CSE 100 // **PA4**

**Six degrees of Kevin Bacon**

|  |
| --- |
| Checkpoint deadline **Friday, June 1 @ 11:59pm**  (**No late days** for checkpoint)  Final submission deadline **Friday, June 8 @ 11:59pm** |

# Assignment Overview

**Starter code:** [**download here**](https://drive.google.com/a/eng.ucsd.edu/file/d/1F2rpVMRfsnUzA3JZlqXcUazk2nj9w3ke/view?usp=sharing) ***Databases updated on 5/30***

✋ **IMPORTANT** Starter code for this PA is *very* limited. You’re in charge of designing your own

classes. Much like in PA3 we’ll simply do make <target> and black-box test your code.

**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.

**Final submission** Pathfinder (Complete) + Actor Connections

You will complete the weighted shortest path version of Pathfinder. You will also write a program

that answers the query: “Given two actors X and Y, after which year did they first become connected?”

**Extra credit** Social network properties and PDF write-up

Compute a set of interesting social network properties on movie\_casts.tsv. You’ll be required to write up what you did, as well as turn in your code.

### Starter Code files

1. **Makefile** Which **you will have to modify** as you add source code
2. **ActorGraph.h/cpp** Contains starter code to read movie\_casts.tsv.

Datasets, test input files and reference solution

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. (Explained in detail 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.tsv** Output file generated by ActorConnections on test\_pairs.tsv
8. **pair.tsv** Text file that contains 100 pairs of actors you can use for testing.

## 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  ... |

1. 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.
2. 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**.
3. [🚨](https://emojipedia.org/police-cars-revolving-light/)[✋](https://emojipedia.org/raised-hand/)[🚨](https://emojipedia.org/police-cars-revolving-light/) 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. [🚨](https://emojipedia.org/police-cars-revolving-light/)[✋](https://emojipedia.org/raised-hand/)[🚨](https://emojipedia.org/police-cars-revolving-light/)
4. 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.

**Note** The **movie\_casts.tsv** file is pretty big, so **DO NOT UPLOAD** to Vocareum. A copy already exists there and will be used when running the grading scripts.

**refpathfinder** and **refactorconnections** are reference solutions you may use to check the expected functionality in various tests.

IMPORTANT NOTES

* **DO NOT HARDCODE ANY TEST CASE. You’ll get a 0 (zero) on this PA if you do.**
* **DO NOT USE ANY OTHER** executable names than the ones in your Makefile. Refer to the **submission instructions** for each part for the **required name of executables.**

# CHECKPOINT [15 points]

## ***1*** Graph design and implementation

|  |
| --- |
| Design and implement *your own* classes to support a graph  implementation for the actor-movie relationships needed in this PA. |

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** 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 so as to allow 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…
* **Test your graph implementation** Load the movie\_casts.tsv file, you should expect to find actors or nodes, movies, and directed edges. **Note**: if we implement our graph with directed edges, every undirected edge will be represented by two directed edges.

Do **NOT** use any pre-built graph data structures, like the Boost Graph Library (BGL). Limit yourself to using the data structures available in the [C++ STL data structures](http://www.cplusplus.com/reference/stl/).

***2*** Pathfinder: unweighted shortest path

|  |
| --- |
| 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

Read the notes that follow for more information.

### ./pathfinder sample usage

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) |

### man pathfinder

**./pathfinder** will take **4 (four)** command-line arguments **in this order**:

1. **Name of text file containing the tab-delimited movie casts** (e.g. **movie\_casts.tsv**).
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 for checkpoint, but you’ll implement it for final submission)

1. **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
2. **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 will 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.

(Refer to the example above.)

### Grading Notes

As per previous PAs, your code will be autograded on submission.

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 the reference solution’s. As long as the *total path* ***lengths*** 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 small graph files along with the starter code. Once you get your code to work on these small graphs, test them on larger graphs.

***3*** Submission instructions

|  |
| --- |
| **Efficiency requirement** All tests are executed with a 60s timeout. Your code must finish within 60s or 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!!!** |

All files must be turned in on Vocareum. Make sure to submit:

1. **Your updated Makefile**. We should be able to compile your code by simply running make pathfinder.
2. **ALL .h/.cpp files you created**. Upload the classes you designed and implemented for this part.

**Do NOT submit any dataset.** All required datasets for grading are already available on Vocareum and will be passed correctly to your program.

FINAL SUBMISSION [85 points]

***1*** Pathfinder: weighted shortest path [25 points]

|  |
| --- |
| Complete your **pathfinder** program by implementing the **weighted**  **edges** version of your program. Edges will have a weight equal to the  age of the movie since 2015 (plus 1). |

The reason for assigned edge-weights based on the age of the movie is 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).

