

$$M_{8\times 8} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Note that it's *symmetric matrix.* If the graph is undirected graph, then we will get the adjacency matrix as symmetric. If the given graph is weighted graph, then we replace the 1's with the weights of the edges.

1.2 Depth First Search

Depth First Search (DFS), and Breadth First Search (BFS) are the graph traversal algorithms. These two algorithms are widely used in many applications. To traverse a graph, we need to choose a starting vertex or sorce vertex. We begin traversing graph nodes from this source node and continue traversing the remaining nodes till all the vertices are visited. In DFS, the vertex nodes are visited depth-wise. The *stack* data structure is used in DFS. Whenever a new vertex is visited, it's adjacent vertices are *pushed* onto the stack. Next node to be visited is *popped* from the stack. This continues untill the stack becomes empty.

While implementing the DFS, we will input the number of vertices and adjacency matrix of the graph. Then, we will also give the starting vertex for traversal. It will be helpful if we provide the adjacency matrix input as redirected input instead of typing the matrix elements from keyboard every time we run the program. Following program shows the C program for DFS.

```c
/* C program for DFS, dfs1.c */

#include<stdio.h>

int a[15][15];
int stack[15], top=-1;
int visited[15];

void dfs(int, int);
void push(int);
int pop();

int main() {
 int n,src;
 int i,j;

 printf("\n Enter Number of Vertices in Graph: ");
 scanf("%d",&n);
 printf("\n Enter Adjacency Matrix of the Graph: ");
 for(i=0;i<n;i++)
  for(j=0;j<n;j++)
   scanf("%d",&a[i][j]);

 printf("\nn=%d ",n);
 printf("\n Adjacency Matrix: ");
 for(i=0;i<n;i++)  {
  printf("\n");
  for(j=0;j<n;j++)
   printf("%5d",a[i][j]);
 }

 for(i=0;i<n;i++)
   visited[i]=0;

 printf("\n Enter The Source Vertex, in terms of index: \n");
 scanf("%d",&src);

 printf("\n node visit order: ");
 dfs(src,n);     /* call dfs() */

 return 0;
}



void dfs(int src,int n) {      /* DFS */
 int i,k;

 push(src);
 visited[src]=1;
 k=pop();
 if(k!=-1)
  printf(" %d ",k);
 while(k!=-1) {
  for(i=0;i<n;i++)
    if((a[k][i]!=0)&&(visited[i]==0)) {
      push(i);
      visited[i]=1;
    }
  k=pop();
  if(k!=-1)
    printf(" %d ",k);
 }

 for(i=0;i<n;i++)
   if(visited[i]==0)
     dfs(i,n);
}

void push(int num) {   /* push into stack */
 if(top==14)
   printf("Stack overflow ");
 else
```

```
   stack[++top]=num;
}

int pop() { /* pop from stack */
 int k;
 if(top==-1)
   return -1;
 else {
   k=stack[top--];
   return k;
 }
}
```

Hopefully, you have typed the program correctly and now it is the time for compilation and execution.

Before that, few words about input. It is really a test of patience to enter the adjacency matrix from keyboard manually everytime we execute the program. Particularly, it's very annoying when we are not getting correct output and have to run several times while debugging. So, we will redirect the input from a text file. We store all the inputs in a text file named *graph_ip1.txt* as follows.

```
8
0 1 1 0 0 1 0 0
1 0 0 1 1 0 0 0
1 0 0 0 0 1 1 1
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0
0
```

First input is the number of vertices in input graph. Next is the 8×8 adjacency matrix of graph of figure 1.1. Last number is the source vertex from where graph traversal is to begin. The output is shown next.

Now, we compile and execute the program. A sample output corresponding to the graph shown in figure 1.1 is given next.

```
$ gcc bfs1.c
$ ./a.out < graph_ip1.txt

 Enter Number of Vertices in Graph:
 Enter Adjacency Matrix of the Graph:
n=8
 Adjacency Matrix:
    0    1    1    0    0    1    0    0
    1    0    0    1    1    0    0    0
    1    0    0    0    0    1    1    1
    0    1    0    0    0    0    0    0
    0    1    0    0    0    0    0    0
    1    0    1    0    0    0    0    0
    0    0    1    0    0    0    0    0
    0    0    1    0    0    0    0    0
 Enter The Source Vertex, in terms of index:

 node visit order:  0  5  2  7  6  1  4  3   $
```

Check if the output is correct.

To run this program on another graph, consider the following directed graph in figure 1.2.

