Assignment 4 README Six Degrees of Kevin Bacon

Checkpoint deadline: Friday, March 3, 11:59pm
Final deadline: Monday, March 13, 11:59pm
Survey and Consent Form deadline: Saturday, March 18

Assignment Overview

In this assignment you will:

Part 0: Survey and Consent Form

Part 1: Checkpoint: Pathfinder

- You will write a program to play (and win) the generalized Kevin Bacon trivia game. Your program will take as input ANY two actors/actresses and find the shortest sequence of shared movies between them. Part of this program (unweighted shortest path) is due at the checkpoint deadline.

Part 2: Final Submission: Pathfinder (Complete) + Actor Connections

- You will complete the weighted shortest path version of Pathfinder. You will also write a program that starts with one actor in the graph and adds new movies step-by-step to find the earliest movie year to connect the start actor to the destination actor.

Part 3: Final Submission: Additional Extension

- You will write a program that will compute some other interesting statistic or action on a different graph of your choosing. Particularly complex extensions will earn a **star point**

"Six Degrees of Kevin Bacon" Overview

"Six Degrees of Kevin Bacon" is a parlor game based on the "six degrees of separation" concept, which posits that any two people on Earth are six or fewer acquaintance-links apart. That idea eventually morphed into this parlor game where movie buffs challenge each other to find the shortest path between an arbitrary actor and prolific Hollywood actor Kevin Bacon. It rests on the assumption that anyone involved in

the Hollywood film industry can be linked through their film roles to Bacon within six steps. The game requires a group of players to try to connect any such individual to Kevin Bacon as quickly as possible and in as few links as possible. In the first part of the assignment, you will implement the generalized version of the Kevin Bacon game.

Getting Started (Important Notes)

Files Provided on Vocareum:

1. **`Makefile`** - which you will have to modify as you add source code
2. **`ActorGraph.h/cpp`** - contains starter code to read the movie_casts.tsv .
3. **`.gitignore`** - allows only certain files to be added to the repo

Files Provided on ieng6:

All these files are in **`/home/linux/ieng6/cs100w/public/pa4`**

1. **`movie_casts.tsv -`** The database that contains the majority of actors/ actresses found in IMDb. See the notes below for more details.
2. **`test_pairs.tsv`** - Text file containing the pairs of actors to find paths/connections(details explained later)
3. **`out_paths_unweighted.tsv`** - Output file generated by Pathfinder that stores the results from finding the unweighted shortest path between two actors in **`test_pairs.tsv`**
4. **`out_paths_weighted.tsv`** - Output file generated by Pathfinder that stores the results from finding the weighted shortest path between two actors in **`test_pairs.tsv`**
5. **`refpathfinder`** - solution executable that implements Pathfinder
6. **`refactorConnections`** - solution executable that implements ActorConnections
7. **`out_connections_bfs.tsv -`** Output file generated by ActorConnections on **`test_pairs.tsv`**
8. **`pair.tsv`** - Text file that contains100 pair of actors you can use for testing.
9. **`/home/linux/ieng6/cs100w/public/pa4/tsv/submissionScriptGraphs -`** This directory contains the input files we use in the submission scripts. It has a Readme file that describes these tsv files. All these are smaller versions of **`movie_casts.tsv and test_pairs.tsv`**. They are described here as well.
10. **`/home/linux/ieng6/cs100w/public/pa4/SocialNetworks -`** This directory contain some datasets that you may use for your extension. It has a readme that describes the dataset

IMPORTANT Notes about movie_casts.tsv:

We have provided you a tab-separated file **`movie_casts.tsv`** that contains the majority of actors/actresses found in IMDb and the movies they have played in. Specifically, the file looks like this ("<TAB>" denotes a single tab character):

```
Actor/Actress<TAB>Movie<TAB>Year
50 CENT<TAB>BEEF<TAB>2003
50 CENT<TAB>BEFORE I SELF DESTRUCT<TAB>2009
50 CENT<TAB>THE MC: WHY WE DO IT<TAB>2005
```

```
50 CENT<TAB>CAUGHT IN THE CROSSFIRE<TAB>2010
50 CENT<TAB>THE FROZEN GROUND<TAB>2013
50 CENT<TAB>BEEF III<TAB>2005
50 CENT<TAB>LAST VEGAS<TAB>2013
50 CENT<TAB>GUN<TAB>2010
...
```

a) The first column contains the name of the actor/actress, the second column contains the name of a movie they played in, and the last column contains the year the movie was made.

b) Each line defines a single actor→movie relationship in this manner (except for the first line, which is the header). You may assume that actor→movie relationships will be grouped by actor name, but do not assume they will be sorted.

c) Note that multiple movies made in different years can have the same name, so use movie year as well as title when checking if two are the same.

d) Some actors have a "(I)" appended to their name - so "Kevin Bacon" is really "Kevin Bacon (I)". **Make sure you DO NOT format the names of actors or movies beyond what is given in the tab-separated input file.** In other words, each actor's name should be taken exactly as the actor's name appears in the **`movie_casts.tsv`** file: you do not have to (and should not) mess with it. During grading, the actor's name in the test file will match the actor's name in the **`movie_casts.tsv`** file.

Note: The **`movie_casts.tsv`** file is pretty big, so we decided to just put all of the tsv files in our public folder on ieng6 (**`/home/linux/ieng6/cs100w/public/pa4/tsv`**) This means you will NOT get these files in the starter code given to you in your Github repo. You should also avoid pushing the tsv files, which should be simple because we provide a **`.gitignore`** file that won't allow you to add these files to your repo. PLEASE DO NOT DELETE THE **`.gitignore`** FILE. If there is some specific file you want to add that is not allowed by the **`.gitignore`** file, simply open the **`.gitignore`** file in a text editor and add a new line at the end with the syntax **`!fileName`** where **`fileName`** is the name of the file you want to add.

We have provided reference solutions for you to use to see what the expected functionality is on various test cases you may want to check. The reference solution can be found in our public folder on ieng6 (**`/home/linux/ieng6/cs100w/public/pa4`**)

Part 0: End of Quarter Survey (complete AFTER filling out CAPEs and finishing most of PA4) -- Deadline: Saturday, March 18 11:59 pm

The link to the survey will be posted after the Checkpoint deadline and the midterm, likely at the end of week 9 or start of week 10.

Consent form: We have been studying different ways to try to improve the experience of students in CSE 100. For one point on this assignment, fill out this consent form indicating whether you are willing to have your data from this course included in this study. We are not doing interviews this quarter, so if you agree to the interview, you might be contacted in a future quarter. You will receive the point for filling out the consent form, whether or not you agree to have your data included.

End of quarter survey: For another point on this assignment, fill out this end of quarter survey. It will ask you whether or not you have filled out your CAPEs, so please fill those out first, and answer the question honestly (don't cheat).

Checkpoint Instructions --- Deadline: Friday, March 3

0. Design and implement classes to support a graph implementation for the actor-movie relationships needed in this assignment.

In order to complete the rest of the assignment, you must first design your graph structure and implement the necessary classes. In your graph, each actor/actress will define a single node. Two nodes (i.e., actors) will be connected by an undirected edge if the corresponding actors played in the same movie. Multiple undirected edges can exist between the same two nodes (which would imply that the two actors played in multiple movies together).