### Formula for edge weights

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

Note that we are using 2015 instead of 2018, which is because the dataset only contains movies released in 2015 and earlier. **Don't accidentally use 2018**!

#### Example

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)--[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) |

**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](http://www.cplusplus.com/reference/queue/priority_queue). 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 w test\_pairs.tsv out\_paths\_weighted.tsv |

**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.

***2*** Actor Connections [60 points]

|  |
| --- |
| Write a program called **actorconnections** to answer the query:  "Given actors X and Y, after which year did they 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 before and including a certain year.

Implementation Checklist

**actorconnections** will give the option to use two different approaches to solve the actor-connections problem. You have to implement **both** options:

1. **Using the graph structure** By solving the **widest path problem**and returning the corresponding year to the bandwidth between actors X and Y – if a path exists. *(Aside: Can you see why this is simply the widest path problem?)*

|  |
| --- |
| **Suggestion** (not required) As covered in [Worksheet 8](https://drive.google.com/open?id=1T3iSfBKPZy1bLoBOPjnX1DJc5q6HgS4M), the [*widest path problem*](https://en.wikipedia.org/wiki/Widest_path_problem) can be solved using a simple modification to Dijkstra’s algorithm. It would be a violation of [the *Don't Repeat Yourself (DRY)* principle](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself) to copy the code for Dijkstra from pathfinder and simply modify the update rule. Given that *only* the update rule changes, all we need is a way to pass in a different [*strategy*](https://en.wikipedia.org/wiki/Strategy_pattern) for the update rule. One way to accomplish that is by [using lambdas in C++](https://stackoverflow.com/questions/29031782/how-to-implement-strategy-pattern-in-c-with-stdfunction). If you’re unfamiliar with lambdas, you can learn more about them here: <https://www.cprogramming.com/c++11/c++11-lambda-closures.html> |

1. **Without using the graph** By using the Disjoint Set ADT (a.k.a **union find**). 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.

Test against refactorconnections (see *(Partial) Reference solution* below)

### ./actorconnections sample usage

Running the following command:

|  |
| --- |
| > ./actorconnections movie\_casts.tsv test\_pairs.tsv out\_connections.tsv ufind |

should run your code (using the union-find algorithm) to produce an output file **out\_connections.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 |

### 

### man actorconnections

**actorconnections** will take **4 (four)** command-line arguments **in this order**:

1. **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 for testing along with the starter code. Once you get it working on these graphs, test it on larger graphs and movie\_casts.tsv.
2. **Name of a text file containing the names of actor pairs** Actor pairs will appear tab-separated on each line (same format as **test\_pairs.tsv**).
3. **Name of your output text file**  This file 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 , your program should output the actor pair and 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 on that line.

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

### (Partial) Reference solution

We included a (partial) reference solution. The reference solution doesn’t take a fourth argument – it will always use the same strategy. You should check the output of your code against the reference solution to make sure everything is fine.

The usage for running the reference solution is

|  |
| --- |
| > ./refactorconnections movie\_cast\_file.tsv pair\_file.tsv output\_file.tsv |

***3*** Submission instructions

|  |
| --- |
| **Efficiency requirement** For the *final submission*, each test is executed with a timeout based on its complexity – varying from 10s to 50s. Your code must finish within the allotted time or 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!!!** |

### 

All files must be turned in on Vocareum. Make sure to submit:

1. **Your updated Makefile**. We should be able to compile your code by simply running make <target> where <target> is one of pathfinder or actorconnections.
2. **ALL .h/.cpp files you created**. Upload the classes you designed and implemented for this part.

**Do NOT submit any dataset.** All required datasets for grading are already available on Vocareum and will be passed correctly to your program.

# EXTRA CREDIT [10 points]

### Introduction

movie\_casts.tsv is simply a social network – one limited to Hollywood connections. Social networks are simply a graph of human relationships. In light of that, many graph properties become even more interesting to be studies in the context of social networks. In the extra credit will take you on a tour of graph properties that highlight interesting patterns when analyzed on a social network.

|  |
| --- |
| **Before you start**  As you can possibly tell, extra credit got more challenging as the quarter went on. Extra credit problems are difficult – because they are *designed to be difficult*. The point of having extra credit problems is to encourage *self*-exploring and researching – it is challenging, but also very fun. Most tutors have not done these extra credit problems before, so there may be limited support in the lab for help with extra credit problems. Good luck! |

### Required deliverables

Turn in all deliverables on Vocareum.

