The following materials have been collected from the numerous sources including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated.

PSet – Graph

Table of Contents

Warming-up: Build a project

As a warming up, you build a project called graph and display a graph menu and a graph. The following files are provided. Build the project with lib/nowic.lib and include/nowic.h. You may use gcc for this project as well.

- graph.h - Don't change this file.
- graph.cpp - You will work on these files
- graphDriver.cpp - You may not need to change this file.
- Graph???.txt – graph files, place them in VS project folder.
- graphx.exe - an executable to compare with

When you start the program, it displays the graph menu as shown below:

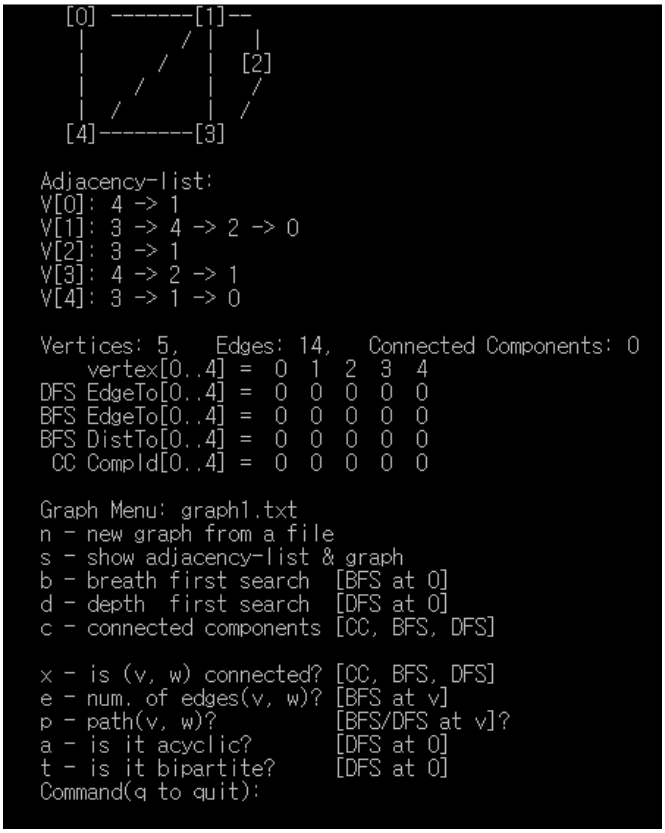
```
Graph Menu:
n - new graph from a file
s - show adjacency-list & graph
b - breath first search  [BFS at 0]
d - depth  first search  [DFS at 0]
c - connected components [CC, BFS, DFS]

x - is (v, w) connected? [CC, BFS, DFS]
e - num. of edges(v, w)? [BFS at v]
p - path(v, w)?          [BFS/DFS at v]?
a - is it acyclic?       [DFS at 0]
t - is it bipartite?     [DFS at 0]
Command(q to quit):
```

Now you can create a graph by entering a graph filename at the menu option n. Once you specified a valid graph file, it reads and display an adjacency list of the graph.

You may specify a graph file in a command-line. Then it will read it and displays

- 'dotted-line graph' that read from the graph text file
- Adjacency-list
- Current status of graph including some content of scratch buffers such as parent[], distTo[] and compId[] etc.



.

You are asked to implement a few options in the menu.

Step 1 – Display Adjacency-list

To understand the graph data structure, let us try to print the content of graph data structure which was populated by reading a graph text file. The graph text file consists of three kinds of lines as shown below.

1. The lines that begin with # and / are comments.
2. The lines that begin with a . (dot) are to be read to display on user's request.
3. The lines that begin with some numbers are nodes and edges.

```
# Graph file format example:
# To represent a graph:
# The number of vertex in the graph comes at the first line.
# The number of edges comes In the following line,
# Then list a pair of vertices connected each other in each line.
# The order of a pair of vertices should not be a matter.
# Blank lines and the lines which begins with # or ; are ignored.
#
# The lines that begins with . will be read into graph data structure
# and displayed on request.
#
# For example:
.  [0] -------[1]--
.   |        / |   |
.   |      /   |  [2]
.   |    /     |   /
.   |  /       |  /
.  [4]--------[3]
#
# DFS [0 4 3 2 1]
# BFS [0 4 1 3 2]
# DFS parent[0..4] = -1  0  1  4  0
# BFS parent[0..4] = -1  0  1  4  0
# BFS distTo[0..4] =  0  1  2  2  1
#  CC compId[0..4] =  0  0  0  0  0
5
7
0 1
0 4
1 2
1 4
1 3
2 3
3 4
```

```
// prints the adjacency list of graph
void printAdjList(pGraph g){
 if (isEmpty(g)) return;

 printf("\n\tAdjacency-list: \n");

 printf("Your code here\n");

}
```

