COMP9024 25T1 Data Structures and Algorithms

Week 5 Practical
Graph Data Structures

1. (Graph properties)

   a. Write pseudocode for computing

   - the degree of each vertex
   - all 3-cliques (i.e. cliques of 3 nodes, "triangles")
   - and the density

   of an undirected graph *g* with *n* vertices.

   Your methods should be representation-independent; the only function you should use is to check if two vertices *v,w∈{0,…n-1}* are adjacent in *g*.

   The *density* of an undirected graph with $|E|$ edges and $|V|$ vertices, is defined as $\frac{2|E|}{|V|^2}$.

   Note: *please use the above definition, even though a more correct definition for an undirected graph would be* $\frac{2|E|}{|V|(|V|-1)}$.

   b. Determine the asymptotic complexity of your three algorithms. Assume that the adjacency check is performed in constant time, *O(1)*.

   c. Implement your algorithms in a program `graphAnalyser.c` that

   1. builds a graph from user input:
      - first, the user is prompted for the number of vertices
      - then, the user is repeatedly asked to input an edge by entering a "from" vertex followed by a "to" vertex
      - until any non-numeric character(s) are entered
   2. outputs the degree of each vertex, considering the vertices in ascending order, i.e. starting with the degree of vertex 0, followed by the degree of vertex 1 and so on;
   3. displays all 3-cliques of the graph in ascending order;
   4. displays the density of the graph formatted to three decimal places.

   You should use `scanf` to read user input. You can assume that input will either be a valid integer or non-numeric.

   Your program should use the Graph ADT from the lecture (`Graph.h` and `Graph.c`). **These files must not be changed.**

   Hint: You may assume that the graph has a minimum of 1 node.

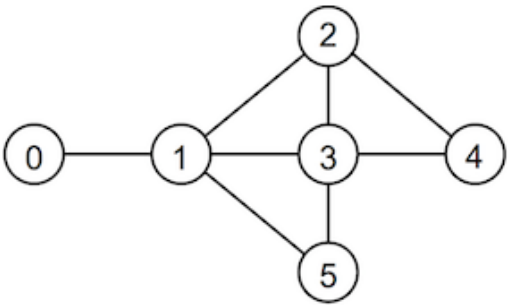
   We have also provided a `Makefile` to simplify the compilation process. Place it in the same directory as your code and simply execute:

```
prompt$ make
```

   If you'd like to delete the object files it creates, you can instead execute:

```
prompt$ make clean
```

   An example of the program executing is shown below for the graph




```
prompt$ ./graphAnalyser
Enter the number of vertices: 6
Enter an edge (from): 0
Enter an edge (to): 1
Enter an edge (from): 1
Enter an edge (to): 2
Enter an edge (from): 1
Enter an edge (to): 3
Enter an edge (from): 1
Enter an edge (to): 5
Enter an edge (from): 2
Enter an edge (to): 3
Enter an edge (from): 2
Enter an edge (to): 4
Enter an edge (from): 3
Enter an edge (to): 4
Enter an edge (from): 3
Enter an edge (to): 5
Enter an edge (from): done
Done.
Degree of node 0: 1
Degree of node 1: 4
Degree of node 2: 3
Degree of node 3: 4
Degree of node 4: 2
Degree of node 5: 2
3-cliques:
1-2-3
1-3-5
2-3-4
Density: 0.444
prompt$
```

Here is an example where there are no 3-cliques:

```
prompt$ ./graphAnalyser
Enter the number of vertices: 3
Enter an edge (from): 0
Enter an edge (to): 1
Enter an edge (from): 1
Enter an edge (to): 2
Enter an edge (from): finito
Done.
Degree of node 0: 1
Degree of node 1: 2
Degree of node 2: 1
3-cliques:
Density: 0.444
prompt$
```

Note that any non-numeric data can be used to 'finish' the interaction.

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to this exercise. It expects to find a program named* `graphAnalyser.c` *in the current directory. You can use dryrun as follows:*

```
prompt$ 9024 dryrun graphAnalyser
```

Note: Please ensure that your program output follows exactly the format shown in the sample interaction above. In particular, the degree should be shown for the vertices in ascending order, the 3-cliques should be printed in ascending order, and the density should be printed to three decimal places.

