CSE 100 PA 4: Six Degrees of Kevin Bacon

Use all of your reading, planning, inquiring, designing, implementing, debugging, runtime analyzing, spacetime analyzing, styling skills here. Read all provided comments and consider all input files. Style guidelines apply as usual.

Abstract

"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 wherein 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 any individual involved in the Hollywood, California, film industry can be linked through his or her film roles to Kevin 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. It can also be described as a trivia game based on the concept of the small world phenomenon.

In this assignment, you will implement a program called pathfinder to play and win the generalized Kevin Bacon trivia game. You will write another 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.

To explore degrees of separation between Hollywood actors who act in the same movies, the graph-based data structures and algorithms to explore large graphs will be designed by you.

Part 1: Checkpoint: Pathfinder (Unweighted)

Due on Friday, Week 4, August 26th, 2016 (No slip day)

Files Required:

1. **pathfinder.cpp**
2. **Makefile** (supporting at least **make pathfinder**)
3. **Any additional files needed to execute Part 1**

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
...
```

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. 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.

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. Also note that 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.

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). Once you 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 that if we implement our graph with directed edges, every undirected edge will be represented by two directed edges. You may NOT use any pre-built data structures, like the Boost Graph Library (BGL), besides what is provided in the C++ STL data structures.

For this part of PA4, you will write a program called **`pathfinder`** (in **`pathfinder.cpp`**) to find paths between actors. It 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. This file is quite large (6.4M), so you should create smaller versions to test your implementation as a first step.
2. The second argument is a lower-case character equal to u or w - u means "build the graph with unweighted edges", while w means "build the graph with weighted edges" (where weights are computed with the formula described later).
3. The third argument is the name of a text file containing the pairs of actors for which you will find paths. The first line of the file is a header, and each row contains the names of two actors separated by a single tab character.
4. The fourth argument is the name for your output text file, which will contain the shortest path between each pair of actors given in the input pairs file in argument 3. The first line of the output will be a header, and each row will contain a path for the corresponding pair of actors in the 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. Note that any given pair of actors may have played in multiple movies together (like Matt Damon and Ben Affleck), and if we are interested in a shortest weighted path, you must display the particular movie between each pair of actors that yields the minimum total path weight when combined with all other movies in the path.

Examples of the input pairs and output paths files are given in **test_pairs.tsv** and **out_paths_unweighted.tsv**, respectively. So calling your program with:

```
> ./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
```

should produce 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)
```

**** For the CHECKPOINT submission (worth 5 of the 30 total points on this assignment) ** you are only required to have the unweighted portion of pathfinder working.** This means we will test your program with all 4 arguments, except the second argument will always be a **u**. The weighted portion will be graded on the FINAL submission This is all that will be graded for the checkpoint submission.

The **complete pathfinder** program (as described below) will be graded at the final submission, so even if you don't get the "unweighted edges" version of your program working for the checkpoint, **you still need to get the whole thing working** (weighted path find and unweighted) for the final submission.

Due on Friday, Week 4, August 26th, 2016 (No slip day)
Files Required:

1. **pathfinder.cpp**
2. **Makefile** (supporting at least **make pathfinder**)
3. **Any additional files needed to execute Part 1**

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

Due: Tuesday, Week 5, August 29th, 2016. (One slip day, but consider the date of the Final Exam)

Files Required:

1. **pathfinder.cpp**
2. **actorconnections.cpp**
3. **Makefile** (supporting at least **make pathfinder**, and **make actorconnections**)
4. **Report.pdf**
5. **Any additional files needed to execute Parts 1-3**

In this part, the first thing you will do is complete `pathfinder` by implementing the "weighted edges" version of your program (which is needed for `pathfinder` for the Final Submission). In this version of your program, you can treat unweighted edges as weighted edges with a weight of 1 (i.e., a "dummy" weight), while truly weighted 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). 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 2016, which is because the dataset only contains movies released in 2015 and earlier. **Don't accidentally use 2016!** Calling your program with:

```
> ./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)
```

In this part of the assignment, you will implement a program called **`actorconnections`**. For a given movie database and a list of actor pairs, the **`actorconnections`** program 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 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 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.**
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.

Your output text file should look like the following:

```
Actor1<TAB>Actor2<TAB>Year
BLANCHETT, CATE<TAB>REEVES, KEANU<TAB>1997
KNAPP, DANIEL<TAB>WILLIS, BRUCE<TAB>9999
…
```

Calling your program with:

```
> ./refactorconnections 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
```

We would like you to implement the **`actorconnections`** program using both BFS and union-find and allow the user to select between the two by specifying the appropriate value for the fourth argument to your executable:

1. **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.
2. **Union-Find:** 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 as well.

