Project 2: Graphs

Introduction

In this Project, you will be implementing a class to represent graphs.

Remember that this is a Project, not a PA, and as such, it is much more involved! To be successful, you **absolutely must** read through the **entire** write-up before even attempting to write a single line of code! Familiarize yourself with the components/sections of the write-up so that, when you finally start coding, you will have a better understanding of what the project entails, and when you inevitably get stuck, you will know where in this write-up to look for tips on how to progress.

Deadline: Sunday, June 9 at 10:00 PM

Late submissions **will not be accepted!** Start early, start often.

Academic Integrity

This Project must be completed **100% independently!** You may only discuss the Project with the Tutors, TAs, and Instructors. You are free to use resources from the internet, but you are not allowed to blatantly copy-and-paste code. If you ever find yourself highlighting a code snippet, copying, and pasting into your Project, you are likely violating the Academic Integrity Policy. If you have any concerns or doubts regarding what you are about to do, *please* be sure to post on the discussion board first to ask us if it is okay.

Grading

- 30 points for Graph Properties
- 15 points for Unweighted Shortest Path
- 15 points for Weighted Shortest Path
- 15 points for Connected Components
- 15 points for Smallest Threshold
- 10 points for no memory leaks

Overview

The mathematical concept of a "graph" is fundamental to Computer Science. In general, any network of connected individuals can be represented using a graph. Almost *all* of the data structures covered in this course are either themselves graphs or can be abstractly represented using a graph.

For example, in the realm of HIV and COVID-19 epidemiology, the current gold-standard method for analyzing viral sequence data to study the spread of the virus is to perform transmission clustering using a tool called HIV-TRACE (Pond *et al*., 2018). If you are given *n* viral sequences (one collected per patient), HIV-TRACE essentially does the following:

- Construct an empty graph with *n* nodes (one per sequence)
- Compute the distance between each pair of sequences (*u*,*v*) under the TN93 model of DNA evolution (Tamura & Nei, 1993)
- If the pairwise distance between *u* and *v* is less than or equal to a given threshold (HIV-TRACE uses 0.015 by default), connect *u* and *v* with an undirected edge
- Each of the connected components of the resulting graph define a "transmission cluster"
- In addition to finding transmission clusters, the graph produced by HIV-TRACE can yield other information relevant for the prevention of the spread of a virus (Grabowski *et al*., 2018)

In this Project, we have provided an API of a `Graph` class that you need to implement. This class will be used to represent undirected graphs. In the starter code, we have provided 4 code files (`Graph.h`, `Graph.cpp`, `GraphTest.cpp`, and `Makefile`) as well as 2 example graph files (`small.csv` and `hiv.csv`).

`Graph.h`

This file contains the basic declarations of the (undirected) `Graph` class that you need to implement. We have declared a series of function signatures that you will need to implement. You are free to modify `Graph.h` however you wish (e.g. adding instance variables to the `Graph` class, adding any helper functions you'd like, etc.). Our only requirement is that the function signatures we have declared remain unchanged (though you are welcome to overload them if you wish).

Please be sure to **thoroughly read the header comments** in `Graph.h` to be sure you understand the functionality, input(s), and output(s) of each function in the `Graph` class.

`Graph.cpp`

This file contains the skeleton of implementing the functions we have declared in the (undirected) `Graph` class. Your task is to fill in the function bodies. You are welcome to add any variables,

functions, classes, etc. you wish, and you are allowed to use anything in the C++ STL.

`GraphTest.cpp`

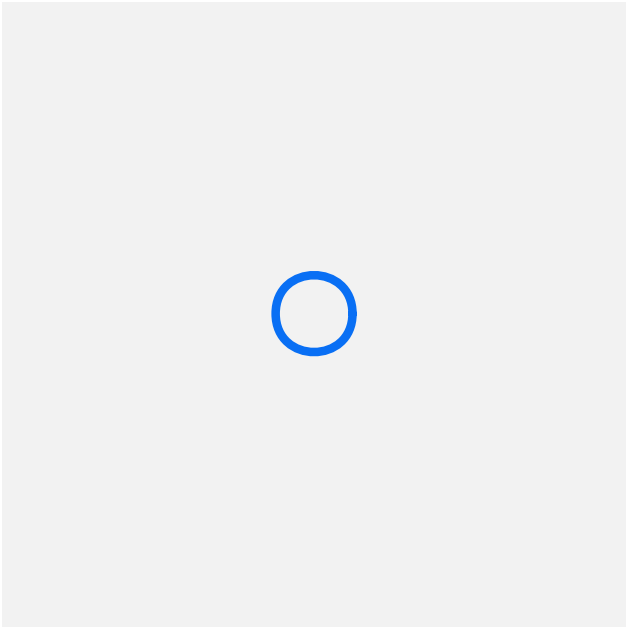
This file contains a tester program to help you run and test your code. We will be using this program to grade your code. You are welcome to modify `GraphTest.cpp` for testing purposes however you wish: we will simply ignore any changes you have made to `GraphTest.cpp` when we grade your code. The `GraphTest` executable that is created from `GraphTest.cpp` can be run as follows:

```
./GraphTest <edgelist_csv> <test>
```

