Graph Theory and Social Networks
Spring 2014 Notes

Kimball Martin

April 30, 2014

Introduction

Graph theory is a branch of discrete mathematics (more specifically, combinatorics) whose origin is generally attributed to Leonard Euler's solution of the Königsberg bridge problem in 1736. At the time, there were two islands in the river Pregel, and 7 bridges connecting the islands to each other and to each bank of the river. As legend goes, for leisure, people would try to find a path in the city of Königsberg which traversed each of the 7 bridges exactly once (see Figure 1). Euler represented this abstractly as a *graph**, and showed by elementary means that no such path exists.

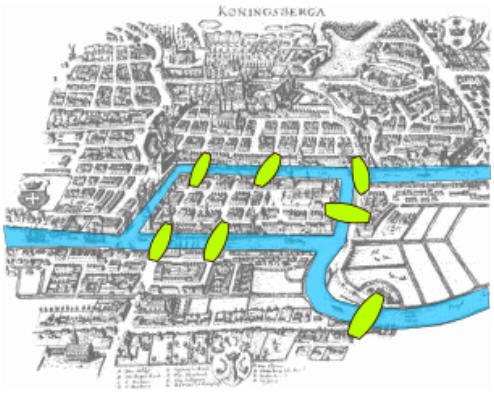


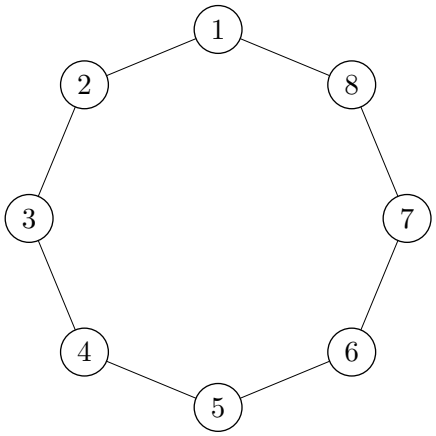
Figure 1: The Seven Bridges of Königsberg (Source: Wikimedia Commons)

Intuitively, a graph is just a set of objects which are connected in some way. The objects are called *vertices* or *nodes*. Pictorially, we usually draw the vertices as circles, and draw a line between two vertices if they are connected or related (in whatever context we have in mind). These lines are called *edges* or *links*.

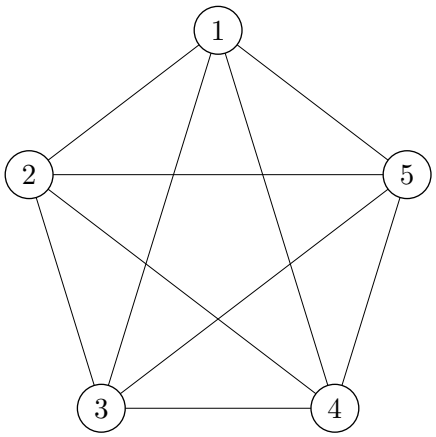
Here are a few examples of abstract graphs.

This is a graph with 8 vertices connected in a circle.

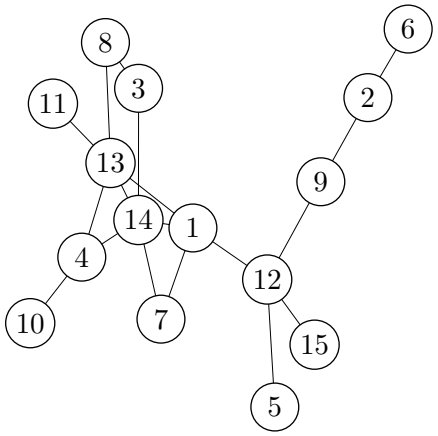
*In this course, graph does not mean the graph of a function, as in calculus. It is unfortunate, but these two very basic objects in mathematics have the same name.



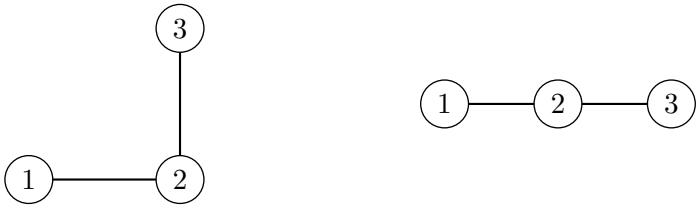
This is a graph on 5 vertices, where all pairs of vertices are connected.



Here is a "random graph" on 15 vertices generated by a computer—in this case, each pair of vertices had a 15% chance of being connected.



Note that the physical location of the vertices in the drawings are unimportant, only which vertices are connected matters. For example, the following two graphs are the same.



Graphs naturally arise in many ways, as they are a convenient way to visualize various situations or complex symptoms. In fact graphs are nearly ubiquitous in mathematics as they can be used to represent different aspects of all kinds of mathematical structures. Consequently, graphs are also prevalent in many scientific fields, where they are often called networks. Technically the two terms are interchangeable, but one typically uses the term network when one is thinking about social or technological connections.

Here are a few examples of situations where graphs arise.

Geometry/Topology: Polygons are vertices and edges in the plane, and can be thought of as graphs. Note that the graph is not the same as the underlying polygon—things like edge length and vertex angles are not considered in the graph. So in some sense, the graphs of a polygon is the polygon without its geometry. This is perhaps not so interesting for polygons, but one can do the same for polyhedra (e.g., the Platonic solids) where looking at the analogous graphs is very useful in the study of surfaces and higher-dimensional objects.

Algebra: Algebra is the study of mathematical structures, e.g., groups, rings and fields, if you know what those are. Graphs are often used to relate how various structures are related to each other (e.g., subgroup lattices), or to understand individual objects, e.g., group actions on graphs). The notion of group actions on graphs has many applications to constructions of *combinatorial designs*, which are important in error-correcting codes and cryptography.

Electric engineering: Electric circuits are graphs, and graph theory has been used in circuit analysis since Kirchoff in the mid 1850's.

Chemistry: Molecular graphs are representations of molecules as graphs—here are the atoms in the molecule and the edges are the bonds. This point of view was first taken up by Cayley in the 1870's.

Geography: Consider a map, say of Europe. Let each country be a vertex and connect two vertices with an edge if those countries share a border. A famous problem that went unsolved for over a hundred years was the *four color problem*. Roughly this states that any map can be colored with at most 4 colors in such a way that no to adjacent countries have the same color. This problem motivated a lot of the developments of graph theory and was finally proved with the aid of a computer in 1976.

Transportation networks: The US Cities and highway system make a graph, with the cities being the nodes and the highways being the edges. More locally, one could consider the Norman Area bus system, where the nodes are the bus stops and two nodes are linked if they are successive bus stops on a single bus line. Similarly, an airline's flight network forms a graph. For these networks, the distance between two nodes is quite important in terms of time/fuel costs, and one can incorporate the distance into the graph by "weighting" the edges. The typical problems in

transportation networks are designing efficient networks and finding efficient ways to route traffic (e.g., what's the best way for you to fly from Oklahoma City to Sydney, Australia?).

As a different kind of transportation network, but entirely analogous, electric power grids and water supply systems also form graphs.

Communication networks: Computer systems in a local network form a graph. So do the landline telephone cable systems and internet routing systems. These can also be thought of as "transportation" networks, where now what is being transported is data. Again, design and routing and principal issues in these networks.

Social networks: In sociology, economics, political science, medicine, social biology, psychology, anthropology, history, and related fields, one often wants to study a society by examining the structure of connections within the society. This could be friend networks in a high school or Facebook, support networks in a village or political/business connection networks. For these sorts of networks, some basic questions are: how do things like information flow or wealth flow or shared opinions relate to the structure of the networks, and which players have the most influence? In medicine, one is often interested in physical contact networks and modeling/preventing the spread of diseases. In some sense, even more basic questions are how do we collect the data to determine these networks, or when infeasible, how to model these networks?

The World Wide Web: One can form a graph of all webpages, and make an edge from Page A to Page B if there is a hyperlink from Page A to Page B. In this case, one should consider directed edges, meaning each edge has a direction which is pictorially indicated with an arrow. Here some basic problems are searching the web and ranking web pages (to get search results in a useful order). Due to the incredible size of the web and amount of information, searching is highly nontrivial. Web page ranking is closely related to the problem of determining how much influence players have in a social network.

Like the web, Twitter also gives rise to a directed social network, where the nodes are the users and the arrows point in the direction of "following."

Game theory/Discrete Dynamical Systems For games (deterministic or not) where there are a finite number of possible moves at each step, you can diagram the game as a graph. Here the vertices are the possible states of the games and the edges represent the moves going from one state to another. (Draw yourself the first few parts of the graph for Tic-Tac-Toe.) Thus the course of the game will be viewed as following a certain path in the associated graph and ending at a "terminal node." More generally graphs can be used to visualize discrete dynamical systems, and some ideas from dynamical systems are extremely useful in social/technological networks (e.g., Google's Pagerank algorithm).

Neurobiology/Artificial Intelligence: The brain is an immensely complex network, and some graph theory can be used to examine the structure of the brain. Many attempts at developing forms of artificial intelligence are based on the idea of neural networks, which are simple feedback networks that can be trained to perform tasks such as optical character recognition or speech recognition.

Being a bit broader with our terminology, we might consider transportation networks, communication networks and the World Wide Web as kinds of social networks as well (they are all products of societies). However I separated them above because they have different features and one typically asks different types for different types of networks. Indeed, communcation and transportation networks motivated a lot of "classical" graph theory, whereas study of social networks has led many newer directions in graph theory. In particular, with the advent of modern computing

and the internet, understanding large networks is a major theme in modernd graph theory.

Our rough plan for the course is as follows.

First, we'll look at some basic ideas in classical graph theory and problems in communication networks. E.g., how can you analyze the efficiency or robustness of a networks. This leads to notions of distance, diameter, k-connectedness and network flow.

Second, we'll give a brief overview of some key themes in social networks and complex (large) networks. Here some topics are notions of centrality, clustering, degree distributions and small world phenomena. In these first two parts we'll get our hands dirty by writing and analyzing some algorithms for these things, but we'll also do exercises by hand.

Third, we'll look at *spectral graph theory*, which means using linear algebra to study graphs, and *random walks on graphs*. This will give us a useful way to study network flow for communication networks and do things like rank webpages or sports teams or determine how influential people are in social networks.

Finally, we'll study *random graphs* to get some insight into large networks. This will be the most "experimental" part of the course, in the sense that, while we hope to do a little theory, much will be learned by generating and working with examples on the computer. Time permitting, we'll spend some time discussing dynamic networks and modeling information flow/disease transmission.

In terms of computer software, we will work initially in Python (current version 2.7.6) for writing simple code to work with graphs, then use built-in algorithms in Sage (current version 5.12, which is compatible with Python 2, but not Python 3) to do more complex things, and possibly use Sage and Python in tandem. While I don't intend to make this a course on Computational Graph Theory or Algorithmic Graph Theory (i.e., how to program everything), I think it's important to have exposure to the nitty-gritty of at least some graph theory algorithms to truly understand the main issues of modern graph theory, which in large part stem from computational complexities.

However, be at peace—no previous experience with this software, or with programming, is required or expected.

The mathematical prerequisites for this course are: (1) some familiarity with proofs, as our Discrete Math or Linear Algebra courses; (2) some linear algebra (eigenvalues, eigenvectors, etc). We'll briefly review some of the necessary linear algebra as we go along, but you really should've have seen it before to hope to be able to get a good understanding of the third part of the course. We'll also need basic probability in the latter two parts of the course. I'll introduce this when the time comes, but it wouldn't hurt to have seen a little before.

Due to both the choice of topics and use of computer software, this will be a very non-traditional course in graph theory, which is why I couldn't find an appropriate textbook and will write my own notes. Based on the goals for the course, I won't be able to cover a lot of things that are covered in a typical graph theory course. Indeed we'll be quite pressed for time with my current goals. I will include some explanations of how to do things in Python and Sage in these notes, and snippets of sample code, but these notes will not contain detailed information on how to use Python and Sage in general—this information can be found elsewhere online, and will be covered in our Computer Lab Meetings.

For fun, I made an example of a social network graph involving some people you may know: the OU Math Department. See Figure 2. Let's take a look at this before we get started in earnest, to give you a better idea of some things we'll be doing throughout the course. This is a collaboration graph. The vertices are current or semi-recent previous faculty and postdocs within the OU Math Department. The postdocs (3-year positions) are all in blue, and regular faculty (tenure/tenure-

track) are in red (current) or black (previous). Two faculty are connected if they have been collaborators (co-authors of the same paper at least once). Faculty who have not collaborated with anyone in the department are not named, but just represented the 11 isolated nodes (nodes not connected to any other nodes). In fact there should be more isolated nodes, but I got tired after drawing 11 of them.

Note, I just made this on my own one afternoon, and there may be some additional collaborations that I've missed—in any case it will probably not be up-to-date for too long. However, let's try to get a little taste of network analysis by making some comments on this graph.

First of all, I was a bit surprised at how connected it is. While our department is quite friendly and social, most mathematics research is quite specialized, and most collaborations tend to occur among people in different universities. (I've written 9 papers since being at OU, of which 3 involve an OU collaborator.) Further, a lot more research in math is solo than in the other sciences. Even taking into account collaborations at the same university, one might expect that each research area is isolated from the other ones: e.g., number theorists just collaborate with other number theorists, geometers just collaborate with other geometers, and so on. However, forgetting the 11 isolated nodes for now, this graph consists of 3 components* of size 2, 1 of size 3, 1 of size 4, and then one large component of size 23: meaning most of the faculty are connected to each other through collaborations, instead of all the different research groups being disconnected from each other. This is part of the *small world phenomena*—in many social networks, things tend to be remarkably well connected.

For the rest of the discussion, let's restrict ourselves to the large component, which looks sort of like a reindeer. In fact, this component looks quite similar to the random graph on 15 vertices presented earlier (which to me, looks like a dog). This visual similarity, in the sense that they both look like balloon animals, was not due to any intentional planning on my part, but just something I noticed after making them. However, it's not mere coincidence—organically-formed networks can be modeled quite well by random graphs. That is what we'll focus on in the last part of the course.

Back to our discussion. Another aspect of small world phenomena, besides having a large component, is what is sometimes known as *six degrees of separation*. This is a notion about "how well connected" a network is. Define the distance $d(u, v)$ between two nodes u and v to be the minimum number of edges one needs to traverse to get from node u to node v. For example, there are many ways one can get from Shankar to Forester—directly, going through Brady, going through Brady, Dani and Clay, and so on. However, the shortest route just consists of 1 edge, so the distance from Shankar to Forester is 1. Similarly the distance from Shankar to Lousma is 2, Shankar to Rafi is 3, and so on.

Six degrees of separation refers to the notion that in many social networks, even though the network may be very large, most people who are connected are not more than 6 steps (distance 6) away from each other. An alternative interpretation is that most people are no more than 6 steps away from a given individual. The number 6 is not so important here, and is not a magic number for all networks—the idea is just that if you pick two people at random, they will be rather close. The farthest apart 2 of these 23 people are is Rubin and Basmajian, at distance 11. But if you pick 2 people in here at random, chances are that their distance will be much smaller. Looking at the alternative interpretation, I'm in the middle in some sense, and I'm within distance 5 from anyone else in the reindeer.

*A *(connected) component* is the part of a graph consisting of all nodes connected to a single given node by some sequence of edges.

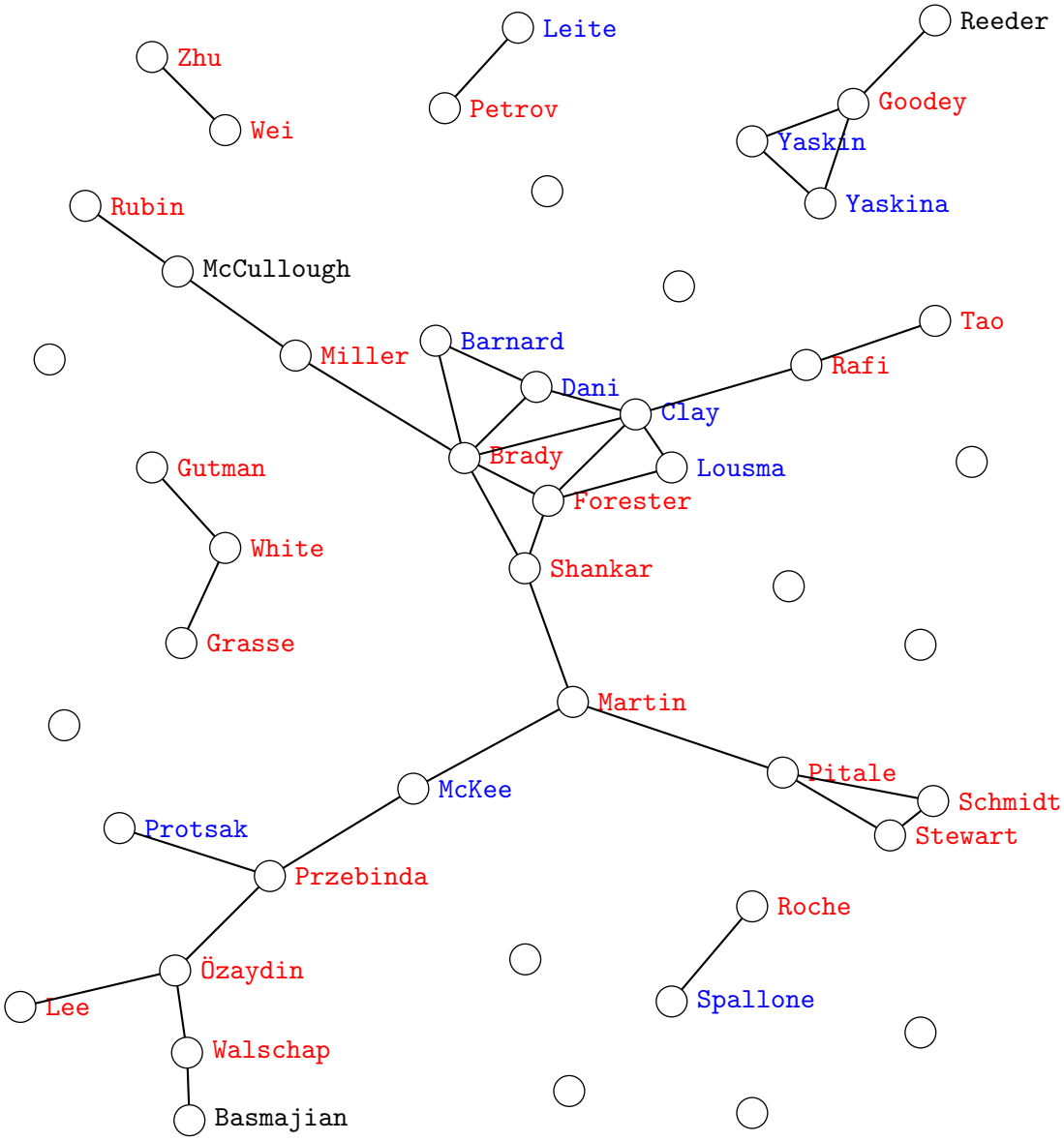


Figure 2: Collaborations within the OU Math Department

One question that's common in many social networks is: who is the "most popular"? For this, we can look at the *degree* of each node, which is the number of edges coming out of it, or equivalently, the number of other nodes it is directly connected to. Then one way to interpret "most popular" is simply having the highest degree, though for a collaboration network, "popular" isn't really a good word—I just mean who's worked with the largest number of people. In this case, Brady has the highest degree (6), followed by Clay (5), then Forester (4).

Another question is who has the most "influence"? You might first think that this is the same question as popularity, but it isn't. In fact for many things, e.g. predicting who might be able to win an election, influence may be more important than popularity, in the way I use these two terms here. Popularity (as we interpreted it) is a local trait. You can determine someone's popularity just by counting their neighbors (the neighboring vertices). Influence is a global trait. For instance, even though Przebinda and Özaydin are tied in terms of "popularity" (both degree 3), Przebinda is one step closer to McKee, and therefore most people in the reindeer, than Özaydin. In other words, Przebinda should be viewed as more "influential" as he is closer to more people in the graph than Özaydin is. Being closer to more people says you are more "central," and let's use the term central now instead of influential. (I don't want you to think our math department is full of clandestine politicking and power struggles, though some departments are! Also remember, this graph also doesn't represent the social structure of our department—only collaborations.)

There are several different ways to measure centrality. Here is a simple one. For each node u, define its centrality as the sum over all other nodes v (in the same component) of $\frac{1}{d(u,v)^2}$. (One could also define it without the squares.) Then the higher the centrality, the closer you are to more nodes. A more sophisticated way to define centrality involves the idea that you should weight these distances by how central v itself is. (Being distance 1 from a more central node is better than being distance 1 from a node on the outskirts.) It may not be clear how to make sense of a definition of centrality that already involves centrality, but this can be done with eigenvalues, and is one of two essential ideas behind Google's Pagerank algorithm.

Another phenomena you may notice is *clustering*. Look at the triangles in the graph. There are 6 in the large component and 1 in the component of size 4. However, the triangles in the large component aren't spread out evenly—5 of them are adjacent (the reindeer's head and hat). This phenomenon that triangles tend to group together in social networks is known as clustering. This just means there tend to be tight-knit groups within social networks. Often one measures clustering to say something about the structure of a network.

If you want to extrapolate qualitative information from the graph, you can, but due to the nature of this particular graph it won't be too precise. For example, let's say you want to use this graph to say something about the research interests of faculty members. (Probably a more natural use for this graph would be as an example in a study of how much being in the same institution factors into collaborations.) While my research interests are closer to those of Brady's or Forester's than say Rubin or Lee as suggested by the distances in the graph, you don't see that my research interests are closer to those of Rafi's than Miller's (or probably Brady's also), even though Rafi and I are farther apart in the graph. In fact, Pitale, Schmidt and I all work in the same area, and we are closer to Roche and Spallone than anyone in the top half of the reindeer. Even though the data is real, you should be careful not to overinterpret it as being a graph of our research interests, or how much we interact. Writing a paper with someone may or may not mean that you both work in the same are most of the time (my paper with Shankar was in an area that neither of us typically work in—in addition, research interests often change over time).

However, there are some things we can do to incorporate more information in the graph. First, one does not get a sense of the strength of the connections. For example Pitale and Schmidt have written many papers together, but Stewart has only written a couple papers with Pitale and Schmidt (these two papers consisted of all 3 authors and 1 additional one). One could weight the edges by counting the number of papers co-authored. This would make it clear that Pitale and Schmidt are closer to each other than either of them are to Stewart. It is also not clear from the graph whether a triangle represents, say, a single paper coauthored by 3 people or 3 different collaborations by each possible pair of the 3 people. This can be incorporated by allowing "faces" (in the sense of a face of a parallelopiped) in graphs, which are called *hypergraphs*. We will look at weighted graphs briefly, but not discuss hypergraphs in this course.

Another aspect not seen in this graph is that this is just a snapshot of all collaborations up to the present time. One can understand more about department research connections by looking at how the network evolves over time. Indeed, social networks tend to be dynamic networks with new nodes being added, nodes being removed and links changing all the time. This is another challenge in data collection and analysis for a given social network. (Further, not all of the collaborations between OU faculty in the graph actually occurred at OU—some collaborations began before one or both parties involved was at OU. Conversely, this might be a factor into who gets and accepts job offers at OU.)

Finally, the main reason for not being able to read too much into this graph is that it's just too small to get sophisticated information. This is the same problem as not having enough data in statistics. If you embed this graph in a larger collaboration graph of all mathematicians in the world, it should be much more clear that Pitale, Schmidt and I all work in the same area, where as Shankar and I typically do not. Even though there may be some "random connections" between people in different research areas, in a huge collaboration graph the number of these random connections will be quite small and can be basically ignored by using more sophisticated analysis techniques. While this may all seem rather frivolous, these ideas actually have immense potential for applications—for example these ideas about using collaboration graphs to distinguish different research areas can be used in things like *machine learning/artificial intelligence* and *face/pattern recognition*.

Chapter 1

Basic Graph Theory: Communication and Transportation Networks

In this section, we will introduce some basics of graph theory with a view towards understanding some features of communication and transportation networks.

1.1 Notions of Graphs

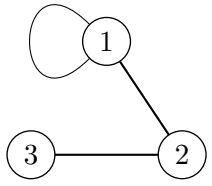
The term graph itself is defined differently by different authors, depending on what one wants to allow. First we will give a fairly typical definition. For elements u and v of a set V, denote by $\langle u, v\rangle$ the unordered pair consisting of u and v.* (Here u and v are not necessarily distinct, and the pair being unordered just means that $\langle v, u\rangle = \langle u, v\rangle$.) Denote by $\mathrm{Sym}(V \times V)$ the set of all unordered pairs $\langle u, v\rangle$ for $u, v \in V$.

Definition 1.1.1. *An* **(undirected) graph** *(or* **network***)* $G = (V, E)$ *consists of a set of* **vertices** *(or* **nodes***)* V *together with an* **edge set** $E \subset \mathrm{Sym}(V \times V)$*. The elements of* E *are called* **edges** *or* **links***. The number of elements in* V *is called the* **order** *of* G*, and we often say* G *is a graph on* V*.*

A priori, the order of a graph could be infinite, i.e., it could have infinitely many vertices. Infinite graphs can be quite useful in theory, but we will focus on networks that arise in real-life situations, which are finite, i.e., they have finitely many vertices.

▶ *Unless otherwise specified, we will always assume our graphs are finite.*

Example 1.1.2. *Let* $V = \{1, 2, 3\}$*. Then* $V \times V$ *is the the set of all order pairs of vertices and* E *should be a symmetric subset of this. Take* $E = \{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle\}$*. To draw the graph, draw and label each node, and draw a link between two vertices if there is an edge between them, like this:*



*This is not standard notation. Most authors write (u, v) or $\{u, v\}$, but I want to reserve (u, v) for the ordered pair and $\{u, v\}$ for the set of u and v.

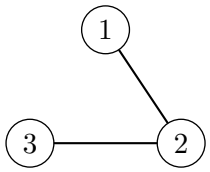


Figure 1.1: The simple graph associated to Example 1.1.2.

In the above example there is an edge from vertex 1 to itself.

Definition 1.1.3. *Let $G = (V, E)$ be a graph. An edge of the form $\langle v, v\rangle \in E$ is called a* **loop**. *If G has no loops, we say G is* **simple**.

It is clear that given any graph, we can make it into a simple graph just by deleting all loops. For instance, the above example gives rise to the simple graph:

Here the edge set is now $E = \{\langle 1, 2\rangle, \langle 2, 3\rangle\}$.

▶ *Unless otherwise specified, we assume all graphs are simple.*

Note that if we are working with simple graphs, then the unordered pair $\langle u, v\rangle$ is simply the set $\{u, v\}$, and you can define an edge as simply a subset of V of size 2. Some authors will use this definition, which implies that any graph for them is simple. (If we are not working with simple graphs, then you can have a loop $\langle v, v\rangle$ whose associated set $\{v, v\} = \{v\}$ has size 1, not 2.) So if you prefer curly braces to angle brackets, feel free to write your edges with those, e.g., $E = \{\{1, 2\}, \{2, 3\}\}$.

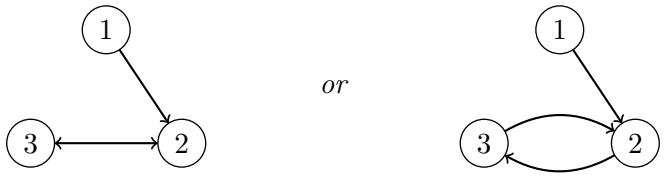
In many instances, the graphs we want to consider are not naturally "symmetric" (undirected). For example, we might want to make a graph of webpages and draw a directed link from one page to another if there is a hyperlink from one page to another. Another example is with citation graphs—graphs of all research documents in a field with a directed link from paper A to paper B if paper A cites paper B. In this case, one should think of these directed links as *ordered* pairs (u, v), rather than unordered pairs $\langle u, v\rangle$. This leads us to the following definition.

Definition 1.1.4. *A* **directed graph** *(or* **digraph***) $G = (V, E)$ consists of a set V of vertices and an edge set $E \subset V \times V$. The elements of E are called* **(directed) edges** *or* **links**. *If G contains no loops, then we say G is* **simple**.

If $e = (u, v) \in E$ is a directed edge, we say e is an edge **from** *u* **to** *v, or* **starting at** *u* **and ending at** *v. Further u is called the* **initial vertex** *of e and v is called the* **terminal vertex** *of e.*

▶ *Except where otherwise specified, the term graph used by itself means undirected graph.*

Example 1.1.5. *Consider $V = \{1, 2, 3\}$ and $E = \{(1, 2), (2, 3), (3, 2)\}$. We draw simple directed graphs as follows. If $(u, v) \in E$ but $(v, u) \notin E$, then we draw a edge from u to v with an arrow pointing towards v. If (u, v) and (v, u) are both in E, then we can either draw a single edge from u to v with an arrow on each end or two different edges, one with an arrow to u and one with an arrow to v.*

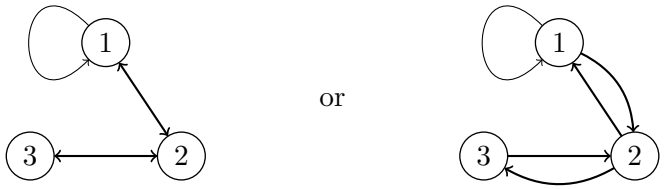


For directed graphs, edges are thought of as having direction, so the edge $(2,3)$ *is considered different than the edge* $(3,2)$*, and this digraph has 3 edges not 2, as one might think from the drawing on the left.*

Note that we can consider undirected graphs as a special case of directed graphs in the following way. Suppose $G = (V, E)$ is an undirected graph. Then one can consider a directed graph $G' = (V, E')$ on the same vertex set V where now the edge set

$$E' = \{(u, v), (v, u) : \langle u, v\rangle \in E\}$$

contains both edges (u, v) and (v, u) for any undirected edge $\langle u, v\rangle$ in G. (For non-simple graphs, the loop $\langle v, v\rangle$ just becomes (v, v).) E.g., for Example 1.1.2 the edge set is $E' = \{(1, 1), (1, 2), (2, 1), (2, 3), (3, 2)\}$. The corresponding directed graph can then be drawn as follows:

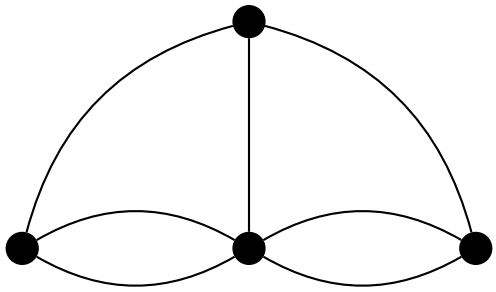


(For non-simple directed graphs, for any loop, we just draw an arrow on one end of the loop.)

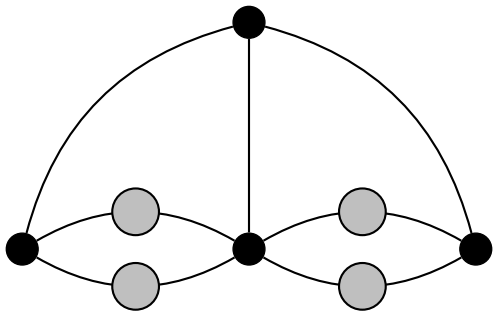
Consequently, we can think of undirected graphs simply as directed graphs whose edge sets E are *symmetric*, i.e., $(u, v) \in E$ implies $(v, u) \in E$. (The set of unordered pairs of elements of V corresponds to symmetric subsets of the set $V \times V$ of ordered pairs, which is why I used the notation $\mathrm{Sym}(V \times V)$ above.) The only real difference is that when counting edges, the directed graph will have more edges (precisely twice as many for simple graphs.) This perspective is useful as we can study both directed and undirected graphs in a unified framework. With this in mind, I will often use the ordered pair notation (u, v) for edges in an undirected graph. In this case, I will write the edge set as a symmetric subset of $V \times V$—for example, for Example 1.1.2, I may write the edge set as $E = \{(1, 1), (1, 2), (2, 1), (2, 3), (3, 2)\}$, with the understanding that (u, v) and (v, u) represent the same edge (so this graph still has 3 edges, not 5).

Realizing the edge set as a subset of the ordered pairs is also more natural from the matrix point of view, and often useful from an algorithmic point of view.

There are two other generalizations of graphs worth mentioning now. Some authors allow multiple edges between vertices in their definition of graph—I will call these **multigraphs**. For instance, consider the Seven Bridges of Königsberg in Figure 1. Euler considered this situation with the multigraph formed by making each landmass a vertex and each bridge an edge:



However, one can reduce multigraphs to graphs by adding in appropriate vertices, e.g., for Euler's example above, we can just add in new vertices along certain edges to get a graph:



Thus we will not have much reason to consider multigraphs except in certain special cases.

Another generalization of graph that we will sometimes consider is a **weighted graph**. This is just a (directed or undirected) graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$ that assigns each edge a weight. For instance, if the graph represents cities on a map, then a natural weight function to consider would just be the distance between the cities. Another possible weight function is the cost to get from one city to the other. Note that a weighted graph generalizes the concept of multigraph: a multigraph can be considered as a weighted graph where the weight of an edge (u, v) is just the number of edges from u to v in the corresponding multigraph. We will say more about weighted graphs later.

Exercises

Exercise 1.1.1. *(a) For $V = \{1, 2\}$ and $V = \{1, 2, 3\}$, draw all possible graphs on V.*

(b) How many possible graphs are there on $V = \{1, 2, 3, 4\}$? Draw all such graphs having 4 edges.

(c) Draw all possible directed graphs on $V = \{1, 2\}$. Then draw all possible directed graphs $V = \{1, 2, 3\}$ which contain the edge $(1, 2)$.

1.2 Representations of Graphs

Now let's discuss different ways one can represent graphs in Python. You can work directly with sets in Python 2.7 (using curly brackets, as in math), so the most naive way you can represent a graph G is with an (ordered) list (denoted by square brackets) consisting of the vertex set V and the edge set E. For example, the graph in Figure 1.1 can be represented as:

Python 2.7
```
>>> V = {1, 2, 3}
>>> V
set([1, 2, 3])
>>> E = [ {1, 2}, {2, 3} ]
>>> E
[set([1, 2]), set([2, 3])]
>>> G = [V, E]
>>> G
[set([1, 2, 3]), [set([1, 2]), set([2, 3])]]
```

(Lines beginning with `>>>` denote input to the Python interpreter, and other lines denote the Python output. I wrote V, E, and G on separate lines after the definitions just so you can see how Python will output this data to you when you want it later. E.g., the Python output `set([1, 2])` just means the set consisting of 1 and 2, or what we would typically write in mathematical notation as $\{1, 2\}$.)

Also note that while spacing (indentation) is important in Python for nested statements over multiple lines (e.g., "for loops" or "if... then" statements), it is not important within individual lines. In particular, I could write something like `V={1,2,3}` or `V = { 1 , 2 , 3 }` for the first line with the same result.)

Here V is represented as a set of vertex names, and the edge set E is an ordered list of edges e, where each edge is represented as a set of size 2. For technical reasons, using the built-in set type in Python, one cannot write E as a set, i.e., `E = { {1, 2}, {2, 3} }` will result in an error, because Python does not by default handle sets of sets, or sets of lists. For similar reasons, G must also be defined as a list `G=[V,E]`, rather than a set `G={V,E}`. Even if it were possible to define `G={V,E}` in Python, it is better to define it as a list, because to actually do things with G, one will need to recover the vertex and edge sets V and E. If you define G as a list, you can just get the vertex set back with `G[0]` and the edge set by `G[1]`. (Python naturally numbers list positions starting at 0, not at 1.) But if one could and did define G as a set, there is no order, so it would not be as easy to recover the vertex set or the edge set.

Many programming languages do not have a built in data structure for sets (i.e., unordered lists), but one can similarly represent a graph in terms of lists (or arrays). In this case one can write the vertex sets as a list, and the edge set as a list of ordered pairs (a list of lists of size 2). For example, this same graph can be represented as

Python 2.7
```
>>> V = [1, 2, 3]
>>> V
[1, 2, 3]
>>> E = [ [1, 2], [2, 1], [2, 3], [3, 2] ]
>>> E
[[1, 2], [2, 1], [2, 3], [3, 2]]
>>> G = [V, E]
>>> G
[[1, 2, 3], [[1, 2], [2, 1], [2, 3], [3, 2]]]
```

This method of representing edges as (ordered) lists of size 2 is of course also advantageous as one can represent directed graphs in the same way.

However, these naive ways of representing a graph in a computer is not so useful in practice. For example, consider the following problem.

Given a node u of a (directed or undirected) graph $G = (V, E)$, we say a node $v \in V$ is **adjacent** to u, or a **neighbor** of u, if $(u, v) \in E$. (Note for undirected graphs, being neighbors is a symmetric relation, but not so for directed graphs, i.e., v may be adjacent to u without u being adjacent to v.) Write an algorithm which, given a node u returns all the neighbors v of u. This is one of the most basic procedures one will want to do when working with graphs.

Let's see how to do this using where we represent V as a list and E as a list of lists of size 2, as in the latter snippet of code. (Thus we are working with directed edges.) In fact, whether V is a set or a list is not important to our algorithm—however it does make a difference in syntax that each edge is a list of size 2, not a set.