Answer:

The following algorithm uses two nested loops to compute the degree of each vertex. Hence its asymptotic running time is $O(n^2)$.

```
degrees(g):
   Input  graph g
   Output array of vertex degrees

   for all vertices v∈g do
      deg[v]=0
      for all vertices w∈g, v≠w do
         if v,w adjacent in g then
            deg[v]=deg[v]+1
         end if
      end for
   end for
   return deg
```

The following algorithm uses three nested loops to print all 3-cliques in order. Hence its asymptotic running time is $O(n^3)$.

```
show3Cliques(g):
   Input graph g of n vertices numbered 0..n-1

   for all i=0..n-3 do
      for all j=i+1..n-2 do
         if i,j adjacent in g then
            for all k=j+1..n-1 do
               if i,k adjacent in g and j,k adjacent in g then
                  print i"-"j"-"k
               end if
            end for
         end if
      end for
   end for
```

The following algorithm uses two nested loops to calculate the graph density. Hence its asymptotic running time is $O(n^2)$.

```
density(g):
   Input  graph g
   Output graph density

   m=0
   for all vertices v∈g do
      for all vertices w∈g, w>v do
         if v,w adjacent in g then
            m=m+1
         end if
      end for
   end for
   return (2*m)/(n*n)
```

Sample `graphAnalyser.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "Graph.h"

// determine and output the sequence of vertex degrees
void degrees(Graph g) {
   int nV = numOfVertices(g);
   int v, w, degree;

   for (v = 0; v < nV; v++) {
      degree = 0;
      for (w = 0; w < nV; w++) {
         if (adjacent(g,v,w)) {
            degree++;
         }
      }
      printf("Degree of node %d: %d\n", v, degree);
   }
}

// show all 3-cliques of graph g
void show3Cliques(Graph g) {
   int i, j, k;
   int nV = numOfVertices(g);

   printf("3-cliques:\n");
   for (i = 0; i < nV-2; i++) {
      for (j = i+1; j < nV-1; j++) {
         if (adjacent(g,i,j)) {
            for (k = j+1; k < nV; k++) {
               if (adjacent(g,i,k) && adjacent(g,j,k)) {
                  printf("%d-%d-%d\n", i, j, k);
               }
            }
         }
      }
   }
}

// calculate and output the density of undirected graph g
void Density(Graph g) {
   int nV = numOfVertices(g);
   int nE = 0;
   int v, w;
   double density;

   for (v = 0; v < nV; v++) {
      for (w = v; w < nV; w++) {
         if (adjacent(g,v,w)) {
            nE++;
         }
      }
   }

   density = (2.0 * (double)nE) / ((double)nV * (double)nV);
   printf("Density: %.3lf\n", density);
}

int main(void) {
   Edge e;
   int nV;
```

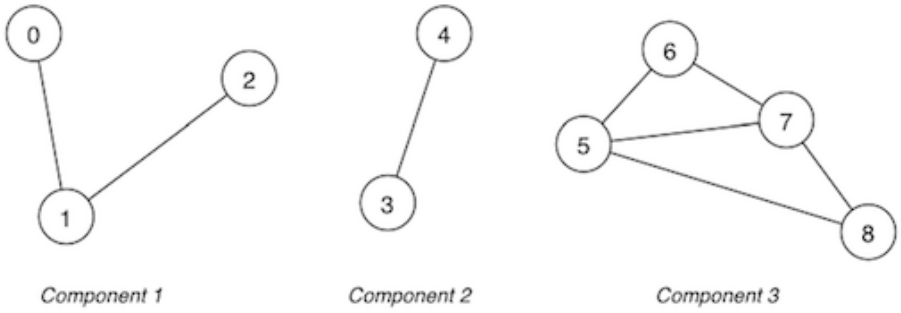
```
    printf("Enter the number of vertices: ");
    scanf("%d", &nV);
    Graph g = newGraph(nV);

    printf("Enter an edge (from): ");
    while (scanf("%d", &e.v) == 1) {
       printf("Enter an edge (to): ");
       scanf("%d", &e.w);
       insertEdge(g, e);
       printf("Enter an edge (from): ");
    }
    printf("Done.\n");

    degrees(g);
    show3Cliques(g);
    Density(g);
    freeGraph(g);

    return 0;
}
```

2. (Cycle check)

   a. Take the "buggy" cycle check from the lecture and design a correct algorithm, in pseudocode, to use depth-first search to determine if a graph has a cycle.

   b. Write a C program `cycleCheck.c` that implements your solution to check whether a graph has a cycle. The graph should be built from user input in the same way as in exercise 1. Your program should use the Graph ADT from the lecture (`Graph.h` and `Graph.c`). **These files must not be changed**.

   An example of the program executing is shown below for the following graph:




```
prompt$ ./cycleCheck
Enter the number of vertices: 9
Enter an edge (from): 0
Enter an edge (to): 1
Enter an edge (from): 1
Enter an edge (to): 2
Enter an edge (from): 4
Enter an edge (to): 3
Enter an edge (from): 6
Enter an edge (to): 5
Enter an edge (from): 6
Enter an edge (to): 7
Enter an edge (from): 5
Enter an edge (to): 7
Enter an edge (from): 5
Enter an edge (to): 8
Enter an edge (from): 7
Enter an edge (to): 8
Enter an edge (from): done
Done.
```

