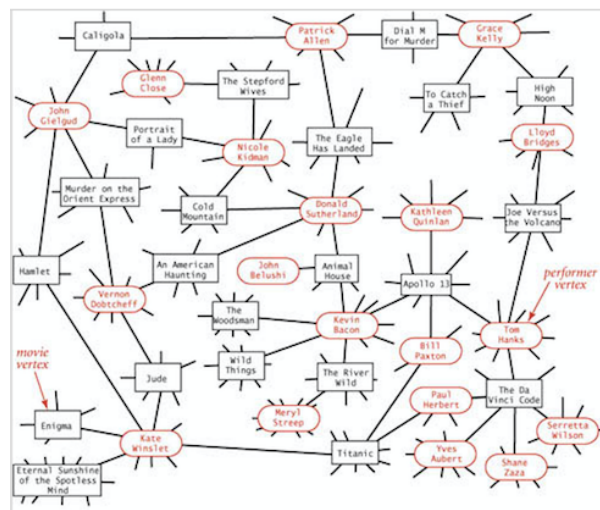


Querying and Traversing Graphs

Part 1: Representing and using a graph

Introduction

In this project you will build a **graph** with a dataset of your choice (from three options given to you) to represent entities and connections between those entities. You will be given some datasets to choose from, such as a variation of *the Hollywood Graph* in which, vertices are actors and edges connect actors that have appeared in a movie together (figure below), or a *Congress Bills Graph* in which, vertices are US congresspersons and edges connect members of congress that have co-sponsored legislative bills together.



Hollywood Graph representing actors and the movies they have appeared.

You will use the graph to query paths between nodes. For example, in the *Hollywood graph*, you query a path between a pair of actors or between a pair of movies, or an actor and a movie, to see how closely they are connected. In the *Congress Bills graph* you query a path between members of congress to see how closely they are connected. This functionality will allow you to “play” the *Six Degrees of Kevin Bacon Game* (a popular on college campuses about a decade ago) for the domain you choose. The game is based on the ideas of *six degrees of separation* and the *small-world experiment*.

Six degrees of separation [https://en.wikipedia.org/wiki/Six_degrees_of_separation] is the idea that all living things and everything else in the world are six or fewer steps away from each other so that a chain of “a friend of a friend” statements can be made to connect any two people in a maximum of six steps.

The small-world experiment [https://en.wikipedia.org/wiki/Small-world_experiment] comprised several experiments conducted by Stanley Milgram and other researchers examining the average path length for social networks of people in the United States. The research was groundbreaking in that it suggested that human society is a small-world-type network characterized by short path-lengths.

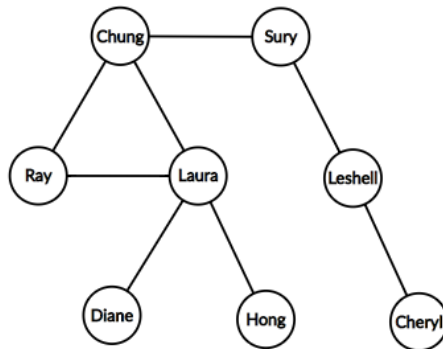
The goal of the *Six Degrees of Kevin Bacon Game* is to try to find the fewest number of connections to link any other actor with Kevin Bacon. It was discovered that you could connect Kevin Bacon with just about any other actor in six steps or so.

Quick Recap of Concepts (1)

This section is a quick recap of the concepts that you need to know for this project. If you don't understand something or can't answer the questions and exercises, you should review the textbook and/or materials provided for this topic.

Definition. A graph is a set of vertices/nodes and a collection of edges that connect some pairs of vertices.

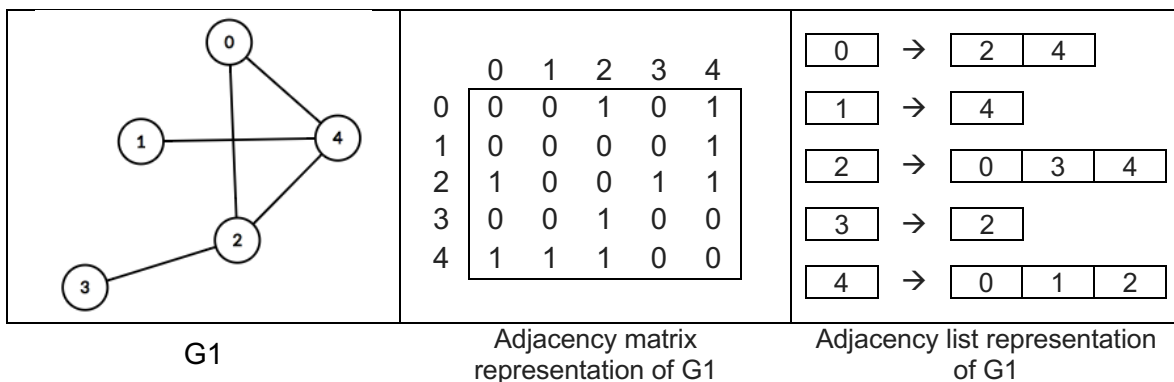
The edges in a graph can be *directed* or *undirected*. For this assignment we will focus on undirected graphs.