Step 2 – DFS

You may implement this most important function, called DFS ().

```
void DFS(pGraph g, int v)
```

It is composed of two steps. First, it initializes all necessary data structures for the processing. Secondly, it calls function DFS() which actually computes DFSrecursive() recursively. This DFS is used in a few functions such as CC_DFS, cycle() – cycleDFS,

Once DFS is done, you are ready to make e and p menu options work.

```
// Recursive DFS does the work
void DFSrecurse(pGraph g, int v) {
  g->marked[v] = true;
  printf("%d ", v);                    // visiting node

  printf("Your code here\n");
}
```

```
// returns a path from s to v using the result of DFS's parent[].
// It has to use a stack to retrace the path back to the source.
// Once the client(caller) gets a stack returned,
// he/she pops the paths and free the stack.
pStack pathToDFS(pGraph g, int v, int w) {
  if (isEmpty(g)) return NULL;

  DFS(g, v);

  printf("Your code here\n");
  // get path from g->parent[], push to a stack, return it


  return NULL;
}
```

Step 3 – BFS

Implement the breadth first search algorithm and e and p options.

```
// runs breadth first search, produces disTo[], parentBFS[] and
// has side effects that prints a list of nodes visited
void BFS(pGraph g, int v) {
  if (isEmpty(g)) return;
  pQueue q = newQueue();
  int N = V(g);
  int E = g->E;

  for (int i = 0; i < N; i++) {
     g->parentBFS[i] = -1;
     g->distTo[i] = -1;
     g->marked[i] = false;
  }

  printf("\tBreadth First Search: "); printf("%d ", v);
  g->distTo[v] = 0;
  g->marked[v] = true;
  enqueue(q, v);
  printf("Your code here\n");
  /*
  while (q->front != NULL) {
     v = dequeue(q);
     // go through all nodes

  }
```

```
 */
 printf("\n");
 freeQueue(q);
}
```

```
// returns a path from s to v using the result of BFS's parentBFS[].
// It has to use a stack to retrace the path back to the source.
// Once the client(caller) gets a stack returned,
// he/she pops the paths and free the stack.
pStack pathToBFS(pGraph g, int v, int w) {
 if (isEmpty(g)) return NULL;

 BFS(g, v); // BFS from v

 printf("Your code here\n");
 // get path from g->parentBFS[], push to a stack, return it

 return NULL;
}
```

Step 4 – Acyclic

Implement a function called cycle() using DFS to find whether or not a graph has a cycle. Once cycle() is done, the menu option a should work.

Step 5 – Bipartite

Implement a function called bipartite() using DFS to find whether or not a graph is bipartite. Also implement a function called bipartiteVerify() to confirm that the graph is really bipartite not using DFS. You may use its adjacency list, knowing that head node's color should be different from all other nodes in its list. Once bipartite() is done, the menu option t should work.

Step 6 – Connected Components

Implement a function called CC() using DFS that labels a component id at each node. The connected nodes are labeled with a same component id. The component id is simple a series of numbers such as 0, 1, 2. Once CC() is done, the menu option x should work.

Submitting your solution

- Include the following line at the top of your every file with your name signed. On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment. Signed: _____

- Make sure your code **compiles** and **runs** right before you submit it. Don't make "a tiny last-minute change" and assume your code still compiles. You will not receive sympathy for code that "almost" works.
- If you only manage to work out the homework partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit

- graph.cpp

Due and Grade points

- Due: Dec. 4, 11:55pm
- Grade points: 6 points
  - 1 point per step.