Edge List

The `<edgelist_csv>` argument is the filename of an edge list representing an undirected graph with non-negative edge weights in the CSV format: each line represents an undirected edge between nodes `u` and `v` with weight `w` in the following format: `u,v,w`

For example, imagine we want to represent the following graph:



We could represent it using the following edge list CSV (note that the order of the rows is arbitrary):

```
A,B,0.1
A,C,0.5
B,C,0.1
B,D,0.1
E,F,0.4
F,G,0.5
```

In the `example` folder of the starter code, we have provided 2 example edge list CSV files: `small.csv`, which is the exact graph shown above, and `hiv.csv`, which is a small example graph produced from an HIV-TRACE analysis of a real HIV dataset from San Diego (Little *et al.*, 2014).

Here are some things you can expect about the edge list CSVs you will be given:

- You are guaranteed that edge weights will be non-negative (i.e., they will be ≥0)
- You are guaranteed there will *not* be any self-edges (e.g. `u,u,?`)
- You are guaranteed that you will *not* be given a *multigraph*, meaning you are guaranteed that there will not exist multiple edges between nodes *u* and *v* in the provided edge list
- Further, because the edges are undirected, if you see an edge from *u* to *v* (e.g. `u,v,?`), you are guaranteed that you will *not* see an edge from *v* to *u* (e.g. `v,u,?`) later in the file
    - Remember: an edge from *u* to *v* is bidirectional and is thus *also* an edge from *v* to *u*
- You are guaranteed that there will be at least 1 edge in the graph
- You are *not* guaranteed that the edges in the CSV will be sorted in any particular order

Test

The `<test>` argument denotes which of the `GraphTest` tests you want to run. The tests are as follows:

- `graph_properties`: Test building a `Graph` object and checking its basic properties
- `shortest_unweighted`: Test the `Graph::shortest_path_unweighted` function
- `shortest_weighted`: Test the `Graph::shortest_path_weighted` function
- `connected_components`: Test the `Graph::connected_components` function
- `smallest_threshold`: Test the `Graph::smallest_connecting_threshold` function

Feel free to investigate the code within `GraphTest.cpp` to see how exactly it works.

`Makefile`

This file will help you compile your code using the `make` command. If you choose to create any additional code files, you are welcome to do so: our only requirement is that, given the original version of `GraphTest.cpp`, we should be able to run your code using the `make` command, which should create an executable called `GraphTest` that can be run as described above.

Project Workflow

As mentioned in the overview, as long as (1) you implement all of the functions we declared in `Graph.h`, (2) we can compile the original starter `GraphTest.cpp` via `make`, and (3) we can run `GraphTest` as described, you are absolutely free to implement the project any way you think is best. However, here are some suggestions just in case you feel a bit lost.

Potential Development Process

Step 1: Reading the Header Comments

Before you do **anything**, please read **all** of the header comments in `Graph.h` **thoroughly** to make sure you understand what exactly each function is supposed to do. You should consider the following questions as you investigate each function declared in `Graph.h`:

- What are the parameters of this function? What do they mean? How are they formatted?
- Given the parameters, what is this function supposed to do? What data structure(s) and algorithm(s) would be relevant to achieve this functionality?
- What is the output of this function? What does it mean? How is it formatted?

Step 2: Designing Your Graph

Before you write even a single line of code, completely map out how exactly you want to implement the graph ADT. You should consider the following questions as you think about your unique design:

- What are the graph operations you will need to support?
- What data structure(s) will you use to represent nodes?
- What data structure(s) will you use to represent edges between nodes?
- Are there any helper variables, functions, classes, etc. that you will want to add?
- What are the time complexities of various standard graph operations if you use the data structure(s) you've chosen? Can you modify your choices to speed things up?