The vertices represent people and there is an edge between two people if they are classmates. The graph is undirected because if a person $P1$ is classmate with a person $P2$ then $P2$ is classmate with $P1$. There is no need to specify the direction or order of the connection.

Graph Representation

The most commonly used representations of graphs are *Adjacency Matrix* and *Adjacency List*.



Programming Exercise 1: Creating and displaying a graph

We will start with an implementation of a graph that defines the fundamental graph operations we will need.

<code>Graph()</code>	Constructor. Creates a graph object.
<code>void add_vertex ()</code>	Adds a vertex to the graph (index automatically assigned)
<code>void add_edge (int source, int target)</code>	Adds an edge connecting source to target
<code>int V() const</code>	Returns the number of vertices
<code>int E() const</code>	Returns the number of edges
<code>set<int> neighbors(int v) const</code>	Returns a set containing the vertices adjacent to v (the neighbors of v)
<code>bool contains(int v) const;</code>	Checks whether vertex v is in the graph

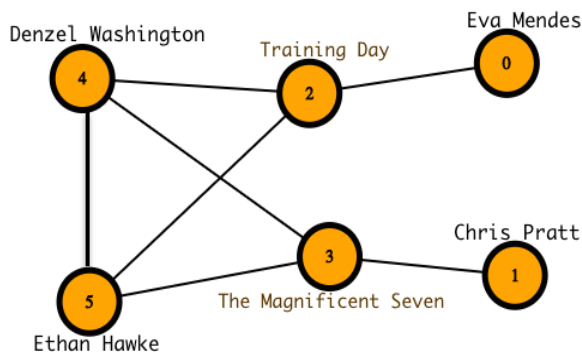
//OVERLOADED OUTPUT OPERATOR
friend std::ostream& operator<< (std::ostream &out, const Graph &g);

Download the starter code for this part of the project and look at it so that you understand how the graph is being represented and handled.

Questions Set 1:

- 1) Look at the member variables of the *Graph* class (in the *Graph.h* file). What representation is being used for the graph in the code provided, Adjacency Matrix or Adjacency List? How is one or the other being stored (what data structure is used)?
- 2) Every function in the *Graph* class has a comment at the top that describes its general purpose. Explain in detail how the functions `add_edge`, `neighbors`, and `contains` do their work. Specifically, for each of these three functions:
 - a. Explain what each line of code in the function does. Do this by adding a comment at the end of the line. Do not repeat the function's general-purpose comment.
 - b. Give the Big-O time of the function. Write it as comment before the function header.

TASK 1. Create a client program (*GraphClient.cpp*) with a main function where you will create and display the graph in the following figure:



You will need to import "*Graph.h*". You can use the following skeleton for your client and write your code in the main function:

```
#include <iostream>
#include "Graph.h"
using namespace std;
int main() {
    //your code here
}
```

The name of your graph object must be a single word consisting of your last name followed by the suffix “graph”. For example, my graph name should be *drzavalagraph* and I would use the following instruction to declare my graph object:

```
Graph drzavalagraph;
```

Use the *add_vertex* and *add_edge* functions on the graph object as needed to construct the graph in the figure above.

After you have added the vertices and edges, you should display the graph to screen. You might have noticed that the output operator has been overloaded, so you can simply output a graph to screen using the output operator:

```
cout << drzavalagraph;
```

If you run your program, you will notice that nothing is outputted; the graph is not displayed. Can you figure out why?

TASK 2. Modify the implementation of the overloaded output operator (*Graph.cxx* file) so that the information about the graph is sent to the output stream. [For a reminder about overloading the output operator: <http://www.learncpp.com/cpp-tutorial/93-overloading-the-io-operators>]. For simplicity, you do not need to display a visual graph. Instead, do a textual display. For example:

```
=====
Graph Summary: 6 vertices, 7 edges
=====
0 --> 2
1 --> 3
2 --> 0    5    4
3 --> 1    5    4
4 --> 3    5    2
5 --> 2    3    4
```

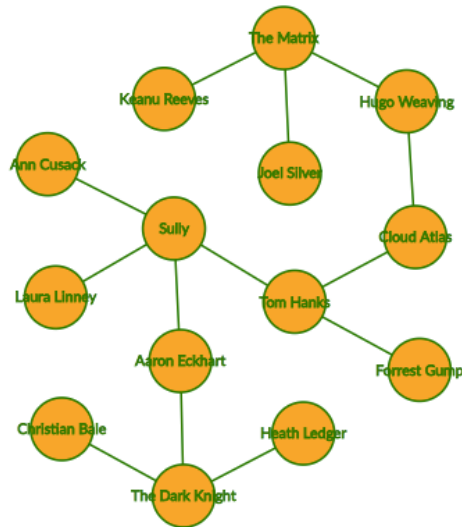
You will need to iterate through the vector of vertices (*edges* variable in the Graph class) and for each vertex, iterate through the set of adjacent vertices or neighbors. [For a reminder about vectors and sets: <http://www.cplusplus.com/reference/stl/>]

Questions Set 2

- 1) The Graph class has an *accessor* function that returns the number of edges: `int E()`, and an *accessor* function that returns the number of vertices: `int V()`. However, the class has a private variable to keep track of the number of edges only (*nedges*), which is increased every time an edge is added to the graph. Why is it that we don't need a variable to keep track of the number of vertices as well?
- 2) Give the *big-O* time for displaying the graph (the overloaded output operator function).
- 3) Give the *big-O* time for displaying the graph (overloaded output operator) if a matrix representation had been used.

