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 DFS/BFS

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 g++ for this project as well.

- graph.h - Don't change this file.
- graph.cpp - You will work on these files
- driver.cpp - You may not need to change this file.
- graph?.txt - graph files, put them **in VS project folder or with your .exe file**
- graphx.exe - an executable to compare with

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

```
Graph
n - new graph file      x - connected(v,w)
d - DFS(v=0)            e - distance(v,w)
b - BFS(v=0)            p - path(v,w)
m - print mode[adjList/graph]
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.

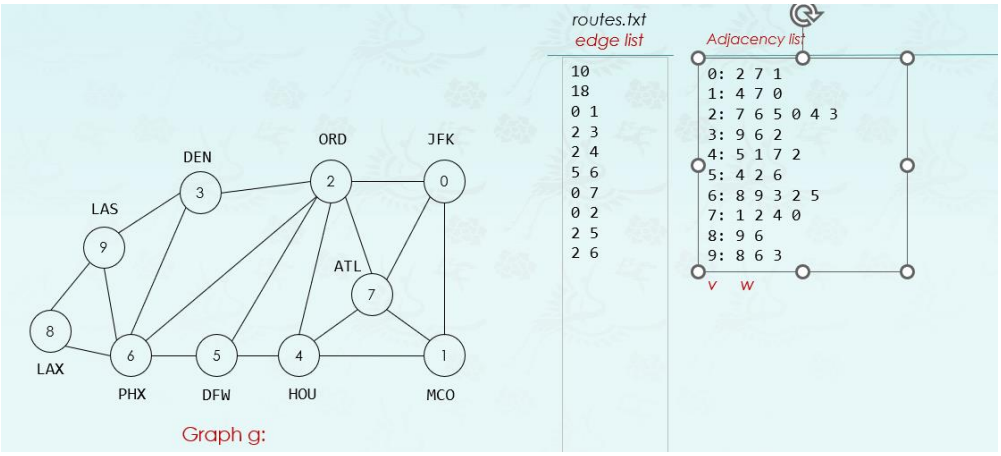
You may specify a graph file in a command-line. Then it will read it and displays

- '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):
```

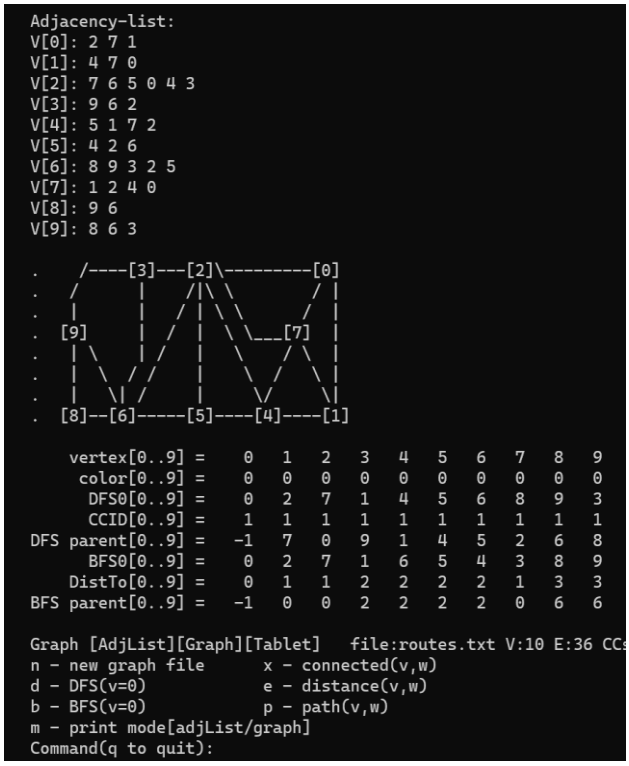
You are asked to implement a few options in the menu.

Step 1- Warming up



Based on the graph shown above, do the following assignment.

1. Get familiar with using graphx.exe and using ~.txt graph files provided in pset. Run DFS/BFS and get familiar with the results, its meaning and menu items.
2. Create an **routes.txt** such that it generates the adjacency list and results as shown below:



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(), DFS_CCs(), DFSpath()

Implement the recursive function DFS() that runs the depth first search in a graph.

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";
}
```

The DFS menu option d actually calls DFS_CCs() which initializes the necessary data structures first and invokes DFS() recursively 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 = {};
}
```

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(), BFS_CCs, distTo[] and parentBFS[]

In this step, most of breadth first search algorithm has been implemented, but not for the connected components.

- Replace the following code to handle the connected components in BFS_CCS().

```
// runs BFS for all vertices or all connected components
// It begins with the first vertex 0 at the adjacent list.
// It produces BFS0[], distTo[] & parentBFS[].
void BFS_CCs(graph g) {
 DPRINT(cout << ">BFS_CCs\n";);
 if (empty(g)) return;

 for (int i = 0; i < V(g); i++) {
    g->marked[i] = false;
    g->parentBFS[i] = -1;
    g->BFS0[i] = -1;
    g->distTo[i] = -1;
 }

 // BFS for all connected components starting from 0
 cout << "your code here to replace the next line\n";
 BFS(g, 0);

 g->BFSv = {};  // clear it not to display, queue<int>().swap(g->BFSv);
```

```
 DPRINT(cout << "\n<BFS_CCs\n";);
}
```

- Complete two lines of code to set values in distTo[] and parentBFS[] inside while() loop. This distTo[] has the distance (values) from v to every w 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);
}
```

- The function distTo(g, v, w) invokes BFS(g, v) to set the values in distTo[] and parentBFS[]. Then, it returns the distance between two vertices, v and w that saved in distTo[].
- As you know, distTo[] has distance values from v to every vertices w's in the graph. The function distTo(g, v, w) is usually invoked by its driver and returns the distance from v and w.

```
// 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

}
```

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

- Step 1: routes.txt
- Step 2 ~ 4: graph.cpp

Due and Grade points

- Due:
- Grade: