CSE100 - Winter 2018

Programming Assignment 3

Checkpoint (Part 1) Deadline: 11:59 pm on Tuesday, 3/6 (not slip day eligible)
Final (Part 1-3) Deadline: 11:59 pm on Wednesday, 3/14 (slip day eligible)

Overview
In this project, you will be required to complete three tasks on graphs built from an actor-movie database. The first task is to implement a path finder, where you will have to find the shortest path between two actors. The second part is to implement a link predictor, which will work on the principle of triadic closure or common neighbors. The third part is to implement a awards ceremony inviter, where you throw out invites to actors who meet a certain threshold criterion. **Your first step is to *read these instructions carefully*.** We've given you a lot of advice here on how to succeed, please take it. And please don't post a question on piazza asking something which is already answered here.

Pair Programming Partner Instructions

Retrieving Starter Code

Part 1: Path Finder

Part 2: Link Prediction and Recommendation

Part 3: Award Ceremony Inviter

Academic Integrity and Honest Implementations

Submitting Your Checkpoint and Final PA

Getting Help

Pair Programming Partner Instructions

If you wish to work with a partner, please be sure to read the guidelines for Pair Programming in the syllabus carefully. Once you are both sure you can work together as a pair, please fill out this form.

NOTE: Even if you paired with a partner previously, you must **RESUBMIT** a partner pair form if you want to continue to pair for this assignment.

Should a relationship dissolve, you can fill out this form to request a break-up. But by doing so, you are agreeing to delete all the work you and your partner have completed. Any code in common between dissolved partnerships is a violation of the Academic Integrity Agreement.

Retrieving Starter Code

For PA3, you are given a tab-separated file movie_casts.tsv that contains a large number of actors/actresses found in IMDb as well as the movies in which they have played.

Note: The **movie_casts.tsv** file is pretty big, so we decided to just put the tsv files in our public folder on ieng6 (**/home/linux/ieng6/cs100w/public/pa3/tsv**). This means you will NOT get these files in the starter code given to you. If you want to have them in your home directory for your convenience, we would advise against copying them, as ieng6 accounts have relatively limited storage space. Instead, we would recommend creating a symbolic link to the directory:
ln -s /home/linux/ieng6/cs100w/public/pa3/tsv

In the provided tsv file, 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. 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
...
```

Note that multiple movies made in different years can have the same name, so use movie year as well as the title when checking if two are the same. Also, note that some actors have "(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) alter it. During grading, the actor's name in the test file will match the actor's name in the movie_casts.tsv file.

We have also 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

We have also provided you with reference binaries for all three parts

To retrieve these files, you need to log into your CSE 100 account. (Use either your lab account or your user account. Log in to a linux machine in the basement or ssh into the machine using the command “ssh *yourAccount*@ieng6.ucsd.edu”). When logging in, you will be prompted for a password. You should already have run ‘‘prep cs100w” by now.

Once you are logged in, open up a linux terminal and run the command “getprogram3 <directory>” where <directory> is the name of a folder that you want to be created to hold the starter code. After this command is run, you should see that the directory you specified has been created if it previously did not exist, and the directory should contain the starting files:

```
ActorGraph.cpp  Makefile            refawardsceremony  refpathfinder     sampleInputPart2
ActorGraph.hpp  MatrixMultiply.hpp  reflinkpredictor   sampleInputPart1
```

Part 1: Path Finder

This part of the project will be due at the checkpoint.

In the first part of the PA, you have two subtasks - finding shortest paths in an unweighted graph and then in a weighted graph.

For the unweighted version of your program, you can treat unweighted edges as weighted edges with a weight of 1 (i.e., a "dummy" weight). Your solution, for this part should therefore find the path with the smallest number of movies between two actors.

In the second subtask, you will complete pathfinder by implementing the "weighted edges" version of your program. 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 as follows:

weight = 1 + (2018 - Y). Note that any given pair of actors may have played in multiple movies together (like Matt Damon and Ben Affleck).

You must display the particular movie between each pair of actors that yields the shortest path when combined with all other movies in the path

As a part of these two tasks, 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).You may NOT use any pre-built data structures besides what is provided in the C++ STL data structures.

For this part of the assignment, 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 tab-delimited file containing the movie casts in the same format as movie_casts.tsv
    a. This file is quite large, so you should create smaller versions to test your implementation as a first step. We have also provided a smaller version in the tsv directory.
2. The second argument will be either **u** or **w** (a single lower-case letter)
    a. u means "build the graph with unweighted edges"
    b. w means "build the graph with weighted edges" (where weights are computed with the formula described later in this write-up)
3. The third argument is the name of a text file containing the pairs of actors for which you will find paths
    a. 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
    a. 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)
    b. 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

c. If no valid path exists, simply output an empty line for this pair

A command like
./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):

Consistent with PA2, we've not provided you with any reasonable tests. You should write many of your own. The grading script will not be sufficient for testing your code. **Tutors will not help you debug your code based on grading script output without clear evidence of substantial testing on your own part.** So if you have not written tests and come to the tutors for help, they will tell you to write tests before continuing to help you.

Part 1: Design Notes

With the -O3 optimization flag, it should not take long (15 seconds max) to run pathfinder (with <20 query paths) on the full movie_casts.tsv file

Your code must build with the make pathfinder command and must clean with the make clean command.

Think about how you want your data structures laid out in a way that will help you solve all the problems in the assignment

- Do you want to have a data structure for edges or represent them as connections?
- 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...
- You should do this planning BEFORE you start coding!

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 you can use the STL C++ priority_queue implementation
    - 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?

Part 2: Link Prediction and Recommending New Links

In the second part of the assignment, you will work on a task on network evolution! Particularly your task involves predicting future links in a social network and answering questions like “Given a snapshot of a social network (say Facebook) , can we infer which new interactions among its members are likely to occur in the near future? Moreover, can we recommend new friends for users?”

In this part of the assignment, you will predict whether two actors would act together in the future in an unweighted graph. (Feel free to get rid of the movie information for this part and the next). This part again will have two subparts - the first where you predict connections between actors who have collaborated in the past and the second where you predict new collaborations between actors. In each case, you will be given the name of an actor and will need to make predictions for that actor.

The idea behind the prediction as follows:
For every pair of actors, we compute the number of common neighbors they share (this is equivalent to the number of triangles the pair of actors would complete if there was a link between the actors (interaction)). This property is called triadic closure. You can learn more about this here.

The input for this part of this assignment is in the form of a text file, which has a header and then an actor for every line in the file. You will be required to create two output files:

1. The first would contain the predictions of the top 4 future interactions (top 4 actors who share the highest number of common neighbors with the given actor) with whom the given actor has already collaborated (there already exists a link between these two actors). The four top predicted actors should be printed on the same line and should be tab separated. Collisions should be resolved alphabetically. This file will also have to contain a header