I want you to literally **write out *every* data structure you can use** to represent the graph, along with each data structure's **worst-case Big-O time and space complexities**, and map the functionality of these data structures to the properties of a graph. This design step is by far **the most important step**, as proper design will make your coding much easier and will make your code much faster. Be sure to refer to the "Summaries of Data Structures" section of the *Data Structures* Cogniterra text to help you brainstorm.

Once you have decided on a design for your graph implementation, you will want to add any

instance variables your design will need to the `Graph` class.

Step 3: Writing the Constructor

The first nontrivial code you should write will likely be the `Graph` constructor. You will need to populate the `Graph` using the edges contained within the given edge list CSV file. You should think about the following questions as you implement your constructor:

- How should you initialize each of the instance variables you added to the `Graph` class?
- Do you want to add any helper functions (or perhaps an overloaded constructor) to help you initialize your graph from the given CSV file?

Step 4: Implementing Basic Graph Operations

Once you are confident that your `Graph` constructor is correct, write the functions that access basic properties of a graph: `num_nodes()`, `nodes()`, `num_edges()`, `edge_weight()`, `num_neighbors()`, and `neighbors()`. You should think about the following questions as you implement these functions:

- How can you use the instance variables you've added to the `Graph` class to perform these operations? What would be the worst-case time complexity of each operation?
- Without blowing up memory consumption, can you save the values of basic graph operations as instance variables, rather than computing them from scratch each time?

Once you have implemented all of these functions, you should be able to use the `graph_properties` test in `GraphTest`. Once you have these basic properties working, the remaining functions can be implemented in any order (they will likely be independent of one another), so feel free to deviate from our recommended order if you prefer.

However, *all* of the following steps require your basic graph functionality to work completely correctly. Thus, **do not continue until you are able to get the basic graph functionality completely working!**

Step 5: Finding a Unweighted Shortest Path

In the `shortest_path_unweighted()` function, you will be finding the shortest unweighted path from a given start node to a given end node. You will want to implement this using the Breadth-First Search (BFS) algorithm. Be sure to map out how exactly the algorithm will operate with the specific graph design you have chosen. You should think about the following questions as you implement this:

- What data structure(s) will help you implement BFS, and do they have built-in C++ implementations?
- How will you use the properties of your graph design to facilitate the BFS algorithm?
- What is the worst-case time complexity of your approach? Are there any optimizations you can make to speed things up?

Be sure to refer to the "Algorithms on Graphs: Breadth First Search" section of the *Data Structures* Cogniterra text for more information about BFS.

Step 6: Finding a Weighted Shortest Path

In the `shortest_path_weighted()` function, you will be finding the shortest weighted path from a given start node to a given end node. You will want to implement this using Dijkstra's Algorithm. Be sure to map out how exactly the algorithm will operate with the specific graph design you have chosen. You should think about the following questions as you implement this:

- What data structure(s) will help you implement Dijkstra's Algorithm, and do they have built-in C++ implementations?
- How will you use the properties of your graph design to facilitate Dijkstra's Algorithm?
- What is the worst-case time complexity of your approach? Are there any optimizations you can make to speed things up?

Be sure to refer to the "Dijkstra's Algorithm" section of the *Data Structures* Cogniterra text for more information about Dijkstra's algorithm.

Step 7: Finding Connected Components

In an undirected graph, a connected component is a subgraph in which any two vertices are connected to each other via some path. Given a graph, you can find all of the components of the graph using the following algorithm:

- Initialize all nodes in the graph to "unvisited"
- While there are still unvisited nodes:
    - Arbitrarily choose one of the remaining unvisited nodes (call it *u*)
    - Perform BFS starting at *u*, and store all nodes visited in the BFS (including *u*) in a set *c*
    - Once BFS is complete, output *c* as a component of the graph

In the `connected_components()` function, you will be finding all connected components in the graph, but with a small catch: given some threshold `thresh`, you will be ignoring all edges with a weight larger than `thresh`. This shouldn't impact the above algorithm very much: the only distinction is that, each time you perform BFS, you will want to only traverse edges with a weight less than or equal to `thresh`.