Python 2.7

```
>>> V = [ 1, 2, 3 ]
>>> E = [ [1, 2], [2, 1], [2, 3], [3, 2] ]
>>> G = [V, E]
>>>
>>> def VE_neighbors(G, u):
...     neigh = set()               # start with an empty set
...     E = G[1]                    # let E be the edge set
...     for e in E:
...         if e[0] == u:           # for each edge of the form (u,v)
...             neigh.add(e[1])     # add v to the set neigh
...     return neigh
...
>>> VE_neighbors(G, 1)
set([2])
>>> VE_neighbors(G, 2)
set([1, 3])
```

Here I have written a function `VE_neighbors` that takes as input two things: the graph `G=[V,E]` represented in the above "vertex set-edge set representation" (I use "VE" at the beginning of this function name to indicate this), and the vertex `u` one wants to find the neighbors of. The algorithm is to just go through each element of the edge set E, and check if the edge starts at u (i.e., is of the form (u, v)), and if so, add the corresponding element to the set of neighbors. (If `e=[u,v]` is a directed edge, then `e[0]` returns the initial vertex `u` and `e[1]` returns the terminal vertex `v`.) The remarks after the hash signs `#` are comments to help you understand the code and ignored by the Python interpreter. (You should always comment your code.)

Note: one uses the double equals `==` in the `if` statement to test if two things are the same—do not use `e[0]=u`, which will set `e[0]` equal to `u`.

Then at the end of this snippet of code, I test the function `VE_neighbors` on the graph from Figure 1.1 for the vertices 1 and 2. As expected, the Python output says the set of neighbors for the vertex 1 is just $\{2\}$, and the set of neighbors for the vertex 2 is $\{1, 3\}$. In this implementation, I encoded the neighbors of u as a set, rather than a list, but one could do this also (Exercise 1.2.2).

When we write programs, we are often concerned with efficiency, particular if we are working with a large amount of data. For small graphs, this not a big deal, but if you want to work with graphs with hundreds or thousands or millions of nodes, it's crucial. The algorithm `VE_neighbors` requires going through each element of the edge set E, so we say the running time is $O(|E|)$.

(This notation, called Big Oh notation, will be explained in detail later—roughly it means that the algorithm requires on the order of $|E|$ steps to finish.) Here $|E|$ denotes the size of the set E, i.e, the number of (in this case directed) edges.

Note that for a not-necessarily-simple directed graph $G = (V, E)$ on n nodes, the maximum number of possible edges $|E|$ is n^2—this is simply the number of ordered pairs $V \times V$ (see Exercise 1.2.1). Thus we can give an upper bound for the run time of this algorithm as $O(n^2)$. This is horribly inefficient for such a basic operation, and we will see we can do much better using a different representation for a graph.

Adjacency matrices

Whenever you have a finite collection of objects, and some relations between them, you can keep track of them in a table. For example, in linear algebra if you're working with two variables x and y, you can keep track of linear combinations

$$3x + 2y$$
$$x - y$$

by just writing the coefficients in a box

$$\begin{pmatrix} 3 & 2 \\ 1 & -1 \end{pmatrix}.$$

Of course one needs to keep track of the order of x and y, so you know x corresponds to the first column, and y the second.

We can do something similar (though not exactly the same) for graphs.

Definition 1.2.1. *Let $G = (V, E)$ be a directed or undirected graph, not necessarily simple.* Write V as an* ordered *set $\{v_1, v_2, \ldots, v_n\}$. The* **adjacency** *(or* **incidence***)* **matrix** *for G (with respect to the ordering $v_1, v_2, \ldots, v_n$) is the $n \times n$-matrix*

$$A = (a_{ij}), \quad a_{ij} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & (v_i, v_j) \notin E. \end{cases}$$

Example 1.2.2. *Let $V = \{1, 2, 3\}$, as an ordered set. Then the adjacency matrix for the undirected graph in Example 1.1.2 is*

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}.$$

For clarity, I labeled which rows and which columns correspond to which vertex in red, but I won't typically do this. In other words, there is an edge from 1 to 1 (a loop), and edge from 1 to 2, an edge from 2 to 1 an edge from 2 to 3, and an edge from 3 to 2.

*From now on, we will often implicitly realize the edge set E for undirected, graphs as a symmetric set of ordered pairs (u, v), rather than a set of unordered pairs.

Similarly, the adjacency matrix for the directed graph on V in Example 1.1.5 is

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}.$$

In other words, there is a (directed) link from 1 to 2, and a link both directions between 2 and 3.

Note that these adjacency matrices depend on the ordering we chose for V. If for some perverse reason, we wanted to use a different ordering, say $V = \{3, 2, 1\}$ then, e.g., the adjacency matrix for the directed graph in Example 1.1.5 is

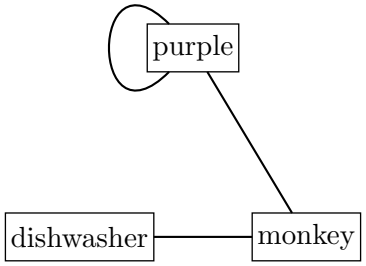
$$\begin{array}{c} \\ 3 \\ 2 \\ 1 \end{array} \begin{array}{c} \begin{array}{ccc} 3 & 2 & 1 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array}.$$

Let's make a couple of elementary observations now. First, having a loop $(v_i, v_i) \in E$ means that the i-th diagonal element of the matrix $a_{ii} = 1$, so having no loops (i.e., being simple) is equivalent to the statement that all the diagonal entries of the adjacency matrix are zero. (As we see from this argument, this fact does is independent of which ordering we choose for V.)

Moreover, note that the (a priori directed) graph G is undirected[†] if an only if the matrix A is symmetric. Again this will not depend on the ordering we choose for V. For any ordering, G being undirected means that $(v_i, v_j) \in E$ is equivalent to $(v_j, v_i) \in E$ for all $1 \leq i, j \leq n$, which means the value of a_{ij} must equal the value of a_{ji} for all i, j, which is equivalent to A being symmetric as asserted.

Next, given some vertex v_i, it is easy to read off its neighbors—just go to the i-th row and look at which spots have a 1. If there is a 1 in the column corresponding to v_j, this means there is a (directed) edge from v_i to v_j. This process requires going through each element of a single row in A, which has n elements, so the running time for such an algorithm is $O(n)$. This is in general much better than the $O(n^2)$ bound we got for using the "vertex set-edge" set representation above.

Note that to represent an arbitrary graph in a computer, we need a little more than the just the adjacency matrix—we also need the ordered list of vertices. For example, the graph



with respect to the vertex ordering $\{\text{purple}, \text{monkey}, \text{dishwasher}\}$ has the same adjacency matrix we saw in the first part of Example 1.2.2. Of course this graph and the graph from Example 1.1.2 are essentially the same—only the names of the vertices have changed, but they are technically different graphs. We will discuss this more when we get to the notion of *graph isomorphisms* below.

[†]Technically, I mean G can be viewed as an undirected graph, i.e., that E is symmetric.

For now, I just want to make the point that to use adjacency matrices to encode the complete information about any graph $G = (V, E)$, we need to store the ordered pair (V, A), where V is an ordered set of vertices and A is the associated adjacency matrix.

For example, we can encode the purplemonkeydishwasher graph in Python as:

Python 2.7
```
>>> V = [ "purple", "monkey", "dishwasher" ]
>>> A = [ [ 1, 1, 0 ], [ 1, 0, 1 ], [ 0, 1, 0 ] ]
>>> G = [V, A]
>>> G
[['purple', 'monkey', 'dishwasher'], [[1, 1, 0], [1, 0, 1], [0, 1, 0]]]
```

Here we represent the adjacency matrix

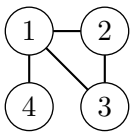
$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

as a list of lists. The lists `[1, 1, 0]`, `[1, 0, 1]` and `[0, 1, 0]` represent the three rows of A, and then the matrix A is encoded in Python as a list of the three row vectors. Then we can access, e.g., the top row of A by the code `A[0]` (this will give you `[1, 1, 0]`) and the individual entries of the top row by `A[0][0]`, `A[0][1]` and `A[0][2]`.

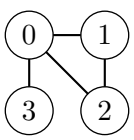
However, we don't really care about purplemonkeydishwashers in this class. We are primarily interested in just studying the structure of graphs in this class. The names of the vertices will only be important when we are looking at specific networks/applications (e.g., the graph in Figure 2). Consequently, to simplify things, we will often assume—at least when we are working by hand—that we are working with an ordered vertex set of the form $V = \{1, 2, \ldots, n\}$. When we are working with graphs on the computer, we will typically assume the vertex set $V = \{0, 1, \ldots, n-1\}$. With this assumption in mind, we can simplify our lives a bit and represent a graph G by just its adjacency matrix A. For instance, by default we will interpret the adjacency matrix

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

as representing the graph



if we are working by hand, but as



if we are working on the computer.

The reason for the difference working by hand versus on the computer is that humans naturally count from 1, where as computers naturally count from 0.* Namely, if $V = \{1, 2, 3, 4\}$ and you want to check if there is an edge from vertex 1 to vertex 3, you would look at the entry a_{13} of $A = (a_{ij})$ working by hand, but you would need to look at `A[0][2]` in Python. The need to shift indices in Python is just an unnecessary complication that we can avoid by assuming our vertex set is $V = \{0, 1, 2, 3\}$, for then the entry `A[0][2]` tells us about the existence or nonexistence of an edge from vertex 0 to vertex 2.

Now let's explain the algorithm to find the neighbors of a given vertex u in a graph G. Assume $V = \{0, 1, \ldots, n-1\}$, and say we want to find vertex i. (We'll often use i and j to denote vertices when are vertex set is $\{0, 1, \ldots, n-1\}$ or $\{1, 2, \ldots, n\}$.) Let A be the associated adjacency matrix. Then `A[i]` gives the i-th row of A, and we just need to go through each element of the row, and if there is a 1 in position j of this row, we add vertex j to our (initially empty) list of neighbors for i. The code, with an example on the above graph, is here.

Python 2.7

```
>>> def neighbors(A, i):
...     n = len(A)                  # let n be the size (number of rows) of A
...     neigh = []                  # start with an empty set neigh
...     for j in range(n):
...         if A[i][j] == 1:        # for each index 0 <= j < n
...             neigh.append(j)     # append j to the list neigh if the i-th
...     return neigh                #     row has a 1 in the j-th position
...
>>> A = [ [ 0, 1, 1, 1 ], [1, 0, 1, 0], [1, 1, 0, 0], [1, 0, 0, 0] ]
>>> neighbors(A,0)
[1, 2, 3]
>>> neighbors(A,1)
[0, 2]
>>> neighbors(A,2)
[0, 1]
>>> neighbors(A,3)
[0]
```

Adjacency Lists

There is a third common way to represent graphs, and this is with *adjacency lists*. Fix a (directed or undirected, simple or not) graph $G = (V, E)$—we do not need to assume V is ordered or consists of numbers. An adjacency list for G is merely a list of all the vertices $v \in V$ together with its set of neighbors $n(v) \subset V$. This can be implemented in Python with a structure known as a *dictionary*.

You can think of a dictionary in Python as basically a table consisting of keywords (called *keys*) and their associated data/definitions (called *values*). A dictionary is defined using curly braces like sets. Each dictionary entry is given in the form `key:value`, and the entries are separated by commas. For example, if I wanted to define a dictionary that gave me course titles associated to the course numbers I am teaching this semester, I can enter this as follows

*This is also one reason why androids don't make good life partners.

Python 2.7
```
>>> courses = { 4383 : "Cryptography", 4673 : "Graph Theory", \
... 5383 : "Cryptography", 5673 : "Graph Theory" }
>>> courses[4383]
'Cryptography'
>>> courses[5383]
'Cryptography'
>>> courses[4673]
'Graph Theory'
>>> courses[5673]
'Graph Theory'
```

Note the keys and the values can be numbers or strings (you can define strings in Python using single or double quotes). In fact, the values can be other things like lists or sets also. The single backslash on the first line just means the input will be continued on the subsequent line. Then we see we can access the entries of the dictionary by using the key in square brackets, in the same way we would access the elements of a list using their index.

Using this dictionary structure, we can encode our purplemonkeydishwasher graph as an adjacency list as follows

Python 2.7
```
>>> G = { "purple" : { "purple", "monkey" }, \
... "monkey" : {"purple", "dishwasher"}, \
... "dishwasher" : { "monkey" } }
>>> G["monkey"]
set(['purple', 'dishwasher'])
```

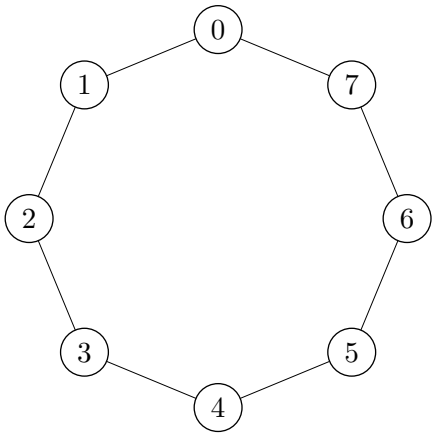
Here the keys are strings, the names of the vertices, and the values are the sets of neighbors, encoded as sets of strings. For instance, the first line says that the node `purple` is assigned the neighbors `purple` and `monkey`. The order in which the vertices are given in the adjacency list is irrelevant. One could alternatively encode the neighbors of each vertex as lists instead of sets.

Note that, conversely, given an adjacency list, one can reconstruct the graph. One simply draws all the vertices (keys) in the adjacency list and draws the (a priori directed) edges from each key to each of its neighbors. (Do this now for the purplemonkeydishwasher adjacency list.) Hence the adjacency list structure gives a valid way to represent a graph (i.e., all the information about the graph is present in the adjacency list).

By design, finding the neighbors of a given vertex using an adjacency list takes only one step! Using our Big Oh notation, which I'll formally get to soon, we would say this can be done in $O(1)$, or constant, time. In other words, it doesn't matter how many vertices there are in the graph, you just look at the entry for the vertex you want, which is the set of neighbors.*

Remarks on implementation: Dictionaries work a bit differently than lists in Python, so you can't append things to a dictionary. If you want to make an adjacency list in Python, and not enter everything by hand the easiest way is to make a list `al` of ordered pairs of the form $(v, \{\text{neighbors of } v\})$. For example, to make an adjacency list for the following graph

*Technically, there are a couple of issues here: (1) We've ignored the time it takes to *locate* the entry for a given vertex, however this can be implemented to be done very very quickly. (2) If we want to actually, say, print out the list of neighbors, the amount of time this takes depends upon the amount of neighbors, which is *a priori* only bounded by the order n of the graph. However, for large graphs that arise in practice, the number of neighbors of any vertex is generally much much smaller than n.



we can use the following code

Python 2.7

```
>>> al = []
>>> for x in range(8):
...     al.append((x, {(x-1)%8, (x+1)%8}))     # for x = 0, 1, 2, ..., 7
...                                # associate the set {x-1 mod 8, x+1 mod 8}
>>> al
[(0, set([1, 7])), (1, set([0, 2])), (2, set([1, 3])), (3, set([2, 4])),
(4, set([3, 5])), (5, set([4, 6])), (6, set([5, 7])), (7, set([0, 6]))]
>>> G = dict(al)
>>> G
{0: set([1, 7]), 1: set([0, 2]), 2: set([1, 3]), 3: set([2, 4]), 4: set([3, 5]),
5: set([4, 6]), 6: set([5, 7]), 7: set([0, 6])}
>>> G[7]
set([0, 6])
```

Here the command `x%8` returns x mod 8, which is the value $r \in \{0, 1, 2, \ldots, 7\}$ such that $8 = qx + r$, i.e., x mod 8 is (at least for $x \geq 0$) the remainder upon dividing x by 8. So, for $0 \leq x \leq 6$, $x + 1$ mod 8 is just x, for $x = 7$ it is `8%8` $= 0$. Similarly, for $1 \leq x \leq 7$, $x - 1$ mod 8 is just x, whereas for $x = 0$ it is `-1%8` $= 7$. In other words, by using the mod function we can use addition/subtraction to right/left shift the numbers $\{0, 1, 2, \ldots, 7\}$ with the convention that we wrap around at the edges.

Adjacency matrices versus adjacency lists

When working with graphs on computers, one typically uses either the adjacency matrix representation or the adjacency list representation. The vertex set-edge set representation that we used for the standard mathematical definition is too cumbersome and slow to work with in actual algorithms. We've seen this for just the problem of finding the neighbors of a given vertex, where the adjacency list representation runs in constant time ($O(1)$), the adjacency matrix representation runs in linear time ($O(n)$), and the vertex set-edge set representation runs in quadratic time ($O(n^2)$).

Adjacency matrices are suitable for small graphs, and have some advantages over adjacency lists. As an example, suppose you have a directed graph G on $V = \{1, 2, \ldots, n\}$ and want to find all the vertices with an edge *to* a fixed vertex j (the inverse to the problem of finding neighbors). With an adjacency matrix, one just looks at the j-th column of an adjacency matrix, where as

things are a bit more complicated with the adjacency list. In addition, it is easier to go between theory and practice using matrices (much of the theory is easier to present in terms of matrices, and some of the coding is also).

For large graphs, the adjacency list representation is typically far superior in practice, particularly for *sparse graphs*, i.e., graphs with relatively few edges (closer to n than n^2). Social networks tend to be rather sparse. (Consider the graph of webpages where the directed edges are hyperlinks. According to Kevin Kelly's *What technology wants* (2010), there are about a trillion webpages and each webpage has, on average, about 60 out of a possible 1 trillion links. If this graph weren't sparse, any useful sort of web searching might be virtually impossible.)

For these reasons, we will primarily use adjacency matrices, at least at the beginning of this course. Towards the end of the course, when we want to work with large graphs, we won't program our own algorithms for everything, and will use the graph theory library in SAGE, which (I believe) uses primarily adjacency lists.

Exercises

Remarks on programming exercises: In all exercises that I ask you to code up a function, you must also test your function on some examples. I will let you choose your own examples to test on (you might choose some from the notes, or some more complicated ones). The more complicated the code is, the more testing you should do. In this section, testing your code on a couple of examples should suffice to convince you (and me) whether it works correctly *all of the time* or not.

In coding, choosing good examples to test your code on is of paramount importance—you should try to test different situations (e.g., directed and undirected, simple or not, include vertices with no neighbors) as it often happens that code will fail for certain very specific cases (for mathematical code, it is often extreme cases, such as code failing when some parameter is minimal or maximal). You also need to choose test cases where you can easily verify that the answer you get is correct (or at least seems reasonable if you don't know the correct answer yourself). (Of course, the first step is to get the code to run without any errors.)

When there is a bug, it is often helpful to choose good examples and examine how the output differs from what it should be to figure out what the bug is. Many times one can figure out what the bug is just by looking a few sample inputs and outputs, and not even looking at the original code! (Though this approach comes easier with experience, but it can be very helpful to try to reason out how the computer is getting from your input to its output.)

If you are having trouble getting your code to run correctly, the first thing you should try to do is test different parts of your code separately. You can also try printing out the values of variables at various steps to help see what is going on.

Exercise 1.2.1. *Let V be a set with n elements.*

(a) How many simple undirected graphs are there on V? What is the maximum number of possible edges? What if we don't require simple?

(b) How many simple directed graphs are there on V? What is the maximum number of possible edges? What if we don't require simple?

Exercise 1.2.2. *Write an analogue of the function* `VE_neighbors`*, called* `VE_neighbors_list`*, that uses a list instead of a set for* `neigh`*, and consequently returns a list instead of a set. (Read the note above about testing your code.)*

Exercise 1.2.3. *Let G be a graph, directed or undirected, simple or not, on $V = \{0, 1, \ldots, n-1\}$. Let A the adjacency matrix for G (with respect to our usual ordering on V). Write a function called* `AM_to_AL`*, whose input is the adjacency matrix A and output is the adjacency list for G.*

Exercise 1.2.4. *Let G be a graph, directed or undirected, simple or not, on $V = \{0, 1, \ldots, n-1\}$, given as an adjacency list. Write a function called* `AL_to_AM`*, whose input is G and output is the adjacency matrix A for G (with respect to our usual ordering on V).*

1.3 Basic Algorithm Analysis

In this section we will explain the notion of *algorithms* and how to analyze their efficiency. To do this, we will first introduce Landau's Big Oh notation and discuss asymptotic growth.

1.3.1 Asymptotic growth and Big Oh notation

Let $\mathbb{N} = \{1, 2, 3, \ldots\}$ and $\mathbb{R}_{>0}$ denote the set of positive real numbers. Recall a function f on $\mathbb{N}$ is just the same thing as a sequence of numbers $(a_n)_n$ by taking $a_n = f(n)$.

Definition 1.3.1 (Big Oh, Version 1)**.** *Consider functions $f, g : \mathbb{N} \to \mathbb{R}_{>0}$, i.e., $(f(n))_n$ and $(g(n))_n$ are sequences of positive real numbers. We say $f(n)$ is* **(big) O** *of $g(n)$ if there exists a constant C such that $f(n) \leq Cg(n)$ for all $n \in \mathbb{N}$. In this case, we write $f(n) \in O(g(n))$ or $f(n) = O(g(n))$.*

Roughly what $f(n) \in O(g(n))$ means is that, for sufficiently large values of n, $f(x)$ grows no faster than $g(n)$. We can think of $O(g(n))$ as the class of functions which don't grow faster than $g(n)$, hence the notation $f(n) \in O(g(n))$. Typically for us $f(n)$ and $g(n)$ will be increasing functions that go to infinity, and you can think of $f(n) \in O(g(n))$ as meaning $f(n)$ is asymptotically $\leq$ (a constant times) $g(n)$. Getting a basic understanding of asymptotic growth rates is essential to understand which how efficient various algorithms are.

We remark that the notation $f(n) = O(g(n))$ is more common, though it is a bit misleading—$f(n) = O(g(n))$ does not mean $g(n) = O(f(n))$. It's usage is probably due to the fact that it is more intuitive for asymptotic expressions. For example, if $f(n)$ is the number of primes less than n, the Prime Number Theorem says

$$f(n) \sim \int_2^n \frac{1}{\log t}\, dt$$

(this is about $n/\log n$), so we can think of

$$f(n) = \int_2^n \frac{1}{\log t}\, dt + \epsilon(n)$$

where ϵ is some error term less than $n/\log n$ for n large. The Riemann Hypothesis gives a bound on the error term: $\epsilon(n) = O(\sqrt{n}\log n)$. Using an equals sign in our Big Oh notation allows us to write our asymptotic for $f(n)$ as

$$f(n) = \int_2^n \frac{1}{\log t}\, dt + O(\sqrt{n}\log n).$$

(Here there are a couple of technicalities with the definition we gave for our Big Oh notation: $\epsilon(n)$ is not always a positive number, and $\sqrt{n}\log n = 0$ for $n = 1$. We'll explain how to define Big Oh notation in a bit more generality below.)

In any case, I will primarily stick to the $f(n) \in O(g(n))$ notation in this course.

Example 1.3.2. *Let $f : \mathbb{N} \to \mathbb{R}_{>0}$ be a bounded function. Then $f(n) = O(1)$.*

Proof. By definition, we know there exists a constant M such that $0 < f(n) < M$ for all n. Consequently, taking $C = M$, we see $f(n) \leq C \cdot 1$ for all $n \in \mathbb{N}$. □

Example 1.3.3. *Consider a polynomial $f(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$ which is positive on each $n \in \mathbb{N}$ (e.g., this is true if each $a_i > 0$). Then $f(n) = O(n^d)$.*

In particular, we have things like $3n^2 + 5n - 2 \in O(n^2)$, so $f(n) \in O(g(n))$ does not necessarily mean that $f(n) \leq g(n)$ for n large—i.e., the constant C in the definition is important. Also, $f(n) = 5n^3$ is $O(n^3)$, $O(n^4)$, $O(n^5)$, and so on, but not $O(1)$, $O(n)$ or $O(n^2)$ (see Exercise 1.3.1).

Proof. Note that for $n \in \mathbb{N}$, we have

$$f(n) \leq |a_d| n^2 + |a_{d-1}| n^d + \cdots + |a_1| n^d + |a_0| n^d \leq C n^d$$

where $C = |a_d| + |a_{d-1}| + \cdots + |a_0|$. □

Proposition 1.3.4 (Transitivity)**.** *Suppose $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$. Then $f(n) \in O(h(n))$.*

Proof. By assumption, we know there are constants C_1 and C_2 such that $f(n) \leq C_1 g(n)$ and $g(n) \leq C_2 h(n)$ for all $n \in \mathbb{N}$. Hence $f(n) \leq C h(n)$ for all n, where $C = C_1 C_2$. □

Again thinking of $O(g(n))$ as the class of functions which grow no faster than $g(n)$, this means if $g(n) \in O(h(n))$, then anything in $O(g(n))$ lies in $O(h(n))$, i.e., $O(g(n)) \subset O(h(n))$. Consequently, our example about polynomials shows we have the following nested sequence of asymptotic classes:

$$O(1) \subset O(n) \subset O(n^2) \subset O(n^3) \subset \cdots$$

If $f(n) \in O(n^d)$ for some d, we say that $f(n)$ has (at most) **polynomial growth**, because it grows no faster than some polynomial. In fact it's not hard to see that all of these $O(n^d)$ classes are different, i.e., the inclusions above are strict inclusions. (See Exercise 1.3.1 below.) For example, $O(n^3)$ contains (positive) polynomials $f(n)$ of degree ≤ 3, whereas $O(n^2)$ will only contain polynomials of degree ≤ 2. (These classes contain other functions besides polynomials as well, e.g., $6n^{2.34567} + n \log n + (-1)^n \in O(n^3)$.)

Now let's give alternative criteria for a function $f(n)$ to be $O(g(n))$, which will give us the right definition even when $f(n)$ and $g(n)$ are not necessarily positive (and sometimes undefined at some values).

Proposition 1.3.5. *Let $f, g : \mathbb{N} \to \mathbb{R}_{>0}$. Then the following are equivalent*

1. *$f(n) \in O(g(n))$;*
2. *There exist constants C, N such that $f(n) \leq C g(n)$ for all $n > N$.*
3. *The sequence of numbers $\left(\frac{f(n)}{g(n)}\right)_n$ is bounded.*

In particular, if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists and is finite, then $f(n) \in O(g(n))$.

Proof. Clearly 1 $\implies$ 2, since they are equivalent if we take $N = 0$. On the other hand suppose 2 holds for some constants C and N. Let $C_0 = \max\{\frac{f(n)}{g(n)} : 1 \le n \le N\}$. Then by definition we have $f(n) \le C_0 g(n)$ for $1 \le n \le N$ and $f(n) \le Cg(n)$ for $n > N$. Thus, for any n, we have $f(n) \le C'g(n)$, where $C' = \max\{C, C_0\}$. Hence 2 $\implies$ 1, and we have the equivalence of the first two conditions.

Now let us show 1 $\iff$ 3. First suppose 1 holds, i.e., there exists C such that $f(n) \le Cg(n)$ for all n. Then, using positivity, we have $0 \le \frac{f(n)}{g(n)} \le C$ for all n, which yields 3. Conversely, 3 implies that there is a constant C such that $f(n) \le Cg(n)$ for all n.

The last statement follows because, if the limit exists, then 3 must hold. □

Definition 1.3.6 (Big Oh, Version 2)**.** *Let $f(n)$ and $g(n)$ be partially-defined real-valued functions on $\mathbb{N}$, but assume they are both well defined for n sufficiently large. Then we say $f(n)$ is* **(big) O** *of $g(n)$, and write $f(n) \in O(g(n))$ or $f(n) = O(g(n))$, if there exist constants C and N such that $|f(n)| \le C|g(n)|$ for all $n > N$.*

The point of this more general, though slightly more technical, definition is that $f(n) \in O(g(n))$ is an asymptotic condition, which means it should only be a statement about sufficiently large n, and for small values of n the condition $f(n) \le Cg(n)$ is not important, and we don't even care if the functions don't make sense for small n.

More precisely, the "partially-defined" condition means that we allow $f(n)$ and $g(n)$ to be undefined on some *finite* subset of $\mathbb{N}$. This is convenient because it allows us to handle functions like $\log(n-1)$ or $\log(\log(n))$, both of which are undefined when $n = 1$, but defined for all $n > 2$.

In addition, if $g(n)$ is not required to be positive, then $f(n) \le Cg(n)$ for all $n > N$ does not imply we can choose a possibly larger value for C' to get $f(n) \le C'g(n)$ for all n like we did in Proposition 1.3.5. The issue is if $g(n) = 0$ for some n. For example, if $f(n) = 3$ and $g(n) = \log(n)$, then we have $f(n) \le g(n)$ for any $n > 20$. In fact, we can get $f(n) \le 5g(n)$ for any $n > 1$, but we will never have $f(1) \le Cg(1)$ for any C since $g(1) = \log 1 = 0$.

The reason to add the absolute values in the definition was simply to give a more general statement of big O notation which is particularly useful in bounding errors in asymptotics, which might be positive or negative, as in the discussion about the Prime Number Theorem above. (Alternatively, one could just put the absolute values on f and require $g(n) \ge 0$ for n sufficiently large). However, for most of our purposes, we will just consider cases where both $f(n)$ and $g(n)$ are positive, at least for sufficiently large n and we can typically forget about these absolute values.

One final remark about this definition versus the previous version: even if your functions are positive everywhere, it is often a bit easier to check that an inequality holds for sufficiently large n than having to find an explicit C that works for all n. For example, suppose you want to check $f(n) = 4$ is $O(\log(n+1))$ by hand from the definition. It is (slightly) easier to use the second definition and simply observe that $\log 3 > 1$ so $f(n) \le 4\log(n+1)$ for $n > 1$, rather than trying to estimate $\log 2$ to find a C such that $4 \le C\log(2) \le C\log(n+1)$ for all n.

The following will be a convenient tool to show $f(n) \in O(g(n))$ in many cases.

Proposition 1.3.7. *Let $f(n)$ and $g(n)$ be partially-defined real-valued functions on $\mathbb{N}$. Assume $g(n) \neq 0$ for n sufficiently large. Then the following are equivalent*

1. *$f(n) \in O(g(n))$;*
2. *For some number N, the sequence of numbers $\left(\frac{f(n)}{g(n)}\right)_{n>N}$ is bounded.*

In particular, if $\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|}$ *exists and is finite, then* $f(n) \in O(g(n))$.

Note we need the condition that $g(n) \neq 0$ for sufficiently large n just to ensure the ratios $\frac{f(n)}{g(n)}$ are well defined for all n large enough. E.g., if we take something like $f(n) = n \sin \frac{\pi}{2} n$ and $g(n) = n^2 \sin \frac{\pi}{2} n$, then $f(n)$ and $g(n)$ are just 0 when n is even n and $\pm n$ and $\pm n^2$ when n is odd. It is true that $f(n) \in O(g(n))$ but we can't say that condition 2 holds because the ratios are never well-defined for n even.

The proof is essentially the same as the $1 \iff 3$ part of the proof for Proposition 1.3.5, except that one includes absolute values and restricts the inequalities to $n > N$ for some N. (See Exercise 1.3.4.)

Example 1.3.8. $O(1) \subsetneq O(\log\log n) \subsetneq O(\log n) \subsetneq O(\sqrt{n}) \subsetneq O(\sqrt{n}\log n) \subsetneq O(n)$.

For increasing functions f and g, the statement $O(f(n)) \subsetneq O(g(n))$ (i.e., every function $h(n)$ in $O(f(n))$ is in $O(g(n))$ but not conversely) means that, asymptotically, f grows strictly slower than g does.

Proof. The structure of the proofs for each part is the same. Namely, we can show $O(f(n)) \subsetneq O(g(n))$ as follows. By transitivity (Proposition 1.3.4), if we show $f(n) \in O(g(n))$ then we will have $O(f(n)) \subset O(g(n))$. Then we show $g(n) \notin O(f(n))$ to get $O(f(n)) \subsetneq O(g(n))$.

The first part, that $O(1) \subsetneq O(\log\log n)$ is obvious because $f(n) = 1$ is bounded, whereas $g(n) = \log\log n$ goes to infinity.

For the second part, that $O(\log\log n) \subsetneq O(\log n)$, we use Proposition 1.3.7. Namely, by l'Hospital's rule, we have

$$\lim_{n\to\infty} \frac{\log\log n}{\log n} = \lim_{x\to\infty} \frac{\frac{d}{dx}\log\log x}{\frac{d}{dx}\log x} = \lim_{x\to\infty} \frac{1/(x\log x)}{1/x} = \lim_{x\to\infty} \frac{1}{\log x} = 0,$$

i.e., $\log\log n \in O(\log n)$. Similarly, an application of l'Hospital's rule on the reciprocal shows

$$\lim_{n\to\infty} \frac{\log n}{\log\log n} = \lim_{x\to\infty} \frac{1/x}{1/(x\log x)} = \lim_{x\to\infty} \log x = \infty,$$

so $\log n \notin O(\log\log n)$. Hence $O(\log\log n) \subsetneq O(\log n)$, as claimed.

The remaining parts are similar to the second part, and left as Exercise 1.3.5. □

Note in the proof of second part, instead of applying l'Hospital's rule a second time, we could just observe that if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $\lim_{n\to\infty} \frac{|g(n)|}{|f(n)|} = \infty$. This observation gives the following corollary of Proposition 1.3.7.

Corollary 1.3.9. *Let* $f(n)$ *and* $g(n)$ *be partially-defined real-valued functions on* $\mathbb{N}$. *Assume* $g(n) \neq 0$ *for* n *sufficiently large. If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, *then* $O(f(n)) \subsetneq O(g(n))$, *i.e.,* $f(n) \in O(g(n))$ *but* $g(n) \notin O(f(n))$.

Example 1.3.10. *Let* $a > 1$ *and* $d > 0$. *Then* $O(n^d) \subsetneq O(a^n)$.

Again, the proof is an exercise. A function $f(n) \in O(a^n)$ for some $a > 1$ is said to have (at most) **exponential growth**. (I include the "at most" because we don't typically say polynomials have exponential growth—they have polynomial growth!) This example should just be a translation

of something you know from calculus—exponential functions grow faster than any polynomial. In algorithm analysis, typically exponential growth is very bad, polynomial growth is good, and logarithmic growth ($O(\log n)$) is outstanding.

For our algorithm analysis, there is one more elementary thing to be aware of—the "arithmetic" of Big Oh.

Proposition 1.3.11. *Suppose $c > 0$ is a constant, $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$. Assume $g_1(n)$ and $g_2(n)$ are positive for sufficiently large n. Then*
(i) $(f_1 + f_2)(n) \in O((g_1 + g_2)(n))$, and
(ii) $(f_1 f_2)(n) \in O((g_1 g_2)(n))$.

Proof. (i) There exist constants such that $|f_1(n)| \leq C_1|g_1(n)| = C_1 g_1(n)$ for $n > N_1$ and $|f_2(n)| \leq C_2|g_2(n)| = C_2 g_2(n)$ for $n > N_2$. Consequently

$$|f_1(n) + f_2(n)| \leq |f_1(n)| + |f_2(n)| \leq C_1 g_1(n) + C_2 g_2(n) \leq \max\{C_1, C_2\}(g_1(n) + g_2(n)),$$

for $n > \max\{N_1, N_2\}$, which is the assertion of (i).
(ii) is similar. □

The assumption about g_1 and g_2 being positive is just to rule out something like $f_1(n) = f_2(n) = n$, $g_1(n) = -g_2(n) = n^2$ where $g_1 + g_2$ cancels out the growth of g_1 and g_2. One could state (i) without the positivity assumption as $(f_1 + f_2)(n) \in O((|g_1| + |g_2|)(n))$. (Positivity is not needed for (ii).)

1.3.2 Algorithms

The point of the above diversion on big Oh asymptotic classes is that now we have some basic tools to explain some simple algorithm analysis, which is extremely important in practice when one wants to work with graphs of even moderate size.

With all this talk of analyzing algorithms, you might already be a little uneasy. Maybe you're thinking to yourself, I don't even know what an algorithm is. That's okay, because it's not entirely well-defined. Don't worry though, this won't cause any problems though—just because Plato wasn't sure what a table was, I'm sure he could build one or use one perfectly well.

For us, an **algorithm** is a (finite) sequence of instructions designed to accomplish a specific task. The instructions themselves might be a little vague, or even a lot vague. For example, consider the following two algorithms.

Algorithm 1.3.12. *Find the "most popular" member of a given social network $G = (V, E)$.*

1. *Go through each node $v \in V$, and count the number of neighbors* $\deg(v)$ *of v (called the* **degree** *of v).*
2. *Find the largest* $\deg(v)$*, and output the corresponding v.*

This algorithm is fairly specific, but there are still some things open to interpretation. First of all, there is the notion of "the most popular" member of a social network. How this should be interepreted might depend upon type the network and whether it is directed or undirected. However, let's assume it is undirected and that by most popular I really mean the node of highest

degree. One issue is that there might be a tie—e.g., for the graph in Figure 2, two nodes (Brady and Clay) are tied for the highest degree (5). In this case, should one output all nodes of highest degree, or just one? Probably it's reasonable to output all nodes of highest degree.

These clarifications make the algorithm rather well-defined, though it's still not as specific as it could be. For example: in what order do you go through the nodes in Step 1? how do you keep d_v and v associated, or don't you? (e.g., make a table?) what is the algorithm to find the largest d_v? (e.g., do you sort first or not?) However, it's good enough for anyone to be able to carry out by hand, or for someone with a moderate amount of programming experience to be able to code up easily.

Now here is a somewhat famous, but much worse, example of an algorithm.

Algorithm 1.3.13. *Find an optimal mate.*

1. *Estimate the number of people N you can date in your lifetime.*
2. *Date $N/e \approx 0.36N$ people, give them scores, break up with them (or get dumped—user choice), and let M be the maximum of these scores.*
3. *For each subsequent person you date, score them. If their score is below M, break up with them. If there score is above M, marry them.*

This comes from a probability exercise about figuring out how to maximize your chances of getting the best possible spouse (here you're not allowed to date multiple people at a time, or marry someone you've already broken up with). The reason I think this is a bad algorithm is perhaps different from the reason you might think this is a bad algorithm (or perhaps not). Sure, maybe you can't accurately estimate N or give your more-than-friends-but-less-than-spouses accurate scores. And maybe anyone who scores above your cutoff M won't want to marry you, i.e., you don't score above the cutoff value in their algorithm. But in some sense, these are problems of implementation of the algorithm, and we're not meant to worry about these issues in this theoretical exercise. (Or maybe you take ontological issue with the existence of such a thing as an "optimal mate." But we're working in the confines of an admittedly absurd exercise.)