Implementation Checklist:

- Review ActorGraph.cpp. This contains starter code to read the movie_casts.tsv file (the code open a file and parses the actor/movie/year from each line). For the implementations below, you may have to create separate .cpp files for your different classes based on your design.
- Design/Implement your node objects (actors). What information does your node need to contain?
- Design/Implement your "edges." How will you connect actors (nodes), relationships (edges), and movies to each other that allows efficient traversal of the graph without needlessly copying whole objects around? Do you want to have a data structure for edges or merely represent them as connections between two nodes? Pointers and/or vector indices might come in handy...
- Check to make sure you did NOT use any pre-built data structures, like the Boost Graph Library (BGL), besides what is provided in the C++ STL data structures.
- Test your graph implementation: load the movie_casts.tsv file, you should expect to find 11,794 actors or nodes, 14,252 movies, and 4,016,412 directed edges. Note: if we implement our graph with directed edges, every undirected edge will be represented by two directed edges.
- In a file called Report.pdf, place the answers to the following under a header called "Graph Design Analysis":
  - Describe the implementation of your graph structure (what new classes you wrote, what data structures you used, etc)
  - Describe *why* you chose to implement your graph structure this way (Is your design clean and easy to understand? what are you optimizing?, etc)
  - NOTE: If at any point your design changes, you MUST update Report.pdf to reflect this. If your report.pdf does not reflect your final implantation, you may lose points.
  - Report.pdf will not be graded in the checkpoint and will be graded only in the final submission.

---

1. Write a program called pathfinder (in pathfinder.cpp) to find the shortest path from one actor to another actor through shared movies.

Implementation Checklist:

- Implement pathfinder to work on an unweighted graph (see ./pathfinder Details below for more information)
- Read the grading notes below
- Read the "Running Time" Notes at the bottom of the PA

./pathfinder Details

Your program should be called like this (see detailed explanation of arguments below):

```
> ./pathfinder movie_casts.tsv u test_pairs.tsv out_paths_unweighted.tsv
```

where **test_pairs.tsv** contains:

```
Actor1/Actress1 Actor2/Actress2
BACON, KEVIN (I)<TAB>HOUNSOU, DJIMON
BACON, KEVIN (I)<TAB>KIDMAN, NICOLE
BACON, KEVIN (I)<TAB>WILLIS, BRUCE
BACON, KEVIN (I)<TAB>GIAMATTI, PAUL
HOUNSOU, DJIMON<TAB>50 CENT
```

and your program produces an output file **out_paths_unweighted.tsv** containing the following (although the particular movies may not match, the total path weights should match your output):

```
(actor)--[movie#@year]-->(actor)--...
(BACON, KEVIN (I))--[ELEPHANT WHITE#@2011]-->(HOUNSOU, DJIMON)
(BACON, KEVIN (I))--[SUPER#@2010]-->(MCKAY, COLE S.)--[FAR AND AWAY#@1992]-->(KIDMAN,
NICOLE)
(BACON, KEVIN (I))--[SUPER#@2010]-->(MORENO, DARCEL WHITE)--[LAY THE FAVORITE#@2012]-
->(WILLIS, BRUCE)
(BACON, KEVIN (I))--[A FEW GOOD MEN#@1992]-->(MOORE, DEMI)--[DECONSTRUCTING
HARRY#@1997]-->(GIAMATTI, PAUL)
(HOUNSOU, DJIMON)--[IN AMERICA#@2002]-->(MARTINEZ, ADRIAN (I))--[MORNING
GLORY#@2010]-->(50 CENT)
```

./pathfinder will take 4 command-line arguments:

1. Name of text file containing the tab-delimited movie casts (such as movie_casts.tsv).

   (movie_casts.tsv is very large, 6.4M. We have provided the smaller graphs that our submission scripts use for testing in **/home/linux/ieng6/cs100w/public/pa4/tsv/submissionScriptGraphs**)
2. Lower-case character **u** or **w**

   u -- builds the graph with unweighted edges

   w -- builds the graph with weighted edges (*you do not need to implement yet for checkpoint, but you will need to implement in final submission*)
3. Name of text file containing the pairs of actors to find paths

   First line in the file is a header, and each row contains the names of the two actors separated by a single tab character
4. Name of output text file

   Pathfinder will create a new file to store the results from finding the shortest path between two actors

   First line of the file is a header, and each row contains the paths for the corresponding pair of actors

and input pairs file (in the same order). Each path will be formatted as follows: `(<actor name>)--[<movie title>#@<movie year>]-->(<actor name>)--[<movie title>#@<movie year>]-->(<actor name>)`....etc where the movie listed between each pair of actors is one where they both had a role.

