

CSE 100 Fall 2018 // PA3

Checkpoint deadline 11:59pm on Thursday, **November 29** (No slip days)

Final submission deadline 11:59pm on Thursday, **December 6** (slip days allowed)

Table of Contents

[Table of Contents](#)

[Pair Programming Partner Instructions](#)

[Assignment Overview](#)

[Retrieving Starter Code](#)

[Checkpoint](#)

[Part 1 Path Finder](#)

[Final Submission](#)

[Part 2 Link Predictions and Recommending New Links](#)

[Part 3 Awards Ceremony Invitation](#)

[Grading Breakdown](#)

[Submitting Your Checkpoint and Final PA](#)

[Academic Integrity and Honest Implementations](#)

[Getting Help](#)

[Acknowledgements](#)

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. **You may only pair before the checkpoint deadline.** 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.

Assignment Overview

Checkpoint Pathfinder

Your program will take as input ANY two actors/actresses and find the shortest sequence of shared movies between them. This program, `pathfinder`, is due at the checkpoint deadline.

Final submission Link Prediction + Awards Ceremony Invitation

For this part you will tackle two problems. The first, `linkpredictor`, will predict and recommend new acting partnerships, based on their mutual co-stars. In `awardsceremony`, you will do some party planning for Hollywood's rich and famous.

Retrieving Starter Code

! IMPORTANT Starter code for this PA is very limited. You're in charge of designing and thoroughly testing¹ your own classes. We'll run `make <target>` and then black-box² test your 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. You are also given an `ActorGraph.h/cpp` class with starter code to read in this file. Starting from that you'll build your graph and solve the posed problems.

1 Retrieve starter code files: 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 cs100f`" 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.

¹ Remember though what Dijkstra said: "*Program testing can be used to show the presence of bugs, but never to show their absence!*"

² We'll just run your compiled executable and verify that for every given test case its output file(s) match what we expect. That is, our grader sees your executable as a black box: it doesn't know anything about its internals. The grader just makes sure the executable fully conforms to the specification.

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:

```
Makefile  create-submission-zip.sh  ref_solutions  sample_inputs  src  test
```

And `src` will contain the files:

```
ActorGraph.cpp  ActorGraph.hpp  MatrixMultiply.hpp  awardsceremony.cpp  linkpredictor.cpp  pathfinder.cpp
```

Under `ref_solutions` you'll find the executables **refpathfinder**, **reflinkpredictor** and **refawardsceremony**. These are reference solutions for Parts 1, 2 and 3, respectively, which you may run on `ieng6` to check the expected functionality and output.

The following make targets are **required**:

- 1. pathfinder** - generates an executable named `pathfinder` which solves the problem defined in Part 1. This target *must* be present in both the checkpoint and final submissions.
- 2. linkpredictor** - generates an executable named `linkpredictor` which solves the problem defined in Part 2. This target *must* be present in the final submission.
- 3. awardsceremony** - generates an executable named `awardsceremony` which solves the problem defined in Part 3. This target *must* be present in the final submission.

If you wish to enable optimization: run with the argument `type=opt`, e.g. `make <target> type=opt` or `make type=opt`. That will compile your code with the `-O3` flag.

Finally, as you'll see, there are files for each part that correspond to the required targets. These are essentially a blank canvas. Feel free to complete them as you see fit.

⚠ There is currently a bug in the Makefile where if you run two make commands in a row (e.g. `make pathfinder && make linkpredictor`), the second call to make may fail due to a linking bug. You can modify the Makefile to fix that, but a simple workaround is to call `make clean` before the second call to `make` - this works fine.



2 Retrieve tsv files: 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/cs100f/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/cs100f/public/pa3/tsv
```

IMPORTANT notes regarding `movie_casts.tsv`

We have provided you a tab-separated file **`movie_casts.tsv`** that contains the majority of actors and actresses found in IMDb and the movies in which they have played. Specifically, the file looks like this (<TAB> denotes a single tab character, i.e. `'\t'`):

```
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
...
```

- 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.**  **THIS IS VERY IMPORTANT**  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**. Not doing so is a common source of headaches encountered by previous students who solved this PA.
- 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**. During grading, the actor's name in the test file will match the actor's name in the `movie_casts.tsv` file.

DO NOT USE ANY OTHER executable names than the ones defined in your Makefile. Refer to the instructions for each part for the **required name of executables**.

Checkpoint

[20 points]

Part 1 Path Finder

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

Implementation Checklist

- ☐ Design and implement graph and **pathfinder** to work on **unweighted graphs**.
- ☐ Extend **pathfinder** to work on weighted graphs. Here you'll implement **Dijkstra's algorithm**.

man pathfinder

./pathfinder will take **four** command-line arguments **in this order**:

1. Name of text file containing the tab-delimited movie casts (e.g. **movie_casts.tsv**).

This file is quite large, so you should create smaller versions to test your implementation as a first step. We've also provided a smaller version (**movie_casts_small.tsv**) in the **tsv** folder.

2. Lower-case character **u** or **w**

u builds the graph with unweighted edges (weights are all 1).

w builds the graph with weighted edges (see the weight formula below).

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, e.g.:

```
Actor1      Actor2
CRYER, JON  TOUSSAINT, LORRAINE
FABIAN, AVA    CHAPMAN, MICHAEL (I)
DUNSMORE, ROSEMARY  GADON, SARAH
HICKOX, ANTHONY (I)  CRAWFORD, RACHAEL (I)
MCGILL, MICHAEL PATRICK  FYFE, JIM
...
```

4. Name of output text file

Pathfinder will create a new file to store the results from finding the shortest path between two actors. (Continued on next page.)

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 **must** be formatted **exactly** as follows:

```
(<actor name>)--[<movie title>#@<movie year>]-->(<actor name>)--[<movie title>#@<movie year>]-->...
```

....etc where the movie listed between each pair of actors is one where they *both* had a role.

./pathfinder sample usage: **unweighted** case

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 **lengths** 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)
```

Note you don't need to have a separate graph implementation for this. In the unweighted case we simply set all edge weights to 1.

./pathfinder sample usage: **weighted** case

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: $weight = 1 + (2018 - Y)$. That way we pick newer movies over older movies when connecting two actors. Note that any given pair of actors may have played in multiple movies together (like Matt Damon and Ben Affleck).

Running the following command:

```
> ./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)
(BACON, KEVIN (I))--[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))
(BACON, KEVIN (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)
```

Note 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](#). Note that it does *not* support an `update_priority` operation, so you'll have to find a way 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?

Reference solution

We included a reference solution in the starter code. The usage for running the reference solution is

```
> refpathfinder movie_casts.tsv u/w test_pairs.tsv out_paths.tsv
```

Where u/w should be either u – for unweighted shortest path – or w – for weighted shortest path.

Note The specific path your pathfinder program outputs may be different than the reference solution's. As long as the *total path lengths (for u) or weights (for w)* are the same, then you are fine.

Submission instructions

Efficiency requirement Your code must finish in under **2x** the time it took the reference solution or **15s**, whichever is greater. If your code times out you won't get any points for that particular test case.

Output format requirement We gave 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. 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!!!

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. We will call your make files with `type=opt` on grading to enable optimization when grading.

Think about how you want your data structures laid out in a way that will help you solve all the problems in the assignment. Ask yourself the following questions:

1. Do you want to have a data structure for edges or represent them as connections?
2. 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...
3. You should do this planning **BEFORE** you start coding!

4. Make sure to write extensive tests verifying your graph is being built correctly. Do the same for every public method you write.
5. Make sure you heed all instructions about the formatting of movie and actor names. This is a common mistake, so take the time to painstakingly compare your output *character by character* with the reference solution's.
6. To efficiently implement Dijkstra's algorithm for shortest path in a weighted graph, you should make use of a priority queue.
 - a. 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?
 - b. For the unweighted graph, you can find the shortest path with much simpler graph traversal algorithms presented in class. Do not try to implement Dijkstra before you've implemented BFS/DFS on your graph and verified it works.
7. You can implement your own, or you can use the STL C++ [priority queue](#) implementation.

⚠ The output files must have the exact name given by the user. Do NOT append an extension if the user doesn't specify one in the file name. Simply write to the given path in text mode.

All files must be turned in on Gradescope. Separate submissions will not be allowed under any circumstances. Your code must build with the **make pathfinder** command and clean with the **make clean** command.

Final Submission

[20 points]

Part 2 Link Predictions and Recommending New Links

Write a program called `linkpredictor` (in `linkpredictor.cpp`) that answers these questions:

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 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

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 is what's known as [triadic closure](#), which boils down to the number of common neighbors shared by two actors in the graph (or nodes more generally). After all, this is equivalent to the number of triangles the pair of actors would complete if there was a link between the actors (i.e. an *interaction*).