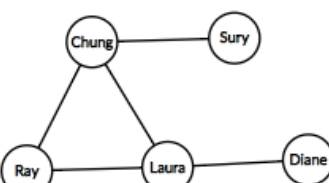
Programming Exercise 2: A Labeled Graph

We now move on to working with a Labeled Graph, we need to be able to create graphs that have labeled vertices, not just numbers. For example:



Our current implementation of a graph does not allow labeled vertices. In this part, we extend the graph implementation to achieve that. To keep things clear and simple, we will work with a new class called *LabeledGraph*.

The representation of a graph will stay the same and vertices will still be numbered. We will simply keep track of the label associated with each vertex. For example:

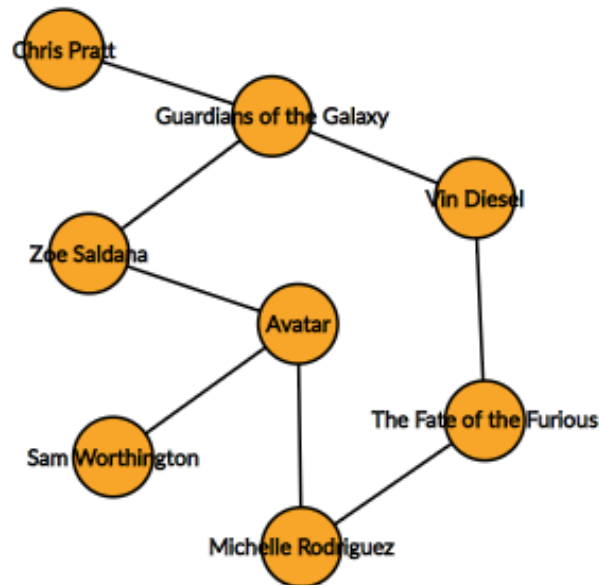
	<table><tr><td>0</td><td>→</td><td>1</td></tr><tr><td>1</td><td>→</td><td>0 2 3</td></tr><tr><td>2</td><td>→</td><td>1 3</td></tr><tr><td>3</td><td>→</td><td>1 2 4</td></tr><tr><td>4</td><td>→</td><td>3</td></tr></table>	0	→	1	1	→	0 2 3	2	→	1 3	3	→	1 2 4	4	→	3	<table><tr><td>Sury</td><td>0</td></tr><tr><td>Chung</td><td>1</td></tr><tr><td>Ray</td><td>2</td></tr><tr><td>Laura</td><td>3</td></tr><tr><td>Diane</td><td>4</td></tr></table>	Sury	0	Chung	1	Ray	2	Laura	3	Diane	4
0	→	1																									
1	→	0 2 3																									
2	→	1 3																									
3	→	1 2 4																									
4	→	3																									
Sury	0																										
Chung	1																										
Ray	2																										
Laura	3																										
Diane	4																										
Graph G	Representation of G	Label-Index mapping																									

Download the starter code for this part of the project and look at it so that you understand how the labeled graph is being represented and handled.

Questions Set 3:

1. What new variables were added to *LabeledGraph* that were not in *Graph* and what do you think they are used for?
2. What changes do you see in the *add_vertex* and *add_edge* functions (*LabeledGraph* vs *Graph*)? Explain why you think those changes were made.

TASK 1. Create a client program (*LabeledGraphClient.cpp*) with a main function where you will create and display the graph in the following figure:



The name of your graph object must be a single word consisting of your name followed by the suffix "LabeledGraph". For example, my graph name should be *rosaLabeledGraph*.

TASK 2. Write the implementation of the overloaded output operator (*LabeledGraph.cxx* file). For simplicity, you do not need to display a visual graph. Instead, do a textual display.

For example:

```

=====
Graph Summary: 8 vertices, 8 edges
=====
Avatar
    Zoe Saldana
    Sam Worthington
    Michelle Rodriguez
Zoe Saldana
    Avatar
    Guardians of the Galaxy
Sam Worthington
    Avatar
Michelle Rodriguez
    Avatar
    The Fate of the Furious
Guardians of the Galaxy
    Zoe Saldana
    Chris Pratt
    Vin Diesel
Chris Pratt
    Guardians of the Galaxy
Vin Diesel
    Guardians of the Galaxy
    The Fate of the Furious
The Fate of the Furious
    Michelle Rodriguez
    Vin Diesel
  
```

Notice that you can make use of the *index()* function to convert a vertex label to an index for use in graph processing and the *label()* function to convert an index into a label for use in the context of the application (such as displaying).