Step 8: Finding the Smallest Connecting Threshold

In HIV transmission clustering, a natural question is the following: Given two individuals *u* and *v*, what is the smallest threshold *d* such that, if we were to only include edges with weight less than or equal to *d*, there would exist a path connecting *u* and *v*?

A trivial but horrendously inefficient algorithm to solve this problem is the following:

- Start with a graph with no edges
- For each unique edge weight *w* in increasing order:
    - Add all edges with weight *w* to the graph
    - Perform BFS starting at *u*
    - If *v* is visited in the BFS, return *w* as the smallest connecting threshold
- If we get here, *u* and *v* were never connected, so no such threshold exists

However, we can utilize the Disjoint Set data structure to speed things up:

- Create a Disjoint Set containing all nodes in the graph, each in their own set
- For each edge (*x*,*y*,*w*) between nodes *x* and *y* with weight *w* increasing order of edge weight:
    - Perform *union*(*x*,*y*)
    - If *find*(*u*) is equal to *find*(*v*) (meaning *u* and *v*, the function arguments, are now in the same set), return *w* as the smallest connecting threshold
- If we get here, *u* and *v* were never connected, so no such threshold exists

In the `smallest_connecting_threshold()` function, you will be finding the smallest connecting threshold between a given pair of nodes *u* and *v*, and you will want to utilize the more efficient algorithm we have described to do so. As such, you will want to implement a helper Disjoint Set class that implements the optimizations discussed in class (e.g. Path Compression and Union-by-Size) to make your code as fast as possible. Be sure to refer to the "Disjoint Sets" section of the *Data Structures* Cogniterra text for more information about Disjoint Sets.

Design Notes

Before you write even a single line of code, it's important to properly design the various components of your `Graph` class. Here, we will discuss some aspects of the design that you should be sure to think about.

Parsing a CSV File

The edge list your `Graph` constructor needs to read will be a CSV file, such as the following CSV file with 3 columns:

```
Niema,Moshiri,100
Ryan,Micallef,95
Felix,Garcia,95
```

In C++, you can parse a CSV file as follows (note that `first`, `second`, and `third` are all `string` objects):

```
ifstream my_file(filename);      // open the file
string line;                     // helper var to store current line
while(getline(my_file, line)) {  // read one line from the file
    istringstream ss(line);      // create istringstream of current line
    string first, second, third; // helper vars
    getline(ss, first, ',');     // store first column in "first"
    getline(ss, second, ',');    // store second column in "second"
    getline(ss, third, '\n');    // store third column column in "third"
}
my_file.close();                 // close file when done
```

- Don't forget to add `#include <fstream>` to the top of your code to be able to use `ifstream`!
    - Be sure to refer to the `ifstream` C++ documentation for more information
- Don't forget to add `#include <sstream>` to the top of your code to be able to use `istringstream`!
    - Be sure to refer to the `istringstream` C++ documentation for more information

Using the C++ `tuple` Class

In order to represent the mathematical concept of a tuple, C++ provides a `tuple` class. Be sure to thoroughly read the `tuple` documentation, but the following code example will hopefully show you how to use it:

```
#include <iostream> // load cout
#include <string>   // load string class
```

```
#include <tuple>    // load tuple class
using namespace std;
int main() {
    tuple<string,string,int> prof = make_tuple("Niema","Moshiri",1993);
    cout << "First Name: " << get<0>(prof) << endl;
    cout << "Last Name:  " << get<1>(prof) << endl;
    cout << "Birth Year: " << get<2>(prof) << endl;
}
```

You may or may not want to use `tuple` objects in your underlying `Graph` design, but you will likely want to use them as you implement the member functions of the `Graph` class (e.g. as you implement a graph traversal algorithm). Pro-tip: if you try to sort a `vector` that contains `tuple` objects using `std::sort`, the `tuple` objects will be sorted based on the first element.

