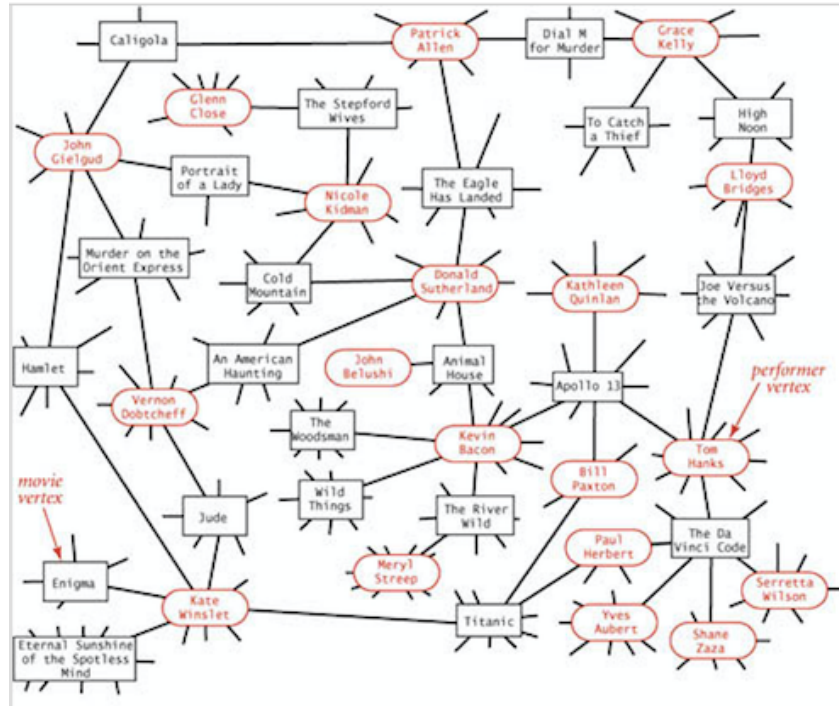


Movies, Actors, and their Connections

Introduction

In this project you will build a variation of *the Hollywood Graph* in which vertices are actors and edges connect actors that have appeared in a movie together. You will build a graph with movies and actors as vertices and edges connecting each movie to its actors. For example:



You will use the graph to query paths between a pair of actors or between a pair of movies, or an actor and a movie, to see how they are connected (if at all). This functionality will allow you to play the *Six Degrees of Kevin Bacon Game*, which was popular on college campuses about a decade ago. 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.

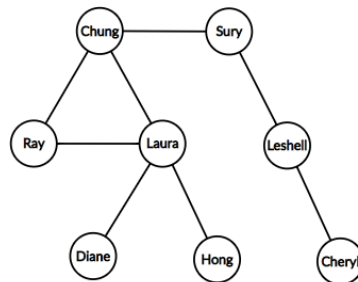
The small-world experiment [https://en.wikipedia.org/wiki/Small-world_experiment] comprised several experiments conducted by Stanley Milgram and other researchers examining the average path length for social networks of people in the United States. The research was groundbreaking in that it suggested 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 6 steps or so.

Quick Recap of Concepts (1)

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, Part 1: Creating and displaying a simple graph

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

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

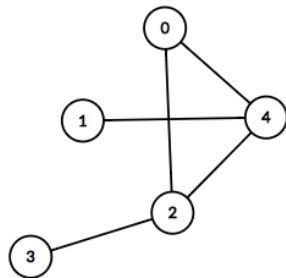
//OVERLOADED OUTPUT OPERATOR
 friend std::ostream& operator<< (std::ostream &out, const Graph &g);

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

Question:

- 1) Look at the member variables of Graph (in the *Graph.h* file). What representation is being used for the graph in the code provided, Adjacency Matrix or Adjacency List?

