CprE 310 / Spring 2019

Lecture 8: Applications of Graph Theory

The Seven Bridges of Konigsberg

The field of graph theory (arguably) was kickstarted by Euler when he solved the *Seven Bridges of Konigsberg problem*. Konigsberg (now a city in Russia called Kaliningrad) looks like this today in Google Maps:

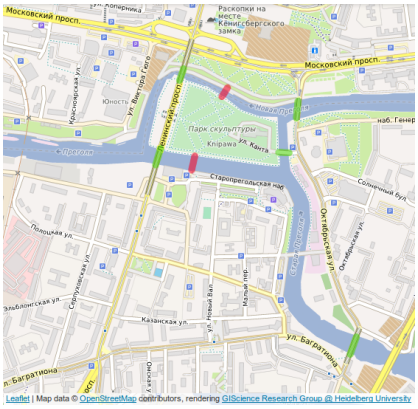

Figure 1: Konigsberg Today

but back in the 1700s, there were a few more bridges crossing the rivers. Then, the city looked like this:

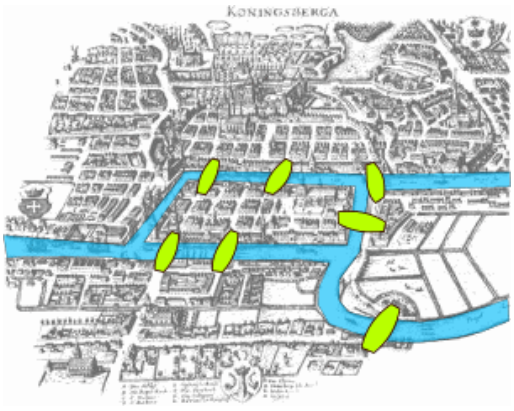

Figure 2: Bridges of Konigsberg

There were 7 bridges across the rivers connecting different landmasses, and a friend of Euler posed to him the following question:

> Is it possible to walk around the city such that each bridge is crossed once, and exactly once?

You can try tracing different walks across the bridges yourself. However, you'll find that a walk that does not repeat edges seems difficult to find. In fact, such a walk is impossible, and Euler proved this using graph theory.

Euler's main idea (like most great ideas) was deceptively simple:

> Draw the map as a graph.

It does not matter where the bridges were located; all that matters is that we represent landmasses as nodes, and bridges between landmasses as edges connecting nodes. Rewriting the map thus gives us the following graph:

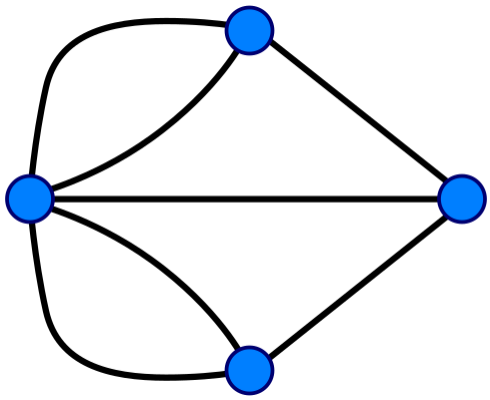

Figure 3: Graph of Konigsberg

Now, we can play around with this graph quite a bit, try out lots of paths etc but always come to the conclusion that a path across the city that crosses each bridge exactly once is impossible. This is in fact a *theorem*:

> There exists no path across the Konigsberg graph which contains each edge exactly once.

But how do we *prove* this statement?

The proof is by contradiction. Let's think like a computer scientist. Observe that traversing to and from any given landmass triggers

- you ENTER via a bridge, and
- you LEAVE via a different bridge.

Therefore, except the initial and final landmasses, every landmass is "touched" by two bridges. This means that each landmass (except the first and second) must be touching an *even* number of bridges.

In the language of graph theory, it is necessary that every node in the path (except first and last) has to have *even degree*.

However, the graph in question has all four nodes of *odd* degree. Therefore, this is a contradiction. No matter what the start and end points were, such a path cannot exist!

Such paths are called *Euler* paths. A necessary condition for an Euler path to exist in a graph is that the graph should be connected, and that the number of nodes with odd degree must be zero or two. In fact, this is also a *sufficient* condition, and this was a neat thing proved by Euler himself.

Let us discuss a couple more applications of graphs, before delving into the theory.

Scheduling

Graphs can arise in somewhat unexpected applications. For example, several problems that involve constructing a *schedule* can be posed in the language of graph theory.

Suppose the Office of the Registrar at Iowa State is trying to build a schedule for the Spring Semester. The following courses can **not** be scheduled in the same slots since they have lots of students in common:

- Physics and ComS
- Physics and Chemistry
- Calc I and Chemistry
- Calc I and Physics
- Calc I and CPRE 310
- Calc I and ComS
- CPRE 310 and ComS
- Psych and Chemistry
- Psych and CPRE 310

The goal is to find the fewest number of **time slots** that can be used to construct a schedule with the above constraints.

If we stare at this a bit and play around with our options, we can probably guess that the answer is **3**, and perhaps come up with a schedule that looks like this:

- Time slot A: Calc I and Psych
- Time slot B: ComS and Chemistry
- Time slot C: Physics and CPRE 310

Guesswork might help us with small problems like this one, but not on (say) a university scale when there are hundreds of such conflicts involving thousands of classes to resolve. Can we solve this problem *systematically*?

Answer: Yes (using graph theory)!

Let us represent each class by nodes of a graph. There are 6 classes in the above example, and therefore there will be 6 nodes. Moreover, if two classes cannot be scheduled in the same slot, we will link them via edges. For example, there is an edge between Physics and ComS, Physics and Chemistry, and so on.

Next, let us consider *coloring* the nodes of the graph with some number of colors, such that no two nodes linked by an edge have the same color. What is the minimum number of distinct colors needed for such a coloring to exist? This is the so-called *graph coloring* problem, for which several solution approaches are known. We will not discuss them in detail; however here are some thoughts:

- Trivially, we need no more than 6 colors; each node can be given its own distinct color.

- We definitely need at least 3 colors; it is clear that only two colors don't suffice (can you argue why that is the case?)
- Playing around a bit, we can assign the following colors (for example) – Red for Calc I and Psych, Blue for ComS and Chemistry, and Green for Physics and CPRE 310. (You could get a different color assignment but confirm that you need no more than 3 colors whichever way you do it.)

Once we get a coloring of the graph, we can assign *slots* to *colors*; automatically, by the constraint we used to set up the colors there will be no conflicts among classes scheduled in the same slot. Done!

Sets, functions, relations, and such

Recall that a graph (as we defined in the previous lecture) consists of a set of nodes, with edges indicating some type of relationship between nodes.

Of course, for this definition to become rigorous, we first need to define what "sets" mean and what "relationship" means from a mathematical standpoint. For that, we require a bit more vocabulary.

Before the math begins, it is useful to make an analogy with software programming: when we write a software program, a key specification is the *type* of a variable or constant in the program. Understanding the type of an entity is absolutely essential for specifying the kinds of objects that software programs can use and manipulate. Analogous to this notion, we will begin the study of (somewhat) more general, *mathematical* data types. In particular, we will dig deeper into three "types": sets, functions, and relations.