The main problem with the algorithm is that often it doesn't give the correct solution to the problem (though sometimes it will). Already 36% of the time, you've broken up with your optimal mate in Step 2, which means in Step 3 you will break up with everyone until the end of (your) time, so the algorithm *doesn't terminate* (until you do). Even when you marry, it doesn't always give the optimal mate. However, within the confines of this theoretical exercise, there is no algorithm with will always produce an optimal mate, i.e., there is no good algorithm for this problem (which I'm sure you already knew). Really this algorithm is not a solution to the problem "find an optimal mate"—it is a solution to the problem "out of a specific class of bad algorithms to find an optimal mate, determine which bad algorithm is the least bad".

The point is that algorithms can be good or bad (they solve the problem always, sometimes, or never). They might terminate or not (e.g., they could get stuck in an infinite loop), or be very quick or very slow. The instructions might be clear or vague. At some point, if the instructions become too vague, we should probably not call it an algorithm anymore (e.g., "solve this problem" is not an algorithm for solving any problem), but there is no clear cut line as to what is "too vague." This is what I meant when I said I don't know exactly what an algorithm is, in the same way Plato wasn't sure exactly what a table was. The notion of an algorithm is like the notion of a

mathematical proof—it's essentially done by consensus. If people are convinced by an argument, it's considered a proof.* If people can figure out how to carry out the instructions, it's considered an algorithm.

Let's return to Algorithm 1.3.12, with programming in mind. If we're trying to write code for this algorithm, there are various ways it could be implemented. (Note: computer code is not the same as an algorithm—it's a specific implementation of an algorithm. For example, changing variable names in code changes the code, but not the algorithm. Or changing a for loop to a while loop changes the code, but not the algorithm. However, we won't try to be precise about when two blocks of code are consider as implementations of the same algorithm, or two different ones. Again this is done by common sense/consensus.)

An experienced programmer would have no trouble coding up this algorithm, however someone with little programming experience (which might be you) might vacillate a little with it. So where possible, particularly as we're getting started with programming, we'll try to make our algorithms a little more explicit. For example, we can write a more detailed algorithm as

Algorithm 1.3.14. *(Algorithm 1.3.12 refined.)*

1. *Set* `maxd` $= 0$.
2. *For each* $v \in V$*, calculate the degree* $\deg(v)$*. If* $\deg(v) >$ `maxd`*, set* `maxd` $= \deg(v)$.
3. *Make a new empty list* `mostpop`.
4. *For each* $v \in V$*, if* $\deg(v) =$ `maxd`*, append* v *to* `mostpop`.
5. *Output* `mostpop`.

This is a lot closer to computer code, and should be easy for you to code up once you're somewhat familiar with Python. (The one point we haven't explained here is how to calculate the degree—this is simple, but it depends upon the implementation of the graph. Let's take this for granted now and come back to it in a moment.) Hopefully, this is fairly straightforward to understand: the first two steps are the algorithm to find the maximum degree `maxd`, and the next two steps find all the vertices having this maximum degree.

Another way to express an algorithm in a ready-to-code way to do this is with *pseudocode*. This is something that looks a lot like computer code, but isn't quite. Typically one makes it a little easier to read than actual code, and sometimes avoids writing down all the details of actual code that will run. Since we're programming in Python, we'll use Pythonesque pseudocode. From the pseudocode, it should generally be a simple matter to write actual Python code (though you may need to look up some commands or syntax).

Here is sample pseudocode for Algorithm 1.3.12.

Pseudocode

```
set maxd = 0
for v in V:
    d = degree(G,v)
    if d > maxd then
```

*Even for mathematicians, who usually want everything to be precise, imprecisely-defined notions like what constitutes a proof are often more useful than exactly defined ones.

```
        set maxd = d
set mostpop = []
for v in V:
    if degree(G,v) == d then
        append v to mostpop
output mostpop
```

This pseudocode is pretty close to actual Python code, and how close you want to make your pseudocode to actual code is up to you. I wrote it essentially as I would Python code, but tried to make it flow more like English when you read it aloud.* Specifically, the differences are: I added the word `set` at the beginning of definitions, I used the word `then` instead of a colon in the `if` statements, the `append` line is different, and the word `output` instead of return. (I also didn't include a line to define the function.) Of course, I also used a function not yet written called `degree`, which computes the degree.

Note this pseudocode is more precise than Algorithm 1.3.14. For example, in our pseudocode, we are recomputing `degree(G,v)` in Step 4 of Algorithm 1.3.14. That is, we don't bother keeping track of the degree $\deg(v)$ for each v when we first compute it—we compute all the degrees once to determine the maximum degree, and then we compute them all again to see which vertices have maximum degree. Alternatively, we could have stored all the degrees in a list and just accessed the previously computed degrees in Step 4. Hence we have (at least) 2 different implementations of Algorithm 1.3.14. (Just to show you there are many ways to do things: a variant of Algorithm 1.3.14 would be to store all the degrees and vertices in a table (a 2-dimensional array) during Step 2, sort the table by degrees, and then output the vertices at the top of the table.)

Now let me tell you how to find the degree of a vertex. After this, you should be able to code up Algorithm 1.3.14 (see Exercises 1.3.7 and 1.3.8).

First, let's see how to do it if the graph is given as an adjacency matrix `A` (with respect to $V = \{0, 1, 2, \ldots, n-1\}$).

Python 2.7

```
>>> def deg(A,i):
...     d = 0                       # initialize the degree d to be 0
...     for j in range(len(A)):     # for j = 0, 1, 2, ..., n-1
...         d = d+A[i][j]           # add A[i,j] to d
...     return d
...
>>> A = [ [ 0, 1, 1, 1 ], [1, 0, 1, 0], [1, 1, 0, 0], [1, 0, 0, 0] ]
>>> deg(A,0)
3
>>> deg(A,1)
2
>>> sum(A[0])
3
>>> sum(A[1])
2
```

Here I define a function `deg(A,i)`, which takes in an adjacency matrix which and returns the degree of the i-th vertex, which is just the sum of the entries in row i. This is how my code computes the

*You can think of pseudocode as programming poetry. Bonus points for pseudocode in iambic pentameter, limerick, or haiku.

degree. (Recall `len(A)` returns the number of rows in `A`, i.e. the size of `A`.) However, Python already has a built-in function `sum`, which returns the sum of the entries in a list, so you can alternatively get the degree of vertex i just by calling `sum(A[i])`.

Now suppose the graph G is given as an adjacency list. Again we could write a function that gets the degree of vertex v, but it can be obtained simply by counting the length of the set of neighbors of v. (We could also use this algorithm in the adjacency matrix implementation, but counting the number of 1's in the i-th row is more straightforward.) If G is given as a dictionary, this can be done as follows.

Python 2.7
```
>>> G = { "purple" : { "purple", "monkey" }, \
... "monkey" : {"purple", "dishwasher"}, \
... "dishwasher" : { "monkey" } }
>>> len(G["monkey"])
2
```

Here `G[v]` returns the set of neighbors of `v`, and we pass this set of neighbors to the function `len`, which returns the length (size) of a set.

1.3.3 Algorithm Running Times

There are two basic constraints in computing: *data storage* and *computing time*. In the olden days, when games came on multiple diskettes and computer screens had 1 color—green—data storage was a serious concern, and programmers had to work hard so as not using any more memory/disk space than necessary. Now, memory/storage capacity is relatively cheap, and data storage is not a serious issue for most computing tasks. It is mostly only a concern for very specialized problems—e.g., keeping tabs on everything on the internet—though I think most program developers don't take data storage issues seriously enough. (Many programs are bloated, and unnecessarily slow down your computer—on the other hand, it's easier to write programs that aren't efficient.)

Nowadays, the main concern about efficiency is typically is the amount of time a program takes to run. This will be our main focus in algorithm analysis as well, though occasionally if the amount of space used becomes egregious we'll discuss it.

How should we gauge the efficiency of a program or algorithm? One way is simply to physically time how long it takes to run. There are a couple of issues with this. One, the amount of time depends on the implementation of the algorithm (both how you write your code, and how your programming language translates your code into machine operations), the task it is performing (what the input is) and the computer it is running on. Since, in the heyday of Moore's law, computer speeds were doubling every 18 months, just measuring physical running times is of limited use (though still useful). Further, trying all possible inputs is typically impractical.

Instead, we'd like a simple theoretical way to analyze algorithms that will allow us to estimate how fast or slow a program will be in practice. This approach will also have the considerable benefit that we don't actually have to write a working program to analyze the algorithm. The procedure is very simple: we just count the number of steps require to complete the algorithm, i.e., the number of lines of code that the program will run.

Let's start off with a simple, straightforward example: the program to find the degree of a vertex using adjacency matrices from Section 1.3.2. Let's recall the code.

Python 2.7
```
>>> def deg(A,i):
...     d = 0                       # initialize the degree d to be 0
...     for j in range(len(A)):     # for j = 0, 1, 2, ..., n-1
...         d = d+A[i][j]           # add A[i,j] to d
...     return d
```

Here there are two inputs, A and i, so we will let $f(A,i)$ be the number of steps required by this code given the input (A,i). Let n be the size of A. The first line (after verb+def+), $d = 0$, is run one time. The second line, you can take as also being run once. The third line, however, is run n times. Finally the last line is run once. Hence $f(A,i) = n+3$. In fact, since f depends only on the order n of the graph, we can think of this as a function of n, i.e., $f(n) = n+3$.

This is not exactly the number of steps the computer will do—there's a lot of stuff going on behind the scenes at the processor level for each line of code. However, it's a reasonable estimate thinking that each line of code takes 1 unit of time to run, and it would be a real headache to analyze what the processor is actually doing. There is one point to be careful about however—in the second line we call the functions `range` and `len`. The function call to `len` takes one step (regardless of how big A is, Python stores the length of A for easy access and you just need to retrieve this value from memory*). However the function `range` returns the list $[0, 1, 2, ..., n-1]$, which takes $n+2$ steps to create (n steps to put all the items in the list, 1 to make an empty list, and 1 to return the list). So perhaps it is better to say $f(n) = (n+3)+1+(n+2) = 2n+6$. (In fact, we could get pickier, but I'm sure none of you want that.)

We can also see here how the implementation makes a difference about the number of steps the algorithm will take—if one uses Python's `xrange` instead of `range`, or a `while` loop instead of the `for` loop, Python doesn't actually need to create a list of size n to do the loop, and we would have something like $f(n) = n+4$ or $f(n) = n+5$.

The point is that, however we do this analysis, and even if we get very picky, the number of steps the computer is doing behind the scenes, this $f(n)$ is a *linear function* in n, i.e., $f(n) \in O(n)$, i.e., $f(n)$ has **linear growth**. In other words, as the order n of the graph grows, the amount of time this function will take to run grows linearly in n. Thus we say the **running time** of this algorithm is $O(n)$. (While technically, we also have $f(n) \in O(n^2), O(n^3), O(2^n)$, etc., we don't say that this algorithm has running time $O(n^2)$ or $O(n^{n^n})$ because that would be morally reprehensible, even if legally permissible.)

Let me quickly give one more example before discussing algorithm running times in more generality. Recall the following algorithm for finding neighbors of a given vertex from an adjacency matrix.

Python 2.7
```
>>> def neighbors(A, i):
...     n = len(A)                  # let n be the size (number of rows) of A
...     neigh = []                  # start with an empty set neigh
...     for j in range(n):
...         if A[i][j] == 1:        # for each index 0 <= j < n
...             neigh.append(j)     # append j to the list neigh if the i-th
...     return neigh                #     row has a 1 in the j-th position
```

*I didn't check the actual implementation of the length function, but Python surely must do this.

To determine the running time of this algorithm, again let $f(A,i)$ denote the number of steps the algorithm takes to run with given input (A,i). The first three lines (after `def`) and the final line contribute 1 step each (not being picky with the `range` function in the `for` loop). The next line `if A[i][j] == 1:` runs n times. The next-to-last line runs somewhere between 0 and n times, depending on how many neighbors vertex i has. Hence $n+4 \leq f(A,i) \leq 2n+4$. Since both our upper and lower bounds for $f(A,i)$ are $O(n)$, we say this algorithm has running time $O(n)$.

In general, an algorithm is a sequence of instructions that takes in some input data, such as an integer, a list, a matrix, a graph, or possibly multiple inputs (17 lists, 3 matrices and a graph). Suppose we have an algorithm, Algorithm **A**, that takes in input $\mathcal{I}$. Let $f(\mathcal{I})$ denote the number of steps Algorithm **A** takes to run given input $\mathcal{I}$. Let $\alpha(\mathcal{I})$ denote the "size" of $\mathcal{I}$. (How we measure the size of the input depends upon the problem and our point of view, but for us it will typically be the order n of some graph.) As we saw in the last example, the number of steps $f(\mathcal{I})$ required may depend upon more than just the size $\alpha(\mathcal{I})$ of $\mathcal{I}$. Consequently, we define three notions of running times.

Definition 1.3.15. *Let $(\mathcal{I}_n)$ denote a sequence of inputs $\mathcal{I}_n$ such that $\alpha(\mathcal{I}_n) = n$.*

- *If $f(\mathcal{I}_n) \in O(g(n))$ for some sequence $(\mathcal{I}_n)$, we say Algorithm* **A** *has* **best case running time** $O(g(n))$.
- *If, on average, $f(\mathcal{I}_n) \in O(g(n))$ for sequences $(\mathcal{I}_n)$, we say Algorithm* **A** *has* **average case running time** $O(g(n))$.
- *If $f(\mathcal{I}_n) \in O(g(n))$ for all sequences $(\mathcal{I}_n)$, we say Algorithm* **A** *has* **worst case running time** $O(g(n))$.

The best case running time tells you what is the fastest your algorithm can run. The average case tells you how long it usually takes, and the worst case gives you an upper bound for all possible inputs.

Here is an alternative, slightly more formal, description. Let $\mathcal{S}$ be the space of all possible inputs $\mathcal{I}$. Define a function $\alpha : \mathcal{S} \to \mathbb{N}$ and assume that for each $n \in \mathbb{N}$, the preimage $\mathcal{S}_n := \alpha^{-1}(n) \in \mathcal{S}$ is finite. Here $\alpha(\mathcal{I})$ is what we called the size $n(\mathcal{I})$ of $\mathcal{I}$ above. Let

$$\begin{aligned} f_{\min}(n) &:= \min_{\mathcal{I} \in \mathcal{S}_n} f(\mathcal{I}) \\ f_{\text{avg}}(n) &:= \frac{1}{|\mathcal{S}_n|} \sum_{\mathcal{I} \in \mathcal{S}_n} f(\mathcal{I}) \\ f_{\max}(n) &:= \max_{\mathcal{I} \in \mathcal{S}_n} f(\mathcal{I}). \end{aligned}$$

Then we say, with respect to our choice of size function α, Algorithm **A** has best case running time $O(f_{\min}(n))$, average case running time $O(f_{\text{avg}}(n))$ and worst case running time $O(f_{\max}(n))$.)

Example 1.3.16. *Consider a graph (possibly directed, non-simple) with adjacency matrix A. Fix a vertex i, and say we want to find the vertices which are either neighbors of i, or neighbors of neighbors of i. (We will define the notion of distance on graphs later, and this is essentially the set of vertices of distance ≤ 2 from i.) Here is our algorithm.*

Algorithm 2-neighbors:

1. *Make an empty set* `2-neigh`.
2. *Find the neighbors of i and add them to* `2-neigh`.
3. *For each neighbor j of i, find the neighbors of j and add them to* `2-neigh`.
4. *Output* `2-neigh`.

Instruction 1 and Instruction 4 both take 1 step each. As we saw above, finding the neighbors of i takes $O(n)$ time (best, average or worst case), and adding these elements to `2-neigh` *should take at most n steps.* Hence, the second instruction always takes $O(n)$ steps. In the third step, we need to run the neighbors algorithm again for each neighbor we had. Let's say there were d neighbors (i.e., d is the degree of i), then this is $O(n) + O(n) + \cdots O(n)$ (d times), or $O(dn)$ steps. Adding the neighbors of neighbors in Instruction 3, takes no more than dn steps, so Instruction 3 runs in $O(dn)$ steps.*

Putting everything together, we see our algorithm runs in $2 + O(n) + O(dn) = O((d+1)n)$ steps (which is the same as $O(dn)$ if $d \neq 0$). Now $0 \leq d \leq n$. If $d = 0$ (so Instruction 3 never runs at all), we are led to the minimum number of steps possible, i.e., a best case running time of $O(n)$. Similarly, the maximum number of steps is clearly when $d = n$, i.e., the worst case running time is $O(n^2 + n) = O(n^2)$. One needs to do a bit more work to rigorously check what is average number of steps. I won't go through this, but it is what you might guess—on average d will be $n/2$, so the average case running time is also $O(n^2/2) = O(n^2)$.

In some sense, knowing the average case running time (how long does the algorithm normally take?) is what you most want to know, but can be more difficult to compute than best case or worst case. Knowing the best case running time is rarely of practical use. Therefore, we will usually just concern ourselves with the worst case running time, which provides an upper bound for the question how long does the algorithm normally take, and in many instances turns out to be the same as the average case running time. Consequently, when we say *the running time* of an algorithm, without further qualification, we mean the worst case running time.

Alternatively, rather than trying to break things up into best case/average case/worse case, we could've just left things at: the running time `2-neighbor` is $O((d+1)n)$. We will sometimes do this.

If an algorithm runs in $O(1)$ time, we say it has **constant running time** (it does not seriously depend upon the size of the input.) If it runs in $O(\log n)$ time, we say it has **logarithmic running time**. If it runs in $O(n^d)$ time for some $d \in \mathbb{N}$, we say it has **polynomial running time** (the special cases $d = 1$ and $d = 2$ are called **linear** and **quadratic** running times). If it runs in $O(a^n)$ time for some $a > 1$, we say it has **exponential running time**. What we can hope for in an algorithm depends upon what the problem is, and how often we plan to call this algorithm. Generally speaking, exponential running time is very bad, arbitrary polynomial running time is okay, linear or maybe quadratic running time is good, and logarithm running time is great. Constant running time is typically impossible.

One other remark about terminology: based on what we've said in this section, you might think of the number of vertices n as the "size" of the graph—this is the default parameter. However,

*Now there is a technicality about how long it takes to add an element to a set—it depends upon the implementation (the issue is that sets should have no repeated elements, so first you have to see if your element is already in the set S or not, which naively takes $O(|S|)$). However, it can be (and I believe is in Python) implemented so that adding an a new element essentially only takes $O(1)$ time, so for simplicity let's assume this is the case.

don't call it that—call it the order of the graph. For a graph $G = (V, E)$, the **size** of G is defined by many authors to be $|E|$, the number of edges, which could be anywhere between 0 and n^2.

Let me close with a brief remark on data storage, which does become important when you've got ridiculously huge graphs like the internet.

If you want to use an adjacency matrix, you need to store an $n \times n$ matrix, which means you'll require $O(n^2)$ space—you need to store each coordinate of the matrix. The exact amount of space needed depends on the actual implementation, and how much space is needed to store the names of the vertices. However, suppose you want a graph with 1 million nodes. Each matrix entry is 0 or 1, so the most efficiently we can store the matrix in a usable form is store each matrix entry as a single bit (in the usual implementation, each matrix entry will be 64 bits, but let's say we do things more efficiently). Then storing this matrix will take 10^{12} bits $\approx$ 100 Gigabytes (GB). If you wanted 10 million nodes, this would require 100 times more space, or about 10 Terabytes (TB). And while we're talking about really large graphs here, this isn't even close to size of a web graph—remember there are an estimated 1 *trillion* webpages out there (Google seems to index tens of billions), and many social network websites have over 100 million users.

Now suppose you want to use an adjacency list. Then you need a dictionary with n entries, and each entry requires a certain space depending on the number of neighbors. The total number of neighbors list in the adjacency list is the same as the number of edges of the graph (for directed graphs, or twice that for undirected graphs). Hence the storage space required is $O(n + |E|)$. For most kinds of graphs, $|E| > n$, so this can be though of as $O(|E|)$. Hence the size $|E|$ of the graph, as defined above, really measures how much space is required to store the graph. How much space would we need to stored a graph with 1 million nodes using adjacency lists? Let's suppose that, on average, each vertex is connected to 200 other nodes (this is quite reasonable in practice—this number is very close to the *average degree* for both Twitter and Facebook graphs). Then the size, $|E|$, is 200 million. If we identify each vertex by a 64-bit number (32-bits is still more than enough), then this would require about 1.6 Gigabytes (GB). This is still quite sizable, but only 1.6% of the space required for the adjacency list representation. (And we've been fairly conservative in our estimates.) If we have 10 million nodes, where the average vertex degree is still 100, then the size only multiplies by 10 to require about 16 GB, about 0.16% of the space require for the adjacency matrix.

Exercises

Exercise 1.3.1. *Let $0 < r < s$ be real numbers. Prove that $O(n^r) \subsetneq O(n^s)$, i.e., that $f(n) \in O(n^r)$ implies $f(n) \in O(n^s)$, but there exist $f(n) \in O(n^s)$ which do are not $O(n^r)$.*

Exercise 1.3.2. *Give an example of positive functions $f(n)$ and $g(n)$ on $\mathbb{N}$ such that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist, but $f(n) \in O(g(n))$.*

Exercise 1.3.3. *Let $f(n)$ and $g(n)$ be positive functions on $\mathbb{N}$. Is it true that either $f(n) \in O(g(n))$ or $g(n) \in O(f(n))$?*

Exercise 1.3.4. *Prove Proposition 1.3.7.*

Exercise 1.3.5. *Complete Example 1.3.8 by showing $\subsetneq O(\log n) \subsetneq O(\sqrt{n}) \subsetneq O(\sqrt{n} \log n) \subsetneq O(n)$.*

Exercise 1.3.6. *Prove the assertion in Example 1.3.10.*

Exercise 1.3.7. *Write Python code for a function* `maxdegvert(A)` *which, given an adjacency matrix A, returns (as a Python set) the set of vertices of maximum degree.*

Exercise 1.3.8. *Write Python code for a function* `AL_maxdegvert(G)` *which, given a graph G as an adjacency list, returns (as a Python set) the set of vertices of maximum degree.*

Exercise 1.3.9. *Determine the (worst case) running times for your functions* `maxdegvert(A)` *and* `AL_maxdegvert(G)` *from the previous 2 exercises.*

Exercise 1.3.10. *Consider the following simple algorithm to find the position of a number i in an ordered list of size n.*

1. *Initialize a position counter variable* `pos = 0`
2. *For each object x in the list:*
3. *if $x = i$, return* `pos`.
4. *otherwise, increase* `pos` *by 1 and continue.*

Assume that the space $\mathcal{S}_n$ of allowable inputs of size n is the set of pairs (π, i) where π is a permutation (an ordering, represented as an ordered list) of $\{0, 1, 2, \ldots, n-1\}$ and $0 \leq i \leq n-1$. Determine the best case, average case, and worst case running times for this algorithm.

1.4 Graph Isomorphisms

▶ In this section, graphs may be simple or not, undirected or directed.

If we are just interested in understanding the structure of a graph, the names of the vertices are unimportant. In other words, we may often want to just consider *unlabelled graphs*, i.e., graphs where the vertices are not labelled. We can do this formally with the notion of an *isomorphism*.

For instance, the two graphs

are technically distinct graphs, because the vertices have different names, but we want to regard them as essentially the same. We will say they are *isomorphic.* Here is the formal definition.

Definition 1.4.1. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be graphs. If there is a bijection $\phi : V_1 \to V_2$ such that $(u, v) \in E_1$ if and only if $\phi((u, v)) := (\phi(u), \phi(v)) \in E_2$, then we say G_1 and G_2 are* **isomorphic***, and we write $G_1 \simeq G_2$. In this case, we say the map ϕ is an* **isomorphism** *of G_1 with G_2 (if $G_1 = G_2$, we say ϕ is an* **automorphism** *of G_1).*

Recall a bijection ϕ is a map which is one-to-one and onto, i.e., ϕ maps distinct elements of V_1 to distinct elements of V_2, and each element of V_2 is in the image of ϕ. There exist bijections from V_1 to V_2 if and only if V_1 and V_2 have the same cardinality.

This definition, in less formal terms, says the following: an isomorphic ϕ is a bijection between the vertex sets V_1 and V_2, such that, regarded as a map of pairs of vertices, it maps edges of G_1 to edges of G_2, and non-edges of G_1 to non-edges of G_2. (I.e., ϕ induces a bijection of the edge sets E_1 and E_2.) Even more colloquiually: two graphs will be isomorphic, if you can turn one graph into the other merely by relabelling the vertices.

Example 1.4.2. *The two graphs pictured above are isomorphic. Let G_1 be the graph on the left, and G_2 the graph on the right. Then we can take for our bijection $\phi : V_1 \to V_2$ the function $\phi(1) = A$, $\phi(2) = B$ and $\phi(2) = C$. Viewed as a map of pairs of vertices, we see $\phi((1,2)) = (A,B) \in E_2$, $\phi((2,3)) = (B,C) \in E_2$ and $\phi((1,3)) = (A,C) \notin E_2$. Hence ϕ is indeed an isomorphism—ϕ takes edges $e \in E_1$ to edges of E_2, and non-edges to non-edges.*

Note, there is another isomorphism we could have taken (in general, there may be many). We could take $\phi'(1) = C$, $\phi'(2) = B$ and $\phi'(3) = A$. One sees again that this is an isomorphism. The fact that there are two distinct isomorphisms is due to the fact that the map of G_1 given by interchanging 1 and 3, but fixing 2, is an automorphism of G_1, i.e., if we switch the labels 1 and 3 on G_1, the graph does not change.

Here are some basic properties of the notion of isomorphic.

Proposition 1.4.3. *Let G_1, G_2 and G_3 be graphs. Then*
(i) $G_1 \simeq G_1$
(ii) $G_1 \simeq G_2 \iff G_2 \simeq G_1$
(iii) If $G_1 \simeq G_2$ and $G_2 \simeq G_3$, then $G_1 \simeq G_3$.

Proof. The proofs are simple—it just involves checking certain maps are isomorphisms, which we leave as an exercise. For (i), check the identity map is an isomorphism. For (ii), if ϕ is an isomorphism from G_1 to G_2, check ϕ^{-1} is an isomorphism from G_2 to G_1. For (iii), if ϕ_1 is an isomorphism from G_1 to G_2 and ϕ_2 is an isomorphism from G_2 to G_3, check $\phi_2 \circ \phi_1$ is an isomorphism from G_1 to G_3. □

This mean being isomorphic defines an equivalence relation among graphs.

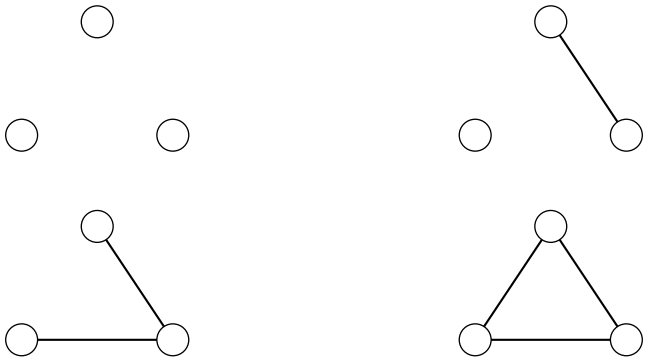
Definition 1.4.4. *An* **unlabelled graph** *is an equivalence (isomorphism) class of graphs.*

This may seem a bit strange definition if you're not familiar with this sort of idea, but the idea is quite simple. Graphs have this extra structure—the names of the vertices—that we often don't care about. So when we don't care about this, we can think of identifying all graphs isomorphic to a given graph G_0 (i.e., the same as G_0 except for this extra structure) as being the same "unlabelled" graph G. The technical way to do this is let G be the set of all graphs which are isomorphic to G_0. Then we think of any specific graph $G_i \in G$ as being a specific manifestation of the idea of G. Because being isomorphic is an equivalence relation, no two isomorphism classes intersect, and each graph corresponds to a unique unlabelled graph.

Occasionally, when we want to emphasize that we are working with honest graphs, not isomorphism classes, we may use the term **labelled graph.**

Since the notions of directed/undirected and simple/non-simple are preserved by equivalence classes (verify this!), it makes sense to say unlabelled graphs are directed/undirected or simple/non-simple if the underlying labelled graphs are.

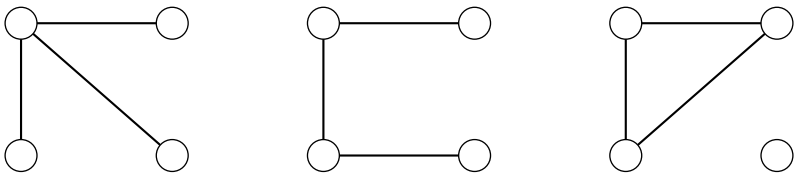
Example 1.4.5. *There are 4 simple, undirected graphs up to isomorphism (i.e., 4 unlabelled simple undirected graphs) on 3 vertices.*



In the above example, the unlabelled graphs are determined simply by the number of edges. This is no longer true when we move to 4 vertices.

Example 1.4.6. *There are two unlabelled undirected graphs on 4 vertices with two edges—either the both edges have a vertex in common or not.*

There are 3 with 3 edges (you can generate them by adding each possible edge to the previous graphs and throw out duplicates):



A basic question in graph theory is, given two graphs G_1 and G_2, determine if they are isomorphic. This is called the **graph isomorphism problem**. This is not easy in general (it might be NP-complete, if you know what that means), however in some cases it is easy to to check that two graphs are not isomorphic. For instance, if two graphs have different number of vertices, or different numbers of edges, it is easy to see there can be no isomorphism between them.

In general, data that can be associated to a graph which does not depend on its isomorphism class will be called an **invariant** of the graph. Then if two graphs have different invariants, we know they are not isomorphic. Some examples of invariants are: the order, the size, the maximum degree of a vertex, the minimum degree of a vertex, the number of isolated (degree 0) nodes, or more generally the number of vertices of degree d. We'll see many more examples of invariants later. An example of something that isn't an invariant could be something like: "the degree of vertex 1"—this evidently depends upon the labeling of the vertices.

Supposing we have two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with the same number of vertices n, then the number of bijections from $V_1 \to V_2$ is $n!$ (there are n choices for where to map the first element of V_1, $(n-1)$ for the second, $(n-2)$ for the third, and so on.) Consequently, using

the naive algorithm to see if $G_1 \simeq G_2$ (check all possible bijections to see if they are isomorphisms) takes $O(n^2 n!)$ time (the extra n^2 is to check if each bijection is an isomorphism or not). By Stirling's approximation, $n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$, so this algorithm has *worse than* exponential growth.

The best known algorithm has worst case running time $O(2^{\sqrt{n}\log n})$, which is *subexponential*—slower than exponential growth but faster than any polynomial growth, i.e., still quite bad. It is not known if there is a polynomial time algorithm which will determine if any two graphs are isomorphic or not (however there are algorithms that work quickly for most pairs of graphs, but have exponential worst case running time). This is a major unsolved problem in computational complexity theory, however we will not focus on this in our class.

Exercise 1.4.1. *Prove Proposition 1.4.3.*

Exercise 1.4.2. *Draw all unlabelled simple undirected graphs on 4 vertices. How many are there?*

Exercise 1.4.3. *Draw all unlabelled simple directed graphs on 3 vertices. How many are there?*

Exercise 1.4.4. *Let $n > 4$. How many unlabelled simple undirected graphs are there with n vertices and 1 edge? What about 2 edges? (You don't need to give formal proofs for your answers, but briefly explain your reasons.)*

Exercise 1.4.5. *How many unlabelled simple undirected graphs are there with 5 vertices and 3 edges? Draw them.*

1.5 Paths, Connectedness and Distance

▶ In this section, graphs may be directed and/or non-simple.

Now that we have various preliminaries out of the way, we can get to discussing some basic issues in networks. We'll start with communication and transportation networks in mind. For such networks, the fundamental issue is how things flow on the network—how do information or passengers or cargo flow? Can they can from point A to point B? If so, how long does it take? In networks, we allow things to travel from one vertex to another vertex along edges. The routes that things can travel along are called *paths* or *walks*.

Definition 1.5.1. *Let $G = (V, E)$ be a graph. We say a (non-empty) sequence of vertices $\gamma = (v_1, v_2, \dots, v_r, v_{r+1})$ in V is a* **path** *or* **walk** *if $(v_i, v_{i+1}) \in E$ for $1 \le i \le r$. The* **length** *of the path is $\mathrm{len}(\gamma) := r$. We say v_1 is the* **start vertex** *and v_{r+1} is the* **end vertex** *of γ. If $v_i \neq v_j$ for $1 \le i \ne j \le r+1$, we say the path is* **simple**.

If $v_{r+1} = v_1$, we say γ is **closed**. *If $\gamma = (v_1, \dots, v_r, v_1)$ is a closed path with $v_i \neq v_j$ for $1 \le i \ne j \le r$, we say γ is a* **(simple) cycle** *or* **circuit**.

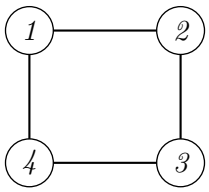
Alternatively, we can specify a path by a sequence of edges, rather than a sequence of vertices. Namely, a sequence of *adjacent* edges $(e_1, e_2, \dots, e_r)$ defines a path of length r. Here adjacent means that e_2 starts where e_1 ends, e_3 starts where e_2 ends, and so on. Thus the length of a path is the number of edges in the path, not the number of vertices. Just as we will allow vertices to repeat in our paths, edges may also repeat. On the other hand, since vertices may not repeat in simple paths or cycles (except for the first and last vertex of a cycle), edges cannot repeat in simple paths or cycles.

We will allow for paths of length 0, i.e., of the form (v) for any vertex $v \in V$. This does not require G having a loop at v, i.e., $(v, v) \in E$. If G does have a loop at v, this means there is a closed path (or cycle) of length 1, denoted (v, v)—which in this case coincides with our edge notation, which starts and ends at v.

Note: this terminology is not entirely standard. Many authors assume all paths are simple. We will not. On the other hand, we will assume all cycles are simple (not all authors do this, or some may only admit cycles of length ≥ 3), and use the term closed path when we want to discuss non-simple cycles.

The terms walk and circuit, however, are fairly standard.

Example 1.5.2. *Let $n > 2$. A* **cycle graph** *of order n is a graph of the form $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$. (Recall $\{v_i, v_j\}$ means an undirected edge, as opposed to (v_i, v_j).) Here is a cycle graph of order 4.*



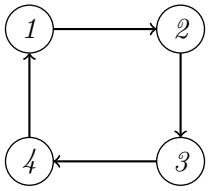
There are infinitely many paths from 1 to 4: (1,4), (1,2,1,4), (1,2,3,4), (1,2,3,4,1,2,3,4), ... However, there are only two simple paths from 1 to 4: (1,4) and (1,2,3,4).

For arbitrary order n, there are $2n$ cycles on G, all of length n—for each vertex v, there are 2 that start and end at v—e.g., (1,2,3,4,1), (1,4,3,2,1). However, there are infinitely many closed paths—you can keep going around the cycle as many times as you want.

All cycle graphs of order n are isomorphic, so we sometimes say the *cycle graph of order n, and denote it C_n.*

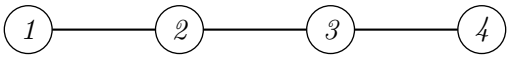
If G is any graph and γ is a cycle of length n, then the vertices and edges of γ define a cycle graph of order n. Hence cycles in any graph may be regarded as cycle graphs.

One can also consider directed cycle graphs, e.g.,



In this case there are exactly n cycles (all of length n) since one can only travel in one direction.

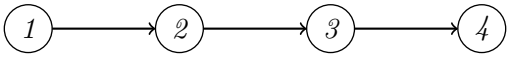
Example 1.5.3. *A* **linear graph** *(or* **path graph***) of order n, is a graph of the form $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$. Here is a line graph on 4 vertices.*



Again, all linear graphs on n vertices are isomorphic, and a simple path of length n yields a linear graph of order n.

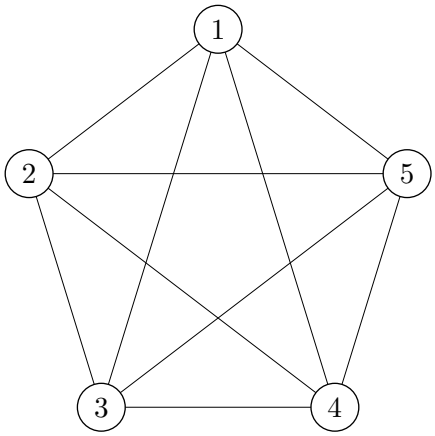
On a linear graph, there is a unique simple path between any given pair of vertices. Of course, if we do not require simple, an infinite number of paths are possible. This has no cycles of length ≥ 3*. (Note: any undirected edge defines a cycle of length 2—e.g., we have the cycle* $(1, 2, 1)$*.)*

We can also consider directed linear graphs, e.g.,



Here, there is a path from 1 to 4, but not from 4 to 1. This has no cycles.