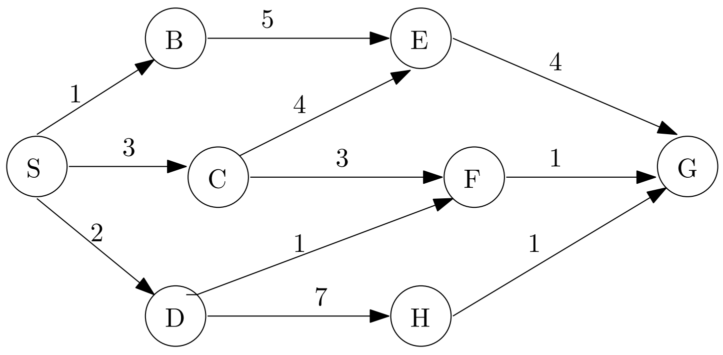

Figure 1.2: Another Graph

Create the adjacency matrix of this graph and test the program execution on it. Note that, it's directed graph. So, be carefule while creating the adjacency matrix. It will not be symmetrix matrix.

1.3 Breadth First Search

In BFS, nodes of the graph are traversed breadth wise. That is, nodes are visited level-wise from the source node. Here, *queue* data structure is used.

One C program that implements BFS is shown next.

```
/* C program for BFS, bfs1.c */

#include<stdio.h>

int a[15][15];
int q[15],front=-1,rear=-1;
int visited[15];

void bfs(int,int);
void insert(int);
int delete();

int main() {
 int n,i,j,src;

 printf("\n Enter Number of Vertices in Graph: ");
 scanf("%d",&n);
 printf("\n Enter Adjacency Matrix of the Graph: ");
 for(i=0;i<n;i++)
  for(j=0;j<n;j++)
   scanf("%d",&a[i][j]);

 printf("\nn=%d ",n);
 printf("\n Adjacency Matrix: ");
 for(i=0;i<n;i++)  {
  printf("\n");
  for(j=0;j<n;j++)
   printf("%5d",a[i][j]);
 }

 for(i=0;i<n;i++)
   visited[i]=0;

 printf("\n Enter The Source Vertex, in terms of index: \n");
 scanf("%d",&src);

 printf("\n node visit order: ");
 bfs(src,n);     /* call bfs() */

 return 0;
```

```
}


void bfs(int s,int n) {      /* BFS */
 int p,i;

 insert(s);
 visited[s]=1;

 p=delete();

 if(p!=-1)
 printf(" %d",p);
 while(p!=-1) {
  for(i=0;i<n;i++)
    if((a[p][i]!=0)&&(visited[i]==0)) {
       insert(i);
       visited[i]=1;
    }
  p=delete();
  if(p!=-1)
   printf(" %d ",p);
 }

 for(i=0;i<n;i++)
   if(visited[i]==0)
     bfs(i,n);
}


void insert(int num) {       /* Q insert */
 if(rear==14)
   printf("Queue Full");      /* Q full */
 else {
  if(rear==-1) {
    q[++rear]=num;
    front++;
  }
  else
    q[++rear]=num;
 }
}

int delete() {       /* Q delete */
 int num;
 if((front>rear)||(front==-1)) /* Q empty */
  return -1;
 else {
  num=q[front++];
  return num;
 }
}
```

Like DFS, to enter the input, we create the input text file as follws:

```
8
0 1 1 0 0 1 0 0
1 0 0 1 1 0 0 0
1 0 0 0 0 1 1 1
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0
0
```

We reiterate the semantics of this text file. First line input is the number of vertices in input graph. Next the matrix is the adjacency matrix of graph. Last line number is the source vertex from where graph traversal is to begin. The output is shown next.

```
$ gcc bfs1.c
$ ./a.out < graph_ip1.txt

 Enter Number of Vertices in Graph:
 Enter Adjacency Matrix of the Graph:
n=8
 Adjacency Matrix:
    0    1    1    0    0    1    0    0
    1    0    0    1    1    0    0    0
    1    0    0    0    0    1    1    1
    0    1    0    0    0    0    0    0
    0    1    0    0    0    0    0    0
    1    0    1    0    0    0    0    0
    0    0    1    0    0    0    0    0
    0    0    1    0    0    0    0    0
 Enter The Source Vertex, in terms of index:

 node visit order:  0 1  2  5  3  4  6  7  $
```

We run this program on the second graph in figure 1.2. Create the 8×8 adjacency matrix of this directed graph and test the program by execution on this graph.