Using the C++ `pair` Class

Much like the `tuple` class, C++ provides the `pair` class that can be used to represent a pair of objects. Be sure to thoroughly read the C++ `pair` documentation, but the following code example will hopefully show you how to use it:

```
#include <iostream> // load cout
#include <string>   // load string class
#include <utility>  // load pair class
using namespace std;
int main() {
    pair<string,string> prof = make_pair("Niema","Moshiri");
    cout << "First Name: " << prof.first << endl;
    cout << "Last Name:  " << prof.second << endl;
}
```

Creating a C++ `struct`

When examining your data types, you may find yourself adding many properties to your `tuple` objects, or may want increased readability by creating custom types. C++ provides the ability to define a `struct`, which is essentially a simple class that can be used to bundle together elements. Be sure to thoroughly read the C++ `struct` documentation, but the following code example will hopefully show you how to use it:

```
#include <iostream> // load cout
#include <string>   // load string class
using namespace std;

// define struct
struct person {
    string first;
    string last;
    unsigned int year;
};
```

```
int main() {
    // create struct object on stack
    person prof = {
        .first = "Niema",
        .last  = "Moshiri",
        .year  = 1993
    };

    // access values of struct object to print
    cout << "First Name: " << prof.first << endl;
    cout << "Last Name:  " << prof.last << endl;
    cout << "Year:       " << prof.year << endl << endl;

    // change a value of the struct object
    prof.first = "Niemster";
    cout << "First Name: " << prof.first << endl;
    cout << "Last Name:  " << prof.last << endl;
    cout << "Year:       " << prof.year << endl << endl;

    // create struct object dynamically on heap
    person* prof_ptr = new person {
        .first = "Niema",
        .last  = "Moshiri",
        .year  = 1993
    };

    // access values of struct object to print
    cout << "First Name: " << prof_ptr->first << endl;
    cout << "Last Name:  " << prof_ptr->last << endl;
    cout << "Year:       " << prof_ptr->year << endl;
}
```

Using a Custom Comparison Class with `priority_queue`

When you implement Dijkstra's Algorithm, you will likely want to store `tuple` objects representing (*total path weight*, *from*, *to*) tuples to keep track of your graph traversal. In C++, you can create a `priority_queue` with a custom comparison class. You may want to create a custom comparison class to compare these tuples in a way that uses the `priority_queue` as a min-heap with respect to total path weight. Be sure to read the C++ `priority_queue` documentation thoroughly to see how to define a custom comparison class and use it with a `priority_queue` object.

Modifying the `Makefile`

As you design your `Graph` class, you may feel as though you want to create additional code files for any helper classes you may potentially create. Note that you don't have to: you can complete this project without creating any additional files. However, if you do choose to do so, you will find that

you will want to modify the `Makefile` in order to properly compile your executable. Feel free to make any modifications you wish, and you may want to refer to this guide to help you.

How to Debug

As you work on this Project, you will almost certainly run into bugs you will need to find and fix. Here, we will provide some general tips for debugging that may prove useful.

Designing and Drawing Out Graphs

Be sure to design your own graph CSV files to test your code, and be sure to actually manually draw out exactly what the graphs look like and exactly how the various graph algorithms should execute on your graphs. Try to think of corner cases you might face and how your code should handle them. When we say that you should "draw out" the graph, we mean **literally draw it out by hand!** Whether you prefer to draw it out using pen and paper, or on your tablet, or using your mouse or touchpad with Microsoft Paint, or using Zoom's whiteboard feature while you're having a video chat with your parents and want to show them all the amazing things you're learning in CSE 100, *please* actually draw out your test graphs and actually step through the graph algorithms by hand!

JSON Pretty Print

The `GraphTest` executable will generally print 2 tab-delimited columns: a descriptor (left) followed by a JSON-formatted output produced by running your code. It outputs the JSON output in a compact manner that may be difficult for you to read as you try to debug your code. Instead of trying to read the JSON output in this compact manner, you can copy it and paste it into a JSON pretty printer, which will reformat it to make it easier to read. There are numerous online tools that do this, with one nice option being JSON Formatter. For example, you can copy the following compact JSON:

```
{"first":"Niema","last":"Moshiri","year":1993}
```

You can then paste this into the left text box on JSON Formatter and click the "Format/Beautify" button. Once you click this button, you should see the "beautified" version of the JSON in the right text box:

```
{
  "first": "Niema",
  "last": "Moshiri",
  "year": 1993
}
```

Because `GraphTest` was actually designed to output in a format that would be easily parseable by Python, some outputs (e.g. the output of `Neighbors` in the `graph_properties` test) are actually not valid JSON (even though they are valid Python syntax). If you try using JSON Formatter and it gives you a syntax error, try using Python Formatter.

