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



# Getting Started: 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 [Graph][Tablet]   file:graph7.txt V:7 E:12 CCs:1 Deg:3
n - new graph file      x - connected(v,w)
d - DFS(v=0)            e - distance(v,w)
b - BFS(v=0)            p - path(v,w)
c - cyclic(v=0)?        m - print mode[adjList/graph]
t - bigraph(v=0)?       a - bigraph using adj-list coloring
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.

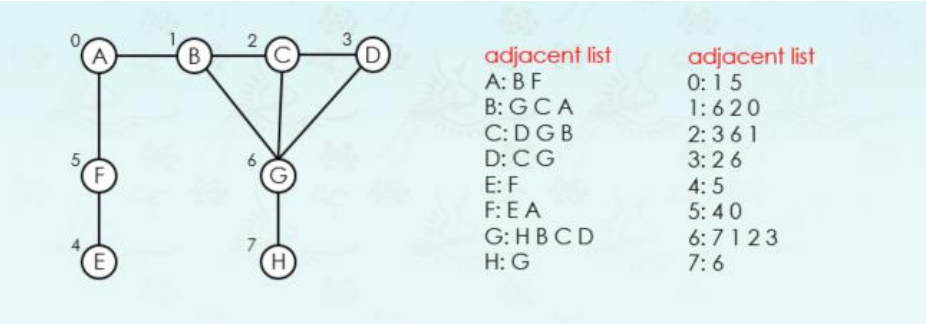
You may specify a graph file in a command-line. Then it will read it and displays

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

```
.  [0] -------[1]--
.   |        / |   |
.   |      /   |  [2]
.   |    /     |   /
.   |  /       |  /
.  [4]--------[3]

    vertex[0..4] =    0   1   2   3   4
     color[0..4] =    0   0   0   0   0

      DFS0[0..4] =    0   4   3   2   1
      CCID[0..4] =    1   1   1   1   1
DFS parent[0..4] =   -1   2   3   4   0
      BFS0[0..4] =    0   4   1   3   2
    DistTo[0..4] =    0   1   2   2   1
BFS parent[0..4] =   -1   0   1   4   0

Graph [Graph][Tablet]   file:graph1.txt V:5 E:14 CCs:1 Deg:4
n - new graph file      x - connected(v,w)
d - DFS(v=0)            e - distance(v,w)
b - BFS(v=0)            p - path(v,w)
c - cyclic(v=0)?        m - print mode[adjList/graph]
t - bigraph(v=0)?       a - bigraph using adj-list coloring
Command(q to quit):
```

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

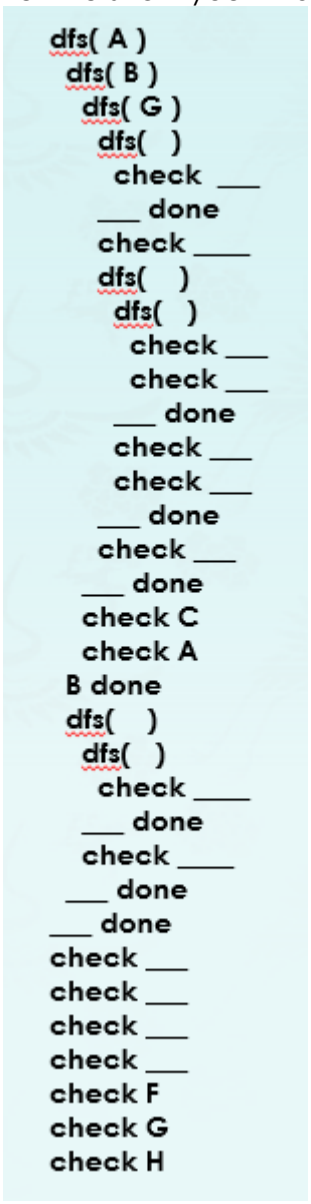
# Step 1- Warming up




Based on the graph shown above, do the following assignment and submit it first.

