CompSci 260P Spring 2023
Project #1: Review of Graph Fundamentals

Due date and time: April 17, 2023 at 6:00 PM.
Remember to read the syllabus to be sure you make yourself eligible to be graded for this assignment. If you were eligible for project zero grading, then you are eligible for the rest of the quarter, too.

This project is eligible to be submitted up to 48 hours late, with a penalty for doing so. Due to the delay posting this project, the first 24 hours of late are free. You need only fill out the form if you are submitting more than 24 hours late. The maximum of 48 hours from the stated due time is still the last chance to submit for credit.

As with project zero, this project asks you to write and submit three short C++ functions. However, as you prepare to write these, you should use this opportunity to confirm that you have reviewed graph fundamentals to the extent you will need for this course. Chapters 5-8, inclusive, of the Erickson text are a great review for graph fundamentals at the level we expect you to know at this point in CompSci 260P. Similarly, if you are following in the textbook of Goodrich and Tamassia, Chapter 13 will serve as an excellent review of graph fundamentals.

For more information about grading, see the relevant section of the lab manual.

The Program’s Requirements

Each of these parts requires you to write a small function in C++. Please use this as an opportunity to gauge your readiness for the class. While solutions to these can easily be found online, doing so would deprive you of a key benefit you will get from this assignment, to say nothing of the risk you would be taking by submitting work that is not yours.

Take the opportunity to do this project yourself, so that you can check that you can write simple functions in C++, read and understand the requirements documents, and successfully use Gitlab to begin and submit the project.

For all three parts of this project, the graph will be represented as an edge list. Your parameter for the graph has two parts: the second is an unsigned integer \( n \), the number of vertices in the graph. The first is a vector of pairs of unsigned integers, representing the edges. For the two parts with undirected graphs (Euler and bipartite), it will always be the case that the first index will be less than the second. This is not guaranteed for the topological sort portion, as that graph is directed. In all cases, you are guaranteed that each unsigned integer in the edge list is a valid vertex; the vertices are numbered \( 0..n-1 \), inclusive.
Part One: Eulerian Graphs

A classic problem in Graph Theory is as follows. We are given a connected\(^1\) undirected graph. I want to know:  can I walk along the graph from a vertex of my (or your) choice and walk across each edge exactly once? I cannot omit an edge from my walk, nor may I walk across an edge more than once. I am allowed to revisit vertices as necessary, and a similar problem with visiting vertices once each is probably harder to solve efficiently than this one.

A graph where I can produce such a walk, but it begins and ends at different vertices, is said to have an Eulerian Walk. If I can produce such a walk, but it begins and ends at the same vertex, the graph is said to have an Eulerian Circuit.

Before you proceed, if you are not familiar with this problem, think for a bit about how you would go about solving it. Draw a few small sample graphs and see if they do or do not meet the criteria. Then consider reading the Wikipedia entry for this\(^3\). Before you do, be advised of two things:

- There are links to implementations on that page. I encourage you to not even click those links. Do not do that, not even “for ideas” for how to code this. Turning in code that you did not write is one of very few ways to get a very poor grade in my class (among other problems).
- You are allowed to draw inspiration from the algorithm descriptions on that page (the pseudo-code, not C++ code linked). If you do so, cite your source in a comment in the relevant file. Specify which section you are trying to implement.

To complete this portion, you need to implement the following function in euler.cpp:

- EulerPossibility checkForEuler(const std::vector<std::pair<unsigned, unsigned>> & graph, unsigned n, std::vector<std::pair<unsigned, unsigned>> & journey )
  - The input graph is presented as a list of undirected edges.
  - Return one of the three of type EulerPossibility, reflecting which of the three correctly describes the graph input.
    - If the graph has neither an Euler walk nor circuit, the return value is sufficient.
    - If the graph has an Euler walk or circuit, indicate this in your return value. Before you return, adjust the reference parameter journey to have the edges that constitute the walk / circuit in order. Return them in such an order that the second endpoint of one is the first point of the next.
  - The parameter journey will be initially an empty vector when this function is called.

---

\(^1\) Recall that this means, in an undirected graph, that for any pair of vertices \(u\) and \(v\), there is a way to walk from \(u\) to \(v\) and vice versa. It does not mean that they necessarily share an edge.

\(^2\) Euler is pronounced “Oiler” not “You-ler”

\(^3\) https://en.wikipedia.org/wiki/Eulerian_path
Part Two: Topological Sort

As an undergraduate, you presumably saw algorithms like SelectionSort or MergeSort, which take a list of comparable elements and puts them into “the correct order.” Sometimes, the artifacts being sorted have a natural total order, such as numbers or words. Other times, we need to define what it means for one element to be “before” the other in order, such as cards in a standard 52-card playing deck⁴.

Other times, it is not meaningful to say that we can compare any two elements. For example, when you get dressed in the morning, your left sock must be donned before your left shoe. However, your left shoe and your hat can be donned in either order. If we do not have a total ordering of artifacts, we cannot use an algorithm like MergeSort. Instead, we have pairwise constraints. This is often represented as a directed graph, where an edge from \( i \) to \( j \) means that artifact \( i \) must appear before \( j \) in the order.