Grading Notes:

1. For the checkpoint, you are only required to have the unweighted portion of pathfinder working i.e. we will test your implementation with all 4 arguments except the second which we will always insert as a u
2. The specific path your pathfinder program outputs may be different than from reference solution. As long as the total path weights are the same, then you are fine.
3. Complete pathfinder is due at final submission, so if you don't get the "unweighted edges" version working for the checkpoint, *you must get it working by the final submission*.
4. You may ONLY use C++ STL data structures and NOT the Boost Graph LIbrary (BGL).
5. We have provided the test files that we use in our submission scripts in `/home/linux/ieng6/cs100w/public/pa4/tsv/submissionScriptGraphs`. Once you get it working on these small graphs, test them on larger graphs.

Final Submission --- Deadline: Monday, March 13

1. Complete your `pathfinder` program by implementing the "weighted edges" version of your program, where edges will have a weight equal to the age of the movie (because we will want to choose newer movies over older movies when connecting two actors).

Implementation Checklist:

- Make pathfinder work for weighted graphs (see below for details)
- Read the "Running Time" Notes at the bottom of the PA

If we are defining an edge between two actors that played in a movie made in year Y, then the weight of that edge will be:

```
weight = 1 + (2015 - Y)
```

Note that we are using 2015 instead of 2017, which is because the dataset only contains movies released in 2015 and earlier. Don't accidentally use 2017!

Example:

```
> ./pathfinder movie_casts.tsv w test_pairs.tsv out_paths_weighted.tsv
```

should produce an output file **`out_paths_weighted.tsv`** containing the following (although the particular movies may not match, the total path weights should match your output):

```
(actor)--[movie#@year]-->(actor)--...
(BACON, KEVIN (I))--[ELEPHANT WHITE#@2011]-->(HOUNSOU, DJIMON)
(BACON, KEVIN (I))--[R.I.P.D.#@2013]-->(HUSS, TOBY (I))--[LITTLE BOY#@2015]-->
(CHAPLIN, BEN)--[CINDERELLA#@2015]-->(MARTIN, BARRIE (II))--[PADDINGTON#@2014]-->
(KIDMAN, NICOLE)
(BACON, KEVIN (I))--[R.I.P.D.#@2013]-->(BELTRAN, JONNY)--[THE WEDDING RINGER#@2015]--
>(ROGERS, MIMI (I))--[CAPTIVE#@2015]-->(WILLIS, BRUCE)
(BACON, KEVIN (I))--[R.I.P.D.#@2013]-->(HOWARD, ROSEMARY (II))--[THE AMAZING SPIDER-
MAN 2#@2014]-->(GIAMATTI, PAUL)
(HOUNSOU, DJIMON)--[THE VATICAN TAPES#@2015]-->(SCOTT, DOUGRAY)--[TAKEN 3#@2014]-->
(HARVEY, DON (I))--[THE PRINCE#@2014]-->(50 CENT)
```

The specific path your pathfinder program outputs may be different than from reference solution. As long as the total path weights are the same, then you are fine.

To efficiently implement Dijkstra's algorithm for shortest path in a weighted graph, you should make use of a priority queue. You can implement your own, or use the STL C++ implementation: http://www.cplusplus.com/reference/queue/priority_queue/. Note that it does not support an **`update_priority`** operation (how can you get around that?). Think about what happens if you insert the same key twice into the heap, but with a lower priority. Which one gets popped first? When you pop a key-priority pair, how do you know if it is valid/up-to-date or not?

Actor Connections

2. Implement a program called `actorconnections`. `actorconnections` should answer the following query: for every actor pair (X, Y) in the given list: "After which year did actors X and Y become *connected*?" By *connected*, we mean that there exists a path between actors X and Y in the equivalent movie graph (similar to that constructed in Part 1) with the exception that the movie graph under consideration only includes movies that were made until (i.e., before and including) a certain year.