Example 1.5.4. *A* **complete graph** *of order* n *is a simple undirected graph on* n *vertices that has all possible* $n(n-1)/2$ *edges. I've shown you one on 5 vertices before:*

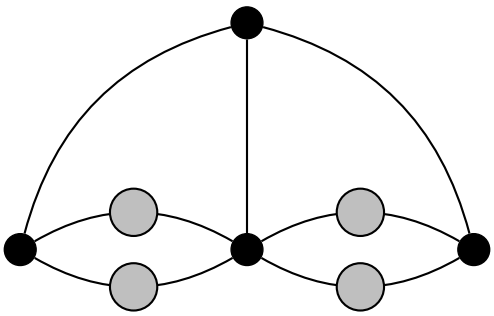


Again, all complete graphs on n *vertices are isomorphic, and we usually speak of* the *complete graph on* n *vertices, and denote it by* K_n *(you know, for komplete).*

In this case, there are loads of paths and cycles. For example, here are the some simple paths from 1 to 5: (1,5), (1,2,5), (1,3,5), (1,4,5), (1,2,3,5), (1,3,2,5), ... (If we want to enumerate them all, it's easiest to be systematic—I started counting by length.) For any two vertices, there are simple paths between them of lengths 1, 2, 3 and 4. There are cycles starting at any vertex of lengths 2, 3, 4 and 5.

The directed complete graph is the same as the undirected complete graph, by our convention of regarding directed graphs with symmetric edge sets as undirected graphs.

Example 1.5.5 (Königsberg bridge problem)**.** *Recall the graph from the Königsberg bridge problem.*



Here each of the black vertices represent landmasses, the edges represent bridges, and the grey vertices are just auxillary vertices used to turn the hypergraph (i.e., the multiedges) into a graph (i.e., ordinary edges). The problem was to find a path that traverses each edge exactly once (note the problem has not changed by our addition of auxillary vertices).

Euler's solution was the following. If there is such a path, then for each vertex in the path, except possible the start and end vertices, one needs to arrive at this vertex the same number of times one leaves this vertex. Hence, the degree of such vertices must be even. However, all black vertices on this graph have odd degree. So such a path is impossible. (Nowadays such paths are called Eulerian paths*, and one can show they exist if and only if the number of vertices of odd degree is either 0 or 2.)*

Connectedness

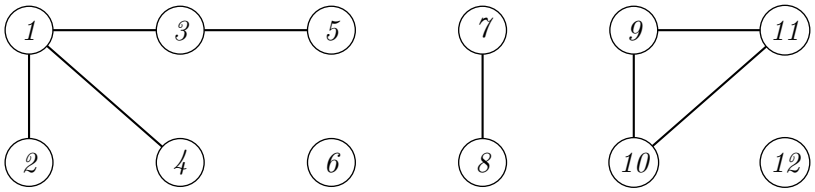
Now we can introduce the notion of *connectedness* which, at least for undirected graphs, will (essentially tautologically) tell us if things can get from point A to point B on a graph.

Definition 1.5.6. *Let $G = (V, E)$ be an undirected graph, and $v_0 \in V$. The* **connected component** *of v_0 is the set of all $v \in V$ such that there exists a path from v_0 to v. The* **connected components** *of G are the subsets of V which arise as connected components of some $v_0 \in V$.*

Proposition 1.5.7. *The connected components of an undirected graph $G = (V, E)$ partition V into disjoint subsets.*

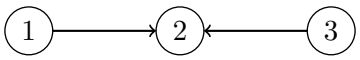
Proof. This follows because being in the same connected component is an equivalence relation; see Exercise 1.5.3. □

Example 1.5.8. *Consider the graph*



The connected component of 1 is the same as the connected component of 2, or 3, or 4, or 5. Similarly for 7 and 8, or 9, 10, and 11. Then the connected components of G are $\{1, 2, 3, 4, 5\}$, $\{6\}$, $\{7, 8\}$, $\{9, 10, 11\}$, and $\{12\}$. Hence the connected componets of G partition the vertices into 5 disjoint sets.

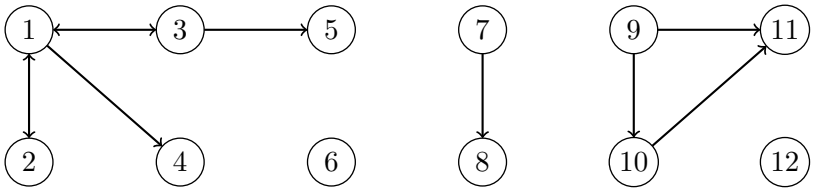
For directed graphs G, we obviously cannot do the same thing. Consider for example



Then, if we were to use the above definition, the connected component of 1 would be $\{1, 2\}$, the connected component of 2 would be $\{2\}$ and the connected component of 3 would be $\{2, 3\}$. So this doesn't give a partition of our digraph. There are a couple of possible ways to try to define connected components for digraphs. Here is perhaps the most naive way.

Definition 1.5.9. *Let $G = (V, E)$ be a directed graph, and let $G' = (V, E')$ be the associated undirected graph, i.e., let $E' = \{(u, v), (v, u) : (u, v) \in E\}$. The* **connected components** *of G are the connected components of G'.*

By definition, the connected components again partition the vertices of a digraph into disjoint subsets. For example, the connected components of the digraph

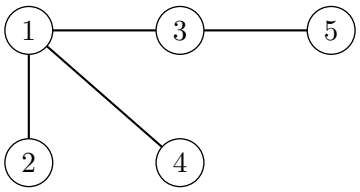


are the same as those for the graph in Example 1.5.8, as the associated undirected graph is the same.

One virtue of this definition of connected components is that it allows us to break up arbitrary graphs into smaller, and hopefuly bite-size, pieces.

Definition 1.5.10. *Let $G = (V, E)$ and $G' = (V', E')$ be graphs. If G is undirected, we assume G' is also undirected. We say $G' = (V', E')$ is a* **subgraph** *of G if $V \subset V'$ and $E \subset E'$.*

Often we will consider connected components as subgraphs of $G = (V, E)$. Note that a subgraph is not determined by just selecting the vertices—you also need to decide which edges to include. However, by convention, if we specify a subgraph only by a subset V_0 of vertices, we mean the graph $G_0 = (V_0, E_0)$ where $E_0 = \{(u, v) \in E : u, v, \in V_0\}$, i.e., we include all possible edges using only the vertices in V_0. For example, the subgraph associated to the connected component of 1 in Example 1.5.8 is



Definition 1.5.11. *Let G be a graph. We say G is* **connected** *if G has exactly one connected component.*

Then any connected component of any graph defines a connected subgraph. The number of connected components as well as their orders/sizes (number of vertices or number of edges), and the property of being connected, are all invariants of graphs. Furthermore, if we know all the connected components of G, we "union" them back together to get the original graph G.

For many problems, one reduces to the study of connected graphs. For undirected graphs G, we can get from vertex u to vertex v if and only if they are in the same connected component. In particular, we can get from any vertex u to any other vertex v if and only if G is connected. Consequently, being connected is one basic property we typically want in things like communication and transportation networks. From a practical point of view—this means we want algorithms to determine if a graph is connected, or to determine the connected components. We will briefly discuss algorithms later.

For directed graphs G (which most communication and transportation networks are not), the notion of connected components is not sufficient—if u and v are not in the same connected component, then v is not reachable from u, but if they are in the same connected component, v may or may not be reachable from u.

Now let's take a look at an alternative notion of connectedness for digraphs.

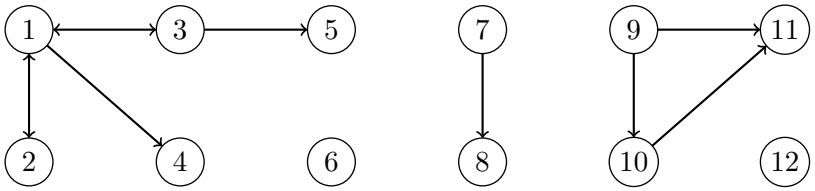
Definition 1.5.12. *Let $G = (V, E)$ be a directed or undirected graph, and $v_0 \in V$. The* **strongly connected component** *of v_0 is the set of all $v \in V$ such that there exists both a path from v_0 to v and a path from v to v_0. The* **strongly connected components** *of G are the subsets of V which arise as strongly connected components of some $v_0 \in V$.*

We say G is **strongly connected** *if it has exactly one strongly connected component.*

Note that if G is undirected, strongly connected components are the same as connected components since having a path from v_0 to v is equivalent to having a path from v to v_0.

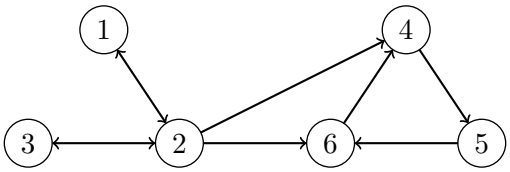
As with connected components, the strongly connected componentes partition the vertices into disjoint subsets (Exercise 1.5.5), and these components are maximal such that one can get from any vertex to any other vertex in same strongly connected component. In particular, G is strongly connected if and only if one can get from vertex u to vertex v for any two vertices u, v in G.

However, knowing the strongly connected components (even together with the connected components) is not enough to completely answer the question can one get from u to v. Namely, tt still may be possible to get from u to v though u and v are in different strongly connected components. For instance, in the digraph



the strongly connected components are $\{1, 2, 3\}$, $\{9, 10, 11\}$ and then the singleton sets $\{4\}$, $\{5\}$, $\{7\}$, $\{8\}$ and $\{12\}$. Just looking at the strongly connected components does not tell us if there is a path from 4 to 5 or a path from 7 to 8, though we can rule out the possibility of a path from 4 to 8 by looking at connected components. In general, there is no way to partition the vertices of a directed graph G in such a way that one can definitively and easily say if there is a path from one given vertex u to another given vertex v. Rather, one can compute the set of all vertices reachable from u (cf. our original definition for connected component for undirected graphs) and check if v is in this set or not.

We remark many authors don't consider our notion of connected components for digraphs—they only consider strongly connected components, and may occasionally just refer to them as the connected components or components of the digraph. (We may sometimes say components of G for connected components of G.) However, I defined the above notion of connected components because is useful for problems where we may want to break up digraphs into smaller digraphs. Note that one typically cannot do this with strongly connected components because one cannot piece together a digraph G from just its strongly connected components (viewed as subgraphs). For instance, the strongly connected component graphs of



are

Just knowing the two strongly connected component graphs does not tell us how to paste them together to get our original graph, since there are many ways these two strongly connected components could be "weakly connected."

Distance

Now, assuming that we can get from point A to point B in the graph, our next question is how do we determine how long it takes? We use the model that it takes 1 time unit to traverse each edge. Later we will account for different time (or money) costs per edge by using weighted graphs.

Definition 1.5.13. *Let $G = (V, E)$ be a graph. For $u, v \in V$, let $\Gamma(u, v)$ denote the set of paths from u to v. We define the* **distance** *$d(u, v)$ between u and v to be*

$$d(u,v) := \begin{cases} \infty & \text{there is no path from } u \text{ to } v; \\ \min\{\text{len}(\gamma) : \gamma \in \Gamma(u,v)\} & \text{else.} \end{cases}$$

In other words, the distance between two vertices is the least number of steps (edges) it takes to get from one to the other (if we are working with directed graphs, which vertex is first is important here). In particular, the vertices which are distance 1 from u are the **neighbors** of u. For any vertex u, $d(u, u) = 0$ since we have allowed paths from u to u of length 0 in our definition of path.

Example 1.5.14. *Let's consider $d(1, 4)$ from our above (undirected) examples. In the cycle graph C_4, 1 and 4 are adjacent, so $d(1, 4) = 1$. In the line graph, $d(1, 4) = 3$. In the complete graph K_5, all vertices are adjacent, so $d(1, 4) = 1$.*

Proposition 1.5.15. *Let $G = (V, E)$ and $u, v \in V$. Suppose $0 \neq d(u, v) < \infty$. Then there is exists a path γ from u to v such that $\text{len}(\gamma) = d(u, v)$. Furthermore, any such γ must be a simple path.*

Proof. The assumptions mean $\Gamma(u, v)$ is non-empty. Since the set $\{\text{len}(\gamma) : \gamma \in \Gamma(u, v)\} \subset \{0, 1, 2, \ldots\}$, it has a least element, i.e., the minimum is well-defined, and so there exists some γ such that $\text{len}(\gamma) = d(u, v)$. Consider any such $\gamma = (u = v_1, v_2, \ldots, v_r = v)$. If γ is not simple, then some $v_i = v_j$ for $i \neq j$. (By assumption $u \neq v$, so $(i, j) \neq (1, r)$.) Say $i < j$. Then we can consider the strictly shorter sequence $\gamma' = (u = v_1, v_2, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_r = v)$. Since v_i, v_j, we must have $(v_i, v_{j+1}) \in E$, whence γ' is also a path from u to v. However, it is shorter than γ, contradicting the minimal length of γ. □

The following proposition shows that graph distances behave at least somewhat like Euclidean ones.

Proposition 1.5.16 (Triangle Inequality)**.** *Let $G = (V, E)$. If $u, v, w \in V$, then*

$$d(u, v) + d(v, w) \geq d(u, w).$$

Proof. Suppose this is not true for some u, v, w. Then $d(u, v) + d(v, w) < d(u, w)$. It suffices to assume all these distances are finite (why?). Hence there is a path from u to v of length $d(u, v)$, and a path from v to w of length $d(v, w)$. "Adding" these paths together (following one, then the other) gives us a path from u to w of length $d(u, v) + d(v, w) < d(u, w)$, contradicting that $d(u, w)$ is the minimum length of paths from u to w. □

If you've studied topology, this makes any undirected graph G into a metric space—i.e., the distance function satisfies all the usual properties ($d(u, v) \geq 0$ with equality if and only if $u = v$, $d(u, v) = d(v, u)$ and the triangle inequality). This is not true for directed graphs, since $d(u, v) \neq d(v, u)$ in general (e.g., consider the directed linear graph on 4 vertices above—$d(1, 4) = 3$ but $d(4, 1) = \infty$).

If we have some sort of communication or transportation network, we want some measure (or measures) of efficiency (i.e., how fast things can travel between two nodes). Here is the most basic one.

Definition 1.5.17. *Let $G = (V, E)$ be a graph (directed or not). The* **diameter** *of G, denoted* $\operatorname{diam}(G)$*, is the maximum distance $d(u, v)$ for $u, v \in V$.*

The diameter is finite if and only if G is connected and undirected, or G is strongly connected and directed.

Example 1.5.18. *For the cyclic graph C_n of order n, we have* $\operatorname{diam}(C_n) = \lfloor \frac{n}{2} \rfloor$.* *(Draw a few examples.) The diameter of the directed cyclic graph of order n is $n - 1$.*

The diameter of a linear graph of order n is $n - 1$. The diameter of a directed linear graph is ∞.

The diameter of the complete graph K_n is $\operatorname{diam}(K_n) = 1$.

The diameter provides an upper bound on the time it takes to get between two points in the graph. Thus smaller diameters indicate higher efficiencies for graphs. You can also think of diameter of being a measure of "how connected" a graph is—the smaller the diameter, the closer together the nodes are, so things are better connected in some sense. For (strongly) connected (di)graphs, the diameter of $n - 1$ for directed cyclic or undirected linear graphs is the worst case possible, as this proposition shows.

Proposition 1.5.19. *Let $G = (V, E)$ be a connected undirected or strongly connected directed graph of order n. Then* $\operatorname{diam}(G) \leq n - 1$.

Proof. Let $u, v \in V$. Then $d(u, v) < \infty$ and there is a path γ of length $d(u, v)$ from u to v. By Proposition 1.5.15, γ must be simple. This means γ has no repeated vertices, i.e., it has at most n vertices, i.e., it has at most $n - 1$ edges. □

*$\lfloor x \rfloor$ denotes the floor, or greatest integer, function—round x down to the nearest integer, while $\lceil x \rceil$ denotes the ceiling function—round x up to the nearest integer.

Another measure of how well connected a graph is to look at the *average distance* between vertices. This will give us a (often times better) estimate on how long it will take to get from a random vertex to another random vertex. This is like looking at the average case running time of an algorithm instead of the worst case running time. Which measure is more appropriate depends upon the particular problem, but as the diameter is an easier quantity to get a handle on, we will focus primarily on that.

However, let us at least give a precise definition. For a directed or undirected graph $G = (V, E)$, define the **average distance** on G to be

$$d_{avg}(G) := \frac{1}{n(n-1)} \sum_{u \in V} \sum_{v \in V, v \neq u} d(u, v).$$

Note $n(n-1)$ is the total number of ordered pairs (u, v) of distinct vertices, and we average the distance over those. This is not much harder to compute than the diameter when working with specific graphs on the computer, but is considerably harder to analyze theoretically. (For instance, try calculating the average distance for a cyclic graph C_n or linear graph of order n in terms of n. It is not horrible, but not nearly as easy calculating the diameter.)

Algorithms

Let's start off with the question of designing an algorithm to find the connected component of a given v_0 of an undirected graph G of order n. The idea is straightforward, though I'll write a reasonable amount of detail which will make it easier to code.

Algorithm 1.5.20. *Find the connected component of v_0.*

1. *Add v_0 to a new set* `visited` *(this keeps track of which vertices we've already visited, and will be the connected component of v_0 when we're done).*
2. *Add each neighbor of v_0 to* `visited`*. Let* `newverts` *be this set of neighbors just added.*
3. *For each vertex in* `newverts`*, find their neighbors. For each neighbor not in* `visited`*, add this vertex to* `visited`*. Then let* `newverts` *be the set of these vertices just added.*
4. *Repeat last step until* `newverts` *is empty (i.e., until you're no longer adding more vertices).*
5. *Ouput* `visited`*.*

In other words, we start at v_0, find its neighbors, find its neighbors' neighbors, find the neighbors' neighbors' neighbors, and so on. This process is known as a breath-first search—we search in layers for all the vertices in the component of v_0 (as opposed to a depth-first search, where one searches in succesive lines out from v_0). At each step in this process, we only travel out from vertices we haven't previously visited. This avoids an infinite loop, and makes our algorithm fairly efficient.

Let's think about how this search is expanding out in a little more detail. (This is what the set `newverts` is at each stage.) From v_0, we go to it's neighbors, i.e., vertices distance 1 from v_0. Then we find the neighbors of the distance 1 vertices that we haven't already seen. These will be of distance ≤ 2 from v_0. Well, the only things we've seen are the things of distance 0 and 1 from v_0. Hence our new set of vertices is precisely the vertices distance 2 from v_0. Continuing in this process, after d iterations, the set `newverts` is precisely the set of vertices of distance d from v_0.

Note: I could've absorbed Step 2 into Step 3 of this algorithm by just letting `newverts = {v_0}` in Step 1. I would do this when coding—I just separated out Step 2 for the purposes of exposition.

Now, let's analyze this algorithm. At some point in the algorithm, for each vertex in the connected component of v_0, I need to find the neighbors of v_0 and go through each neighbor, check if it was already visited and either add it to the connected component or not. Let's say there are m vertices in the connected component of v_0. Each such vertex has at most $m-1$ neighbors (we can ignore loops), hence this algorithm has a running time of $O(m^2)$. Since $m \leq n$, we can also say this algorithm runs in $O(n^2)$ time. In fact, if one uses adjacency lists, this algorithm can be implemented in $O(n + |E|)$ time.

With this algorithm in hand, it is easy to find all connected components of $G = (V, E)$. Pick a random $v_0 \in V$. Find its connected component V_0. Now take a random $v_1 \in V - V_0$, and find its connected component V_1. Continue this process until all vertices have been exhausted.

Similarly, one can determine if G is connected as follows. Pick a random $v_0 \in V$. Find its connected component V_0. Then G is connected if and only if $|V_0| = n$.

Algorithms to find strongly connected components of a digraph G are a bit more involved, and we will not get into them, but just mention this can also be done in $O(n^2)$ time.

Now that we've addressed algorithms pertaining to connectedness, let's move on to distance. Fix two vertices u, v of a graph G (directed or undirected). How can we compute the distance $d(u, v)$? What have we been doing by hand? We've (at least I have, and I assume this is what you've been doing too) essentially been finding all simple paths from u to v, of which there are finitely many and see what path or paths are shortest possible. This is easy to by hand for small graphs, but to do for large graphs, or to automate on the computer, it requires some work to generate all simple paths from u to v.

However, if we remember our algorithm for finding the connected component of u, we organized all vertices in the connected component of u by their distance from u. If we just kept track of that information in our algorithm, we'll have the distance not just from u to v, but from u to any other vertex in the graph (if the other vertex is not in the connected component of u, we know the distance is infinite).

Here is Python code to do just that, using adjacency matrices and our previous function `neighbors`. The function is called `spheres` for the following reason. Given a graph $G = (V, E)$ and a vertex $u \in V$, the **sphere of radius r centered at u** is the set

$$S_u(r) = \{v \in V : d(u, v) = r\}.$$

It is called a sphere because this is the same definition as for spheres in the Euclidean space familiar to you.

Python 2.7

```
def spheres(A, i):
    sph = [ { i } ]
    visited = { i }
    newvert = { i }
    while len(newvert) > 0:
        new = set()
        for j in newvert:
            neigh = neighbors(A,j)
            for k in neigh:
```

```
                if k not in visited:
                    new.add(k)
        newvert = new
        if len(newvert) > 0:
            sph.append(newvert)
            visited = visited.union(new)
    return sph
```

This function returns the list $[S_u(0), S_u(1), S_u(2), \ldots, S_u(m)]$ where m is the maximum distance from u of any vertex in the component of u. Here each $S_u(r)$ is returned as a Python set. Consequently, if you enter `sph = spheres(A, i)`, then you can access $S_u(r)$ simply by `sph[r]`. This function works for directed and undirected graphs. It really is essentially an implementation of Algorithm 1.5.20 where we simply keep track of which vertices are distance r from u, so the same analysis applies and it runs in $O(n^2)$ time.

With this function, we can compute $d(u, v)$ as follows.

Algorithm 1.5.21. *Compute $d(u, v)$.*

1. *Compute the spheres $S_u(r)$ centered at u.*
2. *For each possible value of r, check to see if $v \in S_u(r)$. If so, output r.*
3. *Otherwise, output ∞ (which in the computer we often code as -1).*

Here the first step take $O(n^2)$ times, the second can be done in $O(n)$ time (the number of vertices in the union of the spheres is at most n), and the last step takes $O(1)$ time. Hence this algorithm for computing the distance takes $O(n^2) + O(n) + O(1) = O(n^2)$ time.

We remark one could make this more efficient by not computing all spheres $S_u(r)$ first, but compute them inductively and check at each step if $v \in S_u(r)$.

Lastly, we present an algorithm for computing the diameter. One could simply try to use the definition and compute $d(u, v)$ for all u, v, and take the maximum distance. However, we can do it more efficiently than that.

Algorithm 1.5.22. *Compute* $\operatorname{diam}(G)$*, where $G = (V, E)$.*

1. *For each $u \in V$, do the following:*
2. *Compute the spheres $S_u(r)$ centered at u.*
3. *Let $B(u) = \bigcup_r S_u(r)$. If $|B(u)| < n$, some vertex is not reachable from u, so return ∞.*
4. *Otherwise, let d_u be the maximum r for which $S_u(r)$ is nonempty. (We can get this by `len(spheres(A,i))`.) This is the maximum distance any vertex can be from u.*
5. *Output* $\max\{d_u : u \in V\}$*, which must be the diameter.*

See Exercise 1.5.13 for the analysis. Note that if we were just working with undirected graphs, one could avoid doing Step 3 for each u, and just do it for one u at the beginning to ensure G is connected.

Exercises

Exercise 1.5.1. *Consider the komplete graph K_4 on $\{1,2,3,4\}$.*
(i) Enumerate all simple paths from 1 to 4. How many are there?
(ii) How many cycles of lengths 2, 3 and 4 are there on K_4?

Exercise 1.5.2. *(i) Consider a cycle graph C_5 on $\{1,2,3,4,5\}$. For each vertex j, compute $d(1,j)$.*
(ii) Do the same for the directed cycle graph on $\{1,2,3,4,5\}$.

Exercise 1.5.3. *Let $G=(V,E)$ be an undirected graph. Show that being in the same connected component is an equivalence relation, i.e., show:*
(i) for any $v_0 \in V$, v_0 is in the connected component of v_0;
(ii) if v_1 is in the connected component of v_0, then v_0 is in the connected component of v_1; and
(iii) if v_2 is in the connected component of v_1 and v_1 is in the connected component of v_0, then v_2 is in the connected component of v_0.

Exercise 1.5.4. *Let $G=(V,E)$ be a graph. Show that $v_0 \in V$ is an isolated node (i.e., degree 0) if and only if its connected component has size 1.*

Exercise 1.5.5. *Let $G=(V,E)$ be a digraph. Show the strongly connected components partition V into disjoint subsets by showing that being in the same strongly connected component is an equivalence relation.*

Exercise 1.5.6. *Let G be a connected undirected graph of order n. Show G has at least $n-1$ edges.*

Exercise 1.5.7. *Let $n \geq 2$. Consider the cycle graph $C_{2n}=(V,E_0)$, and form the graph $G=(V,E)$ on the same vertex set $V=\{1,2,\dots,2n\}$ (with the usual choice of cycle, i.e., the edges are $\{1,2\}$, $\{2,3\}$, ..., $\{2n-1,2n\}$ and $\{2n,1\}$) with $E=E_0 \cup \{n,2n\}$. In other words, we add the "diagonal" edge to C_n from n to $2n$. In G, what is $d(1,n+1)$? Determine* $\mathrm{diam}(G)$.

Exercise 1.5.8. *Let $C_m=(V_1,E_1)$ and $C_n=(V_2,E_2)$ be cycle graphs. Consider the graph $G=(V,E)$ obtained by taking the union (or "direct sum") of C_m and C_n and connecting them with a single edge. Precisely, fix $v_1 \in V_1$ and $v_2 \in V_2$. Then $V=V_1 \cup V_2$ and $E=E_1 \cup E_2 \cup \{\{v_1,v_2\}\}$. Determine* $\mathrm{diam}(G)$.

Exercise 1.5.9. *Determine, in terms of n, the running time of the algorithm described in the text (after Algorithm 1.5.20) to find all connected components of G.*

Exercise 1.5.10. *Using the* `spheres` *function, write functions* `component(A,i)`, `components(A)` *and* `is_connected(A)` *to find the connected component of vertex i, all components of G, and determine if G is connected, where A is the adjacency matrix for a directed or undirected graph G. (Caution: remember to convert A to the adjacency matrix for the associated undirected graph.)*

Exercise 1.5.11. *Using the* `spheres` *function, write a function* `distance(A,i,j)` *that computes the distance from vertex i to vertex j given a directed or undirected graph adjacency matrix A.*

Exercise 1.5.12. *Write a function* `diameter(A)` *to compute the diameter of a graph given its adjacency matrix A using the above algorithm.*

Exercise 1.5.13. *Analyze the running times for the following two algorithms to compute the diameter: (i) the naive algorithm of computing all possible distances and taking the maximum, and (ii) Algorithm 1.5.22.*

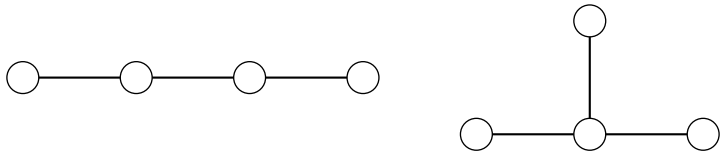
1.6 Network design, trees, k-connectedness and regularity

▶ Here graphs are undirected unless otherwise stated.

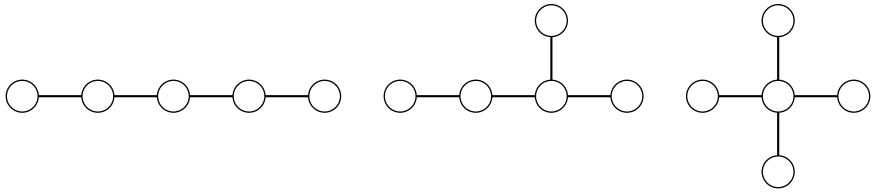
Now let's consider the problem of designing a network. When designing something in real life, there are pros and cons, costs and benefits, that we need to balance out. In network design, there is typically a cost associated to each edge of the nework (for construction, maintenance, or both—e.g., think of a highway network). Hence one wants to minimize the number of edges in the network while maintaining certain standards of performance.

Let's first consider just the following simple constraint: the network should be connected. This is of course a minimum necessity for communication and transportation networks. How few edges do we need to make a connected network on n vertices? Recall Exercise 1.5.6 says we need at least $n-1$ edges. Furthermore we can always make a network connected with $n-1$ edges by using a linear or path graph. What else can we do?

Well, for $n = 2$, there is only 1 graph with 1 edge, and it is connected. For $n = 3$, again there are only 2 possibilities with 2 edges. For $n = 4$, we have 2 possibilities (up to isomorphism):



For $n = 5$, we have 3 possibilities:



The above graphs are all examples of an important family of graphs, namely trees.

Definition 1.6.1. *Let G be a (simple undirected) graph. We say G is a* **tree** *if it is connected and has no cycles of length > 2.*

Proposition 1.6.2. *Let $G = (V, E)$ a connected graph of order n. Then G is a tree if and only if $|E| = n - 1$.*

Proof. Both directions will be proved by proving the contrapositive.

($\Leftarrow$) First we claim that if G has a cycle of length $r > 2$, it must have more than n edges. By relabelling vertices, we may assume $V = \{v_1, \dots, v_n\}$, where there is a cycle of length r on $\{v_1, \dots, v_r\}$. The existence of the cycle means there are at least r edges involving only $v_1, \dots, v_r$. Since G is connected, one of the remaining vertices, say v_{r+1} must have an edge to one of $v_1, \dots, v_r$. Thus there are at least $r + 1$ edges involving only $v_1, \dots, v_{r+1}$. Continuing this argument shows there are at least n edges involving $v_1, \dots, v_n$, $|E| \geq n$ as claimed.

Hence if G is connected with $n - 1$ edges, it has no cycles of length > 2, i.e., is a tree.

($\Rightarrow$) Now suppose $|E| \geq n \geq 3$. We want to show G has a cycle of length > 2.

A **leaf** is a vertex with degree 1. It is clear no cycle of length > 2 will involve a leaf. Thus we may prune all the leaves, i.e., delete the leaves and the corresponding edges from the graph G to

get a subgraph G'. If there were l leaves, now G' is a graph on $n-l$ vertices with at least $n-l$ edges. It is impossible for all vertices of a connected graph on ≥ 3 vertices to be leaves (if this is not clear, think about the argument in ($\Leftarrow$)), so G' indeed has some vertices. Furthermore, since there are no graphs on $m = 1$ or 2 vertices with m edges, G' must have at least 3 vertices.

Now prune G'. Continue this process of pruning leaves until there are no more. (This process must terminate as the number of vertices becomes strictly smaller at each step, with a lower bound of 3.) This leaves (no pun intended) us with a graph G_0 with $m \geq 3$ vertices and at least m edges. Since G_0 has no leaves, each vertex of G_0 has degree ≥ 2. This means G_0 has a cycle of length > 2 by Exercise 1.6.3, as desired.

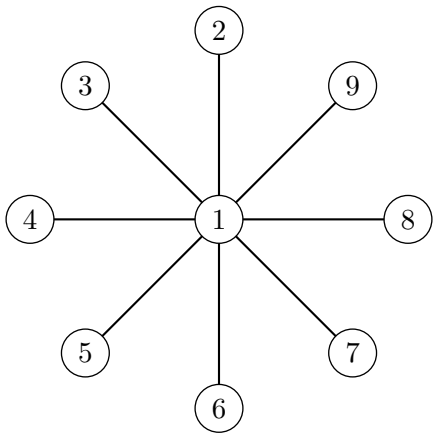
This means that if G is a tree, it has at most $n-1$ edges (we just showed the contrapositive for $n \geq 3$, but this statement is trivial when $n = 1$ or $n = 2$), but it has to have at least $n-1$ since it is connected, i.e., $|E| = n - 1$. □

Remark: the above proof is typical of classical graph theory, and we'd be doing a lot more arguments like this in a standard graph theory course than we will in this one.

In other words, the trees are precisely connected graphs with the minimum possible number of vertices, i.e., the best candidates for our overly-simplified network design problem. Now we can ask, is there any way in which some of the trees might form a better network than others?

Well, another nice property we would like our network to have, besides being connected, is **efficiency**, i.e., one should be able to get between two points in the network relatively quickly, so we want small diameter (or average distance). If we look back at our trees for $n = 4$ and $n = 5$, it is clear the ones on the right are more efficient, and the ones on the left (i.e., the linear graphs) are least efficient. We can generalize the trees on the right to n vertices as follows.

Example 1.6.3. *Let $n \geq 3$. The* **star graph** *of order n is the undirected graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$ and $E = \{\{1, 2\}, \{1, 3\}, \ldots \{1, n\}\}$. That is, vertex 1 is connected to all other vertices, and there are no other edges. We can picture 1 as being the hub at the center of the network. Here is the star graph on 9 vertices is*



This graph is connected with $n-1$ edges and has diameter 2. In particular, it is a tree.

Since the only graph with diameter 1 is the complete graph (why?), the star graph minimizes diameter among all undirected graphs with less than $n(n-1)$ edges, and this uses the fewest number of edges possible for any connected graph on n vertices.

This leads us to the following, perhaps surprising, observation. Naively, you might expect the more edges there are in your graph, the smaller your diameter should be—perhaps if you gradually allow more edges, you can get smaller and smaller diameters. Instead, there's a very sharp trichotomy here. If you have $< n-1$ edges, you must be disconnected so the diameter is infinite. If you have between $n-1$ edges and n^2-n-1 edges (anything short of K_n), you can achieve diameter 2. If you have the maximum possible n^2-n edges, then you must be K_n and have diameter 1. (However, you can get smaller and smaller average distances by adding more edges—see Exercise 1.6.4.)

Rather, what seems to be more important for getting a small diameter is that the edges are well chosen, e.g., making a star graph as opposed to a linear graph. This should indicate that some care should be take in the design to make a good network. On the other hand, we'll encounter a differing philosophy later which says that "random graphs" tend to make good networks. Roughly the idea is that, sure if you pick a tree at random you're unlikely to end up with the star graph, but you're equally unlikely to end up with the linear graph, and chances are that your random tree will have diameter closer to 2 than to $n-1$. This is explored in Exercises 1.6.6 and Exercises 1.6.7.

Okay, so we seem to have given a reasonable answer to the problem of designing an efficient network on n nodes. Is there anything we've overlooked? Well, in a perfect world, not really. There's the aspect that for a physical network, different links will need to be different lengths, and may have different costs associated with them (both for building/maintaining and for travelling along), but we'll revisit this issue later, albeit fairly briefly.

There are two main issues with using a star graph for a network. First, in the real world, things fail all the time. A cable (edge) could get severed, a server (node) might be down for maintenance or have hardware issues, roads (edges) or airports (nodes) might be due to the weather. If the hub of a star graph fails, then the whole network goes down. Or if a single edge goes down, the corresponding outer vertex becomes stranded. A network that can reasonably handle such failures is said to be **robust**.

The second main issue has to do with traffic, or network flow. If we use a star graph as a network, all traffic must pass through the central hub. Then during busy times, it may be that traffic gridlocks at the hub rendering the network essentially non-functional for a period of time. If we want to study traffic issues precisely, then one can define formal notions of the capacity of a network (how much/how fast information/traffic can pass through) and the network flow. However, if our network is robust, this will mean that there are several different ways to get from one point to another, and therefore traffic can be rerouted when necessary to cut down on gridlock. Hence we will focus on robustness now.

Here are a couple of basic measures of robustness.

Definition 1.6.4. *Let $G = (V, E)$ be a graph (possibly directed and non-simple) of order $n > k$ with $n > 1$ We say G is k-**connected**, or k-**vertex-connected**, if the removal of any subset of $< k$ vertices (and involved edges) yields a connected subgraph. The* **vertex connectivity** *$\kappa(G)$ of G is the maximal non-negative integer such that G is $\kappa(G)$-connected. Alternatively, $\kappa(G)$ is the minimal number of vertices one needs to remove to make G disconnected or have order 1.*

A **vertex cut** *is a set of vertices V_0 of V such that the subgraph $V - V_0$ is disconnected. Hence the minimal size of a vertex cut (when one exists) is $\kappa(G)$.*

Note $\kappa(G)$ tells us that if $< \kappa(G)$ nodes of our network fail, the remainder of our network will still be functional (connected). Note that G is 1-connected if and only if G is connected (and

$n > 1$), and $\kappa(G) = 0$ means G is disconnected. However, G being 0-connected does not mean G is disconnected—any k-connected graph is automatically $(k-1)$-connected from the definition (for $k > 0$).

For a directed graph G, being k-connected means the same as the associated undirected graph being k-connected.

Definition 1.6.5. *Let $G = (V, E)$ be a graph (possibly directed and non-simple) of order $n \geq 2$. We say G is k-***edge-connected** *if the removal of any subset of $< k$ edges (but no vertices) yields a connected subgraph. The* **edge connectivity** *$\lambda(G)$ is the maximal non-negative integer such that removing any subset of $< \lambda(G)$ edges (but no vertices) leaves G connected. Alternatively, $\lambda(G)$ is the minimal number of edges one needs to remove to make G disconnected.*

A **cut***, or an* **edge cut** *is a subset of edges E_0 such that the graph $(V, E - E_0)$ is disconnected. The minimal size of a cut equals $\lambda(G)$.*

Note $\lambda(G)$ means that if $< \lambda(G)$ edges of our network fail, our network will still be functional (connected). Again G is 1-edge-connected if and only if G is connected (and $n > 1$), and $\lambda(G) = 0$ means G is disconnected. Also, k-edge-connected implies $(k-1)$-edge-connected (assuming $k > 0$).

For a directed graph G, being k-edge-connected is not the same as the associated undirected graph G' being k-edge-connected, as 1 edge in G' might correspond to 1 or 2 edges in G. However, an edge cut of size k in G corresponds to an edge cut in G' of size $\leq k$, so we can say $\lambda(G') \leq \lambda(G)$.