Implementation Checklist

- ☐ Implement **linkpredictor** to report the **top four future interactions** for an actor with respect to other actors with whom they *have already collaborated* in the past (i.e. a link exists between them already). These will be the *top four neighbors of the given actor with whom the given actor shares the highest number of common neighbors*.
- ☐ Extend **linkpredictor** to report the **top four new collaborations** for an actor, with respect to other actors with whom they *have NOT collaborated* in the past (i.e. a link DOES NOT exist between them already). These will be the *top four actors two hops away of the given actor who share the highest number of common neighbors with the given actor*.
- ☐ For both cases, make sure that the output file contains a header (refer to the reference solution's outputs for a reference header, but yours may be different), and that:
 - The four top predicted actors are *printed on the same line*.
 - The listed actors are **tab separated**. (Please make sure you're using `'\t'`, not spaces!)
 - *Collisions are resolved alphabetically* (i.e. in ascending lexicographical ordering).

man linkpredictor

`./linkpredictor` will take **four** command-line arguments **in this order**:

1. Name of text file containing the tab-delimited movie casts (e.g. `movie_casts.tsv`).

This file is quite large, so you should create smaller versions to test your implementation as a first step. We've also provided a smaller version (`movie_casts_small.tsv`) in the `tsv` folder.

2. Name of text file containing the actors for which to predict links

First line in the file is a header, and each row contains the names of the actors for which future interactions and new collaborations are to be predicted, e.g. head of `sample_inputs/part2.txt`:

```
Actors
50 CENT
AARON, CAROLINE
AARONS, BONNIE
...
```

3. Name of output text file for predictions of *future interactions*

`linkpredictor` will create a new file to store the results of finding the predictions for the **top four actors with whom the given actor has already collaborated**. There should be *four* actors on each line and they should be *tab separated*, e.g. head of output of `reflinkpredictor` for `sample_inputs/part2.txt`:

```
Actor1,Actor2,Actor3,Actor4
DE NIRO, ROBERT LEBLANC, ELTON HALEY, EMILY D. LEBLANC, CYNTHIA
JACKSON, SAMUEL L. HAWN, PHIL RIEHLE, RICHARD SARANDON, SUSAN
MARSHALL, GARRY ELIZONDO, HECTOR (I) MCGEE, JACK (I) MILLER, LARRY (I)
ROACH, MARTIN (I) PAUL, BRIAN (I) FITZPATRICK, RICHARD (I)PODHORA, ROMAN
...
```

4. Name of output text file for predictions of *new collaborations*

`linkpredictor` will create a new file to store the results of finding the predictions for the **top four actors with whom the given actor has not collaborated**. There should be *four* actors on each line and they should be *tab separated*, e.g. head of output of `reflinkpredictor` for `sample_inputs/part2.txt`:

```
Actor1,Actor2,Actor3,Actor4
CHREST, JOE MONTGOMERY, RITCHIE SARANDON, SUSAN JENSEN, DAVID (II)
GOLDBERG, WHOOP! DAVID, KEITH (I) FREEMAN, MORGAN (I) DE NIRO, ROBERT
ROCK, CHRIS (I) DIAZ, CAMERON COOPER, CHRIS (I) BOSTON, DAVID (IV)
BANKS, BOYD BERGSCHNEIDER, CONRAD ELDRIDGE, CRAIG RICHINGS, JULIAN
...
```

Reference solution

We included a reference solution in the starter code. The usage for running the reference solution is

```
> reflinkpredictor movie_casts.tsv actors.tsv future_interactions.tsv new_collaborations.tsv
```

Submission instructions

Efficiency requirement Your code must finish in under **2x** the time it took the reference solution or **30s**, whichever is greater. If your code times out you won't get any points for that particular test case.

Output format requirement We gave 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. 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!!!

With the `-O3` optimization flag, it should not take long (30 seconds max) to run `linkpredictor` (with `<20` actors) on the full `movie_casts.tsv` file. We will call your make files with `type=opt` to enable optimization when grading.

Most of the notes from *Part 1* still apply.

⚠ The output files must have the exact name given by the user. Do NOT append an extension if the user doesn't specify one in the file name. Simply write to the given path in text mode.

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

1. using a 1 or 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 (`MatrixMultiply.hpp`). **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!

(For another interesting property of adjacency matrix exponentiation, i.e. reachability, refer to this worksheet from CSE 100 Spring 2018: <https://goo.gl/ByT12t>.)

All files must be turned in on Gradescope. Separate submissions will not be allowed under any circumstances. Your code must build with the **make linkpredictor** command and clean with the **make clean** command.

Part 3 Awards Ceremony Invitation

Write a program called **awardsceremony** (in **awardsceremony.cpp**). Since this is the last part of your last programming assignment in this course, you've decided to play host to an awards ceremony and your task involves inviting actors to the party. 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 after all!)

This boils down to finding **k -cores**³ within a graph. Graph decompositions like this task are performed on social networks to identify dense subgraphs and communities, evaluate collaborations in the networks, as well as identify influential nodes in a network.