Comparing JSON Files

When debugging, it may be useful to use a program to compare your code's JSON output to the known correct output to make sure it's correct. Since JSON's key-pair values may be stored in any order (similarly to a hash map), using regular `diff` or comparing by hand may prove difficult. You may use whatever you like, but JSON Diff is one useful tool for performing such comparisons:

1. First, paste the known correct JSON output into the box on the left
2. Then, paste your code's JSON output into the box on the right
3. Remove any extra newline characters that may be in either box
4. Click the "Compare" button, which will display the two JSONs side-by-side and show the differences (if any)
    - If there *are* any differences, rows highlighted in blue represent values that differ, and rows highlighted in green represent missing (or extra!) key-value pairs
    - If there *are not* any differences (i.e., the JSONs were identical), it will display the following text: "The two files were semantically identical."

Using `gdb` to Help You Debug

As you implement your `Graph` class, you will want to be able to debug effectively. Be sure to make use of `gdb` (the **G**NU **d**e**b**ugger) to help you step through your code, print variables, etc. You can refer to this excellent cheat sheet to help you use `gdb` if you don't remember how. If your code is producing the incorrect output on a specific graph, try physically drawing out the graph and manually stepping through the graph algorithm while simultaneously stepping through your code line-by-line using `gdb`, printing variables as needed, in order to pinpoint where and how exactly your code diverges from your logic.

Speeding Up Code

If your code is taking too long to run, you will want to investigate *which parts* of your code are slow so you can speed up those specific parts. On pretty much any Linux platform, you can use `gprof` (**G**NU **Prof**iler) to perform "per-function profiling". There are more complicated profiling tools that exist, so if you feel comfortable learning how to set up / use one of those, feel free, but `gprof` works for me. The basic workflow is as follows:

```
# Step 1: Compile your program with the -pg flag (we provide a Makefile target for convenience)
make gprof

# Step 2: Run your program (it will create a file gmon.out)
./MyProgram

# Step 3: Run gprof to profile (I pipe to less -S to make it easier to read)
gprof MyProgram | less -S
```

The `gprof` output tells you what percentage of your runtime was spent on each function that was called in your code. Many times, the runtime will be some built-in C++ STL function, so you may think "Well, that's not my code, so that's not something I can speed up", but that would be incorrect! If a built-in C++ STL function is taking a bunch of runtime, it's because *you're calling* that function :-) So try to find where in your code you're calling it a bunch of times, and try to refactor that part of your code!

Here's a lengthier tutorial about `gprof`:

https://www.thegeekstuff.com/2012/08/gprof-tutorial/

Common Issues

Code is Too Slow

The most likely culprit for code that is too slow is the use of inefficient data structures, either within your implementations of the member functions or even within your `Graph` class design itself. One way to help pinpoint the specific slow parts of your code are to use `gprof` (see the "How to Debug" part of this write-up for usage information). In general, there is no "one size fits all" way to speed up your code: for each operation or algorithm your `Graph` class implements, think of each step of your code, then think about the actual data structures and algorithms that your code is employing, and consider the time complexity of each step.

Infinite Loop in Graph Traversal

Remember that the edges in your `Graph` class are undirected edges, meaning an edge from *u* to *v* can also be traversed from *v* to *u*. As such, one common mistake is to accidentally re-add nodes you've already visited back into the container you're using to keep track of your graph traversal (e.g. a queue for Breadth-First Search or a priority queue for Dijkstra's Algorithm). You can check for this by stepping through your code (e.g. using `gdb`) and seeing if you accidentally visit any nodes multiple times.

Valgrind Reports Memory Leak with `make gprof`

If your code was timing out because the runtime was too long, you likely investigated the runtime of the parts of your code using `gprof` (which is excellent!), and in doing so, you likely compiled your code using `make gprof`, which adds the `-pg` compilation flag to the compilation command (the `-pg` compilation flag is required in order to use `gprof`).

However, for reasons out-of-scope of this class, Valgrind doesn't play nicely with the `-pg` flag, and it will (potentially) incorrectly report a memory leak even if your code doesn't actually have a memory leak. Because of this, you will want to remove the `-pg` flag from your compilation command before checking for memory leaks (and before submitting your code, as the grader will be checking for memory leaks). For more information about Valgrind and `-pg`, see this post.

Code Works on Example Datasets but Fails Grader

The small example datasets we provide are by no means exhaustive, nor are they intended to be. If your code seems to be working correctly on all of the example datasets we provide but fails the grader, that means that there is a bug in your code that needs to be fixed, and the only way to find

the bug is to **design your own test datasets** to try to cover any corner cases you can think of in order to trigger the bug. The ability to design appropriate test datasets is one of the intended learning outcomes of this assignment.