2. The second would contain the predictions of the top 4 new collaborations (i.e. in the current state of the graph - there is no direct link between the two actors in consideration). Again, this will be based on the highest number of common neighbors. The four top predicted actors should again be printed on the same

line and should be tab separated. Collisions should be resolved alphabetically. This file will also have to contain a header

For this part of the assignment, you will write a program called linkpredictor(in linkpredictor.cpp).
It will take 4 command-line arguments:

1. The first argument is the name of a tab-delimited file containing the movie casts in the same format as movie_casts.tsv This file is quite large, 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 a header and then an actor for every line. This is the set of actors for whom you will be making the predictions.
3. The third argument is the name for your output text file which contains the predictions for the top 4 actors with whom the given actor has already collaborated. There should be four actors on each line and they should be tab separated.
4. The fourth argument is the name for your output text file which contains the predictions for the top 4 actors with whom the given actor has not collaborated. There should be four actors on each line and they should be tab separated.

Part 2: Design Notes

With the -O3 optimization flag, it should not take too long (30 seconds max) to run linkpredictor on the full movie_casts.tsv file with less than 20 actors for whom to make predictions.

Your code must build with the make linkpredictor command and must clean with the make clean command.

Most of the notes from part 1 still apply.

In addition, the number of common neighbors, can either be computed by:

1. using a 1/2 level BFS and then taking an intersection of the neighbors
2. using a matrix multiplication operation.

Note: As a part of the starter code, you have also been given starter code to perform matrix multiplication. You may choose to not use/ modify the given starter code.

Remember that in your adjacency matrix, a 1 at the (i,j) index indicates that there exists a connection between actor ‘i’ and actor ‘j’. So when the adjacency matrix is multiplied by itself, the (i,j) index in the resultant matrix is the product of the i’th row and j’th column of the adjacency matrix. The aforementioned row and column are indicative of the links to neighbours and therefore the product gives us the number of common neighbors. Work out a simple example to convince yourself!

Part 3: Award Ceremony Invitation

In the third part of the assignment, you will work on a task on graph decomposition! This task is again to be performed on the unweighted actor graph. Since this is the last part of your last programming assignment in this course you have decided to play host to an awards ceremony and your task involves inviting actors to the ceremony. **The invitation process though involves a constraint - every actor invited to the ceremony should know at least ‘k’ other actors who would be present at the awards ceremony (You have to keep them engaged!).**

The output for this part of the project, should be a text file containing the list of actors you would invite. Format your output so as to have one actor in a line and have them ordered alphabetically.

For this part of the assignment, you will write a program called awardsceremony (in awardsceremony.cpp).
It will take 4 command-line arguments:

1. The first argument is the name of a tab-delimited file containing the movie casts in the same format as movie_casts.tsv This file is quite large, so you should create smaller versions to test your implementation as a first step
2. The second argument is the positive integer value ‘k’
3. The third argument is the name of your output text file which contains the list of actors to be invited to the ceremony

Graph decompositions, like this task are performed on social networks to identify dense subgraphs and communities, evaluate collaborations in the networks as well as identifying influential nodes in a network.

Part 3: Design Notes

With the -O3 optimization flag, it should not take too long (20 seconds max) to run awardsceremony on the full movie_casts.tsv file for any particular positive value of k.

Your code must build with the make awardsceremony command and must clean with the make clean command.

You may find these resources extremely useful.
https://en.wikipedia.org/wiki/Degeneracy_(graph_theory)#k-Cores
http://vlado.fmf.uni-lj.si/pub/networks/doc/cores/cores.pdf

Most of the notes from part 1 and part 2 still apply.

Grading

The grading for part 1 of the project is out of 20 points, and is broken down into 3 categories:

- The code builds properly - if your code does not build and clean properly using your makefile, you will score 0 points.
- 4 points for no memory leak. Memory leaks are notoriously hard to debug if not caught immediately, so test your code frequently, and create checkpoints you can revert to regularly.
- 8 points for the unweighted path finder
- 8 points for the weighted path finder
- If you miss points on the checkpoint, you can gain ½ of them back at the final submission. You can get all the bonus points at the final submission, that does not need to be implemented by the checkpoint.

The code for part 2 of the project is out of 10 points, and is broken down into 3 categories:

- The code builds properly - if your code does not build and clean properly using your makefile, you will score 0 points.
- 1 points for no memory leak. The same warning about memory leaks applies.
- 5 points for the link predictor with existing connections
- 4 points for the link recommendations

The code for part 3 of the project is out of 10 points, and is broken down into 2 categories:

- The code builds properly - if your code does not build and clean properly using your makefile, you will score 0 points.
- 1 points for no memory leak. The same warning about memory leaks applies.
- 4 points if you get at least half the invitees correct
- All 9 points if you get all invitees correct

Run the grading scripts (grade_pa3_checkpoint.py and grade_pa3_post_check.py) to see a sample score. If your code is nondeterministic, then we cannot guarantee that the score you see will be the score you get. The final grading will use the same logical tests, but will be run on different .tsv files than the ones given to you.

Lastly, we will be running tests not seen in the grading script to look for the use of unauthorized libraries or data structures. In other words, if you try to solve this using another data structure from non-STL libraries or from the web and not build your own, you will receive a zero on the assignment and possibly face an Academic Integrity Violation.

Your files that will be collected for grading are: the Makefile, and any *.cpp and *.hpp files in your project's directory. Everything else will be ignored.

- Please note that you do not need to have your name or PID in the turn in files. In fact, by adding your name/PID to these files, they will be exposed to a 3rd party server (so feel free to remove them).

Be sure to test your code on ieng6. That's where it will be run and graded.

Submitting Your Checkpoint and Final PA

Submission command: bundlePA3
If you are submitting with a partner, make sure that you two have registered that you two are working together.

You can submit as many times as you like before the deadline: only your last time running will be counted. If you submit the assignment within 24 hours after the deadline (even a minute after the deadline) you will be charged a slip day. If it is more than 24 but within 48 hours, you will be charged 2 slip days. If you've already used some of your slip days for the previous PA, you won't be able to use those slip days now.

Academic Integrity and Honest Implementations

We will hand inspect, randomly, a percentage of all submissions and will use automated tools to look for plagiarism or deception. **Attempting to solve a problem by other than your own means will be treated as an Academic Integrity Violation.** This includes all the issues discussed in the Academic Integrity Agreement, but in addition, it covers deceptive implementations. For example, if you use a library (create a library object and just reroute calls

through that library object) rather than write your own code, that's seriously not okay and will be treated as dishonest work.

Getting Help

Tutors in the labs are there to help you debug. TA and Professor OH are dedicated to homework and/or PA conceptual questions, but they will not help with debugging (to ensure fairness and also so students have a clear space to ask conceptual questions). Questions about the intent of starter code can be posted on piazza. Please do not post your code to piazza either publicly or privately - debugging support comes from the tutors in the labs.

Format of your debugging help requests
At various times in the labs, the queue to get help can become rather long (all the more reason to start early). To ensure everyone can get help, we have a 5 minute debugging rule for tutors in that they are not supposed to spend more than 5 minutes with you before moving onto a new group. Please respect them and this rule by not begging them to stay longer as you're essentially asking them to NOT help the next group in exchange for helping you more.

5 minutes?!
Yes, 5 minutes. The job of tutors is to help you figure out the *next step in the debugging process*, not to debug for you. So this means, if you hit a segfault and wait for help from a tutor, the tutor is going to say "run your code in gdb, then run bt to get a backtrace." Then the tutor will leave as they have gotten you to the next step.

This means you should use your time with tutors effectively. Before asking for help, you will want to already have tried running your code in gbd (or valgrind, depending on the error). You should known roughly which line is causing the error and/or have a clear idea of the symptoms. When the tutor comes over, you should be able to say:
What you are trying to do. For example, "I'm working on Part 1 and am trying to structure my actor vertices properly."
What's the error. For example, "the code compiles correctly, but when I add an actor for a second time, it overwrites the adjacency list for that actor."
What you've done already. For example, "I added a method to print out the entire state of the graph with pointers and vector contents. I print it as each actor is added and it seems like a whole new vertex is made rather than an adjacency list being overwritten, but I can't seem to figure out why based on the code <here>.. What do you suggest I try next?"

Acknowledgements
Special Thanks to Jonathan Margoliash, Dylan McNamara, Niema Moshiri, Adrian Guthals, Christine Alvarado, and Paul Kube for creating the base on which this assignment is built.