1. Create an **antenna.txt** such that it generates the adjacent list as shown.
   Refer to some graph files provided with Pset to see how the graph file format works.
2. Find DFS/BFS and parent vertices, respectively and record the results in antenna.txt.
   Add the distances, distTo[], from the starting vertex as well during BFS.
3. Show how you get **the sequences of DFS and BFS** as good as you can.
   Submit this part in a separate file as named like antenna.docx or antenna.hwp, etc.

Let me show you what I did for DFS as an example:




4. Optionally, make sure that your antenna.txt can be processed by graphx.exe and produces the same adjacent list.

# Step 2 – Display Adjacency-list (menu m)

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]
#
#            vertex[0..4] =    0   1   2   3   4
#             color[0..4] =    0   0   0   0   0
#               DFS0[0..4] =    0   4   3   2   1
#               CCID[0..4] =    1   1   1   1   1
#        DFS parent[0..4] =   -1   2   3   4   0
#              BFS0[0..4] =    0   4   1   3   2
#            DistTo[0..4] =    0   1   2   2   1
#        BFS parent[0..4] =   -1   0   1   4   0
5
7
0 1
0 4
1 2
1 4
1 3
2 3
3 4
```

```
// prints the adjacency list of graph
void print_adjlist(graph g){
  if (empty(g)) return;

  cout << "your code here \n";

}
```

# Step 3 – DFS_CCs(), DFS(), DFSpath() using menu d and p

Implement the recursive function DFS() that runs the depth first search in a graph. This function is called by DFS_CCs() which initializes the necessary data structures first and invokes DFS() for each component.

The following code works for a graph with only one connected component. Modify or add some code that makes it work with multiple connected components.

```
// runs DFS for all components and produces DFS0[], CCID[] & parentDFS[].
void DFS_CCs(graph g) {
  if (empty(g)) return;
  for (int i = 0; i < V(g); i++) {
    g->marked[i] = false;
    g->parentDFS[i] = -1;
```

```
    g->CCID[i] = 0;
  }

  queue<int> que;
  cout << "your code here: make it work with multiple CC's\n";
  DFS(g, 0, que);
  setDFS0(g, 0, que);

  g->DFSv = {};
}
```

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

```
// runs DFS for at vertex v recursively.
// Only que, g->marked[v] and g->parentDFS[] are updated here.
void DFS(graph g, int v, queue<int>& que) {
 g->marked[v] = true;  // visited
 que.push(v);        // save the path

 cout << "your code here (recursion) \n";
}
```

Once you implement DFS() and DFS_CCs() successfully, then complete DFSpath() as shown below:

```
// returns a path from v to w using the DFS result or parentDFS[].
// It has to use a stack to retrace the path back to the source.
// Once the client(caller) gets a stack returned,
void DFSpath(graph g, int v, int w, stack<int>& path) {
  if (empty(g)) return;
  for (int i = 0; i < V(g); i++) {
    g->marked[i] = false;
    g->parentDFS[i] = -1;
  }
  queue<int> q;
  DFS(g, v, q);     // DFS at v, starting vertex
  g->DFSv = q;      // DFS result at v
  path = {};

  cout << "your code here\n"; // push v to w path to the stack path
}
```

# Step 4 – BFS(), distTo[] and parentBFS[] (menu b, e, p)

In this step, the breadth first search algorithm has been implemented except distTo[] and parentBFS[]. Complete two lines of code to update distTo[] and parentBFS[] inside while() loop. Once you update them, you must code distTo() function that computes the distance between to vertices.

```
// runs BFS starting at v and produces distTo[] & parentBFS[]
void BFS(graph g, int v) {
```

```
 queue<int> que;            // to process each vertex
 queue<int> sav;       // BFS result saved

 // all marked[] are set to false since it may visit all vertices
 for (int i = 0; i < V(g); i++)  g->marked[i] = false;
 g->parentBFS[v] = -1;
 g->marked[v] = true;
 g->distTo[v] = 0;
 g->BFSv = {};

 que.push(v);
 sav.push(v);

 while (!que.empty()) {
   int cur = que.front(); que.pop();  // remove it since processed
   for (gnode w = g->adj[cur].next; w; w = w->next) {
     if (!g->marked[w->item]) {
       g->marked[w->item] = true;
       que.push(w->item);// queued to process next
       sav.push(w->item);// save the result

       cout << "your code here";   // set parentBFS[] & distTo[]