Implementation Checklist

- ❑ Implement **awardsceremony** to produce the guest list for your awards ceremony. The guest list must have a header, be sorted in increasing lexicographical order and fulfil the constraint that every invited actor knows at least k other actors who would be present at the ceremony.

man awardsceremony

./awardsceremony will take **three** command-line arguments **in this order**:

1. Name of text file containing the tab-delimited movie casts (e.g. **movie_casts.tsv**).

This file is quite large, so you should create smaller versions to test your implementation as a first step. We've also provided a smaller version (**movie_casts_small.tsv**) in the **tsv** folder.

2. The value k This will be a *positive integer* value.

3. Name of output text file for the guest list **awardsceremony** will create a new file to store the guest list. This file must have a header, and there should be one actor on each line, e.g. head of output of **refawardsceremony** for **movie_casts.tsv** and **k=300**:

```
Actor
ABELL, ALISTAIR
ABRAHAMS, DOUG
ACHESON, MARK
ADACHI, LEANNE
...
```

³ You may find these resources extremely useful: [https://en.wikipedia.org/wiki/Degeneracy_\(graph_theory\)#k-Cores](https://en.wikipedia.org/wiki/Degeneracy_(graph_theory)#k-Cores) & <http://vlado.fmf.uni-lj.si/pub/networks/doc/cores/cores.pdf>

Reference solution

We included a reference solution in the starter code. The usage for running the reference solution from the root directory of PA3 (save for passing the right paths):

```
> refawardsceremony movie_casts.tsv k guest_list.tsv
```

Where k should be replaced by a positive integer value.


Submission instructions

Efficiency requirement Your code must finish in under **2x** the time it took the reference solution or **20s**, whichever is greater. If your code times out you won't get any points for that particular test case.

Output format requirement We gave 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. 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!!!

With the `-O3` optimization flag, it should not take long (20 seconds max) to run `awardsceremony` (with <20 actors) on the full `movie_casts.tsv` file for *any* particular value of k . We will call your make files with `type=opt` to enable optimization when grading.

 The output files must have the exact name given by the user. Do NOT append an extension if the user doesn't specify one in the file name. Simply write to the given path in text mode.

Most of the notes from *Part 1* and *Part 2* still apply.

All files must be turned in on Gradescope. Separate submissions will not be allowed under any circumstances. Your code must build with the `make awardsceremony` command and clean with the `make clean` command.

Grading Breakdown

Restriction on non-STL Libraries If you try to solve this PA using any data structures from non-STL libraries, including but not limited to the Boost Graph Library (BGL), or from the web, and not build your own, you will receive a **zero** on the assignment and possibly face an **Academic Integrity Violation**.

[Checkpoint]

PART 1 The grading for Part 1 of the project is **out of 20 points**, and is broken down into three categories:

- The code builds properly - if your code does not build and clean properly using your makefile, you will score 0 points.
- **Memory leak:**
 - **2** 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.
- **Shortest path:**
 - **9** points for the unweighted path finder.
 - **9** points for the weighted path finder.

If you miss points on the checkpoint, you can gain $\frac{1}{2}$ of them back at the final submission.

[Final Submission]

PART 2 The code for Part 2 of the project is **out of 10 points**, and is broken down into three categories:

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

PART 3 The code for Part 3 of the project is **out of 10 points**, and is broken down into two categories.


- The code builds properly - if your code does not build and clean properly using your makefile, you will score 0 points.
- **Memory leak:**
 - **1** point for no memory leak. The same warning about memory leaks applies.
- **Awards ceremony invitees:**
 - At least half of the invitees correct: **4** points
 - All invitees correct: **9** points


If your score fluctuates on different runs, assume that's because a bug in your code is being uncovered at random. There are many sources of error, but in this PA in particular you may run into sneaky undefined behavior that'll make you go crazy. The following Piazza post may come in handy: [@519](#)

Submitting Your Checkpoint and Final PA

As in the last PA, there is a **PA3Checkpoint** assignment on gradescope. The **PA3Final Pathfinder** and **PA3Final LinksAndAwards** (link predictions/recommendations and awards ceremony) assignments will be opened up closer to the final deadline. Use the utility we've given you (`create-submission-zip.sh`) to create the submission bundle for both the checkpoint and final submissions.

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).

 **Make sure you submit the same zip file to PA3Final Pathfinder and PA3Final LinksAndAwards.** We will be grading them separately, but you need to hand in the same code. (We *will* make sure you did.)

 **If you're doing pair programming** Only *one* of you should make a submission. Do not make separate individual submissions. Make sure to [add your PA partner](#) to your last/active submission. And make sure to [submit the Pair Programming sign-up form](#), even if you partnered with the same person last PA.

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 `gdb` (or `valgrind`, depending on the error). You should know 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 the error is. 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 already done. 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 Carlos Mattoso, Sander Valstar, Jonathan Margoliash, Dylan McNamara, Niema Moshiri, Adrian Guthals, Christine Alvarado, and Paul Kube for creating the base on which this assignment is built.