We avoided defining vertex and edge connectivity for a graph of order 1 (which is connected) to avoid putting more technicalities in the definitions.

We remark that (if $n \geq 2$) edge cuts always exist (you can cut off all edges from a given vertex to isolate it), but vertex cuts do not. For instance, take the linear graph of order 2—we can only remove 1 vertex and still be left with a subgraph (I'm not allowing "empty graphs" on 0 vertices), and either vertex you remove leaves you with a (connected) graph on 1 vertex. This is why the alternative definition of $\kappa(G)$ includes the condition that removing $\kappa(G)$ vertices leaves you with a single vertex.

Example 1.6.6. *The linear graph L_n of order $n \geq 2$ has $\kappa(L_n) = \lambda(L_n) = 1$. To see this, note L_n is connected so $\kappa(L_n), \lambda(L_n) \geq 1$. If $n = 2$, we can only remove 1 vertex as discussed above, so $\kappa(L_2) = 1$. If $n > 2$, we can remove any vertex "in the middle" and this will disconnect the graph, so $\kappa(L_n) = 1$. Similarly, for any $n \geq 2$, if we remove any edge, we disconnect the graph, so $\lambda(L_n) = 2$.*

Example 1.6.7. *More generally, let T be any tree of order $n \geq 2$. Then again $\kappa(T) = \lambda(T) = 1$. To see this, note T must have at least one leaf (otherwise, it has minimum degree 2 and therefore a cycle of length > 2 by Exercise 1.6.3). We can cut the edge from the leaf to disconnect T, so $\lambda(T) = 1$. Again, for $n = 2$ it is trivial to see $\kappa(T) = 1$, so assume $n \geq 3$ now. Then removing a neighbor of a leaf cuts of the leaf from the rest of the tree, so we have a vertex cut of size 1, i.e., $\kappa(T) = 1$.*

Example 1.6.8. *Consider the cycle graph C_n, $n \geq 3$. We can isolate any vertex by removing the two adjacent vertices or edges, however removing any single vertex or edge leaves us with a connected subgraph (a linear graph of order $n - 1$). Hence $\kappa(C_n) = \lambda(C_n) = 2$.*

Example 1.6.9. *The complete graph K_n, $n \geq 2$, has $\kappa(K_n) = \lambda(K_n) = n - 1$. To see this, observe removing any set of $k < n$ vertices leaves us with a complete subgraphs K_{n-k}, i.e., $\kappa(K_n) =$*

$n-1$. On the other hand, suppose there is an edge cut that leave two vertices u and v in different components. How many ways can we get from u to v in K_n? There are many, but here are $n-1$ possibilities: we can go straight from u to v, or for any of the other $n-2$ vertices w, we can go from u to w, then w to v. Note these paths are independent *in the sense that they have no edges in common. Hence to disconnect u from v, we need to get rid of at least 1 edge from each of these paths, i.e., the minimum size of a cut is at least $n-1$. In fact it is exacly $n-1$ since we can just remove all $n-1$ edges from u. Thus $\lambda(K_n) = n-1$.*

Remark: A fundamental theorem about connectivity is **Menger's Theorem**. It states that the number of edges needed to disconnect u and v is the maximum number of independent paths from u to v. There is also a vertex connectivity version. This theorem is a special case of the famous *Max Flow-Min Cut Theorem*, which is a generalization to the setting where one considers each edge having a certain capacity for traffic.

Up till now, we haven't seen any examples with $\kappa(G) \neq \lambda(G)$, so you may be wondering if they exist. Well, they do. How might we construct one? First observe the following.

Proposition 1.6.10. *Let $G = (V, E)$ be a connected graph (possibly directed non-simple) of order $n \geq 2$. Suppose C is an edge cut of minimal size. Then the cut graph $G' = (V, E - C)$ has two connected components, i.e., C partitions V into two disjoint subsets.*

Proof. Exercise. □

Proposition 1.6.11. *Let $G = (V, E)$ be a graph (possibly directed non-simple) of order $n \geq 2$. Then $\kappa(G) \leq \lambda(G) \leq n-1$ if G is undirected and $\kappa(G) \leq \lambda(G) \leq 2n-2$ if G is directed.*

Proof. Consider a minimum edge cut $C = \{e_1, e_2, \dots e_k\}$. This cannot contain any loops, otherwise we would remove the loops and get a smaller edge cut. Thus $k \leq n-1$ if G is undirected and $k \leq 2n-1$ if G is directed.

Now let's show $\kappa(G) \leq \lambda(G) = k$. We may assume now G is undirected, otherwise we can replace it with the associated undirected graph G', which satisfies $\lambda(G') \leq \lambda(G)$.

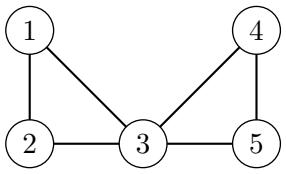
The idea is the following: for each edge in the edge cut, remove a vertex at one end to get a vertex cut. However, one has to be a little careful because doing this arbitrarily may not give us a vertex cut (indeed, they don't exist for K_n).

By definition, $\kappa(G) \leq n-1$, our proposition is true whenever $\lambda(G) \geq n-1$. Hence we may assume $\lambda(G) \leq n-2$. (This rules out K_n.) The previous proposition says that C partitions V into two disjoint subsets V_1 and V_2. We claim that there exist $u \in V_1$ and $v \in V_2$ such that (u, v) is not an edge in C. Otherwise, there must be an edge from each vertex in V_1 to each vertex in V_2, which would give us $|V_1| \times |V_2|$ edges. Consequently, for C to disconnect V_1 from V_2, we would need $k = |V_1| \times |V_2|$. It is easy to see $|V_1| \times |V_2|$ is minimized when either $|V_1|$ or $|V_2|$ is 1, and the other is $n-1$, so $k = \lambda(G) \geq n-1$, which we are assuming is not the case. Therefore, the claim is true.

Now each edge $e_i \in C$, pick a vertex v_i at one end of e_i such that $u \in V_1$ and $v \in V_2$ do not lie in $V_0 = \{v_1, v_2, \dots, v_k\}$. Here the v_i's need not be distinct, so this set may have less than k elements. Removing V_0 removes all the edges in C also, so V_0 forms a vertex cut of size $\leq k$ (it must disconnect u from v). Hence $\kappa(G) \leq k = \lambda(G)$. □

The above argument suggests that, in order to construct a graph with $\kappa(G) < \lambda(G)$, we need a graph where there is a minimal edge cut that involves repeated vertices. Here is an example.

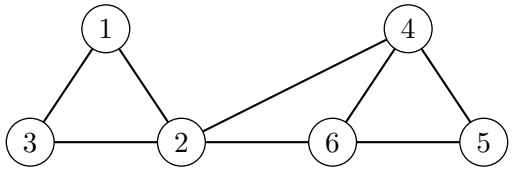
Example 1.6.12. *Consider the following graph G.*



Then $\kappa(G) = 1$ ($\{3\}$ is a vertex cut), but there is no edge cut of size 1. However, there are edge cuts of size 2, e.g., $\{\{1,2\},\{1,3\}\}$, so $\lambda(G) = 2$.

In fact, we can construct an infinite family of examples.

Example 1.6.13. *Take two cycle graphs C_m and C_n, and make a new graph G by connecting one vertex v_0 of C_m to two different vertices of C_n. For example, if $m = n = 3$ (the smallest size of cycle graphs), we can do this*



Then removing any single edge from C_m, or from C_n, or one of the 2 connecting edges will not disconnect the graph since C_m and C_n are 2-edge-connected. Hence our new graph is also 2-edge-connected. However, removing the vertex v_0 of C_m (2 in the above picture) which connects to C_n will disconnect C_n from C_m minus v_0. Thus this graph satisfies $\kappa(G) = 1$ and $\lambda(G) = 2$.

Let's return to the question of network design. First consider the problem of designing a robust network at minimal cost (let's not worry about efficiency yet). Say we want a network that will still be functional (connected) if some number of nodes or edge fail. Then we can set a threshold number k, depending on our expectations of this network, such that if any set of $< k$ nodes or edges fail, our network is still connected. That is, we want a k-connected network. Now we can ask what is the minimum number of edges we need.

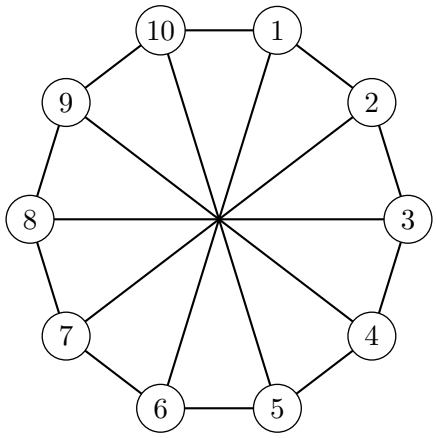
Proposition 1.6.14. *Let $G = (V, E)$ be an (undirected) graph on n nodes. If G is k-connected (or k-edge-connected), then each vertex of G has degree at least k. In particular, $|E| \geq \frac{nk}{2}$.*

For k-connected directed graphs, we will have $|E| \geq nk$.

Proof. Clearly the first statement implies the second, so we just need to prove the first. Fix any vertex $v_0 \in V$, and let d be the degree of v_0. Then if we remove the d edges coming out of v_0, we disconnect the graph. Hence we have an edge cut of size d. Thus $\kappa(G) \leq \lambda(G) \leq d$, i.e., $d \geq k$. □

Now we can ask if one can actually construct a k-connected graph on n vertices with $\frac{nk}{2}$ edges. Well, clearly this is impossible if nk is odd, so really the best one can ask for is a k-connected graph with $\lceil \frac{nk}{2} \rceil$ edges. When $k = 2$, we want a 2-connected graph with n edges. Here we see the cycle graph C_n is 2-connected with n edges, so this is possible for $k = 2$. For general k, such graphs were constructed by Frank Harary, and now known as Harary graphs, and denoted $H_{n,k}$.

Example 1.6.15. *We will construct $H_{2n,3}$, i.e., a 3-connected graph on $2n$ vertices with the minimum possible number of edges, $3n$. Start with a cycle graph C_{2n} on $V = \{1, 2, \ldots, 2n\}$. Now connect each vertex to its diametrically opposite pair (this is why we assume an even number of vertices, but the construction is similar for an odd number of vertices). For example, for $n = 10$ we have:*

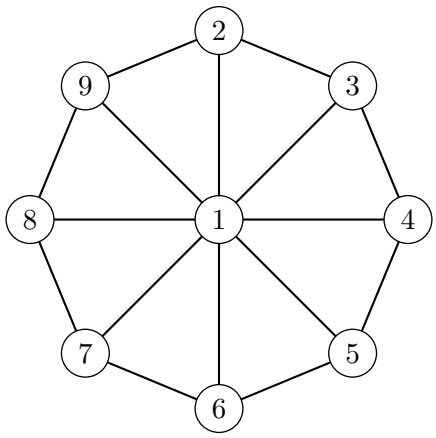


It is an exercise to check $\kappa(H_{2n,3}) = \lambda(H_{2n,3}) = 3$.

We won't bother going through the general construction of Harary graphs, though see the exercises for $k = 4$. The reason is the following: they don't have very small diameter, and therefore aren't appropriate for making efficient networks. For example, when $k = 2$, $H_{n,2} = C_n$ has diameter $\lfloor \frac{n}{2} \rfloor$. Similarly, the diameter of $H_{2n,3}$ grows linearly in n (see Exercise 1.6.10).

If we allow ourselves to increase the cost (i.e., the number of edges), we can bring ourselves down to diameter 2, by combining the star graph with a cycle graph (or if you prefer, adding a vertex to the center of $H_{2n,3}$).

Example 1.6.16. *Let $n \geq 4$. The* **wheel graph** *of order n is the undirected graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$ obtained by taking a cycle graph on $\{2, 3, \ldots, n\}$ and connecting the vertex 1 to each of the vertices 2, 3, ..., n. Here is a picture of the wheel graph of order 9.*



The vertex 1 acts as the hub for the network.

It is easy to see the wheel graph has diameter 2 and is 3-connected (and 3-edge-connected), with $2n - 2$ edges (compared to a minimum possible $\lceil \frac{3}{2}n \rceil$ edges for 3-connected graphs).

Now how good would a wheel graph be for an actual network? Well, it depends on what exactly we want, but let's suppose 3-connectivity is enough for us. While it is somewhat robust, and efficient, and not too costly (the number of edges grows linearly in n, as opposed to quadratically), it is highly reliant on the hub (the central vertex 1) for most short paths. For many networks, hubs (nodes that are connected to a relatively large number of other nodes) are quite desirable. For organization/administrative purposes, it's often convenient to have a few central nodes (think of a flight path network). However, the wheel graph has only one hub, so it will experience all of the traffic and if there is a problem with the hub, the network, while still connected becomes very inefficient (a cycle graph with diameter $\lfloor \frac{n}{2} \rfloor$). Imagine if 90% of all US flights went through Chicago. Now imagine this during a winter snowstorm.

In practice, networks may have a number of hubs, of varying sizes. You may not be able to get everywhere in 2 steps, but maybe you can get to most places in 2 or 3 steps, and in several ways, so traffic can be rerouted. This provides some sort of compromise between our three desired qualities: efficiency, robustness and cost effectiveness.

At the other end of the spectrum, we could have highly decentralized networks, meaning an absence of hubs. It turns out these make excellent networks in practice also, provided there aren't administrative reasons for wanting a centralized network.

Definition 1.6.17. *Let G be a graph (simple or not, but undirected). We say G is* **k-regular** *if each vertex of G has degree k. We say G is* **regular** *if it is k-regular for some k.*

Note C_n, K_n and $H_{2n,3}$ are regular graphs, where as linear graphs, star graphs and wheel graphs are not. We know by Proposition 1.6.14 that a k-regular graph is at most k-connected, and the Harary graphs show we can achieve k-connected k-regular graphs when nk is even. Now we can ask, how efficient can k-regular graphs be?

Proposition 1.6.18. *Let G be a k-regular graph on n nodes with $k \geq 2$. Then* $\operatorname{diam}(G) > \log_k(n(k-1)) - 1$.

Proof. Assume $G = (V, E)$ is connected and fix a vertex $v_0 \in V$. Now run through the algorithm to find the connected component of v_0. In other words, we find the neighbors of v_0, then its neighbors' neighbors, and so on. At the first step, we find k neighbors, i.e., k elements distance 1 from v_0. At the next step, for each neighbor, we have at most k neighbors of this neighbor, so there are at most $1 + k + k^2$ vertices of distance ≤ 2. Similarly, there are at most $\frac{k^d-1}{k-1} = 1 + k + k^2 + \cdots + k^d$ vertices of distance $\leq d$ from v_0. Our algorithm cannot terminate before this number is $\geq n$, i.e., there exists some vertex v of distance d from v_0 with $\frac{k^d-1}{k-1} \geq n$, which implies there are two nodes in G which are distance $d > \log_k(n(k-1)) - 1$ apart. □

This is not the best possible lower bound for the diameter of a k-regular graph, but it is not too far off. The point is the diameter has to grow at least logarithmically in n. It turns out that if we look at a random k-regular graph, it will with very high probability be k-connected (say nk is even, otherwise k-regular graphs don't exist) and have diameter that is essentially logarithmic in n, which is much much better than the linear growth of diameter for graphs like C_n or $H_{2n,3}$. Time permitting, we will explain this in greater detail when we begin our discussion of random graphs in earnest.

In closing this section, you might notice that we don't have a single measure of how good a network is, we have several—diameter/average distance, vertex/edge connectivity, and size. We also don't have any direct measure of how "robustly efficient" a network is—meaning, if not too

many nodes go down, will the network still be efficient? This of course is very important in practice. It's reasonable to guess that if a graph is k-connected but not too many nodes or edges go down in comparison to k (e.g., $k/4$ or $\sqrt{k}$), that the graph will still be fairly efficient. This is not necessarily always true (e.g., if you have a few central hubs, and they all go down), but it's often true. When we get to spectral graph theory, we will see that looking at eigenvalues provides a way to measure how good the "network flow" is. This will give us a convenient quantity which provides a nice balance of the qualities of efficiency, robustness and robust efficiency.

Exercises

Exercise 1.6.1. *Draw all possible unlabelled trees of order 6.*

Exercise 1.6.2. *Draw all possible unlabelled trees of order 7.*

Exercise 1.6.3. *Let G_0 be a (simple undirected) graph with minimum degree ≥ 2, i.e., each vertex has degree ≥ 2. Show G_0 has a cycle of length > 2. (Hint: start at any vertex try to trace out a simple path, and show it must eventually lead to a repeated vertex.)*

Exercise 1.6.4. *Let $\mathcal{G}_{n,e}$ denote the set of (simple undirected) connected graphs of order n with e edges. Let $f_n(e)$ denote the minimum possible average distance for a graph $G \in \mathcal{G}_{n,e}$. Show that $f_n(e+1) < f_n$ for $n-1 \leq e < n(n-1)$. In other words, unlike diameter, you can always get smaller average distances by adding in more edges.*

Exercise 1.6.5. *Let G be a connected graph of order n with n edges. What is the maximum possible value for $\operatorname{diam}(G)$? Explain why, and explain how to construct graphs with this diameter.*

Exercise 1.6.6. *For $n = 4, 5, 6, 7$, do the following. Compute the number of (unlabelled) trees of a given diameter $2 \leq d \leq n-1$, and determine the probability that a given tree of order n has diameter d. Assume each tree of a given order is equally likely (this is not the case in practice if you try to randomly generate trees by a reasonable method, e.g., it is not very likely you will generate a linear graph).*

Exercise 1.6.7. *Write a Python function* `randtree(n)`*, that randomly generates a tree on n nodes as follows. Start with vertex 2. Now add vertex 2 and connect it to vertex 1. Then add vertex 3, randomly select one of vertices 1 and 2, and connect 3 to that vertex. Continue in this process until you have n nodes, and return the adjacency matrix. Then, using this function:*

(i) By generating 100 random trees of order 4, estimate the probability of getting each type of unlabelled tree of order 4. Do the same for order 5. (Hint: for $n = 4, 5$, you can determine the isomorphism type of the tree by looking, e.g., at the diameter, maximum degree, or number of leaves.)

(ii) For each $n = 5, 10, 20, 50, 100$, generate 100 random trees of order n, and estimate the expected diameter of a random tree of order n.

Exercise 1.6.8. *Let G be a simple undirected graph on n nodes. Show if $G \neq K_n$, then G has a vertex cut.*

Exercise 1.6.9. *Prove Proposition 1.6.10.*

Exercise 1.6.10. *Show that $H_{2n,3}$ from Example 1.6.15 has vertex and edge connectivities of 3. Determine the diameter.*

Exercise 1.6.11. *For $n = 5, 6, 7$ vertices, construct a graph with $2n$ edges which is 4-connected. Can you generalize this to arbitrary n?*

1.7 Weighted Graphs and Travelling Salesmen

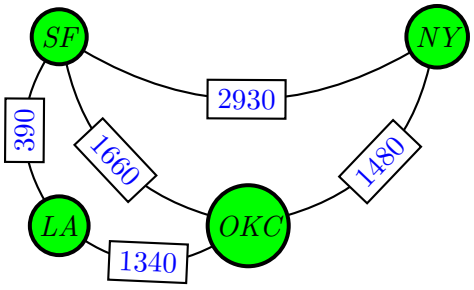
▶ Here graph means undirected graph.

Definition 1.7.1. *A* **weighted graph** $G = (V, E, w)$ *is a graph* (V, E) *together with a weight function* $w : E \to \mathbb{R}_{>0}$.

In other words, a weighted graph is a graph to which we assign each (undirected) edge a weight, which is a positive real number. (One can also consider nonpositive weights, but for our applications, we want positive weights.) The weight of an edge is typically thought of as the cost of using this edge (which might be a physical distance, or a financial cost, or a time cost, or some combination of these relevant for the problem at hand). We draw this graphically by drawing our graph as usual, and then writing the weights on or next to each edge. Much of what we have done so far can be done in the context of weighted graphs.

First, we can still represent graphs with matrices. If the vertex set is $V = \{1, 2, \ldots, n\}$, put $w_{ij} = w(i, j)$ if $(i, j) \in E$ and $w_{ij} = 0$ else. Then we can represent the weighted graph $G = (V, E, w)$ with the weighted adjacency matrix $A = (w_{ij})_{ij}$.

Example 1.7.2. *Here is a weighted graph which depicts some approximate road distances among four cities: New York, Oklahoma City, San Francisco and Los Angeles.*



The weight between two cities is an approximate road distance (in miles). We did not include an edge between LA and NY because going through OKC is approximately the shortest way to get from LA to NY. The weighted adjacency matrix with respect to the vertex ordering {*NY, OKC, SF, LA*} *is*

$$A = \begin{pmatrix} 0 & 1480 & 2930 & 0 \\ 1480 & 0 & 1660 & 1340 \\ 2930 & 1660 & 0 & 390 \\ 0 & 1340 & 390 & 0 \end{pmatrix}.$$

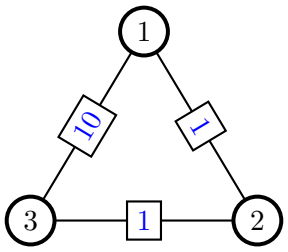
Paths are defined the same way for weighted graphs as for unweighted graphs, except now one might define the length of the path to be sum of the weights of the edges. To avoid confusion of terminology, we won't use the word length for weighted paths (so you won't just think the number of edges), but we'll use the word cost. That is, if γ is a path in G represented by a sequence of edges $(e_1, e_2, \ldots, e_k)$, then the **cost** of γ is $\sum_{i=1}^{k} w(e_i)$. For instance, in our example above the cost of the path from LA to NY given by (LA, OKC, NY) is $1340 + 1480 = 2820$.

Note that if $G = (V, E, w)$ is a weighted graph where we assign each edge weight 1, the cost is the same as our definition of length for the unweighted graph (V, E). Indeed, we can view the theory of graphs as a special case of the theory of weighted graphs where all edges have weight

1. (One can define weighted directed graphs similarly, however we will only discuss the weighted undirected case here.)

Then one defines distance in the same way: $d(u, v)$ is the minimum possible cost of a path from u to v, or sets $d(u, v) = \infty$ if no such paths exist. Again, we have the triangle inequality so distance defines a metric on weighted graphs (i.e., it satisfies the primary properties you expect from a notion of distance). Diameter again is the maximum distance between two vertices. The notions of (vertex or edge) connectedness are the same for weighted graphs as for unweighted graphs, as the weights on the edges play no role in vertex or edge cuts.

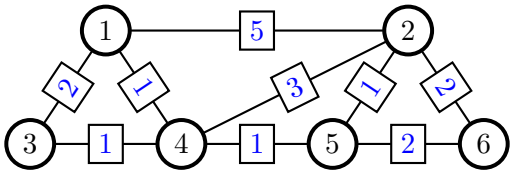
A path of minimal cost between u and v will be simple by the same argument given for unweighted graphs. One thing to be careful of is that the cheapest (i.e., lowest cost) path (or paths) from u to v may not use the fewest number of edges. For instance, consider the weighted graph



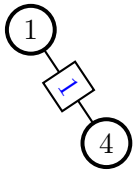
Here the shortest path from 1 to 3 goes through 2, rather than taking the direct edge from 1 to 3.

For this reason, the algorithm for computing distances that we discussed for unweighted graphs needs to be modified to work for weighted graphs. The basic idea is the same as the `spheres` function. Starting at some vertex u, we will do a breadth-first search to find the closest vertices, then the next closest, and so on. In the process, we will construct what is known as an **minimum spanning tree**. This is a subgraph of the connected component of u, which is a tree that contains only the edges needed to reach any vertex in the component of v with a shortest possible path.

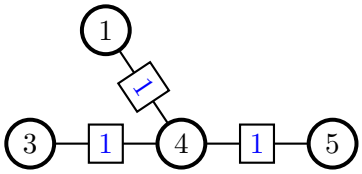
We will just explain the algorithm by way of an example. Consider the following weighted graph.



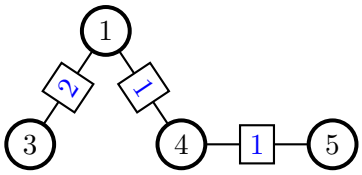
Here's how we can grow out a minimum spanning tree (MST) starting from vertex 1. Initially, the MST just contains 1. First we look at the neighbors of 1: there is 2, 3, and 4. Of these 4 is the closest (distance 1). Therefore, the shortest path from 1 to 4 must be the direct edge from 1 to 4 (any path from 1 to 4 must start from going to either 2, 3 or 4—if we go to 2 first, the path must have cost greater than 5, and if we go to 3 first, the path must have cost greater than 2). Thus we will add the edge from 1 to 4 to our MST, so it looks like this.



Now we look at the closest neighbors of 4—besides 1, there are two: 3 and 5. This gives us two paths to consider from 1 to 3, either $(1,3)$ or $(1,4,3)$, both of which have cost 2. We can choose either of these to be in our MST since they have the same cost. We also get one path from 1 to 5, namely $(1,4,5)$ which has cost 2. Again, looking at the neighbors of 1 tells us no other path to 5 can be shorter to this one, so we will add the edge $(4,5)$ to our tree. Hence at this stage, we have either the MST

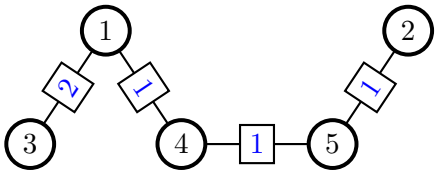


or the MST

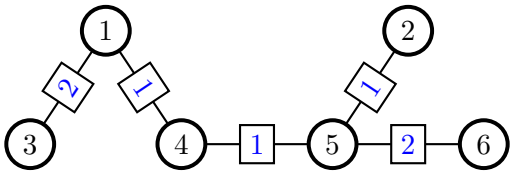


depending on which of the shortest paths we used to get from 1 to 3. For this example, let's choose the latter.

Now we can explore the new neighbors of 3 and 5. The neighbors of 3 don't get us anywhere new, so just consider the new neighbors of 5: 2 and 6. The closest one is 2, and the corresponding path cost $(1,4,5,2)$ is cost 3. Now we compare this with the other paths we've already found to 2: $(1,2)$ and $(1,4,2)$. They have costs 5 and 4, so $(1,4,5,2)$ is the shortest. Thus we will add the edge $(5,2)$ to our MST:



Now we look at the new neighbors of 2: there is just 6. We've now found 2 paths to 6: $(1,4,5,2,6)$ and $(1,4,5,6)$. The latter is shorter, so we add the edge $(5,6)$ to our MST, giving:



Since 6 has no new neighbors, and now we've included all vertices we've encountered, so this completes our minimal spanning tree. In a tree, there is a unique path between any pair of vertices (Exercise 1.7.1) so we can unambiguously read off the distance from 1 to any other vertex by looking at the path in the MST. Namely, we see $d(1,4) = 1$, $d(1,3) = d(1,5) = 2$, $d(1,2) = 3$ and $d(1,6) = 4$.

The reason this algorithm works, and works efficiently, is the following: if we have a path from u to v of minimal cost that passes through w, the part of the path going from u to w must also be of minimal cost. For example, when we were considering paths from 1 to 6, we didn't need to consider paths like $(1, 2, 6)$ or $(1, 3, 4, 5, 6)$ because at that point in our algorithm we already knew that $(1, 2)$ is not the most efficient way to get to 2 and $(1, 3, 4)$ is not the most efficient way to get to 4. This algorithm runs in $O(n^2 \log n)$ time, or to be more precise, $O(|E| \log n)$ time.

Of course, an MST will not tell you all paths of least possible cost—there are 2 from 1 to 2, but only 1 can be in the MST. Finding all paths of least possible cost is a different problem, but can be done with a simple variant of this algorithm.

With this algorithm to compute an MST for a vertex u, one can compute distance as mentioned above or the diameter. Again the algorithm to compute diameter is similar. If the graph is disconnected, it is infinity. Otherwise, do the following. Given an MST for u, one can find the vertex (or vertices) furthest from u, and record this maximum possible distance d_u. Now do this for every u and take the maximum.

Consequently, even for weighted graphs, it is not too difficult to find an optimal way to get from Point A to Point B. However, what if we want to the optimal way to visit multiple places? This might seem like a problem that should not be too much harder than finding an optimal route between two locations, but it is not so simple because trees no longer suffice to address this problem.

Definition 1.7.3. *Let G be a weighted or unweighted graph on n nodes. A* **Hamiltonian cycle** *(or* **Hamiltonian circuit***) is a cycle on G containing all n nodes.*

If $n > 1$, this is equivalent to being a cycle of cost n. Such cycles sometimes exist and sometimes do not, and deciding whether they do or not is a computationally hard problem. Technically, it is what is known as an NP-hard problem—in particular, it is believed this problem is not solvable in polynomial time. There are at most $n!$ possible cycles of cost n in a given graph (Exercise 1.7.2), and constructing a given cycle takes $O(n)$ time, so we can at solve this problem (and construct a Hamiltonian cycle when they exist) in $O(n \cdot n!)$ time. Recall Stirling's approximation says $n! \sim \sqrt{2\pi n}(n/e)^n$, which is superexponential. In the 1960's, an $O(n^2 2^n)$ algorithm was discovered by Bellman and Held–Karp (independently). This is exponential as $O(n^2 2^n) \subset O(3^n)$.

Given a weighted graph G, the **travelling salesman problem (TSP)** is to find a Hamiltonian circuit of minimum possible cost. More colloquially, suppose there are n cities you need to visit for business, but the order in which you go is not important. How can you plan a route to all the cities, and go back home, that is as short (or cheap) as possible? Hence the name travelling salesman problem. The TSP is a fundamental problem in optimization, and has applications to areas such as logistics problems, microchip design and DNA sequencing. (In DNA sequencing, the nodes are DNA fragments, and the distance measures similarity between two fragments.)

If you think about it a little, you might notice there's a slight difference between my definition of the TSP in terms of Hamiltonian circuits and my colloquiual description. Namely, a Hamiltonian circuit visits each node except the start node exactly once, whereas the least cost tour of n cities may involve taking a path through a city you've already visited. However, we can account for this with Hamiltonian cycles as follows. Let G be the weighted graph representing n cities, with an edge representing a direct physical route between 2 cities. (There may be more than one direct physical route, but for this problem it suffices to include only the one of minimal cost, which will be the weight of the edge.) Now it may be that there are two cities u and v with no edge between them (i.e., no direct physical route—e.g., no road or direct flight between the two), or it may happen that the direct route from u to v is not the most economical. In this case, we make an edge (or

replace the existing one) between u and v whose weight is the cost of the most economical path from u to v. This transforms G into a complete weighted graph G' (a weighted graph with edges between all pairs of distinct vertices, i.e., K_n with weights on the edges) where the weight of any edge (u, v) is precisely $d(u, v)$. Solving the TSP on G' really is equivalent to finding the least cost physical tour of the n cities in G, though one needs to keep track of what physical route in G is represented by each edge in G'. The construction of G' from G can be done in polynomial time, as distances can be computed in polynomial time.

Assume now G is a complete weighted graph. For complete graphs, it is easy to generate a Hamiltonian cycle—we can visit all nodes in any order we like! Consequently, we get $n!$ Hamiltonian cycles (cf. Exercise 1.7.2). Once we specify a starting vertex, there are $(n-1)!$ Hamiltonian cycles. Computing the costs of each of these cycles takes $O(n)$ time, so it is possible to solve the TSP in $O(n \cdot (n-1)!) = O(n!)$ time. Again one can do better—$O(n^2 2^n)$ run time is possible, but the TSP is also an NP-hard problem, and we expect that it cannot be solved in polynomial time—in fact, exponential time may be the best possible*.

Even though these two problems—the TSP and finding Hamiltonian circuits—are closely related and solvable in the same amount of time (essentially the same algorithm solves them both, and you can provably reduce† solving one problem to solving the other), here is one feature of TSP that is harder than the Hamiltonian cycle problem. Given a possible solution to the TSP, i.e., some Hamiltonian cycle, it is still hard to determine if this proposed solution has minimal cost. On the other hand, given a possible solution to the Hamiltonian cycle problem, it is easy to determine if it is a Hamiltonian cycle or not.

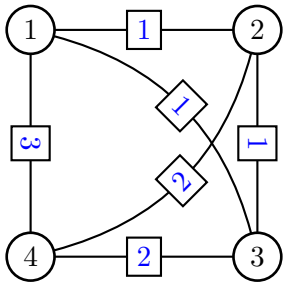
Since the TSP is hard, but of practical importance, what can we do? We have a few options: we can try to solve the TSP for special classes of graphs, we can try to find probabilistic algorithms to solve the TSP in faster time (this means, they will work some percentage of the time, but they won't give the correct solution, or at least not quickly, in some cases), or we can try to find faster *heuristic algorithms* which give approximate solutions to the TSP (i.e., find Hamiltonian cycles of relatively low cost, but not necessarily the minimum possible). Of these approaches, the latter is typically the most practical, and we'll discuss this briefly now.

The simplest algorithm you might imagine is, starting from your home vertex, travel to the neighbor of minimum distance away (or pick one if there are several). From there, again travel to the closest neighbor (or pick one if there are several) that you haven't already visited. Repeat this until you have visited all nodes, and take the unique path home. Remember, we are working with complete graphs, so this algorithm will always give some Hamiltonian cycle. This is called *the greedy algorithm*, because at each step it chooses the cheapest available. However, this may not be cheapest in the long run as the following example shows.

Consider this graph:

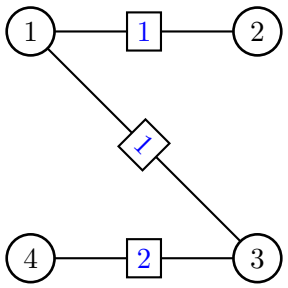
*There a subexponential/superpolynomial range of functions growing faster than polynomials but slower than exponential functions, so not being solvable in polynomial time does not mean exponential running time is the best possible. On the other hand, there are some things that take longer than exponential time (superexponential) to solve! The TSP can be solved in exponential time, but the naive $O(n!)$ algorithm is superexponential—it is essentially of the same order as $O(n^n)$.

†Meaning a polynomial-time reduction.



Let's try to find a minimal cost Hamiltonian cycle starting and ending at vertex 1. What happens if we use the greedy algorithm. At first, there are two cheapest options available—we can go to 2 or 3. Let's say we go to 2. Then there is a unique cheapest option available—going to 3. From 3, we have to go to 4, and back to 1. Thus we have taken the path $(1, 2, 3, 4, 1)$, with cost $1+1+2+3 = 7$. If we had alternatively selected the path from 1 to 3 at the first step, this would force us into the cycle $(1, 3, 2, 4, 1)$, which in this example also has cost 7. Neither of these is as cheap as possible, as both require traveling along the most expensive edge $(4, 1)$ in the graph. The best we can do for this graph is the cycle $(1, 2, 4, 3, 1)$ or the cycle $(1, 3, 4, 2, 1)$, both of which have cost 6.

Many times the greedy algorithm gives an approximate solution to the TSP that is not too far from optimal, but sometimes is can be much worse. However, it is certainly fast. It runs in $O(n^2)$ time. In 1976, Christofides discovered an polynomial-time algorithm based on a minimal spanning tree which find a Hamiltonian cycle that costs no more than 1.5 times the cost of an optimal solution. Roughly the idea is to make a minimal spanning tree, travel along a path in the tree until it ends, then jump to another path. However, this jumping paths is done in an intelligent way. In the above example, we can take for a minimal spanning tree from 1 the following:



We can start out by either picking the path $(1, 2)$ or $(1, 3, 4)$ in the MST. Then we should jump to the end of the other one, and travel back to 1. By chance, either of these choices give a Hamiltonian cycle of minimal cost, either $(1, 2, 4, 3, 1)$ or $(1, 3, 4, 2, 1)$. Of course for larger graphs, one needs to be more careful about how to jump from one path to another as there are many choices to make, and we won't typically get an optimal cycle, but at least one that's not too far off.

After 3 decades of essentially no progress along the lines of Christofides' algorithm, there have been exciting developments in this direction over the past several years, and now one can find a Hamiltonian cycle that is no more than 1.4 times the cost of an optimal one in polynomial time (Sebö–Vygen, 2012).

This is not to say that research on the TSP was stagnant from the late 70's to the mid 2000's. There has been, and still is, much work on alternative kinds of fast heuristic algorithms to approximate solutions to the TSP. However, it is very difficult to accurately assess how close the

approximate solution is to the optimal solution—and of course, it should be difficult to assess, since we don't have a good way to get a handle on the optimal solutions for comparison.

One simple alternative approach is the *Monte Carlo* method. We just start from our initial vertex, and at each stage, travel to a new vertex chosen at random, and repeat until we have to go home. We do this many times and keep track of the best solution so far. In other words, the Monte Carlo approach just tries a fairly large number of random paths, and selects the best among those. This will work well if we are in a situation where most Hamiltonian cycles are relatively cheap, and we just need to avoid certain bad paths. However, it won't work well if there are only a few good paths to find.

There are also a lot more interesting approaches, such as genetic algorithms (algorithms that evolve themselves based on past performance), simulated annealing, Tabu search and ant colony optimization (based on artificial intelligence models of ant colony behavior). In fact, the TSP is often used as a benchmark to compare different kinds of general optimization philosophies. The TSP has also crept into cognitive psychology—psychologist have studied how good humans are at solving the TSP (we're pretty good, though I'm not sure if we're as good as ants), and what algorithms most closely model how Earthlings "naturally" (approximately) solve the TSP.