`actorconnections` will give the option to use two different approaches to solving the actor-connections problem: (1) using the graph structure itself, with BFS over particular edges in the graph (i.e. those up to a certain year) to determine whether two actors are connected and (2) without using the graph, but instead using the Disjont Set ADT that we will discuss in class.

Implementation Checklist:

- Implement actor connections using **the graph itself (and BFS):** To answer queries about the connection between actor pairs using BFS, we recommend you start with an empty graph containing only actor names and incrementally add movies in increasing order of the year of the movie. Every time you add a new set of movies made in a specific year, actors that were not connected before may become connected, which can be determined by running BFS on the updated graph.
- Implement actor connections **without using the graph, instead using the Disjoint Set ADT:** Alternatively, the disjoint-set (i.e., "union-find") data structure allows you to keep track of all

connected sets of actors without maintaining the corresponding graph structure. You might still consider adding movies incrementally, and if a movie creates a path between two actors that were not connected before, two disjoint sets would be merged into a single set in your union-find data structure. You should be able to then query your data structure about the connectivity of any specific actor pairs. The performance of your implementation will naturally depend on the efficiency of your Union-Find data structure. We will go over these topics in lecture and discussion as well. Your implementation MUST be very efficient.

- Review the edge cases checklist below
- Test against refactorconnections (see Reference Solution below)
- Read the "Running Time" Notes at the bottom of the PA

./actorconnections Details

```
> ./actorconnections movie_casts.tsv test_pairs.tsv out_connections_bfs.tsv ufind
```

should run your code (using the union-find algorithm) to produce an output file **`out_connections_bfs.tsv`** containing the following:

```
Actor1<TAB>Actor2<TAB>Year
BACON, KEVIN (I)<TAB>HOUNSOU, DJIMON<TAB>1992
BACON, KEVIN (I)<TAB>KIDMAN, NICOLE<TAB>1991
BACON, KEVIN (I)<TAB>WILLIS, BRUCE<TAB>1990
BACON, KEVIN (I)<TAB>GIAMATTI, PAUL<TAB>1992
HOUNSOU, DJIMON<TAB>50 CENT<TAB>2003
```

`actorconnections` will take 4 command-line arguments:

1. The first argument is the name of a text file containing the movie casts in the same format as **`movie_casts.tsv`**. **Again, this file is quite large (6.4M), so you should create smaller versions to test your implementation as a first step.** We have provided the smaller graphs that our submission scripts use for testing in **`/home/linux/ieng6/cs100w/public/pa4/tsv/submissionScriptGraphs.`** Once you get it working on these graphs, test it on larger graphs and movie_casts.tsv.
2. The second argument is the name of a text file containing the names of actor pairs on each line separated, with the two actor names are tab-separated (same format as **`test_pairs.tsv`**).
3. The third argument is the name of your output text file, which should contain in each line an actor pair followed by the year (tab-separated) after which the corresponding actor pair became connected (you will do all actor pairs specified in the file from step 2, one on each line). If two actors are never connected or if one or both of the actors is not in the movie cast file given to you, simply append 9999 in the corresponding line of the output file. To further clarify, if the second argument was a file containing the actor pair "BLANCHETT, CATE" and "REEVES, KEANU" and they only became connected after adding a movie made in 1997, your program should output the actor pair and 1997 in their line of the output file. If they never became connected even after adding all the movies from in the movie cast file to your graph, you should output 9999 on that line.
4. The fourth argument should be specified as either **`bfs`** or **`ufind`**. This option determines which algorithm will be used in your program. If the fourth argument is not given, by default your algorithm should use the union-find data structure (i.e., the equivalent of specifying **`ufind`** as the fourth argument). We will test your code with both flags.

Edge Cases Checklist:

- We will not test you on the corner case where both the actors are the same. You can handle it however you want.