```
The graph has a cycle.
prompt$
```

If the graph has no cycle, then the output should be:

```
prompt$ ./cycleCheck
Enter the number of vertices: 3
Enter an edge (from): 0
Enter an edge (to): 1
Enter an edge (from): #
Done.
The graph is acyclic.
prompt$
```

Hint: You may assume that the graph has a minimum of 1 node and a maximum of 1,000 nodes.

To test your program you can execute the `dryrun` *program that corresponds to this exercise. It expects to find a program named* `cycleCheck.c` *in the current directory. You can use dryrun as follows:*

```
prompt$ 9024 dryrun cycleCheck
```

Note: Please ensure that your output follows exactly the format shown above.

Answer:

a.

```
hasCycle(G):
|  Input  graph G
|  Output true if G has a cycle, false otherwise
|
|  mark all vertices as unvisited
|  for each vertex v∈G do           // make sure to check all connected components
|  |  if v has not been visited then
|  |     if dfsCycleCheck(G,v,v) then
|  |        return true
|  |     end if
|  |  end if
|  end for
|  return false

dfsCycleCheck(G,v,u):      // look for a cycle that does not go back directly to u
|  mark v as visited
|  for each (v,w)∈edges(G) do
|  |  if w has not been visited then
|  |  |  if dfsCycleCheck(G,w,v) then
|  |  |     return true
|  |  |  end if
|  |  else if w≠u then
|  |  |  return true
|  |  end if
|  end for
|  return false
```

b. The following two C functions implement this algorithm:

```
#define MAX_NODES 1000

int visited[MAX_NODES];

bool dfsCycleCheck(Graph g, Vertex v, Vertex u) {
   visited[v] = true;
   Vertex w;
   for (w = 0; w < numOfVertices(g); w++) {
      if (adjacent(g, v, w)) {
```

```
            if (!visited[w]) {
               if (dfsCycleCheck(g, w, v)) {
                  return true;
               }
            } else if (w != u) {
               return true;
            }
         }
      }
      return false;
}

bool hasCycle(Graph g) {
      int v, nV = numOfVertices(g);
      for (v = 0; v < nV; v++) {
         visited[v] = false;
      }
      for (v = 0; v < nV; v++) {
         if (!visited[v]) {
            if (dfsCycleCheck(g, v, v)) {
               return true;
            }
         }
      }
      return false;
}
```

Due Date

Tuesday, 25 March, 11:59:59. Late submissions will not be accepted.

Submission

You should submit your files using the following `give` command:

```
prompt$ give cs9024 week5 graphAnalyser.c cycleCheck.c
```

Alternatively, you can select the option to "Make Submission" at the top of this page to submit directly through WebCMS3.

Important notes:

- Make sure you spell all filenames correctly.
- You can run `give` multiple times. Only your last submission will be marked.
- Where you're expected to submit multiple files, ensure they form a single submission. If you separate them across multiple submissions, each submission will replace the previous one.
- Whether you submit through the command line or WebCMS3, it is your responsibility to ensure it reports a successful submission. Failure to submit correctly will not be considered as an excuse.
- You cannot obtain marks by e-mailing your code to tutors or lecturers.

Assessment

- Each question is worth 1 mark, for a total of 2 marks.
- Submissions will be auto-marked shortly after the deadline has passed.
- It is important that the output of your program follows exactly the format of the sample output, otherwise auto-marking will result in 0 marks.
- Ensure that your program compiles on a CSE machine with the standard options, i.e. `-Wall -Werror -std=c11`. Programs that do not compile will receive 0 marks.

- Auto-marking will use test cases different to `dryrun`. (Hint: do your own testing in addition to running `dryrun`).

Collection

Once marking is complete you can collect your marked submission using the following command:

```
prompt$ 9024 classrun -collect week5
```

You can also view your marks using the following command:

```
prompt$ 9024 classrun -sturec
```

You can also collect your marked submission directly through WebCMS3 from the "Collect Submission" tab at the top of this page.

Plagiarism

Group submissions will not be allowed. Your programs must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for similar assessments in previous years, if applicable) and serious penalties will be applied, including an entry on UNSW's plagiarism register.

- ***Do not copy ideas or code from others***
- ***Do not use a publicly accessible repository or allow anyone to see your code***

Please refer to the on-line sources to help you understand what plagiarism is and how it is dealt with at UNSW:

- Plagiarism and Academic Integrity
- UNSW Plagiarism Policy Statement
- UNSW Plagiarism Procedure