Exercises

Exercise 1.7.1. *Let T be a tree, and u and v be nodes in T. Show there is a unique simple path from u to v.*

Exercise 1.7.2. *Let G be a (simple) graph of order $n > 1$. Show that G has at most $n!$ cycles of length n, with exactly $n!$ occurring in the case that G is complete.*

Exercise 1.7.3. *Let G be a complete weighted graph of order $n > 1$. Suppose you have enumerated all $n!$ Hamiltonian cycles and computed their costs and stored them in a table. Show that you can find the smallest possible cost in $O(n!)$ time. Is it possible to do better than this (just using this table)? (Note: even for the naive algorithm to solve TSP, one would not store all Hamiltonian cycles and their costs in a table as this would require superexponential space—instead, we can just keep track of the best so far.)*

Exercise 1.7.4. *Let G be the weighted complete graph on $V = \{1, 2, 3, 4\}$, where the weight of an edge (i, j) is given by $\min(i, j)$. Solve the TSP by hand for G, with initial vertex 1. (Give a minimal cost Hamiltonian cycle, and the cost.) Do the same for initial vertices 2, 3, and 4.*

Exercise 1.7.5. *Let G be the weighted complete graph on $V = \{1, 2, 3, 4, 5\}$, where the weight of an edge (i, j) is given by $\min(i, j)$. Solve the TSP by hand for G, with initial vertex 1. (Give a minimal cost Hamiltonian cycle, and the cost.)*

1.8 Further topics

Since graphs arise in many ways in many situations, there are a plethora of angles from which one can come to the study of graph theory. We've barely touched the surface of classical graph theory, and now it's time to move on. (By classical graph theory, I mean something like: the aspects in graph theory that whose study began before humans started sending things to the moon, or the parts of graph theory whose study involves mostly just combinatorics, or what I knew something

about when I was an undergrad. The important thing is I mean certain parts of graph theory that people thought about before they had to worry about really large graphs or being bothered by sociologists and economists.) In fact, a 1-semester course just on classical graph theory still isn't enough to cover all the "basics."

I'd like to give you a little overview of the classical graph theory that we're skipping in this class, but it a vast, sprawling field, sort of like a a big, complicated graph, and difficult to summarize succinctly. At a very general level, a lot of graph theory is studying invariants of graphs and seeing what they tell you—e.g., if you have two graph invariants (e.g., number of edges and vertex connectivity), does one of them imply anything about the other? Often people study these questions restricted to certain types of graphs, e.g., trees or regular graphs, where one often gets nicer answers. Another general type of question is: what conditions imply certain properties of the graph (e.g., when can we guarantee the existence of a Hamiltonian cycle?).

One large subarea is *extremal graph theory*, where one tries to determine the optimal bounds on one invariant in terms of others. We've touched on this above—if the vertex connectivity of an undirected graph on n nodes is k, then it must have at least $\lceil \frac{nk}{2} \rceil$ edges, and this bound is optimal because one can construct Harary graphs $H_{n,k}$ with vertex connectivity k and exactly $\lceil \frac{nk}{2} \rceil$ edges. Another typical question is: what is the minimum number of edges required for a graph on n nodes to have *clique* of order m, i.e., a subgraph isomorphic to K_m. (We'll say a little about cliques later.) Or: given a k-regular graph on n nodes, what is the minimum possible diameter (we gave a lower bound, but it is not optimal).

There is a large overlap of graph theory with the field of *enumerative combinatorics*. Here the typical question is to count the number of graphs (or subgraphs, or path, or cuts, etc) with certain properties. For example, count all undirected graphs (or trees, or k-regular graphs) on n vertices up to isomorphism. Sometimes one is interested in a question not originally phrased in terms of graphs, and then one interprets it in terms of certain kinds of graphs, and tries to count these kinds of graphs (or prove something about them).

Another subarea is *algebraic graph theory*, which uses linear algebra and group theory (groups are a fundamental object in algebra—a group is essentially the symmetries of some object). to study graphs. E.g., what do the eigenvalues of the adjacency matrix tell us, or what do the group of automorphisms of a graph tell us? Conversely, graphs are often used as tools to study groups. We'll look at eigenvalues in the third part of the course.

Graphs are also closely related to *finite geometries*—these are finite sets of points and lines which satisfy a set of axioms like Euclid's axioms for plane geometry. This is part of *algebraic combinatorics* and has applications to cryptography and the theory of error-correcting codes, which are important in engineering and communications.

There are many other aspects and areas of graph theory that we won't get to in this course, but let me just tell you about what is perhaps the most famous result in classical graph theory: the *four-color theorem*. Draw any map on a piece of paper. That is, draw a set of continuous curves on your paper that divide the space up into a finite number of contiguous regions. This is what we'll call a planar map. We can turn this map into a graph by making each region a vertex and connecting two vertices with an edge when the corresponding regions share a common border (not just a point). This is essentially what we did for the Königsberg bridge problem, except we used edges to denote bridges, not borders there.

A graph is called **planar**, if we can draw it in the plane $\mathbb{R}^2$ with no two edges overlapping. It is clear by construction that if you start with a planar map, the associated graph is also planar.

Definition 1.8.1. *Let S be a set of size k, which we think of as denoting k different colors. A* **k-coloring** *of a graph $G = (V, E)$ is a map $\alpha : V \to S$ such that if $(u, v) \in E$, then $\alpha(u) \neq \alpha(v)$. The minimal k such that G has a k-coloring is called the* **chromatic number** *$\chi(G)$ of G.*

In English, a k-coloring of a graph G is just a way to color the vertices of G with k distinct colors in such a way that no two adjacent vertices are the same color. If G has order n, then we can clearly color G with n different colors. The chromatic number $\chi(G)$ is just the smallest number of colors we need to color the vertices of G with the above rule.

Theorem 1.8.2 (Four-color theorem)**.** *Let G be a planar graph. Then $\chi(G) \leq 4$.*

In other words, any planar graph can be colored with at most 4 different colors. Thus we can color the regions of any map in the plane using at most 4 colors so that no two bordering regions have the same color.

This is a nice, simple-to-understand result, of course, but the reason it's so famous is because of it's history. Despite it's simplicity, it resisted proof for over 100 years, the proof was controversial at the time, and we still don't have a good way to understand *why* it is true.

The four-color theorem was originally stated in 1852, but with an incorrect proof. Many "proof" and "counterexamples" were since proposed, but were later discovered to also be incorrect. On the other hand, in 1890, Heawood provided a simple (correct) proof that any planar graph can be colored with at most 5 colors. Finally, in 1976, Appel and Haken announced a proof of the four-color theorem that is now believed to be correct. Based on Heawood's result, it suffices to show no planar map requires 5 colors. The basic idea of the proof is to assume there is a planar graph G that requires 5 colors, and use a reduction argument to yield a smaller graph that requires 5 colors. Appel and Haken, through much work, reduced the problem to considering an explicit set of 1482 cases which were checked by a computer to all be 4-colorable. At the time, the computer calculations themselves were a great achievement, which took over 1000 hours of computing time.

Proofs are deemed correct or flawed or incomplete by consensus. Typically, especially for problems of significant interest, the proofs are carefully checked by other experts by hand. However, this was the first serious example of a computer-proved theorem and doubts remained about it's validity—both general doubts about computer proofs because it could not feasibly be verified by hand and specific doubts about the actual code. People are always skeptical of new things, but it really is very hard to verify correctness of complicated computer output. First, there is the issue of guaranteeing that the machine is doing exactly what you tell it (no hardware/environment issues), then verifying the correctness of the code itself, which can easily have a minor, hard-to-find bug. (For example, there is a problem in combinatorics/coding theory known as Berkekamp's Switching Game, proposed by Berlekamp in 1960. This was "solved" by computer in 1989. I had two undergraduates work on a generalization of this in the summer of 2002 and, the night before the end of the summer program, they unexpectedly discovered, again by computer, that the original solution was wrong!)

In response to some of these doubts, Appel and Haken published a very detailed monograph of the proof of the four color theorem in 1989, including computer calculations, which was over 700 pages. This is now generally accepted, and since then other researchers have done separate computer proofs of the four-color theorem to double-check its correctness, but there is no known complete proof by hand.

The other issue with this computer proof is that it is not very enlightening—traditional proofs (at least good ones) typically do not just verify the truth of a statement, but also give us intuitive

understanding of why it is true. Reducing a problem to 1500 cases, which are checked individually, fails to give a good *reason* why it is true. This is not to take away from Appel and Haken great accomplishment—there was a very hard result, and there may be no real simple or enlightening proof of the four-color theorem.

There are innumerably many texts on graph theory and combinatorics that you can see for more information on the topics discussed on this chapter. Eventually, I may add a list of specific references here or in the introduction or at the end, but most of what we have talked about can be found on almost any introductory text on graph theory, though many books will stick to the case of undirected and possibly simple and/or unweighted graphs. (Part of the problem is, there are so many books, it's hard to choose what to put on a reference list.) One exception is the TSP, for which you should turn to a book on algorithmic graph theory, or a general book on algorithms or combinatorial optimization, for more details.

Exercises

Exercise 1.8.1. *Show K_4 is planar, but K_5 is not.*

Exercise 1.8.2. *Determine the chromatic number of the cycle graph C_n.*

Exercise 1.8.3. *Determine the chromatic number of the complete graph K_n.*

Exercise 1.8.4. *Determine the chromatic number of the star graph of order n.*

Exercise 1.8.5. *Determine the chromatic number of the wheel graph of order n.*

Chapter 2

An overview of social networks

A society is a collection of individuals which are interrelated. At its simplest form, a society may be represented by a network or graph. The graphs that arise this way are called *social networks*. Examining these networks can give us insight into how we interact with each other and the world around us, and what influences our behavior or success in different areas.

For instance, here are a couple of well-known examples of social networks.

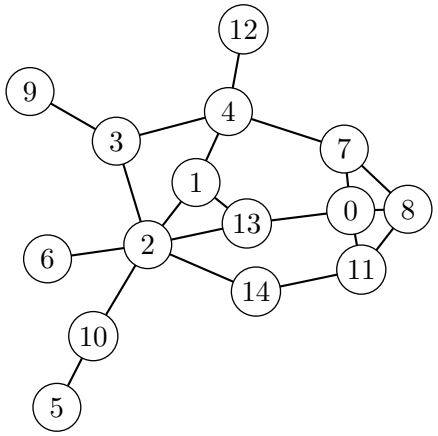


Figure 2.1: Florentine families ca. 1430

In Figure 2.1, each vertex represents a key Florentine family, and the edges denote a connection by marriage (Padgett and Ansell, 1993). One of these families rose to prominence and consolidated much of the power in 15th century Florence. Can you guess which one by looking at the graph? (Think about it now.)

Figure 2.2 depicts members of a Karate club, where the links represent friendship (Zachary, 1977). If I told you there was a schism in the club, and it later split into two rival clubs, can you guess how the split went just from looking at the graph? (Think about it now.)

While you may not have been able to give precise, definite answers to these questions, you probably guessed something pretty close to what actually happened. The fact that you can do this, even without any training, is a testament to how understanding the structure of networks can give you real insight into the workings of society. The Florentine family that rose to the top wasn't the one that was richest or most politically influential at first—its rise to prominence can instead explained by the social interrelationships of these 15 families.

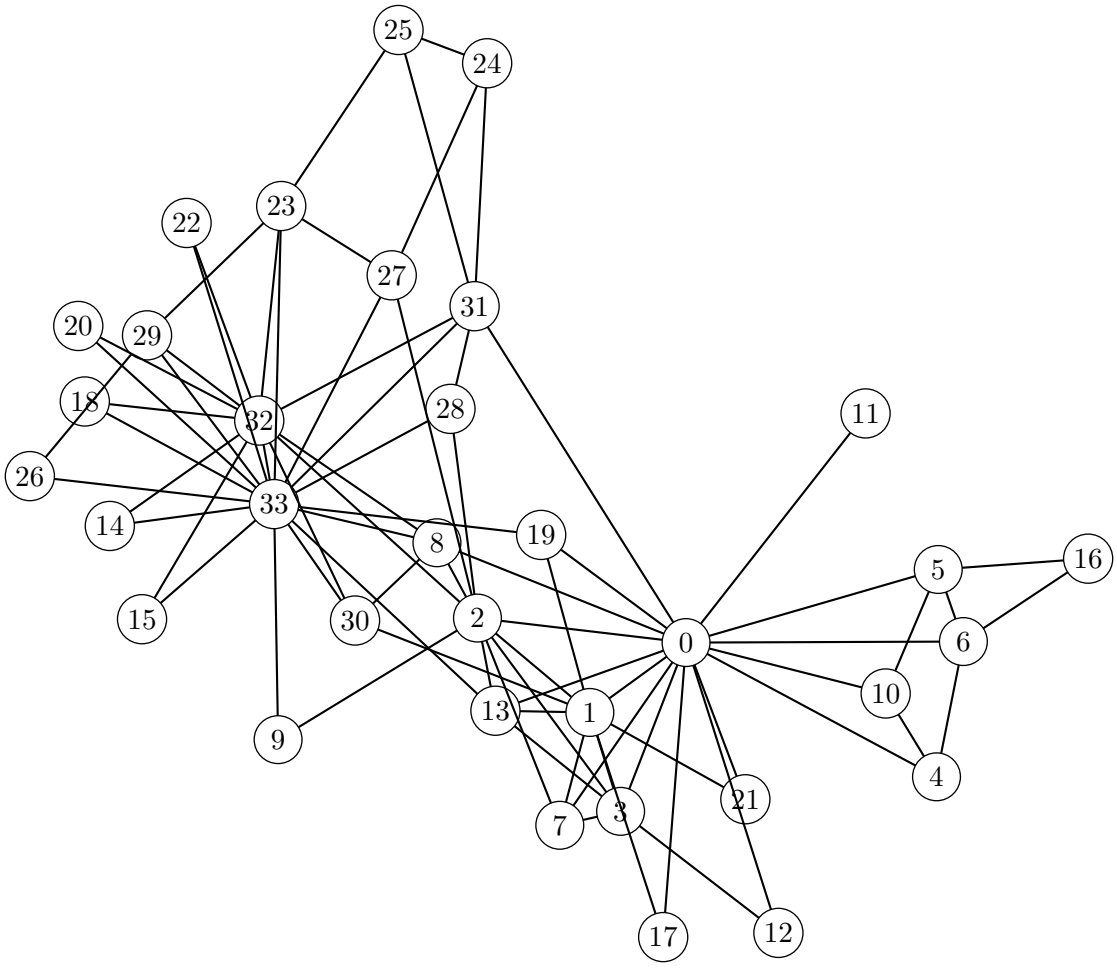


Figure 2.2: The Karate Club

Just like with classical graph theory, there are many points of view from which one can enter the study of social networks. One might come to social networks as a sociologist, or anthropologist, or linguist, or economist, or computer scientist, or engineer, or biologist, or businessperson, or investigator. Two principal questions a social scientist would ask are:

- What does the structure of a social network tell us about society?
- How do social networks form?

The first question can be taken at an local, individual level or a broader, global le Your position in society, who you're connected to, and who your connections are connected to, and so on, plays a large role in forming your beliefs, what opportunities you have, and your decision making.

Notice the second question, how do networks form?, implies that social networks change over time. They are dynamic. Vertices may come an go, edges may appear and disappear. Understanding how networks form will tell us how they may evolve and give us insight into questions such as how much do you your present connections play a role in forming future connections.

For good or for ill, I am not a social scientist, but I'll try to guide you into the area of social networks from the point of view of one mathematician. This means we'll ignore some questions

that are important to a social scientist (what is the best way to sample a network? how do you collect your data and turn it into a graph, e.g., if you're constructing a friendship network, how do you tell if two people are friends? how do you interpret graph invariants? etc), but focus on getting a deeper, structural understanding of networks. For us, two guiding questions will be

- How can we measure the structure of a network?
- How can we model social networks?

We've already seen some ways to get a handle on the structure of a network in the last chapter—e.g., connectedness, diameter, vertex degrees. Now we'll start to consider other measurable quantities of social networks that gives additional insight into the social structure of the network. For instance, we'll provide different ways to measure how important a given node is in a network. Oh, by the way, in the Florentine families network, the family that rose to power was the de Medicis (vertex 2). This was despite the Strozzis (vertex 4) being previously richer and more powerful. One graphic explanation is that vertex 2 has the highest degree, but a better explanation is that the de Medicis are more centrally positioned in the network. We'll look at some measures of *centrality* later.

Another aspect of this question is, we would like to characterize social networks in terms of important properties. We've seen that the graph isomorphism problem is difficult, which means we won't in general be able to determine a graph up to isomorphism by looking a few simple, easy-to-compute invariants. (Note: things like the adjacency matrices are not invariants, since they depend upon an ordering of vertices—there are up to $n!$ possible adjacency matrices for a given graph.) So instead, we'll examine what are some of the more important features/invariants of a graph from the point of view of social networks. This is particularly important when we are dealing with *complex networks*, i.e., really big networks like the internet.

This notion of characterizing social networks by key properties is important when it comes to the question of modeling social networks. By modeling social networks, we mean finding an algorithmic way to generate graphs whose key properties are essentially the same as those of social networks found in nature. These methods typically depend upon a random process—maybe at each stage you add/remove a node or edge with some given probability. Depending on these probabilities (which could be different for different nodes or edges), the kinds of networks you grow will look different. If these *random graphs* we generate have the same good-looking features as certain types of social networks, we'll consider these random graphs a model for these social networks. Note this also gives a model for how such social networks can form.

Over the past 20 years or so, there has be an explosion of interest in social networks, and there are lots of books on social networks now. Unfortunately, most of them are either not mathematical enough for our purposes or require a much more serious mathematical background. There are also recent introductory mathematics books about graph theory which discuss some aspects of social networks, but not everything we want to talk about. Two references I can recommend are *Social and Economic Networks* by Matthew O. Jackson (2008), an economist at Stanford, and *Networks, Crowds and Markets* by David Easley and Jon Kleinberg, an economist and computer scientist at Cornell (2010). The latter book is available freely online in preprint form at: http://www.cs.cornell.edu/home/kleinber/networks-book

In the rest of this chapter, we'll explain what are some "landmarks" or key features/invariants of social networks, and discuss a couple of basic ideas in network modelling, before delving more

deeply into some of these topics in our remaining two chapters. But first we'll talk about working with graphs in Sage.

Graphs in Sage

From now on, we'll do most of our computational work with graphs in the (free, open-source) mathematical package Sage, that has many built-in functions for working with and visualizing graphs. At present, the current version is Sage 6.1.1, though when I started writing these notes it was Sage 5.12. Most, if not all, of the things we will do should work on either of these versions of Sage, as well as any versions of Sage released in the near future. Sage is written in Python 2, and all of the things we've done in Python will also work in Sage.

(You may now wonder why we didn't start with Sage, and there are a couple of reasons. One, I think it is pedagogically better to start with Python first. I feel you get a better understanding of graph theory algorithms and how they are implemented by having to write some without yourself without too many tools available. Two, once you can use Python, Sage is a cinch. In addition, it's a useful skill to be able to program, and Python is a widely-used programming language, whereas most non-math people don't know what Sage is. So if you can put Python skills on your resume, that can be a big plus when looking for jobs.)

Let me also mention that there is a Python package called `networkx` for working with graphs. For some things, Sage is better, while `networkx` may be better for other things. I think both are equally difficult to install for Windows, but on a Mac, Sage is easier to install. Sage has the additional benefit that one can run it online without installation (e.g., https://cloud.sagemath.com/).

In any case, we will use Sage in this course, and now briefly explain how to get started working with graphs in Sage. See the lab page http://www.math.ou.edu/~kmartin/graphs/lab5.html for some other details and more references. This lab page also includes the data for the Florentine families and karate club graphs, in the Python adjacency matrix form we have been using so far.

First, Sage has a built-in graph object type, so to use the graph theory features of Sage on a graph we represented in Python with an adjacency matrix A, we'll need to convert it to a Sage graph object. We can do this by converting A to a Sage matrix (Sage also has a built-in matrix type), and then using the Sage matrix to generate a graph. For instance

Sage 6.1
```
sage: A = [ [0, 1, 0], [1, 0, 1], [0, 1, 0] ]
sage: m = matrix(A)
sage: G = Graph(m)
sage: G
Graph on 3 vertices
sage: G.show()    # or G.plot()
```

Here the lines that begin with `sage:` are what you input into Sage, and the other lines are the Sage output.

The commands `show()` and `plot()` are two functions that will draw the graph (slightly differently) on the computer for us. (Note neither of these commands draw the graph in a canonical way, so if you try plotting the same graph multiple times, it will not look the same each time.) I will not typically include the plots in these notes—you should try playing around with these or similar examples in Sage yourself and see the plots on your computer. There is also a `plot3d()` command

that gives you a 3-d visualization of your graph. Go to the lab page now, and use Sage to plot the Florentine family and Karate club graphs.

There are also many built-in constructors for generating graphs in Sage. You can see a list by typing `graphs.` (note the period) followed by *tab*. For example, try the following

Sage 6.1
```
sage: C10 = graphs.CycleGraph(10)
sage: C10.show()
sage: K5 = graphs.CompleteGraph(5)
sage: K5.show()
```

Pretty much everything we have talked about so far is already implemented in Sage. Here is a list of some basic commands. They are all run in the form `G.`[command], like the `show()` and `plot()` commands.

- `am()` — returns a (Sage) adjacency matrix
- `vertices()` — returns the list of vertices
- `order()` — returns the order
- `edges()` — returns the list of edges
- `size()` — returns size
- `degree(u)` — returns the degree of u
- `average_degree()` — returns the average degree
- `connected_components()` — returns the connected components
- `distance(u,v)` — returns the distance from u to v
- `shortest_path(u,v)` — returns a shortest path from u to v
- `diameter()` — returns the diameter
- `average_distance()` — returns the average distance
- `vertex_connectivity()` — returns the vertex connectivity
- `edge_connectivity()` — returns the edge connectivity
- `chromatic_number()` — returns the chromatic number

2.1 Landmarks of Social Networks

Here we discuss some, but not all, prominent features of social networks.

2.1.1 Connectedness

We've presented 3 real examples of social networks—the OU Math collaboration graph in Figure 2, the Florentine families graph in Figure 2.1 and the Karate club graph in Figure 2.2. Of these, the first is not connected but the latter two are. Some social networks are connected, but there's no reason to expect this in general. E.g., consider a graph of Facebook users, who are connected by a link if they are friends. There will be small groups of people who are only friends with each other, and therefore constitute make a connected component.

However, empirically, social networks tend to have a *giant component*. We'll give a precise definition in the context of random graphs (there's not a good precise definition for a single graph), but you can get some idea of what we mean by looking at the OU Math collaboration again—there's one connected component which is much bigger than the other ones, and it comprises a significant percentage of the total number of nodes in the network.

Here's another example—the Web Data Commons Hyperlink Graph (`http://webdatacommons.org/hyperlinkgraph/`). This is a directed graph where the nodes are webpages, and there is an edge from page A to page B if there is a hyperlink from page A to page B. This has (at present) about 3.5 billion nodes and about 128 billion links. If we look just at connected components in the undirected sense, this has a giant component consisting of about 3.34 billion nodes (94%). If we look at strongly connected components, there is a giant component consisting of about 1.8 billion nodes ($\sim 51\%$).

Of course, one has to be careful in interpreting this because of how the data was collected. This graph does not represent all webpages at a given time (this may not even be a sensible notion, as many webpages are dynamic—e.g., Google search result pages). Instead this data was collected from crawling the web. You find a node at random to start at, and then follow its links to find other nodes. If you only do this once, you'll only stay inside the connected component of the node you start at, and won't necessarily visit all pages with a link to your initial node. So you do this many times and collect the results. Still, this procedure tends to give you a graph that may be "more connected" than just picking a whole bunch of nodes at random. Of course, another issue is how do you find random webpages without finding them as links from another webpage? There are some methods, but there is no way to find all webpages with equal probability without being omniscient. (You'll never find hidden webpages.) However, this graph gives a good sense of the part of the web that is in common use.

2.1.2 Degree distributions

Euler solved the Königsberg bridge problem just by looking at the degree of vertices in the graph. Similarly, we can learn quite a bit about a social network just by looking at the degree of each node. We could just look at a list of the degrees of each node, and this can be done in Sage with the `degree_sequence()` function. However, it is often more convenient to count the number of nodes of a given degree, which can be done in Sage with the `degree_histogram()` function. We can either look at straight vertex counts (as the `degree_histogram()` function does), or normalize by the total number of vertices. The latter is what is known as the degree distribution, and this normalization allows us to easily compare degree distributions for graphs with different numbers of vertices.

Denote by $\mathbb{Z}_{\geq 0}$ the set of nonnegative integers $\{0, 1, 2, 3, \ldots\}$.

Definition 2.1.1. *Let $G = (V, E)$ be an undirected (not necessarily simple) graph. The* **degree**

distribution *of G is the function $P : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ defined by $P(d)$ is the proportion of nodes in V of degree d.*

Note the degree distribution defines a *probability function* on $\mathbb{Z}_{\geq 0}$—it satisfies $P(d) \geq 0$ for all d and the probabilities

$$\sum_{d=0}^{\infty} P(d) = 1$$

sum to one. (We'll review some basic probability theory later when we really need it.) For a given graph G of order n, of course we'll have $P(d) = 0$ whenever $d > n$ so the above sum is really a finite sum, and we can consider P as a probability function on the finite set $\{0, 1, 2, \ldots, n\}$. However, it is theoretically convenient to consider P as a function on all nonnegative integers so that (i) we can compare degree distributions for graphs with arbitrary number of nodes, and (ii) we can model some degree distributions with continuous functions like $P(d) = c/d^2$, where c is an appropriate normalization constant to make the total probability 1. (More on how to interpret the latter type of distribution below.)

We will also want to define degree distributions for directed graphs, to be able to talk about the Web Data Commons Hyperlink Graph for instance. In this case there are two kinds of degrees we can look at for a node.

Definition 2.1.2. *Let $G = (V, E)$ be a directed graph, and $u \in V$. The* **in degree** *of u is $|\{v \in V : (v, u) \in E\}|$, i.e., the number of edges which point to (end at) u. The* **out degree** *of u is $|\{v \in V : (u, v) \in E\}|$, i.e., the number of edges going out of (starting from) u.*

Definition 2.1.3. *Let $G = (V, E)$ be an directed (not necessarily simple) graph. The* **in degree distribution** *of G is the function $P : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ defined by $P(d)$ is the proportion of nodes in V of in degree d. The* **out degree distribution** *of G is the function $P : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ defined by $P(d)$ is the proportion of nodes in V of out degree d.*

Again, the in and out degree distributions define probability functions.

Note if G is given as an adjacency matrix A, we can find the out degree by counting the number of 1's in the row corresponding to u, and the in degree by counting the number of 1's in the column corresponding to u. On the other hand, if G is given as an adjacency list, it is easy to get the out degree of u. However, one has to go through the whole adjacency list to get the in degree, which is more time consuming (still polynomial time, in fact $O(n^2)$, but this is nontrivial if n is on the order of 1 billion).

If one is working with a really large directed graph, such as the Web Data Commons Hyperlink Graph, it is not feasible to store the graph as an adjacency matrix, so one has to store it as an adjacency list. So for large directed graphs, in degree is harder to get at than out degree. This is apparent if one thinks about how one has to collect the data. Given a webpage A, it is easy to find the out degree—just count the number of links to different pages on page A. However, if we want the in degree, we need to find all webpages that link to A, which is not easy. Consequently, it's not such a simple task to see how "popular" an individual page is (thinking of popularity in the sense of how many webpages link to this page—on the other hand, the server that hosts the webpage keeps track of how often it is accessed, so popularity in the sense of being accessed a lot is relatively easy to keep track of).

Let's consider two extreme degree distributions to see what kinds of things they tell us about a network. For purposes of illustration, let's think about friendship networks, i.e., the vertices are people and we connect them with a link if they are friends.

At one extreme, we could have a "delta-type" distribution, e.g., $P(5) = 1$ and $P(d) = 0$ for $d \neq 5$. This says the probability that a vertex has degree 5 is 1, so the probability a vertex has any other degree is 0. Since there are only a finite number of vertices, this means ever vertex has degree 5, i.e., our friendship network is 5-regular, i.e., everyone has exactly 5 friends. Of course this is not very likely, but we could loosen things up and have a "bump" distribution where most people have exactly 5 friends, some have 4 or 6 friends, and a few have 3 or 7 friends.

At the other extreme, we could have a distribution that runs the gamut of all possible degrees. There are different ways to do this—you could try for a "flat" distribution of the form $P(d) = c$ for $d \leq k$ and $P(d) = P(0) = 0$ for $d > k$. (This is not possible for $c = 1/n$; see Exercise 2.1.1.) (We could also choose to allow $P(0) = c$ so that there are nodes with no friends.) This would mean there are the same number of really popular people (with maximum degree k) as the same number of really lonely people (with only 1 friend), which is also the same as the number of people of average popularity (say $k/2$ friends). However a distribution like this is quite unlikely for social network, so let's think of how else we could do this.

If we broader our thought process a bit, we can think about another scenarios where people range over a gamut of possibilities. One would be something like exam grades, or IQ scores, which are often modeled with a Bell curve, which is a *normal* (or *Gaussian*) *distribution*. However, this is basically the same as what I called a bump distribution above. Another type of distribution where people run a whole spectrum of possibilities is a wealth distribution. Think about the distribution of wealth in the US. There are a lot of poor people, quite a few middle class, and a very few Richie Riches. The wealth distribution is often modeled by a **power law distribution**. This is a distribution of the form

$$P(d) = cd^{-\alpha}, \qquad \alpha > 1.$$

(Note this expression is infinite for $d = 0$, so we will set $P(0) = 0$ so there are no isolated nodes.) Here c is a normalization constant to make the total probability 1. Namely, for this to be a probability function, we need that

$$\sum_{d=1}^{\infty} \frac{c}{d^\alpha} = c \sum_{d=1}^{\infty} \frac{1}{d^\alpha} = c\zeta(\alpha) = 1,$$

where $\zeta(s)$ denotes the Riemann zeta function

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} = 1 + \frac{1}{2^s} + \frac{1}{3^s} + \cdots,$$

which converges for $s > 1$. (This is why we insist $\alpha > 1$.) Hence, the constant $c = \frac{1}{\zeta(\alpha)}$, and we can write our power law distribution in the form

$$P(d) = \frac{1}{\zeta(\alpha) d^\alpha}, \qquad \alpha > 1.$$

We remark that explicit formulas for $\zeta(\alpha)$ is known when α is an even integer, e.g.,

$$\zeta(2) = \frac{\pi^2}{6}, \quad \zeta(4) = \frac{\pi^4}{90}, \quad \zeta(6) = \frac{\pi^6}{945}, \quad \zeta(8) = \frac{\pi^8}{9450}, \quad \cdots$$

In practice, ones need to compute $\zeta(\alpha)$ numerically, which is not hard. (Of course, in practice, even when α is even, one also has to approximate π.)

A graph that follows a power law distribution is called a **power law graph**. Many authors also call this a *scale-free graph* (and the power law distribution a *scale-free distribution*), though for some people the term scale-free has a more specific usage. The power law distribution is scale free in the sense that if you plot the graph in the range (.1, 10) or (.001, 1000), the graph will look the same, i.e., zooming out by a factor of 100 does nothing to the visual appearance of the graph. Consequently, if we have graphs with 100 nodes, 10,000 nodes and 1 billion nodes all with scale-free degree distributions with the same parameter α, then the degree distributions for these graphs will follow exactly the same shape. Roughly, the refined notion of a scale-free graph is a graph that looks the same when you look at the whole thing or zoom in to just see a part of it, similar to the way a fractal behaves. Such a graph must have a scale-free (power law) degree distribution, but having a power law degree distribution does not guarantee fractal like properties for a graph. We will use the term scale-free graph in its refined sense, and say power law graph when we just mean some graph with a power law distribution.

Now it may seem perplexing that for all $d = 1, 2, 3, 4, \ldots$, the proportion of nodes of any degree d can be nonzero. For instance, if we have a graph G on 100 nodes that follows a power law degree distribution $P(d) = \zeta(2)^{-1}d^{-2}$, how is it possible that $P(100) = 6/(100\pi)^2 \approx 0.00016 > 0$? Unless we allow loops or multiedges, the maximum degree of a vertex is 99. In addition, for a given graph G, the proportion of nodes of degree d must be a rational number, whereas $P(d) = 6/(\pi d)^2$ is irrational for all d. The answer of course is that no individual (finite) graph will have a degree distribution which is exactly a power law distribution. Rather, a power law distribution will be a convenient mathematical approximation of an actual distribution. It is helpful to think of $P(d)$ as being the probability that a given node has degree d, so $100P(d)$ should be approximately the number of nodes of degree d. Since $100P(100) \approx 0.016$, which rounds down to zero, this says we probabilistically expect 0 nodes of degree 100 (and similarly for higher degrees), and this indeed is what we logically expect.

Put another way, we can think of a power law distribution as a *model* for the degree distributions for certain networks. Allowing $P(d) > 0$ for all $d > 0$ in fact confers an advantage on us—this model is valid for all n. Returning to our example with $\alpha = 2$, we saw that $P(100) \approx 0.00016$. This means if we have a growing network following this degree distribution, by the time $n = 10000$, we expect to see 1 or 2 nodes of degree 100.

Many people think social networks roughly follow power-law distributions, so one often asks, given a social network, how close is the degree distribution to a power law distribution? The easiest way to see this visually is to look at what is called a *log-log plot* of the degree distribution. Note that if we have a power law graph $y = cx^{-\alpha}$, taking logs of both sides gives

$$\log y = \log(cx^{-\alpha}) = \log c + \log x^{-\alpha} = \log c - \alpha \log x.$$

So if we have data (x_i, y_i) that we suspect follows a power law for some exponent α, the data $(\log x_i, \log y_i)$ must satisfy a linear relation, and we can determine α by looking at the slope. The plot of the data $(\log x_i, \log y_i)$ is called the log-log plot. See Figure 2.3. There is a standard statistical procedure known as simple linear regression which can give us an objective measure of how close the data $(\log x_i, \log y_i)$ is to being linear, and approximating the slope α, but we won't discuss this here.

The web page http://webdatacommons.org/hyperlinkgraph/topology.html has log-log plots of the distributions of in degrees and out degrees for the Web Data Commons Hyperlink Graph. If

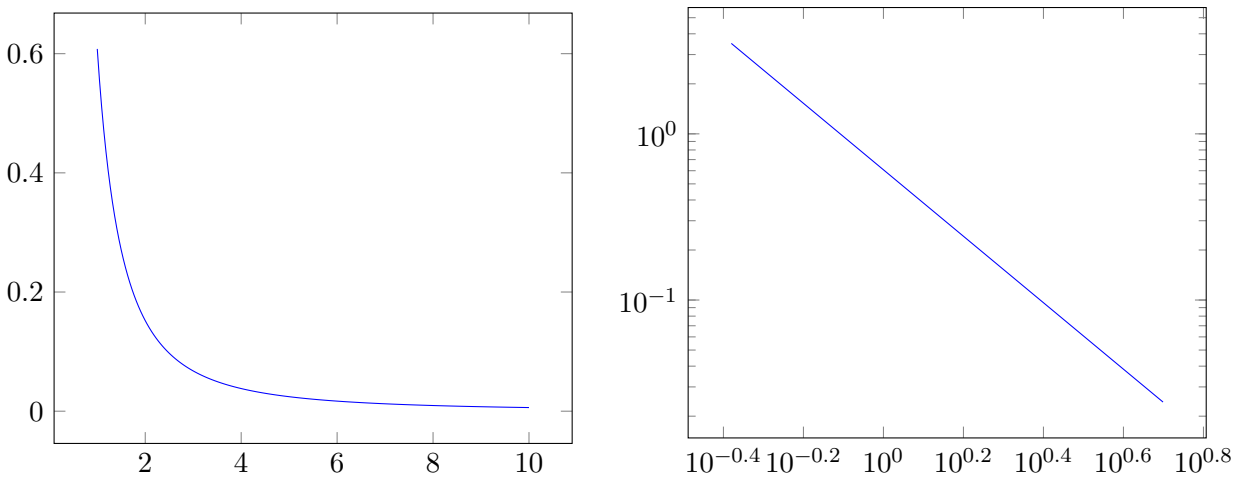


Figure 2.3: A power law distribution: the standard plot and a log-log plot for $\alpha = 2$

you take a look, you'll see the in degrees follow a power law distribution rather closely, at least for quite a while, but the out degree distribution does not look too linear in the log-log plot.

Let's take a look at the Karate Club graph from Figure 2.2. This has degree distribution given by $P(1) = P(9) = P(10) = P(12) = P(16) = P(17) = 1/34$, $P(2) = 11/34$, $P(3) = P(4) = 6/34$, $P(5) = 3/34$, $P(6) = 2/34$ and all other $P(d) = 0$. We've plotted the degree distribution in Figure 2.4 using both the standard and log-log plots. (Note, since $\log 0 = -\infty$, we need to omit the d's such that $P(d) = 0$ in the log-log plot.) Note the log-log plot does not look too linear, and indeed the standard plot does not look too much like a power function. One issue to take into account in this comparison is that the order of the graph is rather small ($n = 34$), so a few stray points on the graph will greatly alter its shape.

Looking at these graphs more closely, what are the differences with a power law graph? The first thing to notice is the Karate Club graph starts with a large spike, whereas a power law graph starts with a steep decline. If this friendship network really followed a power law graph, there should be a large number of people with only 1 friend, but here there's only one. Rather there are a large number of people with 2 friends, then the degree distribution drops sharply, and in the range $2 \leq d \leq 8$ (looking at the standard plot), it doesn't seem so far off from a power law shape. Indeed, the middle section of the log-log plot does not appear to be too far from linear. However, the last section (of both plots) go more-or-less horizontal. This is a function of working with a small value of n, so for the larger degrees where we don't expect too many such nodes, there are only really two possibilities that happen—for $d \geq 7$, either $P(d) = 0$ or $P(d) = 1$.

So what's our conclusion—is a power law a good model for this graph or not? Well, it's not my job to tell you how to think, so you can come up with your own conclusion. But we will revisit this question later when we talk about network models.