Reference Solution:

We have made a reference solution available on ieng6 in the following location:

```
/home/linux/ieng6/cs100w/public/pa4/refactorconnections
/home/linux/ieng6/cs100w/public/pa4/refpathfinder
```

The usage for running union find is:

```
./refactorconnections movie_cast_file.tsv pair_file.tsv output_file.tsv ufind
```

OR:

```
./refactorconnections movie_cast_file.tsv pair_file.tsv output_file.tsv
```

The usage for running the BFS implementation is:

```
./refactorconnections movie_cast_file.tsv pair_file.tsv output_file.tsv bfs
```

Actorconnections running time comparison (Report.pdf)

3. Compare the run times of each implementation on a file containing actor pair pairs that you generate yourself. See how the run times compare when you repeat the same query multiple times. The file should be in the same format as test_pairs.tsv.

Implementation Checklist:

- Calculate runtimes of each implementation. The first argument should be **`movie_casts.tsv`**.
- In your input file (specified as the second argument), have the same actor pair appear 100 times and calculate the time it took to answer all 100 queries using your BFS implementation and then using your union-find implementation. For your code, you can use the timing classes given to you in PA2.
- In your input file (specified as the second argument), have 100 different actor pairs and calculate the time it took to answer all 100 queries using your BFS implementation and then using your union-find implementation. For your code, you can use the timing classes given to you in PA2. **`pair.tsv`** has 100 different actor pairs.
- In the same Report.pdf where you explained the class design (**see part 0 of the checkpoint for more details**) , place the answers to the following questions under a header called "Actor connections running time":
    1. Which implementation is better and by how much?
    2. When does the union-find data structure significantly outperform BFS (if at all)?
    3. What arguments can you provide to support your observations?
- Do not forget that the Report.pdf will contain answers to questions on the class design (see part0 of the checkpoint) and the above questions.

Grading

The grading breakdown is as follows: This assignment is meant to be open-ended, so you may define classes/methods/data-structures how ever you wish (i.e., you can rename **`ActorGraph.h/cpp`** to what ever you want). We will only grade your assignment based on the correctness of your **`pathfinder`** and **`actorconnections`** output as well as on the correctness of your extension and its write-up. We will develop our own input graph files to test all edge cases - at a minimum, these graphs will contain at least 2 nodes and 1 movie. **All testing will be performed on connected graphs**, which are graphs that contain at least one path between all pairs of nodes. **`pathfinder`** will always be tested with pairs of unique nodes as input (i.e., no self-paths). The assignment is out of 35 points and the breakdown is as follows:

1. **5 points** for unweighted **`pathfinder`** correctness at CHECKPOINT. To receive **any** points, your program must compile and output some path (in the correct format) for each input pair on a connected graph.
2. **8 points** for the complete **`pathfinder`** correctness at FINAL SUBMISSION. Note that this might include a test of unweighted as well as weighted. To receive **any** points, your program must compile and output some path (in the correct format) for each input pair on a connected graph. 1 point will go towards managing memory correctly (i.e., no memory leaks).
3. **7 points** for **`actorconnections`** correctness at FINAL SUBMISSION. We will run your program on a number of graph files, the largest of which will be the provided **`movie_casts.tsv`** file. For each input file, we will check that your output set of movie years is correct for your BFS implementation and union-find data structure. 2 points are for your analysis of how the run time of the two implementations compare.
4. **2 points** for the Actor connections running time analysis in **`Report.pdf`**
5. **4 points** for an extension of your choosing
6. **7 points** for style and class design (including the information in the report). That is, bad style can lose you up to 20% of your final grade. Stick to the Minimum Style Guide
7. **2 points** for Survey (and CAPEs). See Part 0

Format

We are giving very specific instructions on how to format your programs' output because our auto-grader will parse your output in these formats, and **any deviation from these exact formats will cause the autograder to take off points**. There will be no special attention given to submissions that do not output results in the correct format. Although we will still give partial credit to the correctness of your results, if you do not follow the exact formatting described here, you are at risk of losing all the points for that portion of the assignment. **NO EXCEPTIONS**.