Project 2: Graphs

Did you already read through the ***entire*** write-up before coming here to code? We know it's long, but we promise that you'll be way more ready to start coding once you've read everything and mapped out how you want to tackle the project. If you skip any part of the write-up, you'll make this poor puppy sad:

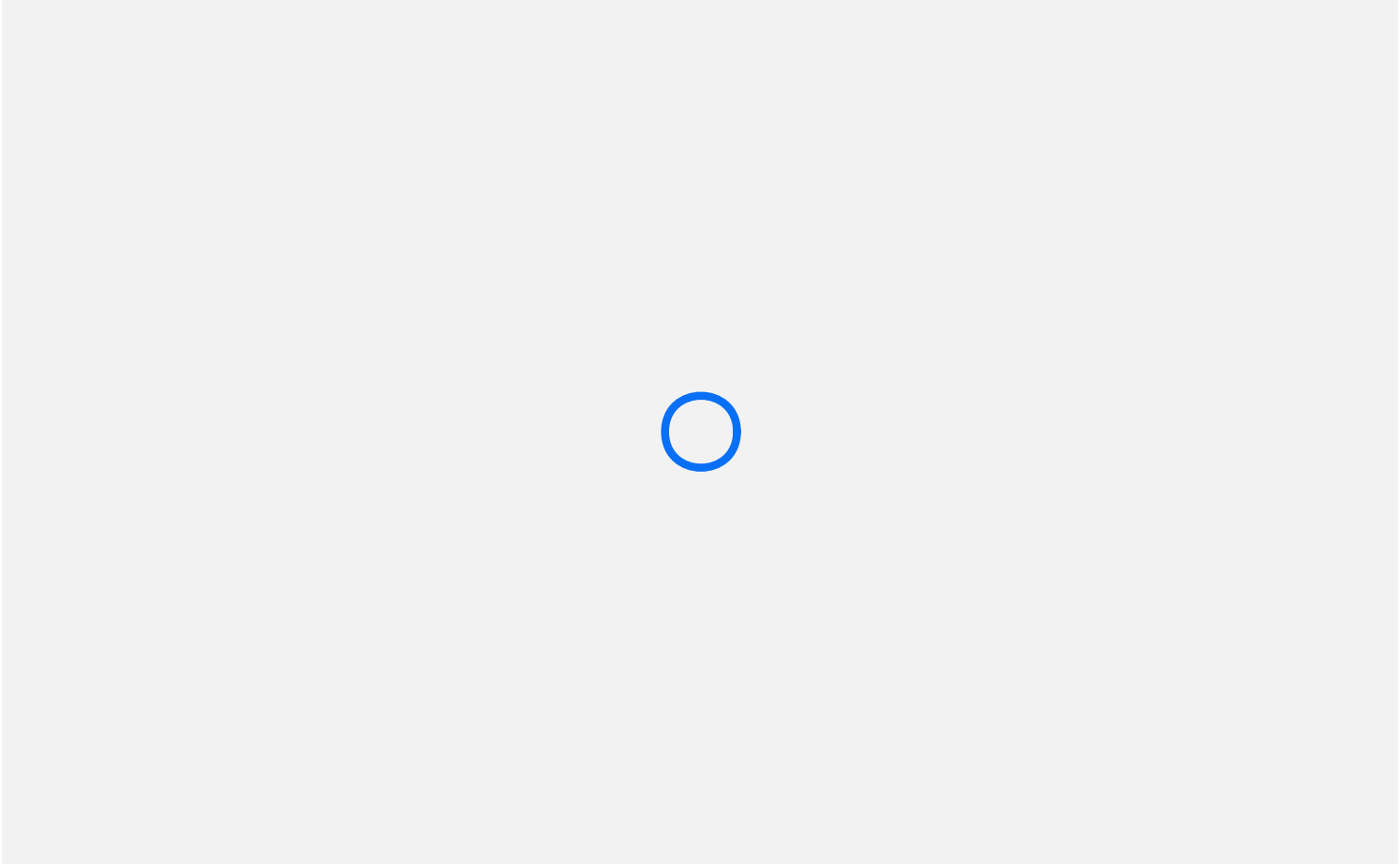


Figure: *The poor puppy you'll make sad if you start coding without reading the entire write-up*