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*" (the starter code you downloaded). 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() {

}
```

The name of your graph object must be a single word consisting of your 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: 5 vertices, 5 edges
=====
0 --> 2    4
1 --> 4
2 --> 0    3    4
3 --> 2
4 --> 0    1    2
```

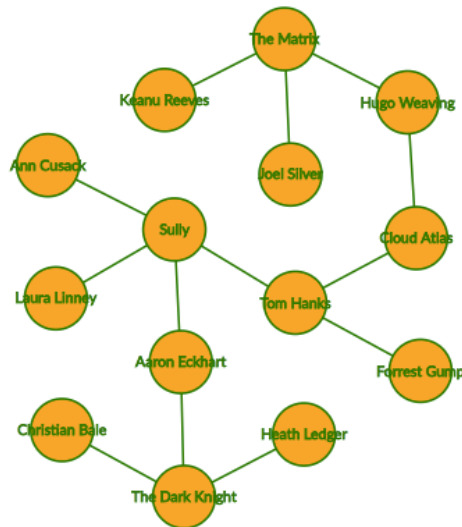
You will need to iterate through the vector of vertices and for each, retrieve the set of adjacent vertices (neighbors). [For a reminder about vectors and sets: <http://www.cplusplus.com/reference/stl/>]

Questions

- 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 don't we need a variable to keep track of the number of vertices? How can we quickly know the number of vertices in the graph?
- 2) What is the *big-O* time for displaying the graph (the overloaded output operator)?
- 3) What would be the *big-O* time for displaying the graph (overloaded output operator) if a matrix representation had been used?


Programming Exercise, Part 2: A Labeled Graph

For this project, 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:

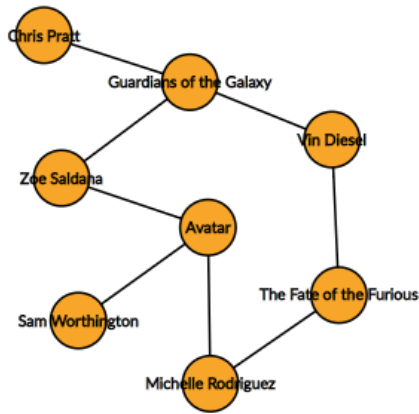
| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|--|---------------------|---|---|---|---|-------|---|---|-----|---|---|-------|---|---|---|---|------|---|-------|---|-----|---|-------|---|-------|---|
|  | <table><tr><td>0</td><td>→</td><td>1</td></tr><tr><td>1</td><td>→</td><td>0 2 3</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 take a look at it so that you understand how the graph is being represented and handled.

Questions:

4. What new variables were added to *LabeledGraph* that were not in *Graph* and what do you think they are used for?
5. 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 3. 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 *drzavalalabeledgraph*.

TASK 4. Just like in the previous exercise, in order for the graph to be displayed, you will need to 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
  
```

Programming Exercise, Part 3: Querying movies and their performers

Download the input files *movies_short.txt* and *movies.txt*, which contain information from the Internet Movie Database (IMDB: www.imdb.com) and consist of lines listing a movie name followed by a list of the performers in the movie. The file *movies_short.txt* has a snapshot of *movies.txt* so that you can work with a small file while testing. The file *movies.txt* has the complete list of movies and performers.

TASK 5. Add a constructor to *LabeledGraph* so clients can build a graph from an input file (with the format of the input file provided). You will need to modify both, the header and the implementation files (*LabeledGraph.h* and *LabeledGraph.cxx*). The name of the input file should be passed as parameter to the constructor so that a *LabeledGraph* object can be created using:

```
LabeledGraph g("movies_short.txt");
```

TASK 6. Write a client program, *LabeledGraphQueryClient.cpp*, to take queries from standard input. The user enters a vertex label and gets the list of vertices adjacent to that vertex (the neighbors). When you run the program, if you type the name of an actor, you should see the list of the movies in the database in which that actor appeared, or if you type the name of a movie, you should see the list of actors that appear in that movie.

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).

Programming Exercise, Part 4: The Kevin Bacon Game

In the next assignment