Before you continue, think for a moment about how you would go about deciding a valid order within these, if the directed graph were provided to you as a drawing on paper. You may then wish to read the Wikipedia entry for Topological Sort⁵ for more information. Before you do, be advised of three things:

- There are links to implementations on that page. I encourage you to not even click those links. Do not do that, not even “for ideas” for how to code this. Turning in code that you did not write is one of very few ways to get a very poor grade in my class.
- You are allowed to draw inspiration from the pseudo-code on that page. If you do so, cite your source in a comment in the relevant file. Specify which batch of pseudo-code you are using.
- Don’t bother with the parallel implementation in 260P. While it’s nice to know, the input graphs will be sufficiently small that a decently good implementation of the basic serial algorithms will run in well under the required running time.

Your input is a directed graph, which may or may not be acyclic (meaning it has no cycles). You are going to output a topological order of the graph, sometimes known as a topological sort.

To accomplish this, you need to implement the following function in topologicalSort.cpp:

- `std::vector<unsigned> topologicalSort(const std::vector<std::pair<unsigned, unsigned>> & graph, unsigned n)`
  - If the graph has a cycle, throw a NoTopologicalSortException. The default start code does this in all cases.
  - If the graph does not have a cycle, return a vector that is a permutation of 0..\( n \) representing a topological order. In particular, if edge \((i,j)\) is in the input, then \( i \) must appear before \( j \) in the returned vector, although not necessarily immediately before it.

---

⁴ The typical ordering for playing cards is to compare suits alphabetically, so the 3 of Clubs comes before the King of Diamonds. If the suits are equal, we use the rank of the cards, treating an ace as one, jack as eleven, queen as twelve, and king as thirteen. Some games use a different ordering. It won’t matter for this project.

Part Three: Bipartite Checking

A simple, undirected graph is called bipartite if we can perform the following. You are given two crayons, one blue and one gold. You want to color in each vertex in such a way that:

- Each vertex is exactly one color -- none are omitted, and none are both blue and gold.
- Each edge has two endpoints of different colors.

A graph for which this is possible is bipartite. In general, I could prove to you that a graph is bipartite by giving you the coloring of the vertices, and you could inspect each edge to confirm that it has two different color endpoints. Perhaps surprisingly, we can prove that a graph is not bipartite by providing a cycle within the graph of odd length. In particular, a graph is bipartite if and only if every cycle within the graph has even length.

You will need to determine, given a graph, if it is bipartite or not. The typical way we do this is as follows; this describes how you would do it for a single connected component -- if the graph has multiple connected components, you would do this for each component.

Select a vertex arbitrarily. Tentatively color that vertex blue. Color each of its neighbors gold. Color each of the newly-painted gold vertices' neighbors in blue. Color each of the newly-painted blue vertices' neighbors in gold. Repeat until each vertex has a color. Now check every edge: if every edge has a blue and a gold endpoint, the graph is bipartite (and the coloring would be proof of that, if you were asked for such).

If any edge has two same-color endpoints, we will use it to find proof that the graph cannot be done with two colors (i.e., it isn't just that the attempt we made did not get the job done). Start at the two endpoints $u$ and $v$; walk up to their "parent" nodes (the vertices whose painting in the opposite color caused these to be painted). Repeat this until you find a single common ancestor. The path from this common ancestor to $u$ is some integer $k$ edges. The edge from $u$ to $v$ is one edge, and from $v$ back to the common ancestor is also $k$ edges. This is a cycle of $2k+1$ edges (and also $2k+1$ vertices), which cannot be painted with only two colors (do you see why?).

To accomplish this, you need to implement the following function in bipartite.cpp:

- `bool isBipartite(const std::vector<std::pair<unsigned, unsigned>>& graph, unsigned n, std::vector<std::pair<unsigned, unsigned>>& evidence)`
  - If the graph is bipartite, return true. You do not need to provide evidence for this project.
  - If the graph is not bipartite, this function must modify the reference parameter evidence with a simple (no repeating edges) cycle of odd length from the input graph. The order of the edges listed must be within a walk around the cycle, not an arbitrary ordering of edges that constitute a cycle.
  - The input graph is undirected; all edges are listed as a pair $(a,b)$, where $a < b$. This means that $(b,a)$ is also an edge in the graph, but might not be listed.
Your grade on this project

There are 6 points possible on this project; they are only available by test cases. We will run test cases with the code you submit; each test case is worth some fraction of the grade. In order to have either part graded, that part’s required test cases must pass; there are no points for these, other than that you are eligible to have other test cases we use graded. Each of the three parts is worth two points, and each is graded independently, other than that a single instance of non-compiling code will prevent our program from grading at all, resulting in a score of zero.

While some of the code used to perform verification is provided to you in the project, please be aware that we may rename these functions in the grading. This imposes two limits on you: first, while you may use the code in writing your own test cases, you should not change it; second, you may not use it in your code outside of test cases, as you may not be sure that it will still be there.

Test cases that take longer than two minutes to run on the grader’s computer may be deemed incorrect runs, even if a longer amount of time available to them would cause a correct answer. Note that the grader’s computer may be slower than yours, especially if you have a very new computer, such as an M1/M2. Do not assume that the speed on your computer will match that on the grader’s! Furthermore, note that the running time for grading includes the time to run the test cases, which include verification functions in some cases. We considered this when writing the cases we will use for grading.

We will only choose test cases that can reasonably run within this constraint. Each of the problems in this assignment can be solved by a linear time algorithm, although that aspect is not a requirement for this project, as long as the observed running time is sufficiently short.

There is also a style concern for this quarter: under no conditions may your code contain the following declaration: using namespace std; Projects that contain that declaration are worth zero points, regardless of other considerations. This rule is in effect for all projects in CompSci 260P this quarter.

For more information about grading, including how to submit a late project, see the relevant section of the lab manual.