     }
   }
 }
 g->BFSv = sav;                // save the result at v
 setBFS0(g, v, sav);
}
```

```
// returns the number of edges in a shortest path between v and w
int distTo(graph g, int v, int w) {
 if (empty(g)) return 0;
 if (!connected(g, v, w)) return 0;

 BFS(g, v);
 cout << "your code here\n";      // compute and return distance
 return 0;
}
```

Once you implement BFS() successfully, then complete BFSpath() as shown below. Since the code you are adding here is the same as in DFSpath(), just copy and paste them.

```
// returns a path from v to w using the BFS result or parentBFS[].
// It has to use a stack to retrace the path back to the source.
// Once the client(caller) gets a stack returned,
void BFSpath(graph g, int v, int w, stack<int>& path) {
 if (empty(g)) return;

 BFS(g, v);                    // g->BFSv updated already.

 path = {};                    // clear path, stack<int>().swap(path);

 cout << "your code here\n";  // push v to w path to the stack path

}
```

# Step 6: bigraph: Two-Colorability using BFS (menu a)

Implement two-colorability using BFS and adjacent-list. You may test it with the following graph files provided.

- graph6.txt, graph7.txt - graphs with one component, not bipartite
- graph6cc.txt, graph7cc.txt - graphs with multiple CCs, not bipartite
- graph6ccb.txt, graph7ccb.txt – graphs with multiple CCs, bipartite or two-colorable

The function main() calls bigraphBFS2Coloring() to check whether or not a graph is bipartite as shown below:

```
// runs two-coloring using BFS
bool bigraphBFS2Coloring(graph g) {
  if (empty(g)) return true;
  init2colorability(g);
  BFS2Coloring(g);
  return check2colorability(g);
}
```

It initializes the necessary data structure first, then invokes BFS function for two-coloring. As it goes through vertices, it changes the color alternatively and set its color. It begins with BLACK which is 0 and WHITE 1 following.

```
// runs two-coloring using BFS - no recursion
void BFS2Coloring(graph g) {
 DPRINT(cout << ">BFS2Coloring" << endl;);
 queue<int> que;
 int v = 0;
 que.push(v);

 // while (!que.empty()) {
 cout << "your code here \n";
 // }

 DPRINT(cout << "<BFS2Coloring" << endl;);
}
```

# Step 5: bigraph: Two-Colorability using DFS (menu a)

Implement two-colorability using DFS and adjacent-list. You may test it with the following graph files provided.

- graph6.txt, graph7.txt - graphs with one component, not bipartite
- graph6cc.txt, graph7cc.txt - graphs with multiple CCs, not bipartite
- graph6ccb.txt, graph7ccb.txt – graphs with multiple CCs, bipartite or two-colorable

The function main() calls bigraphDFS2Coloring() to check whether or not a graph is bipartite as shown below. However, the following function does **NOT** work with some graph files provided even if DFS2Coloring() works fine. Make it work with all cases.

```
// runs two-coloring using DFS
bool bigraphDFS2Coloring(graph g) {
  if (empty(g)) return true;
  init2colorability(g);

    DFS2Coloring(g,0);

  return check2colorability(g);
  } // This function does not work in some cases.
```

It initializes the necessary data structure first, then invokes DFS function for two-coloring recursively. As it goes through vertices, it changes the color alternatively and set its color. It begins with BLACK which is 0 and WHITE 1 following.

```
// runs two-coloring using DFS recursively
void DFS2Coloring(graph g, int v) {  // DFS
 DPRINT(cout << ">DFS2Coloring v=" << v << " color=" << g->color[v] << endl;);
 g->marked[v] = true;          // v is visited now

 cout << "your code here (recursion)\n";

 DPRINT(cout << "<DFS2Coloring visits v=" << v << endl;);
}
```

# 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

- 1st submission: antenna.txt & word/hwp file
- 2nd submission: graph.cpp

# Due and Grade points

- Due:
  - Step 1:
  - Step 2 ~ 6:
- Grade points: 12 points
  - 2 point per step.