Once you have completed and tested both implementations, compare the runtime of each implementation on a file containing actor pair pairs that you generate yourself. The file should be in the same format as `test_pairs.tsv`. See how the run times compare when you repeat the same query multiple times. For example, 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. To time your code, you can use the timing classes given to you in PA3. Analyze the timing results for the two implementations and **write your analysis in the file Report.pdf.**

In addition, **answer the following questions in `Report.pdf`:**

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?

We have made a reference solution available on ieng6 in the following location:
```
/home/linux/ieng6/cs100v/public/pa4/refactorconnections
/home/linux/ieng6/cs100v/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
```

<u>Files Required</u>:

1. **pathfinder.cpp**
2. **actorconnections.cpp**
3. **Makefile** (supporting at least **make pathfinder**, and **make actorconnections**
4. **Report.pdf**
5. **Any additional files needed to execute Parts 1-3**

Formating

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**.

Grading

The grading breakdown is as follows: This assignment is meant to be open-ended, so you may define classes/methods/data-structures however you wish (i.e., you can rename **ActorGraph.hpp/cpp** to whatever you want). We will only grade your assignment based on the correctness of your **pathfinder** and **actorconnections** output. 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 30 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. **9 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. **9 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 runtime of the two implementations compare.
4. **2 points** for **Report.pdf** (see the end of "Part 2" for more information about what should be in the report)
5. **5 points** for style. That is, bad style can lose you up to nearly 16% of your final grade. Stick to the Minimum Style Guide

Running Time

With the `-O3` optimization flag, it should not be taking too long (5 seconds max) to run `pathfinder` (with <20 query paths) or `actorconnections` on the full `movie_casts.tsv` file. But we will predominantly test your program with smaller versions of this file (as should you) containing <100 nodes. This means you will not lose more than a point or two on each of the `pathfinder/actorconnections` segments if your program processes the small test files successfully but takes too long on the full file. As a rule of thumb, we expect that your program runs in **less than 2x the runtime of the reference**, so ensure that it does. Practically, this is because your code might time out if it takes far too long, but realistically, the purpose of this course is to implement fast algorithms, so within twice the time of the reference solution should be reasonable, if not expected.

In general, aim for the reference run times or better.

For Part 1:
We will still accept solutions that take up to 3 times the specified runtime. Anything more than that will receive a 4 point penalty

For Part 2:
We expect your solutions to run under 1 minute for any of the provided input files, the largest of which is the `movie_casts.tsv` database and the `pair.tsv` file with 100 actor pairs. The reference solution runs in a few seconds. No deductions for this part unless your solution takes a really long to run (say 2 to 3 minutes)

Hints

1. We have provided you some starter code on how to read the **`movie_casts.tsv`** file inside **`ActorGraph.cpp`**. All this code does is open a file and parse the actor/movie/year from each line. It is your responsibility to insert this information into your graph data structure and make sure edges between actors are properly defined.
2. We have also provided you the binary executable for our reference **`pathfinder`** implementation. You can use these to benchmark against your implementation. Note that this executable can only be run on ieng6.
3. Think about how you want your data structures laid out in a way that will help you solve all the problems in the assignment.
    a. Do you want to have a data structure for edges or represent them as connections?
    b. 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? Pointers and/or vector indices might come in handy...
4. 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?
5. Remember that a hash table is given to you as a std::unordered_set and a hash map is given to you as a std::unordered_map. You may NOT use any pre-built data structures, like the Boost Graph Library (BGL), besides what is provided in the C++ STL data structures.
6. If your implementation is slow compared to the reference program, first make sure you have compiled with the **`-O3`** flag (see section above). Otherwise, consider this: Given two programs with the same Big-O running time, the one that runs faster tends to be the most space-efficient. Below are some clarifications relating to this assignment:
    a. Say we were trying to optimize BFS traversal for **`pathfinder`**. Program A implements each adjacency list of outgoing edges (for each node) as an **`unordered_map<string, Edge>`** mapping connected actor_name -> outgoing edge pointer. Program B implements each adjacency list instead as a **`vector<Edge>`** (assume both have the same **`Edge`** class). Obviously program A would be faster if we needed to check if two nodes are directly connected by an edge in the graph, but BFS traversal only requires that we explore all outgoing edges for each node. Both programs are O(V+E) in terms of running time, but

program B will be faster for many reasons, mainly because it uses less memory. First of all, there are no extra strings (each actor's name could be stored *once* for each node, not copied every time a node is connected by an edge) and secondly, 25% of a hash table's capacity is reserved for "empty space" to reduce collisions. Using a `map` instead if an **`unordered_map`** would result in similar performance issues since a **`map`** is implemented as a Red-Black Tree, requiring extra information (such as node left/right/parent pointers) stored for each item in the adjacency list.

7. Compilation problems? Try having a target **`.o`** file for each **`hpp`** that corresponds to a class. For example:

```
ActorGraph.o: ActorGraph.hpp Movie.o ActorNode.o OtherClass.o
Movie.o: Movie.hpp
ActorNode.o: ActorNode.hpp
OtherClass.o:  OtherClass.hpp
```