Here's how we can look at degree distributions is Sage.

Sage 6.1

```
sage: # K is the Karate club graph
sage: dh = K.degree_histogram()
sage: dh
[0, 1, 11, 6, 6, 3, 2, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1]
```

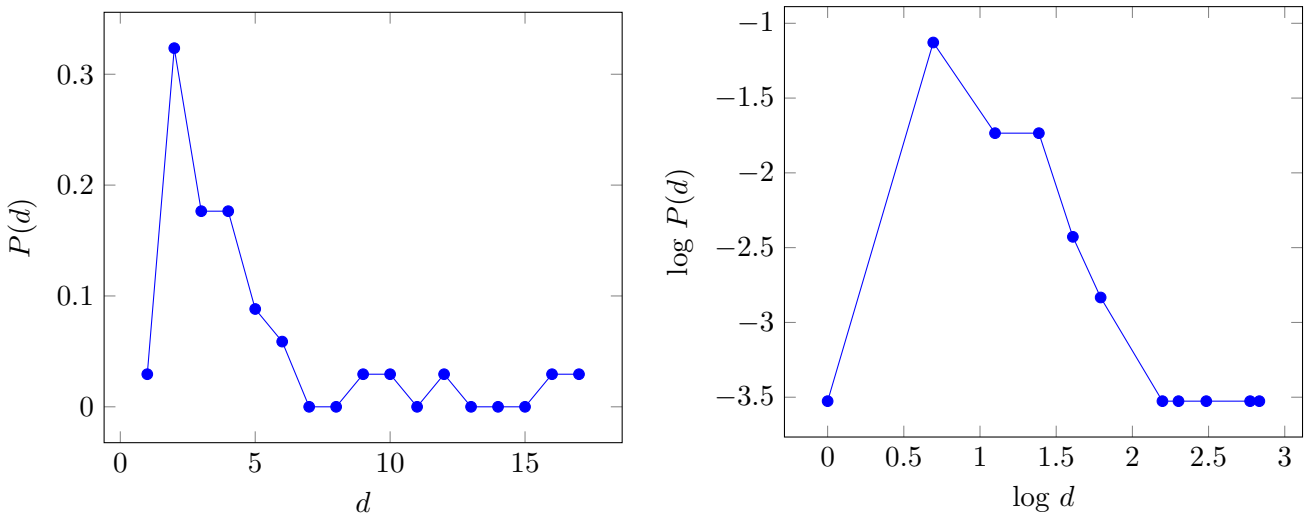


Figure 2.4: The Karate Club graph degree distribution — standard and log-log plots

```
sage: from sage.plot.bar_chart import BarChart
sage: bar_chart(dh)
```

There is no built-in function to get the degree distribution, but the `degree_histogram()` function returns a list whose d-th entry is the number of nodes of degree d (starting from $d = 0$). One can easily get the degree distribution from this, but for purposes of plotting, the degree histogram is sufficient. Sage supports line plots and bar chart plots—we chose to do a draw a bar chart for this example.

A related function is the `degree_sequence()` function, which returns a list consisting of the vertex degrees for each vertex, ordered reverse numerically.

2.1.3 Centrality

For our question about the Florentine families graph in Figure 2.1—which family rose to power—we want to understand how the "position" of a node in a network determines how much power or influence it has. There are different ways to try to measure this, and these notions fall under the heading of *centrality*.

The most basic measure of centrality is **degree centrality**. Given a (simple undirected) graph G, the degree centrality of a node is simply its degree divided by $n-1$. So if our node is connected to all other nodes, it has the maximum possible degree centrality 1, whereas if it is isolated, it has minimum possible degree centrality 0. In the Florentine families network, we see vertex 2 (the de Medici's, which is the family that rose to power) has the highest centrality, namely 6/13, or nearly 1/2.

However, degree centrality is a local notion, which I termed popularity in the Introduction. That is to say, the degree centrality of a vertex depends only on the neighbors of the vertex, and not how the rest of the network looks. According to (Padgett and Ansell, 1993), just looking at degree centrality in the Florentine families graph is not sufficient to explain how the de Medici's rose above the Strozzi's (vertex 4), who were richer and more politically powerful at the time.

In the Introduction, we observed in the collaboration graph in Figure 2 that while Przebinda and Özaydin have the same degree centrality, Przebinda is one step closer to most people in the large component. Similarly, the de Medici's are closer to more families than the Strozzi's in the Florentine graph.

To measure how close a node is to other nodes in a graph, we consider measures of **closeness centrality**. Then the inverse distance measure of closeness centrality is simply

$$\sum_{v \neq u} \frac{n-1}{d(u,v)}.$$

(For any v where $d(u,v) = \infty$, we regard $\frac{1}{d(u,v)} = 0$.) It is also natural to consider an inverse square measure:

$$\sum_{v \neq u} \frac{n-1}{d(u,v)^2}.$$

We normalized these measures by the maximum possible degree $n-1$ so if u has an edge to all other vertices v, these measures give the maximum value of 1. Of course one could consider arbitrary exponents $k \geq 1$ on $d(u,v)$ in these sums.

An alternative way to measure closeness centrality it to look at what is called *decay centrality*

$$\sum_{v \in V} \delta^{d(u,v)}, \qquad 0 < \delta < 1,$$

where δ is a parameter that can be adjusted. This does not require G to be connected as $\delta^\infty = 0$ for $0 < \delta < 1$. Note the limit as $\delta \to 1$ of the decay centrality of u is simply the size of the connected component of u.

Another notion of centrality is **betweenness centrality**. This measures the number of times a given vertex u lies on a (shortest length) path between other vertices v_1 and v_2, and gives a higher score the more times u appears. I won't give a precise definition, but both closeness and betweenness centralities give more "global" notions of centrality, better reflecting how well positioned a node is in a network. The de Medici's have high closeness and betweenness centralities, and this seems to be what gave them the ability to rise to power in 15th century Florence.

Sage has some of these centrality measures built in. Let's do an example.

Sage 6.1

```
sage: # F is the Florentine families graph
sage: # vertex 2 is de Medicis, vertex 4 is Strozzis
sage: F.centrality_degree(2)
0.42857142857142855
sage: F.centrality_degree(4)
0.2857142857142857
sage: F.centrality_closeness(2)
0.56
sage: F.centrality_closeness(4)
0.4666666666666667
sage: F.centrality_betweenness()
{0: 0.10256410256410256,
 1: 0.09157509157509157,
 2: 0.521978021978022,
 3: 0.21245421245421245,
```

```
4: 0.25457875457875456,
5: 0.0,
6: 0.0,
7: 0.1043956043956044,
8: 0.02197802197802198,
9: 0.0,
10: 0.14285714285714288,
11: 0.05494505494505495,
12: 0.0,
13: 0.11355311355311357,
14: 0.09340659340659341}
```

Here `centrality_degree()` is the normalized degree centrality, `centrality_closeness()` is the first measure of closeness centrality described above, and `centrality_betweenness()` returns the betweenness centrality of all nodes.

There is another, more sophisticated notion of centrality. How important a node is should be based on who it's neighbors are. The more important it's neighbors, the more important that node should be. This makes intuitive sense, but how can we use this idea to give a precise measure of centrality? We're basing our notion of importance on the notion of importance. There are various ways to resolve this recursive conundrum, the most elegant in my mind being **eigenvector centrality**. This notion forms the basis of the Google PageRank algorithm, and we will see how to define it in the next chapter with the notion of random walks.

2.1.4 Small world phenomena

Many social networks exhibit phenomena that are called *small world phenomena*. Loosely this means the network is very well connected, often surprisingly so. We already explored part of this idea a little bit with the notion of connectedness and giant components. Let's consider some large social network that you're part of—maybe the nodes are email or Skype or cell phone or Facebook users, and two people are connected if they've been in contact in the past month, or worse, are Facebook friends. For concreteness let's consider an actual (as opposed to Facebook) friendship network of students at OU.

Pretend you're in a story. Maybe you're really antisocial—you live off campus, you don't really meet other students, you just come to campus to go to class and leave, but on your way out from class, you randomly bump into this guy/girl who seems pretty cool, so you show him/her you're pretty cool too by not talking to him/her. Afterwards you realize this was pretty stupid, since you have no way of meeting him/her again. Still, you have this nutty professor for your Bokonism, Robotics and the African Diaspora class who makes you work in group projects, so you got together with this guy Kilgore in your group once or twice, and you guess you're sort of friends. He's also pretty antisocial, but even he has a couple other friends, and one of them has a bunch of friends in the Science Fiction Club. Some of the Sci-Fi'ers have friends in the Math Club. Everyone in the Math Club is friends, including that guy/girl you bumped into one and sort of liked, from whom you're only 4 steps away now. Gradually you meet Kilgore's friends, and eventually you start hanging out with some of the Sci-Fi kids. Then one of them who's in the Math Club tells you about this cool lecture he heard in the Math Club, and the good pizza. So you decide to check out the Math Club, and the next Wednesday you go up the the 11th floor of PHSC at 4pm only to discover no one's there. That's cause they don't meet every week. All of a sudden, the roof flies

off the building and you get sucked up in a tornado. Across from you, you see the guy/girl you kind of liked, and you work up the courage to wave and say hi. Only he/she doesn't wave back, because he/she is screaming uncontrollably, because the lower half of his/her is being eaten by a shark. But then the shark starts screaming because the lower half of his/her body (it's hard to tell the gender of a shark, especially in a tornado) is being eaten by a dinosaur. "All's well that ends well!" you scream. The end. True story. I have the screenplay rights.

The point is, even in you hang out with just a small group of people, though some random connections, you're connected to larger groups, which are connected to other large groups, and you'll see that most of the network is connected, or at least being eaten by sharks and dinosaurs. Sure, there may be some really antisocial people who don't make any connections, or little groups which are not connected to the rest of the network, but for friendship networks like this, we expect most people in the network to be connected to each other through some sequence of friends.

Another type of small world phenomena is what is usually referred to as *six degrees of separation*. In 1929, Frigyes Karinthy wrote a story (if you can call something without a sharknado a "story") about how the world was shrinking (not literally, unfortunately). In there, a character bets that if you choose any two humans on Earth at random, they are connected by a sequence of at most 5 personal acquaintances (not including themselves, so distance at most 6 in the "personal acquaintance network"). You could interpret this as a conjecture that the personal acquaintance network is connected with diameter at most 6, but this seems not likely to be strictly true, as there are pockets of societies with little-to-no contact with the rest of the world. However, a better interpretation is that if you choose two people at random, they are unlikely to be of distance more than 6 apart. I don't know if this is true or not—there's nothing inherently magical about the number 6—it was just some number Karinthy pulled out of his, um, writing cap—but I expect it is true for some relatively small distance. Incidentally, he phrase "six degrees of separation" was made famous by a play of the same name written by John Guare, but in his usage it means everyone is distance at most 7 apart.

Here is a heuristic reason. Let's make a very conservative estimate that you have made 100 acquaintances in your couple of decades on this earth. Similarly, your acquaintances will have at least 100 acquaintances each. So this yields potentially $100^2 = 10,000$ people of distance at most 2 from you. Of course there will be a lot of repetition in your acquaintances' acquaintances, but it seems reasonable to assume there are at least 1,000 people distance at most 2 from you. Continuing this form of estimation, you might estimate there are at least 10,000,000 people of distance at most 6 from you. Only a few more steps, and you're at 10 billion! See, you've got everyone covered! (This heuristic follows the same reasoning as the proof of Proposition 1.6.18.)

This seems like a hard problem to actually study at the level of personal acquaintances (how do you determine them, and how do you gather data?), but some studies have been done. One famous study was by Stanley Milgram in the 1960's who estimated the average distance between someone in Omaha and a specific person in Boston is less than 6. For this study, Milgram randomly chose 96 people in Omaha from the phone book, and sent them small packages, with instructions that they should send them to someone they know whom they think will be closer to the "target." That person should should do the same, recording the steps of the package. The target was a friend of Milgram's and specified by name, address and occupation. Of these packages, 18 reached Milgram's friend, with average path length about 5.9.

Another famous "study" is *six degrees of Kevin Bacon*. In 1994, Kevin Bacon was the center of the entertainment world (he said so himself!). Here we take the graph of all actors, and put

an edge between two if they worked on a film together. Any well-known actor will be in the same component as Kevin Bacon, which will be the giant component. At the time, there was a game to find a shortest path to Kevin Bacon, and usually the distance was less than six. Note this does not mean most pairs of actors are distance at most six away, though this may also be true—it only formally implies the distance between two random actors is usually less than 12.

Mathematicians have something similar, which is the notion of an Erdös number, and this is quite well-known too. Paul Erdös (1913–1996) was an itinerant Hungarian mathematician who wrote about 1500 papers with 511 different collaborators (though he almost never wrote anything up himself). Here we consider the collaboration graph of all mathematicians—two mathematicians are connected if they co-authored a paper together. The Erdös number of a mathematician is his or her distance from Erdös in the collaboration graph. Of course many mathematicians are not in the same component as Erdös, but of those that are (and there are many—it is a giant component), the average Erdös number is 4.65, the median is 5, almost all are below 8, and the maximum is 15. There are 511 with Erdös number 1 and 9,267 with Erdös number 2. (These numbers of course can change over time, and may be slightly dated now.) This provides some evidence that our estimate of 1,000 people of distance at most 2 from you in the acquaintance graph is reasonable (though Erdös is of course a special node), and that most people in a giant component are can be connected in a relatively small number of steps.

We can rephrase this mathematical notion of six degrees of separation as the statement: in the giant component of a social network, the average distance is relatively small and the diameter is is also not too large. We can be precise about the terms "relatively small" or "not too large" when we discuss random graphs.

Another aspect of small world phenomena is that given two nodes A and B, there tend to be many short paths from A to B. This is the feature that usually causes people to declaim, "It's a small world." For example, say there are two friends Abbey and Camille, and Camille introduces Abbey to her new ocarina teacher, Barry. Guess what? Abbey already knows Barry because he plays the card game Set with Abbey's brother Doug on Tuesday nights. Now there are two short paths which represent the ways Abbey knows Barry—through Camille, and through Doug. As they say, it's a small world.

This idea that there are often many short paths between two vertices is related to the notions of cliques and clusters, which we'll discuss next.

2.1.5 Cliques and clusters

A **clique** in an (undirected) graph G is a subgraph of G isomorphic to a complete graph. Cliques of order 1 are just single vertices. Cliques of order 2 are just pairs of vertices connected by an edge, so for simple undirected graphs the number of cliques of order 2 is just the number of edges. Cliques of order 3 are "triangles" in the graph, or just cycles of length 3 if we ignore the initial vertex (i.e., if we consider the cycles (a, b, c, a), (b, c, a, b) and (c, a, b, c) to be the same).

For instance, consider K_4. This has one clique of order 4 and 4 of order 3, 6 of order 2 and 4 of order 1. On the other hand, the cycle graph C_n for $n > 3$ has no cliques of order > 2.

Thinking in terms of a social network, say a friendship network, a clique is a subset of of people such that any two people in this set are friends. Knowing what the cliques are in a network will tell us a lot about the society. If you and I are friends, are most of your friends my friends also? Is the formation of the network very clique-ish (i.e., are most connections made by "networking"?), or are most connections made by chance encounters? In the first case, we expect many fairly large

cliques, where as in the second, not many at all.

The **clique number** of G is the maximum order of a clique in G. For K_n, it is clearly n. For C_n it is 2 unless $n = 3$, in which case it is 3 as $C_3 = K_3$. In the OU Math collaboration graph, the clique number is 3, but there are 7 cliques of order 3. For the Florentine families graph, it is also 3, with 3 cliques of order 3. For the karate club graph, the clique number is 5—this graph has 2 cliques of order 5 and 11 cliques of order 4 (2 of which are not contained in a clique of order 5).

Sage has a many built-in functions for cliques. Here is an example of a few.

Sage 6.1

```
sage: # Here F is the Florentine families graph
sage: F.clique_number()
3
sage: F.cliques_maximum()
[[0, 7, 8], [0, 8, 11], [1, 2, 13]]
sage: F.cliques_maximal()
[[0, 8, 7],
 [0, 8, 11],
 [0, 13],
 [2, 1, 13],
 [2, 3],
 [2, 6],
 [2, 10],
 [2, 14],
 [4, 1],
 [4, 3],
 [4, 7],
 [4, 12],
 [5, 10],
 [9, 3],
 [11, 14]]
```

As you can guess, `clique_number()` returns the clique number of the graphs. The function `cliques_maximum()` returns all the cliques of maximum possible order. The function `cliques_maximal()` returns all *maximal cliques*, meaning they are not contained in larger cliques.

One issue with counting cliques in a social network is that cliques are unstable under minor changes. Remember that social networks are dynamic, so we want to look at measures of graphs which are robust in the sense that they do not vary wildly under small changes to the network. For instance, just adding the single edge $\{7, 11\}$ would change the clique number of the Florentine families graph from 3 to 4.

Even from a "static" point of view, the notion of clique is rather rigid for what we want to measure. We might want to look for "generalized cliques" in the network—tightly-knit groups in the network where perhaps not every pair in this network is directly connected, but most are. This brings us to the notion of *cohesiveness* and graph decompositions.

Let's go back to the example of the karate club in Figure 2.2. The karate club graph has 2 "hubs", vertex 33 of degree 17, and vertex 0 of degree 16. One is the instructor of the club and the other is the student founder. From a networks point of view, this graph can be roughly partitioned into 2 subgraphs around these hubs which are cohesive (relatively tightly-knit) groups. Here is a picture of how the split went.

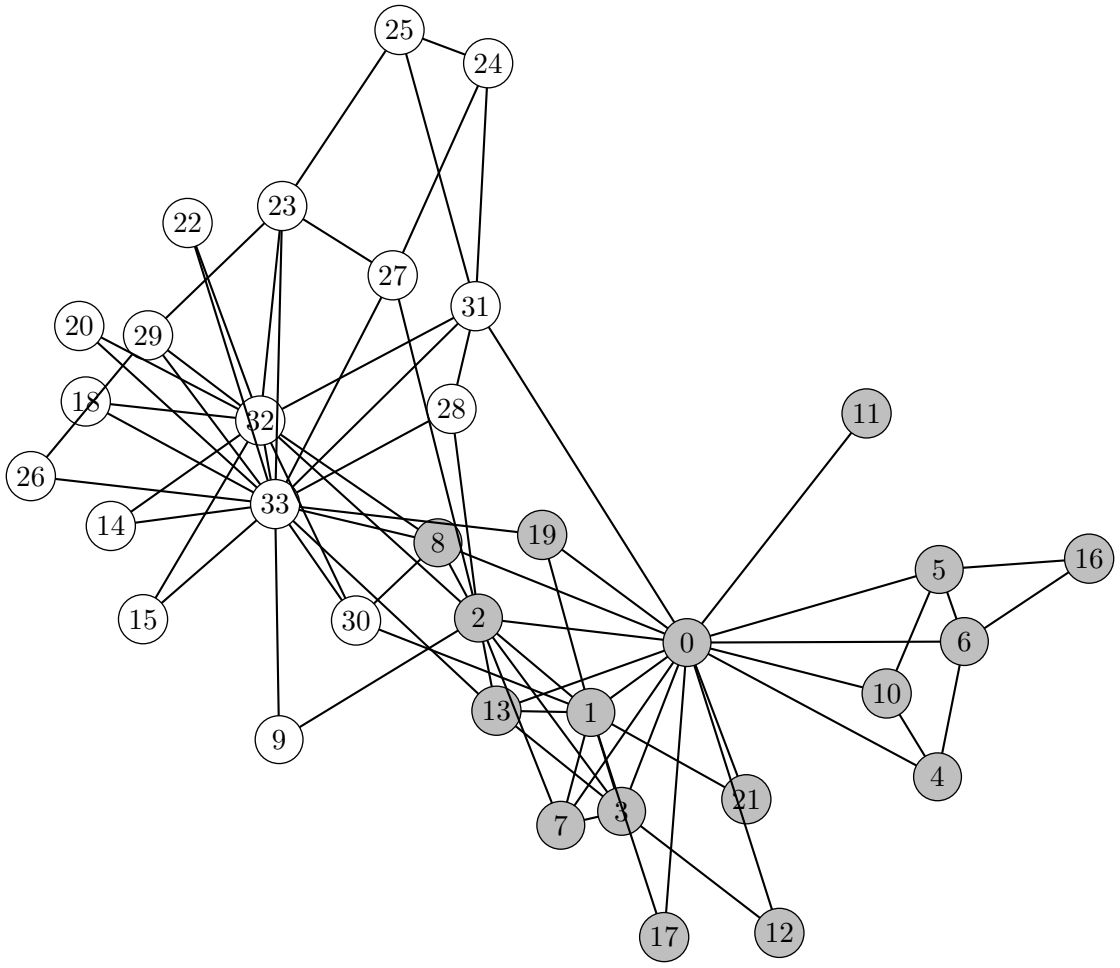


Figure 2.5: The Karate Club, split

For the most part, the split can be determined in the following way—*karateka* i went with vertex 0 or 33, according to whoever they were closer to. However, this does not provide a complete explanation. For instance, 19 is friends with both 0 and 33, but chose to stay with 0. Why? Well, a likely explanation is that 19 is also friends with 1, who is friends with 0 but not 33. When faced with a choice, people often do what their friends do. This reasoning does not explain all choices made—e.g., 8 went with 0 despite having more friends who went with 33. We don't capture all relevant information in this network (e.g., strength of connections and any personal details are not measured), but the point is we can mathematically guess how the split would go almost exactly even before it happened by just looking at the network structure.

Here is one way of mathematically formulating the idea that friends tend to go with friends' choices. We partition the graph into two components, one containing 0 and one containing 33, in such a way that we minimize the number of triangles (cliques of order 3) broken. For example, if 19 went with 33, we would loose the clique $\{0, 1, 19\}$, but if 19 goes with 0, no clique of order 3 is destroyed.

How can we identify cohesive subgraphs? While cliques are certainly cohesive, this measure is too crude to identify the two components the karate club split into. One strategy is to use the

notion of *clustering*. Visually this is the notion of how many triangles are in the graph and how much they bunch together. There are different ways we can measure clustering.

Definition 2.1.4. *Let $G = (V, E)$ be a simple undirected graph. The* **(individual) clustering coefficient** *of a node $u \in V$ is the probability $Cl(u)$ that two neighbors of u are themselves neighbors, i.e.,*

$$\begin{aligned} Cl(u) &= \frac{|\{(v,w) : v, w \in V; v \neq w; (u,v), (u,w), (v,w) \in E\}|}{|\{(v,w) : v, w \in V; v \neq w; (u,v), (u,w) \in E\}|} \\ &= \frac{|\{(v,w) : v, w \in V, v \neq w; (u,v), (u,w), (v,w) \in E\}|}{\deg(u)(\deg(u)-1)}. \end{aligned}$$

The **average clustering coefficient** *is the average of $Cl(u)$ over all $u \in V$. The* **overall clustering coefficient** *or* **transitivity** *of G is*

$$Cl(G) = \frac{\sum_{u \in V} |\{(v,w) : v, w \in V; v \neq w; (u,v), (u,w), (v,w) \in E\}|}{\sum_{u \in V} |\{(v,w) : v, w \in V; v \neq w; (u,v), (u,w) \in E\}|}.$$

The meaning of the individual and average clustering coefficients are straightforward—e.g., for friendship networks this measures how many of your friends are friends with each other. We'll explain transitivity momentarily.

First let's look at some calculations for the karate club graph `K`.

Sage 6.1

```
sage: K.clustering_coeff([0,33])
{0: 0.15, 33: 0.11029411764705882}
sage: K.cluster_triangles([0, 33])
{0: 18, 33: 15}
sage: K.clustering_average()
0.5706384782076823
sage: K.cluster_transitivity()
0.2556818181818182
```

The function `clustering_coeff` returns the individual clustering coefficients for the list of nodes specified (or all nodes by default). The function `cluster_triangles` returns the number of triangles involving node u for each u in the specified list (or all nodes by default). The `clustering_average()` and `cluster_transitivity()` functions return the average and overall clustering coefficients for the graph.

Note that even though the vertices 0 and 33 are involved in more triangles than any other vertices, their individual clustering coefficients are much lower than the average clustering coefficient. This is actually to be expected with hubs, because they are friends with so many different people. On the other hand, vertices like 18 or 20, who are only friends with 32 and 33 (who are friends), have clustering coefficients 1. When we take an average of individual clustering coefficients, the clustering coefficients for all vertices are weighted the same.

If instead we want to put more weight on vertices with higher degree, we can see this in the overall clustering coefficient. We call a "potential triangle" in a graph a *triad*, i.e., a triad is a set of 3 vertices in the graph with at least 2 edges among them. Then the overall clustering, or transitivity, as defined above is simply the number of triangles in the graph divided by the number of triads.

Both the average and overall clustering coefficients give some sense of how cohesive the graph is. Let's examine these clustering measures for the 2 graphs `K1` and `K2` of how the karate club split. Here `K1` is the subgraph of `K` consisting of people that went with 0, and `K2` is the subgraph of `K` consisting of people that went with 33.

Sage 6.1

```
sage: K1.cluster_transitivity()
0.39195979899497485
sage: K1.clustering_average()
0.7215686274509804
sage: K2.cluster_transitivity()
0.25139664804469275
sage: K2.clustering_average()
0.631279178338002
```

In both cases we see there is a sizable jump in the average clustering coefficients over the original graph `K`. There is also a significant jump in overall clustering for `K1`, but almost no change (in fact a slight drop) for `K2`.

How does this compare with clustering coefficients for random subsets of `K`? I generated a couple of random subsets of vertices of size 17, and for both of these the overall clustering was not too far from 0.25, but the average clustering dropped to about 0.43.

This suggests that clustering coefficients can provide useful measures of how cohesive certain groups in a network are. Clustering will also give us some insight into how a network forms. A lot of clustering in, say, a friendship network suggests that many friendships were formed through mutual friends. On the other hand, in networks with relatively low clustering suggests that most connections were formed not through mutual connections, but other methods, such as chance encounters or strategic choices (e.g., in the Florentine families network, it is likely that many of the connections (marriages) were strategically formed for mutual gain).

Exercises

Exercise 2.1.1. *Fix $k \in \mathbb{N}$. Show there is no simple undirected graph G with exactly 1 vertex of each degree $1, 2, \ldots, k$ and no vertices of higher degree. (Hint: what can you say about the number of vertices of degree > 0?)*

Exercise 2.1.2. *Is there a simple undirected graph G with exactly 2 vertices of each degree $1, 2, 3, 4$ and no vertices of higher degree?*

Exercise 2.1.3. *Write a Sage function* `degree_distribution(G)` *that takes in Sage graph* `G`*, and returns the degree distribution. This should be returned as a list, just like* `degree_histogram()`*, where the d-th entry is the proportion of nodes of degree d (starting with $d = 0$. Test this on the Florentine families and karate club graphs. (You define functions in Sage just as in Python.)*

Exercise 2.1.4. *Write a Sage function* `loglogplot(dd)` *that takes in a degree distribution* `dd` *and produces a log-log plot. (Use a line plot—see the Sage documentation. The function* `log` *is built into Sage. You may also want to use the* `float` *function which converts numbers into floating point numbers.) Test this on the karate club graph and compare with Figure 2.4.*

Exercise 2.1.5. *Write a Sage function* `decay_centrality(G,u,delta)` *which returns the decay centrality for vertex* `u` *with parameter* `delta`*.*

Exercise 2.1.6. *Write a Sage function* `clique_count(G,m)` *that returns the total number of cliques of order* m *in* `G`*. Note you cannot read this information directly off the* `cliques_maximal()` *result as this only returns maximal cliques. Test your function on the Florentine families and karate club graphs.*

Exercise 2.1.7. *Design an algorithm to predict which nodes went with vertex 0 and which went with vertex 33 in the karate club split. Explain your algorithm, code it up in Sage, and compare your results to data in Figure 2.5.*

2.2 Social Network Models

To properly study social networks, it is important to have a notion of *social network models*. The basic idea is that we have some procedure, usually involving an element of randomness, that generates graphs. (We've seen one example with random tree generation in Exercise 1.6.7.) Typically there are some adjustable parameters in this procedure. If, for some choice of parameters, this procedure generates graphs which typically look very similar to (have the same landmarks as) a given social network G_0, this can give us a lot of insight into how the social network G_0 has formed, and how it will evolve over time.

In addition, models will allow us to formalize intuitive notions about networks and prove theorems about them, which will give us baseline expectations for our social networks. For instance, we'll be able to say what it precisely means to have a giant component and show that certain types of networks almost always have giant components. On the other hand, when a network looks different from our expectations, this will tell us that there is something interesting going on about how this social network forms.

2.2.1 Probability for n00bs*

Because we want to talk about randomness, we first need to establish some basic notions from probability.

Definition 2.2.1. *A* **(discrete) probability space** (S, P) *is a set* S *with a function* $P : S \to [0, 1]$ *such that*

$$\sum_{s \in S} P(s) = 1.$$

The function P *is called the* **(discrete) probability function***, or* **(discrete) probability distribution***.*

It is convenient to extend the definition of the probability function P to subsets $A \subset S$ by

$$P(A) = \sum_{s \in A} P(s).$$

Then it is clear that P satisfies the properties

$$0 \leq P(A) \leq 1, \quad A \subset S$$

*Certainly not for dummies, but smart people who haven't seen or don't remember this stuff.

(with $P(\emptyset) = 0$ and $P(S) = 1$) and

$$P(A \cup B) = P(A) + P(B), \quad A, B \subset S,\ A \cap B = \emptyset.$$

We will call subsets $A \subset S$ **events** in the probability space.

Once P is understood, we often just refer to S as the probability space. We can think of this probability space as follows. We have some infinitely repeatable experiment (under identical conditions), and the elements $s \in S$ represent the different possible mutually-exclusive outcomes of this experiment (exactly one outcome s in the space S occurs after each trial of the experiment). The probability $P(s)$ represents the fraction of the time we get outcome s. Similarly, the probability $P(A)$ represents the fraction of the time we get an outcome $s \in A$.

Example 2.2.2. *We can model a fair coin flip with a probability space $S = \{H, T\}$, where H represents heads and T tails, where the probability function is defined by $P(H) = P(T) = 1/2$.*

Example 2.2.3. *We can model a fair die roll with a probability space $S = \{1, 2, 3, 4, 5, 6\}$, where $P(s) = 1/6$ for all $s \in S$. Then the probability of rolling an even number is $P(\{2, 4, 6\}) = 1/6 + 1/6 + 1/6 = 1/2$. Here the set $A = \{2, 4, 6\}$ represents the event of an even die roll.*

This gives a little indication as to why it is convenient to define probabilities of subsets, rather than just elements, of S. So does the next example.

Example 2.2.4. *Consider an urn with n balls, m of which are red, and $n - m$ are green. You draw one at random (the balls are mixed so that each ball is drawn with equal probability). We can model this with a probability space as follows. Think of the balls as being numbered from 1 to n, and for convenience assume that balls 1 through m are red, and the remaining balls are green. Formally, take $S = \{1, 2, \ldots, n\}$ and $P(s) = \frac{1}{n}$ for each n. Then $A = \{1, 2, \ldots m\}$ represents the event of drawing a red ball and $B = \{m + 1, m + 2, \ldots n\}$ as represents the event of drawing a green ball. Hence the probability that you pick a red ball is $P(A) = P(1) + P(2) + \cdots + P(m) = \frac{m}{n}$, (unless you are red-green color blind, in which case the probability is 1).*

Alternatively, we can model this experiment with the probability space (S', P') where $S' = \{R, G\}$ and $P'(R) = m/n$ and $P'(G) = (n - m)/m$. However the first model is more convenient if one want to allow things like drawing k balls (without replacement) and counting how many are red or green.

Example 2.2.5. *Let $G = (V, E)$ be an undirected graph. Then (V, P) is a probability space where P is the degree distribution.*

All of the examples of probability spaces we just have have been finite spaces, but we do not need to restrict ourself to finite spaces. In fact, we've seen a non-finite discrete probability space before.

Example 2.2.6. *Fix $\alpha > 1$. Let $S = \mathbb{N} = \{1, 2, 3, \ldots\}$ and $P(s) = \frac{1}{\zeta(s)s^\alpha}$. Then (S, P) is a probability space, where P is the power law distribution described in Section 2.1.2.*

The other major type of probability space is a *continuous probability space*. (One can also have mixtures of continuous and discrete probability spaces.) For instance if $S \subset \mathbb{R}$ is an interval, then the distribution function will be an integrable function $f : S \to [0, \infty)$ such that $\int_S f(x)\,dx = 1$. Events will be subsets $A \subset S$ and the probability of an event is given by $P(A) = \int_A f(x)\,dx$.[†]

[†]Technically, A should be what is called a *measurable* subset of S (think of a union of intervals) since there exist strange sets that don't make sense to integrate over. This is also why I assumed S was an interval.

To be a little more concrete, think of $S = [0, 1]$, and our random process is picking a number between 0 and 1. If we model this with any continuous distribution f, then if $A = \{a\}$ is a single point, we see $P(A) = \int_a^a f(x)\,dx = 0$. In other words, the probability of picking any specific number must be 0, in contrast to the case of discrete distributions. This of course doesn't mean you never get any specific number—you always get some specific number, but you can think of this as follows: even if you pick a (countably) number of random numbers between 0 and 1, according to this distribution, you will never pick the same number twice.

Rather for continuous distributions, it is only meaningful to discuss the probability that our random number lies in a given range. The simplest possible example is if we take $f(x)$ to be the constant function $f(x) = 1$—this is called the **uniform distribution** (on $[0, 1]$). For the uniform distribution (on $[0, 1]$), we have $P([a, b]) = P((a, b)) = \int_a^b dx = b - a$, i.e., the probability our random number lies in any given interval is simply the length of that interval (and it doesn't matter whether we include endpoints or not, since the probability of an endpoint value is 0).

Python (and by extension Sage) has a built in (pseudo)random number generator called `random()`, which returns a random number between 0 and 1.

Python 2.7
```
>>> from random import random   # you don't need this line in Sage
>>> random()
0.22445061772144392
>>> random()
0.103016249389979
>>> random()
0.90772991525930202
```

Technically this is a pseudorandom number generator, not a random number generator, because the method is deterministic, i.e., not truly random. However, it is close enough to random for our purposes, in the sense that each time `random()` is called, the probability the result x lies in a specific range $0 \le a \le x \le b \le 1$ is effectively $b - a$. In particular, if we want to do something with probability p, say add 1 to some variable a with $p = 0.5$, we can write this in Python/Sage as follows:

Python 2.7
```
>>> a = 0
>>> p = 0.5
>>> if random() < p:
...     a = a + 1
...
>>> a
0
>>> if random() < p:
...     a = a + 1
...
>>> a
1
```

This does what we want, because the probability that `random() < p`, i.e., that `random()` lies in $[0, p]$ is $p - 0 = p$.

There is one other basic concept we need from probability, and that is the notion of *independent events*. To understand this concept, consider the following example. Suppose we have an urn with

3 red balls and 5 green balls, and draw 2 balls in sequence (without replacing the first one). There are two events that we are interested in—the color of the first ball and the color of the second ball. The probability that the first ball is red is clearly 3/8. What about the probability the second ball is red? It depends on whether the first ball was red or green—it is 2/7 if the first ball was red, and 3/7 if the first ball was green. Hence these events are not independent. Though it is still possible to calculate the absolute probability the second ball is red:

$$\frac{3}{8} \cdot \frac{2}{7} + \left(1 - \frac{3}{8}\right) \frac{3}{7} = \frac{3}{8}.$$

(And indeed, it makes sense that the probability the second ball is red is the same as the probability the first one is red, provided we don't know anything about what color the first ball is.) However, these events would be independent if we replaced the first ball before drawing the second ball, in the sense that then the probability the second ball is red would not depend upon whether the first ball is red or not.

To give the formal definition of this sense of independence requires defining *conditional probabilities*. This is not hard, but we will not do it as there is is an alternative, equivalent way to state the notion of independence of two events, and this alternative formulation will actually be more useful for us. Namely, if two events are independent, then the probability of both of them happening is equal to the product of the probabilities of each of them happening separately. For instance, in our urn example, if we replace the first ball we draw before drawing the second one, then the probability that both balls are green is just the probability that the first is green times the probability that the second is green, i.e., $\frac{3}{8} \cdot 38$. Here is the formal definition.

Definition 2.2.7. *Let $A, B \subset S$ be two events in a probability space (S, P). Then A and B are* **independent** *if $P(A \cap B) = P(A)P(B)$.*

Note $A \cap B$ represents the event that both A and B happen. (Similarly, $A \cup B$ represents the even that either A or B, or both, happen.)

Example 2.2.8. *Consider our fair die roll example: $S = \{1, 2, 3, 4, 5, 6\}$ and $P(s) = 1/6$ for all $s \in S$. Let $A = \{2, 4, 6\}$ be the event of an even die roll, and $B = \{1, 2\}$ be the event of rolling ≤ 2. Then*

$$P(A \cap B) = P(2) = \frac{1}{6} = \frac{1}{2} \cdot \frac{1}{3} = P(A)P(B).$$

Hence A and B are independent events.

We remark that it is the property of independence that makes most random number generators on computers fail to be truly random—when we call our random number generator many times, the results should look pretty close to the uniform distribution, but the results are not actually independent. For instance, the simplest kind of random number generators repeat, although maybe not until you run them 4 billion times. This is an issue for some applications (e.g., cryptography) where (close to) true randomness is important, but it does not really for simple modeling/simulation like what we will do.

2.2.2 Erdös–Rényi Model

Notions of random graphs were introduced by Paul Erdös and Alfréd Rényi, and independently by Edgar Gilbert, in 1959. Two models were proposed, and these both now go by the name of the **Erdös–Rényi model.**

Here is the first model.