1. **extracredit.pdf** A write-up with one section per problem, containing: 1) the answer to the problem (this may be a value or a visualization) 2) *If applicable,* a clear description of your solution and and analysis of the observed phenomenon.
2. **Your updated Makefile**. We should be able to compile your code by simply running make extracredit. When we run ./extracredit movie\_casts.tsv we should observe output that is consistent with the answers on your report.
3. **ALL .h/.cpp files you created**. Upload the classes you designed and implemented for this part. For this part make sure they are clearly documented.

### **Problem 1** Centrality [2 points]

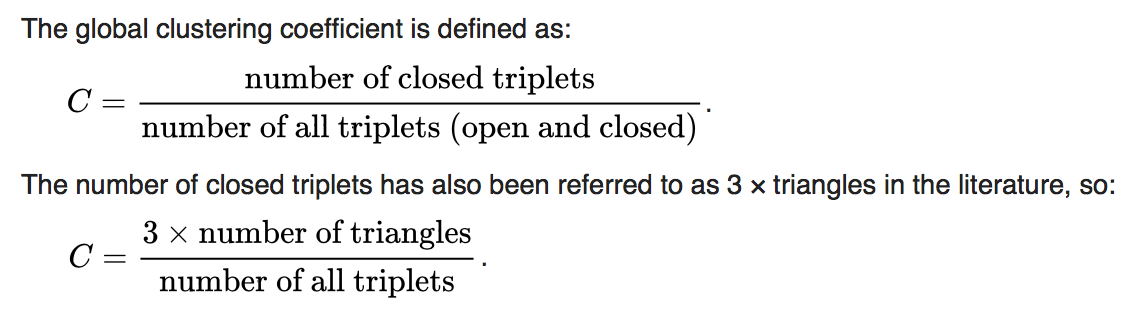
|  |
| --- |
| *Centrality.* In graph theory and network analysis, indicators of centrality identify the most important vertices within a graph. (source: [Wikipedia](https://en.wikipedia.org/wiki/Centrality)) |

This PA revolves around the meme of six-degrees of Kevin Bacon – an actor so prolific most actors are said to be six hops away from him. Is that really so? For this part identify and report the **ten most central** vertices in the network. On your report indicate the measure of centrality you adopted and explain it briefly. Is Kevin Bacon truly deserving of this meme? Were you able to reveal other Kevin Bacons? Is there an actress as central – or possibly more central – than Kevin Bacon?

### **Problem 2** Clustering coefficient [2 points]

|  |
| --- |
| *Clustering coefficient.* In graph theory, a clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. (source: [Wikipedia](https://en.wikipedia.org/wiki/Centrality)) |

In particular, we want you to measure the **global clustering coefficient** as defined on Wikipedia:



For this problem compute and report the **global clustering coefficient** for the movie\_casts.tsv network. In your report make sure to indicate and explain the algorithm you used to compute the number of closed triplets. Also, what does the clustering coefficient you found say about this network?

### **Problem 3** Degeneracy [2 points]

|  |
| --- |
| *Degeneracy.* In graph theory, a *k*-degenerate graph is an undirected graph in which every subgraph has a vertex of degree at most *k*: that is, some vertex in the subgraph touches *k* or fewer of the subgraph's edges. The *degeneracy of a graph* is the smallest value of *k* for which it is *k*-degenerate. (source: [Wikipedia](https://en.wikipedia.org/wiki/Degeneracy_(graph_theory))) |

For this problem compute and report the degeneracy of the graph. In your report be sure to indicate the algorithm you used to compute it and write a concise and clear explanation of how the algorithm works.

### **Problem 4** Interesting maximal cliques [4 points]

|  |
| --- |
| *Clique.* In the mathematical area of graph theory, a clique [...] is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent. [...] A *maximal clique* is a clique that cannot be extended by including one more adjacent vertex. (source: [Wikipedia](https://en.wikipedia.org/wiki/Clique_(graph_theory))) |

For this problem you must implement, report and explain an algorithm of your choice to find maximal cliques in the movie\_casts.tsv network. In doing so, sieve through the cliques to find an *interesting* one that unearths an amusing pattern in the graph (e.g. is there a clique of Hollywood actors who participated in underground poker games ran by an individual known as [Molly](https://en.wikipedia.org/wiki/Molly_Bloom_(author)#Poker_game)?). Include in your report a visualization of the clique (e.g. using something like [graphviz](https://www.graphviz.org)) and write a few sentences about why it’s interesting.