Definition 2.2.9. *Let $n \geq 1$ and $0 \leq M \leq n(n-1)/2$. The $G(n, M)$* **model for random graphs** *is the probability space (S, P), where S denotes the set of simple undirected graphs G on $\{1, 2, \ldots, n\}$ with M edges (or equivalently, the set of all subgraphs G of K_n with n nodes and M edges) and each graph $G \in S$ is assigned equal probability $P(G) = 1/|S|$.*

Let's explain how to compute $P(G)$ in terms of n and M.

Recall the following elementary fact from discrete mathematics: the *binomial coefficient* $\binom{n}{m} = \frac{n!}{m!(n-m)!}$, often read "$m$ choose n", is the number of ways to choose m distinct objects out of n total (distinct) objects. For instance, the maximum number of possible edges in a (simple undirected) graph on n nodes is $\binom{n}{2} = \frac{n(n-1)}{2}$, which is the number of way to choose 2 out of n vertices. Similarly, the maximum number of possible triangles in a graph is $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$. Note $\binom{n}{m} = \binom{n}{n-m}$—choosing m objects to include in something is the same as choosing $n - m$ objects to exclude from something.

Thus the number of graphs in S in the $G(n, M)$ model is simply the number of edges in K_n choose M, i.e., $\binom{n(n-1)/2}{M}$, as we just need to choose which M edges to include in our graph. Hence

$$P(G) = \frac{1}{\binom{n(n-1)/2}{M}}, \quad G \in S.$$

It is not hard to implement an algorithm to generate $G(n, M)$ graphs in Python (they are already implemented in Sage). Here is pseudocode

Pseudocode

```
set V = { 1, 2, ..., n }
set Eall = []
for i from 1 to n:
    for j from i+1 n:
         append {i, j} to Eall
set E = M randomly chosen edges from Eall
return G = (V, E)
```

That is, first we generate all possible undirected edges `Eall`, and then we randomly choose M of them to include in our graph. This is the only tricky part, but there is a command `shuffle` in the Python `random` module which makes this easy. For example, here is an example of how we can randomly choose 5 elements from a list.

Python 2.7

```
>>> from random import shuffle
>>> l = range(20)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> shuffle(l)
>>> l
```

```
[6, 4, 9, 3, 18, 13, 12, 16, 19, 11, 17, 10, 0, 15, 7, 1, 2, 8, 5, 14]
>>> l[:5]
[6, 4, 9, 3, 18]
```

Now we'll present the second Erdös–Rényi random graph model.

Consider the following random procedure for making a graph $G = (V, E)$. Fix a probability $0 \leq p \leq 1$.

Pseudocode

```
set V = { 1, 2, ..., n }
set E = { }
for i from 1 to n:
    for j from i+1 to n:
        with probability p, add {i, j} to E
return G = (V, E)
```

In other words, we start with n vertices, and for each (unordered) pair of vertices $\{i, j\}$, we include the edge $\{i, j\}$ in our graph with probability p. (Recall, we saw how to do something with probability p in Python/Sage in Section 2.2.1.) Note in the pseudocode above we take $i + 1 \leq j \leq n$ to ensure that we loop through each unordered pair of distinct vertices $\{i, j\}$ exactly once.

This is the random process for the $G(n, p)$ model. Note the $G(n, p)$ model models a random generation of graphs on n nodes where the probability of link formation is p, independent of the choice of the link. To properly analyze the $G(n, p)$ model, we need to be able to calculate the probability of getting a given graph G of order n.

Let G be a random $G(n, p)$ graph. For $1 \leq i < j \leq n$, let $A_{i,j}$ denote the event that the edge $\{i, j\}$ is included G, and $B_{i,j}$ the event that the edge $\{i, j\}$ is not included in G. Say G has m edges: $\{i_1, j_1\}, \ldots, \{i_m, j_m\}$. Let $\{i_{m+1}, j_{m+1}\}, \ldots, \{i_N, j_N\}$ denote the remaining pairs of non-edges, where $N = n(n-1)/2$. Since the $A_{i,j}$'s and $B_{i',j'}$'s are all pairwise independent events, we see

$$P(A_{i_1,j_1} \cap A_{i_2,j_2} \cap \cdots \cap A_{i_m,j_m}) \cap B_{i_{m+1},j_{m+1}} \cap \cdots \cap B_{i_N,j_N}) =$$
$$P(A_{i_1,j_1})P(A_{i_2,j_2}) \cdots P(A_{i_m,j_m})P(B_{i_{m+1},j_{m+1}}) \cdots P(B_{i_N,j_N}) = p^m(1-p)^{N-m}.$$

Because this list of edges and non-edges determines G, we get that $P(G) = p^m(1-p)^{N-m}$, i.e., the probability only depends on how many edges m has.

With this in mind, we can also define of the $G(n, p)$ model in terms of the probability space.

Definition 2.2.10. *Let $n \geq 1$ and $0 \leq p \leq 1$. The $G(n, p)$* **model for random graphs** *is the probability space (S, P) where S is the set of all (simple undirected) graphs on $\{1, 2, \ldots, n\}$, where for $G \in S$ with m edges, we define the probability function to be $P(G) = p^m(1-p)^{n(n-1)/2-m}$.*

Note, if $p = 0.5$, then $P(G) = p^m(1-p)^{n(n-1/2)-m} = (0.5)^m(0.5)^{n(n-1)/2-m} = 0.5^{n(n-1)/2}$, independent of m.

Example 2.2.11. *Let $n = 3$ and $p = 0.5$. On $V = \{1, 2, 3\}$, there is $\binom{3}{0} = 1$ graph with 0 edges, $\binom{3}{1} = 3$ graphs with 1 edge, $\binom{3}{2} = 3$ graphs with 2 edges and $\binom{3}{3} = 1$ graph with 3 edges. Let*

A_m denote the subset of graphs on V with m edges. Hence, in the $G(n,p) = G(3, 0.5)$ model.

$$P(A_0) = P(0 \text{ edges}) = 1 \cdot (0.5)^3 = 0.125$$
$$P(A_1) = P(1 \text{ edge}) = 3 \cdot (0.5)^3 = 0.375$$
$$P(A_2) = P(2 \text{ edges}) = 3 \cdot (0.5)^3 = 0.375$$
$$P(A_3) = P(3 \text{ edges}) = 1 \cdot (0.5)^3 = 0.125.$$

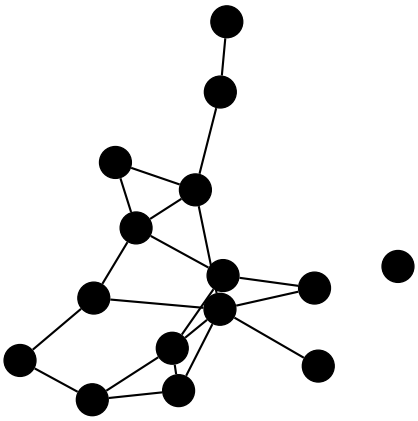
The $G(n, M)$ and $G(n, p)$ models are clearly different (the $G(n, p)$ model can result in any possible number of edges), but in many ways they behave rather similarly, particularly for large n. These are both models where edge formations are independent—we can view the $G(n, M)$ models as randomly distributing M edges to the $n(n-1)/2$ unordered pairs of distinct vertices, with each pair getting equal consideration. Also note that in either model, all graphs with a fixed number of edges m are equiprobable—it's just that in the $G(n, M)$ model this probability is 0 unless $m = M$. However, the $G(n, p)$ model is more commonly studied. Indeed, if people talk about random graphs without any further qualification, they probably have in mind the $G(n, p)$ model.

Let's look at our two favorite social networks, and see how well these can be modeled by random graphs. Let's first try this with the $G(n, M)$ model, since there's less work for us to do here. Denote the Florentine families network by F, and the karate club graph by K.

Start with F. Since F has 15 nodes and 20 edges, we can generate a random $G(n, M) = G(15, 20)$ graph in Sage as follows

Sage 6.1
```
sage: G = graphs.RandomGNM(15,20)
sage: G.show()
```

Then maybe you'll get something that looks like this.



This particular example isn't connected, but almost is (only 1 isolated node), and often this $G(n, M)$ random graph will be (we'll be more precise below). If we compare some landmarks of this graph G with F, they seem reasonably similar:

Sage 6.1
```
sage: F.degree_histogram()
[0, 4, 2, 6, 2, 0, 1]
sage: G.degree_histogram()
[1, 2, 4, 4, 3, 0, 1]
sage: F.cliques_maximum()
```

```
[[0, 7, 8], [0, 8, 11], [1, 2, 13]]
sage: G.cliques_maximum()
[[1, 2, 14], [1, 5, 14], [6, 8, 11]]
sage: F.cluster_transitivity()
0.19148936170212766
sage: G.cluster_transitivity()
0.1836734693877551
```

They both have 3 cliques of size 3, very close overall clustering, and fairly close degree distributions.

To be a bit more scientific, I wrote a function `testGNM(n,M)` which test some statistics about graphs in the $G(n, M)$ model. Namely, it generates 1000 random $G(n, M)$ graphs and counts the number of times they are connected and the averages of the following quantities: maximum degree, average distance (among connected graphs), average diameter (among connected graphs), max closeness centrality, max betweenness centrality, clique number and transitivity. Here is the result.

Sage 6.1

```
sage: testGNM(15,20)
Connected:   476 / 1000 times
Average max degree:   5.249
Average distance:   2.5818927571
Average diameter:   5.4243697479
Closeness centrality:    0.530094239059
Betweenness centrality:    0.353480946831
Average clique number:   2.985
Average transitivity:   0.173241028715
```

To compare, F has maximum degree 6, average distance 2.4857..., diameter 5, clique number 3, max closeness centrality 0.56, max betweenness centrality 0.52, and transitivity 0.1914... All in all, not so far off, except perhaps for betweenness centrality, so the $G(n, M)$ models many features of F rather accurately.

What would it mean if the $G(n, M)$ model is a good model for how the network F formed? It might me that rather than the de Medici family having blessed foresight in forging connections to let them rise to the top, maybe they just happened to be the lucky node that was most central. If one generates a few more examples of random graphs, then we'll see the above degree distribution sometimes looks like our previous example, but not most of the time. For another randomly chosen G, one gets the degree histogram `[0, 2, 4, 6, 3]`, where there are many vertices with relatively high degree centrality. Further, from doing a few trials, and looking at betweenness centrality, it seems quite unusual for one node to have betweenness centrality much higher than all other nodes, as the de Medicis do in F. This supports the hypothesis that indeed the de Medicis (as well as other families) did not choose their connections at random, but selected them strategically (and wisely in the case of the de Medicis).

Now let's consider K, which has 34 nodes and 78 edges. Again, we can run our program to test some statistics on a $G(n, M) = G(34, 78)$ model.

Sage 6.1

```
sage: testGNM(34,78)
Connected:   796 / 1000 times
Average max degree:   8.979
Average distance:   2.41009190337
Average diameter:   4.78140703518
```

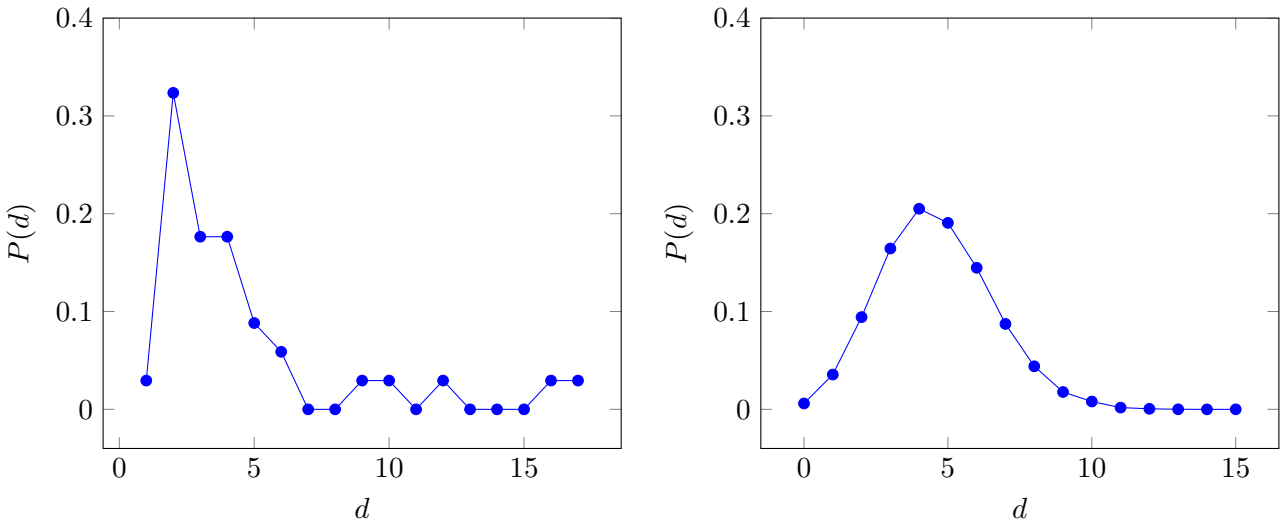


Figure 2.6: Degree distibution for K and average degree distribution for 1000 $G(34, 78)$ graphs

```
Closeness centrality:    0.520352742539
Betweenness centrality:    0.152535379873
Average clique number:   3.24
Average transitivity:   0.136935568331
```

Compare this to K, which has max degree 17, average distance 2.408..., diameter 5, max closeness centrality 0.5689..., max betweenness centrality 0.4376..., clique number 5 and transitivity 0.2556... Some of the statistics are close, but a few are far off: max degree, betweenness centrality, clique number and transitivity.

In particular, the degree distribution in the $G(n, M)$ model is way off from K. To see this clearly, I computed the average degree distribution for 1000 $G(34, 78)$ graphs and plotted this next to the degree distribution for K in Figure 2.6. If $G(n, M)$ is a good model for K (that is, if K looks similar to an "average" graph from $G(n, M)$ rather than a low probability one), these graphs should have pretty close to the same shape, but the $G(n, M)$ degree distribution has a much gentler, rounder slope, with peak around degree 4, rather than degree 2.

Unlike the Florentine families graph, where at least sometimes the degree distributions of our random graphs matched pretty closely, they never match closely for the Karate club graph. In fact, the max degee ever achieved in this trial of the $G(34, 78)$ model was 13, which only occured twice (the max for K is 17). Part of this may be attributed to the size of the graphs—the Florentine families graph is much smaller, so there is less possible variation for networks of this size. However, part of it is undoubtedly due to the difference in the nature of the links in F and K—the links in F are marriages, likely largely chosen strategically, whereas the links in K are friendships, where one expect things like clustering and other small world phenomena.

Thus we can safely conclude that the friendships formed in K cannot be explained (at least not entirely) by random, independent formation (independent chance encounters). We can't explain the large hubs (one degree 16 node and one degree 17 node) by random formation—they are distinguished individuals in the clubs, the instructor and student founder. Moreover, the amount of sizable cliques and clustering/transitivity in K is not present in a random $G(n, M)$ graph.

What about the $G(n,p)$ model? We will get very similar results as with the $G(n,M)$ model, but let us explain how we can compare a given graph with the $G(n,p)$ model. The difference here is that while it was easy to see what M should be in the $G(n,M)$ model, it is not immediately obvious how to chose the appropriate probability p for edge formation.

Proposition 2.2.12. *The average, or expected, number of edges in a graph in the $G(n,p)$ model is*

$$\sum_{m=0}^{n(n-1)/2} m \binom{n(n-1)/2}{m} p^m (1-p)^{n(n-1/2)-m}.$$

Proof. To average a random quantity over a discrete probability space, instead of just summing up all values and dividing by the number of entries, we sum up all values weighted by the probability of obtaining that value. So the expected number of edges in a graph in the $G(n,p)$ model is

$$\sum_{m=0}^{n(n-1)/2} P(m \text{ edges}) \cdot m.$$

Now the probability of m edges is simply $p^m(1-p)^{n(n-1/2)-m}$ times the number of graphs with m edges, which we saw above is $\binom{n(n-1)/2}{m}$. □

Using this proposition, we can with trial and error, find a value of p that gives us the desired expected number of edges. For example, when $n = 15$, $p = 0.19$ gives an expected 19.95 edges, so we can try to model F with $G(n,p) = G(15, 0.19)$. Similarly, with $n = 34$, $p = 0.139$ gives an expected 77.979 edges, so this is reasonable choice to try to model K. I wrote a Sage function `testGNP` which is the analogue of `testGNM` for the $G(n,p)$ model, and you can see the results below are very similar to those for $G(n,M)$.

Sage 6.1

```
sage: testGNP(15,0.19)
Connected:   426 / 1000 times
Average max degree:   5.252
Average distance:   2.45504135927
Average diameter:   5.10328638498
Closeness centrality:    0.516331967533
Betweenness centrality:    0.340416128774
Average clique number:   2.944
Average transitivity:   0.171173755959

sage: testGNP(34,0.139)
Connected:   793 / 1000 times
Average max degree:   9.043
Average distance:   2.40368150011
Average diameter:   4.75283732661
Closeness centrality:    0.520815091198
Betweenness centrality:    0.154727157339
Average clique number:   3.255
Average transitivity:   0.136305968809
```

2.2.3 Preferential Attachment Models

We saw in the last section that the Erdös–Rényi model is not suitable for modeling some friendship networks, such as the karate club graph. There are a couple of issues here. One, there is not enough clustering in this model of random graphs. Two, we don't see nodes of high degree.

Let me briefly mention a couple of other simple models to deal with these specific issues.

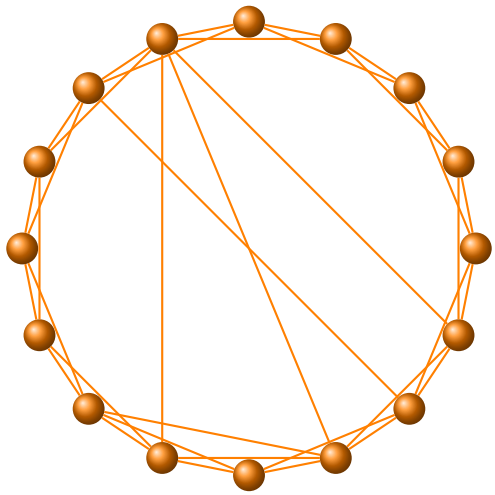
The **Watts–Strogatz model** is one model to generate random graphs with a specified average degree and large clustering. Here we fix a number of nodes n, the average degree κ, and a probability $0 \leq \beta \leq 1$. The algorithm is as follows Arrange the n nodes in a circle, and initially connect each node u to the κ nodes closest to u on the circle (say $\lfloor \kappa/2 \rfloor$ nodes on the left and $\lceil \kappa/2 \rceil$ nodes on the right). So for $\kappa = 2$, this is just a cycle graph at this stage. (In general, it is a kind of generalization of cycle graphs called *circulant graphs*.) Now, for each edge (u, v) in the graph, with probability β replace it with an edge (u, v') where v' is chosen at random among all other vertices.

For whatever reason, Sage has not implemented the Watts–Strogatz model, but the closely related Newman–Watts–Strogatz model, which is just like the Watts–Strogatz model except that instead of randomly replacing existing edges, it simply adds new ones. The can be accessed via the function `graphs.RandomNewmanWattsStrogatz(n,kappa,beta)` in Sage as follows.

Sage 6.1

```
sage: n = 16
sage: G = graphs.RandomNewmanWattsStrogatz(n,4,0.1)
sage: pos_dict = {}
sage: for i in range(n):
....:     x = float(cos(pi/2 + ((2*pi)/n)*i))
....:     y = float(sin(pi/2 + ((2*pi)/n)*i))
....:     pos_dict[i] = [x,y]
....:
sage: p = G.graphplot(pos = pos_dict)
sage: p.show()
```

While we could've just done `G.show()` after the second line, Sage does not naturally arrange these vertices in a circle, so we did a little more work to plot it like this. This code should give a graph that looks something like this.



Note that in the Watts–Strogatz model, we will get precisely $n\kappa$ edges. If $\beta = 0$, we get a κ-regular graph with n cliques of order $\lfloor \frac{\kappa}{2} \rfloor$. On the other hand, as $\beta \to 1$, the Watts–Strogatz random

graphs get closer to the random graphs in a $G(n, M) = G(n, n\kappa)$ model. Hence this model gives us a hybrid of regular graphs with a lot of clustering and Erdös–Rényi random graphs. Consequently, we will almost never get nodes of high degree like in the karate club graph. Another issue is that the Watts–Strogatz model will almost always have a cycle of length n, unless β is quite high, and this is not typically expected in social networks (e.g., look at the Florentine families or karate club graphs).

A completely different idea for generating random graphs is **preferential attachment**, that is, nodes tend to form links with nodes that have already high degree. The idea comes from consideration of citation networks. Here we consider graph the nodes represent certain publications—e.g., all mathematics publications as indexed by the American Mathematical Society (MathSciNet)—and we draw a directed edge from paper A to paper B if paper A cites paper B. This is similar in spirit to webpage hyperlink graphs.

Note that both citation networks and hyperlink graphs are dynamic in nature—new publications and new webpages are always being born. Right away, this is completely different from the Erdös–Rényi and Watts–Strogatz models, which are by nature static, in that the number of nodes do not change in the generation process. (However, we could generate graphs in the $G(n, p)$ model by adding nodes at each stage.)

Preferential attachment is based on the following idea. When I do research, how do I find what is already known about a subject? I look at other research papers or books, or go to conferences, or talk to experts. If another paper in this area is highly cited, I am likely to come across it and/or find it relevant, and likely to cite this also. Whereas, if a paper is not cited much, I am less likely to both come across this paper and find it relevant. If we're a bit more precise about this, we'll see this heuristic predicts that the probability I cite another paper is proportional to the number of citations that paper already has, i.e., that paper's in degree in the citation network. The exact same heuristic makes sense for hyperlink graphs. Furthermore, it is not hard to see this heuristic predicts a scale-free (power law) degree distribution.

One well-known preferential attachment model is the **Barabási–Albert model**. The algorithm begins with an initial connected network of m_0 nodes. There are fixed parameters of n (total number of nodes, or papers), and m (the "out degree" or number of citations each new paper will make). At each stage (until there are n nodes), we add a new node to the network and connect it to m existing nodes chosen at random with respect to a "preferential" probability distribution. Here we can use either directed or undirected edges—directed edges makes sense for a citation/hyperlink network, but the Barabási–Albert model just works with undirected edges.

The actual description of the algorithm for this preferential attachment by Barabási–Albert in their papers is a little vague but here is how I interpret it. Suppose we are at a stage with n_0 nodes numbered $1, 2, \ldots, n_0$. Let d_i be the degree of node i. Set

$$p_i = \frac{d_i}{\sum d_i}.$$

Add a new vertex $n_0 + 1$, and do the following m times. Connect $n_0 + 1$ to one of the vertices in $\{1, 2, \ldots, n_0\}$, where vertex i gets selected with probability p_i. This choice can be practically implemented as follows. Choose a random number x in $[0, 1]$. Divide $[0, 1]$ into n_0 intervals, $I_1, \ldots, I_{n_0}$ where interval I has length p_i. For instance, if $n_0 = 3$, we may take the following partition:

$$[0, 1] = [0, p_1) \cup [p_1, p_1 + p_2) \cup [p_1 + p_2, p_1 + p_2 + p_3 = 1] = I_1 \cup I_2 \cup I_3.$$

If the random number x lies in I_i, connect $n_0 + 1$ to p_i.

The ambiguity here, if $m > 1$, is what to do if you're trying to add an edge you've just added, e.g., if you randomly pick the edge $(n_0 + 1, 1)$ twice. One can either allow multi-edges or pick another edge. Allowing multi-edges makes the analysis easier in some ways, but it seems that at least the Sage implementation chooses m distinct edges. (I'm not sure what Barab'asi and Albert originally did.) Picking m distinct edges can be done in one of two ways—(1) if we happen to choose an edge we've just added, we can just randomly choose again until we get a new edge; or (2) before a new edge is selected, we can update the probabilities for connecting to each vertex—that is, compute the probabilities as above, but simply leave out all the vertices we already connected to $n_0 + 1$. For instance, in approach (2), say $n_0 = 3$, $m = 2$ and we just added an edge from the new vertex 4 to vertex 1. Then we set

$$p_1 = 0, \quad p_2 = \frac{d_2}{d_2 + d_3}, \quad p_3 = \frac{d_3}{d_2 + d_3}$$

and now add an edge from vertex 4 to vertex i with probability p_i.

It is convenient to be able to do this without starting with an initial connected network of m_0 nodes, but just start with the parameters n and m. Then one can start with a initial empty graph on m nodes, and connect the next node added to each of these m nodes. Then proceed as above. This appears to be how the algorithm is implemented in Sage.

While the algorithm is given for a fixed n, this model is dynamic as the networks are generated by adding more nodes. Furthermore this graph generation can be done in stages, and the output of each stage can be put in as the "input graph" in the next stage without affecting the probability distribution in the model. For instance if we generate a preferential attachment graph G_{99}, with $(n, m) = (99, 3)$, we can generate a graph G_{100} with $(n', m) = (100, 3)$ in this model by starting by using G_{99} as our initial network of m_0 nodes, and adding 1 node in the above process. Note we cannot do this sort of thing in the Watts–Strogatz model or the $G(n, M)$ model, though we can for the $G(n, p)$ model.

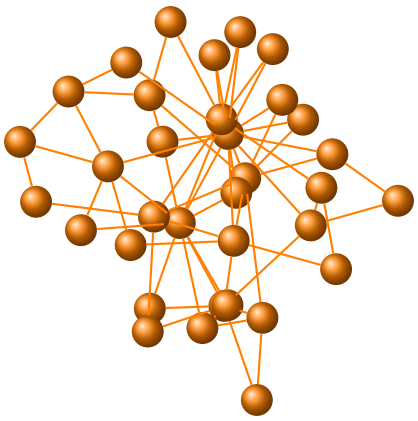
Assuming this implementation, the Barabási–Albert model always has $(n - m)m$ edges. In addition, it will always be connected, hence it is a tree for $m = 1$.

Now if we want to try to model the karate club graph this way, we can take $n = 34$ and either $m = 2$ or $m = 3$. These will give us 64 and 93 edges respectively. Remember the karate club graph as 78 edges.

Here is an attempt with $m = 2$.

Sage 6.1

```
sage: G = graphs.RandomBarabasiAlbert(34,2)
sage: G.degree_histogram()
[0, 0, 17, 5, 4, 1, 2, 2, 1, 0, 1, 0, 0, 0, 1]
sage: len(G.degree_histogram()) - 1
14
sage: G.show()
```



We see this example has maximum degree 14, which is higher (by 1) than what we ever got with 1000 random $G(n, M) = G(34, 78)$ graphs. However, one can get higher maximum degrees in this model. Here are the maximum degrees and degree distributions for 10 such random graphs.

Sage 6.1

```
sage: for i in range(10):
....:     dh = graphs.RandomBarabasiAlbert(34,2).degree_histogram()
....:     print len(dh) - 1, ":  ", dh
....:
12 :   [0, 0, 12, 9, 6, 2, 1, 0, 2, 1, 0, 0, 1]
10 :   [0, 1, 14, 8, 3, 2, 0, 0, 2, 3, 1]
11 :   [0, 0, 14, 7, 4, 1, 4, 2, 0, 1, 0, 1]
13 :   [0, 0, 18, 6, 2, 2, 1, 1, 0, 1, 1, 1, 0, 1]
17 :   [0, 0, 13, 10, 4, 2, 1, 2, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1]
13 :   [0, 0, 16, 6, 4, 1, 4, 1, 0, 0, 0, 0, 0, 2]
17 :   [0, 0, 18, 5, 5, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1]
15 :   [0, 0, 13, 12, 1, 2, 0, 3, 2, 0, 0, 0, 0, 0, 0, 1]
14 :   [0, 0, 16, 4, 5, 3, 3, 0, 1, 1, 0, 0, 0, 0, 1]
10 :   [0, 0, 17, 4, 3, 1, 5, 0, 2, 1, 1]
```

We see the maximum degree varies a lot, and we happened to get a couple instances where the maximum degree is the same as that of the karate club graph, however it seems unlikely that we get one node of degree 16 and one of degree 17. Of course these graphs have fewer edges than our karate club graphs. If instead we look at degree distributions for preferential attachment graphs with $m = 3$, we get something like the following.

Sage 6.1

```
sage: for i in range(10):
    dh = graphs.RandomBarabasiAlbert(34,3).degree_histogram()
    print len(dh) - 1, ":  ", dh
....:
18 :   [0, 0, 0, 9, 9, 7, 3, 0, 2, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1]
16 :   [0, 0, 1, 13, 6, 3, 2, 1, 3, 1, 0, 1, 0, 1, 1, 0, 1]
16 :   [0, 0, 0, 11, 5, 8, 2, 1, 2, 2, 0, 1, 0, 1, 0, 0, 1]
21 :   [0, 0, 1, 15, 5, 2, 3, 1, 0, 2, 1, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1]
18 :   [0, 0, 0, 14, 5, 5, 1, 3, 2, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1]
16 :   [0, 0, 1, 11, 6, 3, 2, 4, 4, 0, 0, 1, 0, 1, 0, 0, 1]
17 :   [0, 0, 0, 10, 11, 2, 3, 1, 0, 1, 4, 1, 0, 0, 0, 0, 0, 1]
12 :   [0, 0, 0, 13, 4, 5, 3, 1, 1, 3, 0, 2, 2]
```

```
17 :   [0, 0, 1, 15, 3, 4, 2, 2, 1, 2, 0, 1, 1, 0, 0, 1, 0, 1]
16 :   [0, 0, 1, 13, 5, 4, 4, 1, 1, 0, 1, 0, 1, 0, 2, 0, 1]
```

Here we see instances where we get a couple of nodes of large degree, though now have too few nodes of degree 2 (the karate club had 11 such nodes, and 1 of degree 1).

This suggests that the Barabási–Albert model is not too bad at modeling the degree distribution for the karate club graph, though ideally we would want to choose something like $m = 2.5$ in this model. While m has to be an integer, we could modify this algorithm to allow fractional m. In this case, $m = 2.5$ would mean that at half the stages in our algorithm we add 2 edges to new vertices, and the other half of the time we add 3 edges.

In fact, analysis of this algorithm shows that the degree distribution follows a power law given roughly by

$$P(d) = \begin{cases} \frac{2m^2}{d^3} & d \geq m \\ 0 & \text{else.} \end{cases}$$

This is not exactly true of course for fixed n—for instance $P(d) = 0$ for $d > n$ and some of the initial m_0 vertices may have degree less than m—but this is the limit of the degree distribution as $n \to \infty$.

While we may view the Barabási–Albert model as a relatively good fit for the degree distribution of the karate club graph, there are other ways in which it is less than ideal. First, we cannot choose appropriate m to get the desired number of edges, but this can be remedied by allowing fractional values of m. Second, whenever $m > 2$ there will never be many nodes of degree $< m$, but nodes of small degree are common in many social networks. Third, graphs in this model tend to have relatively low clustering compared with friendship networks.

Exercises

Exercise 2.2.1. *Consider the process of flipping a fair coin twice. Write down a probability space S to model this process, and say what the probability function P is. For the following pairs of events A and B, write the events A and B explicitly as subsets of S and determine (with proof), whether A and B are independent or not:*

(i) A is getting heads on the first flip, and B is getting heads on the second flip

(ii) A is getting heads on the first flip, and B is getting heads on both flips

(iii) A is getting the same result on both flip, and B is getting different results on each flip

(iv) A is getting heads on the first flip, and B is getting heads on at least one flip

Exercise 2.2.2. *Write a function* `GnM(n,M)` *in Python that returns the adjacency matrix for a random graph in the $G(n, M)$ model.*

Exercise 2.2.3. *Write a function* `Gnp(n,p)` *in Python that returns the adjacency matrix for a random graph in the $G(n, p)$ model.*

Exercise 2.2.4. *Let $n = 4$ and $p = 0.5$. For each $0 \leq m \leq n(n-1)/2$, compute the probability a graph in the $G(n, p)$ model has m edges.*

Exercise 2.2.5. *Write a function* `Prmedge(n,p,m)` *in Sage that returns the probability a graph in the $G(n, p)$ model has m edges. Then for $n = 5$, each $p \in \{0.1, 0.2, 0.3\}$, and each $0 \leq m \leq 10$, compute the probability that a graph in the $G(n, p)$ model has m edges.*

Exercise 2.2.6. *Write a function* `testGnpconn(n,p)` *in Sage that generates 1000 graphs in the* $G(n,p)$ *model and outputs (prints, returns, or both, your choice) the number that are connected. For* $n = 20$*, how large should* p *be (to 2 decimal places) in order for random* $G(n,p)$ *graphs to be connected at least 99% of the time.*

Exercise 2.2.7. *Write a Sage function* `WattsStrogatz(n,kappa,beta)` *that generates and returns a graphs in the Watts–Strogatz model.*

Exercise 2.2.8. *Write a Sage function* `PrefAttach(n,m)` *that generates a Barabási–Albert type graph, but allows* m *to be fractional. Precisely if we write* $m = m' + p$ *where* m' *is an integer and* $0 \leq p < 1$*, at each stage we add* m' *edges to our new node with probability* $1 - p$*, and* $m' + 1$ *nodes with probability* p*.*

Generate 20 such graphs with $n = 34$ *and* $m = 2.5$*, and print out the maximum degrees and degree distributions. Does this seem to match closely with the degree distribution of the karate club graph?*

Exercise 2.2.9. *(i) For Barabási–Albert graphs with* $(n,m) = (34,2)$*, generate 100 random graphs and compute the average of the following quantites: diameter, maximum closeness centrality, maximum betweenness centrality, and transitivity. Compare these with the corresponding values for the karate club graph.*

(ii) Do the same for $(n,m) = (34,3)$*.*

Exercise 2.2.10. *(i) Design your own algorithm for generating random graphs to model friendship network (not necessarily the karate club graph). Explain your reasoning behind your algorithm.*

(ii) Code up your algorithm in Sage. By computing some examples, see how well it models some features of the karate club graph, and compare it with the other models we've studied.

Chapter 3

Spectral graph theory and random walks on graphs

Algebraic graph theory is a major area within graph theory. One of the main themes of algebraic graph theory comes from the following question: what do matrices and linear algebra tell us about graphs? One of the most useful invariants of a matrix to look in linear algebra at are its eigenvalues and eigenspaces. This is also true in graph theory, and this aspect of graph theory is known as *spectral graph theory*.

Given a graph G, the most obvious matrix to look at is its adjacency matrix A, however there are others. Two other common choices are the *Laplacian matrix*, motivated from differential geometry, and what we will call the *transition matrix*, motivated from dynamical systems—specifically random walks on graphs. The Laplacian and transition matrices are closely related (the eigenvalues of one determine the eigenvalues of the other), and which to look at just depends upon one's point of view.

We will first examine some aspects of the adjacency matrix, and then discuss random walks on graphs and transition matrices. On one hand, eigenvalues can be used to measure how good the network flow is and give bounds on the diameter of a graph. This will help us answer some of the questions we raised in the first chapter about communication and transportation networks. On the other hand, eigenvectors will tell us more precise information about the flow of a networks. One application is using eigenvectors to give a different kind of centrality ranking, and we will use this idea when we explain Google PageRank.

Here are some other references on these topics.

- **Algebraic Graph Theory**, by Norman Biggs.
- **Algebraic Graph Theory**, by Chris Godsil and Gordon Royle.
- **Modern Graph Theory**, by Béla Bollobás.
- **Spectra of Graphs**, by Andries Brouwer and Willem Haemers.
- **Spectral Graph Theory**, by Fan Chung.

The first two books are "classical graph theory" books in the sense that they do not discuss random walks on graphs, and cover more than just spectral theory. Bollobás's book covers many

topics, including some spectral theory and random walks on graphs (and random graphs). The latter two books focus on spectral theory. Brouwer–Haemers cover the adjacency and Laplacian spectra but does not really discuss random walks, whereas Chung's book discusses random walks but focuses entirely on the (normalized) Laplacian matrix.

3.1 A review of some linear algebra

Before we get started on applying some linear algebra to graph theory, we will review some standard facts from linear algebra. This is just meant to be a quick reminder of the relevant theory, as well as fixing some notation, but you should refer to a linear algebra text to make sense of what I'm saying.

Let $M_n(\mathbb{R})$ denote the set of real $n \times n$ matrices, and $M_n(\mathbb{C})$ denote the set of complex $n \times n$ matrices. In what follows, we mainly work with real matrices, though most of the theory is the same for complex matrices.

Let $A \in M_n(\mathbb{R})$. We may view A as a linear operator

$$\begin{aligned} A : \mathbb{R}^n &\to \mathbb{R}^n \\ v &\mapsto Av \end{aligned}$$

where we view $v \in \mathbb{R}^n$ as a $n \times 1$ matrix, i.e., a column vector. We may sometimes write a column vector $v = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$ as $v = (x_1, \dots, x_n)$ out of typographical convenience.

Let A^t denote the transpose of A. Then $(A^t)^t = A$. For $A, B \in M_n(\mathbb{R})$, we have $(AB)^t = B^t A^t$. Recall AB does not usually equal BA, though it happens from time to time.

Denote by $I = I_n$ the $n \times n$ identity matrix. This means $AI = IA = A$ for any $A \in M_n(\mathbb{R})$. We use 0 for the number 0 as well as zero matrices.

Denote by $\mathrm{diag}(d_1, \dots, d_n)$ the diagonal matrix with $A = (a_{ij})$ with $a_{ii} = d_i$ and $a_{ij} = 0$ if $i \neq j$. Note $I = \mathrm{diag}(1, 1, \dots, 1)$.

Recall A is invertible if there is a matrix $B \in M_n(\mathbb{R})$ such that $AB = I$. If there is such a matrix B, it must be unique, and we say B is the inverse of A and write $B = A^{-1}$. It is true that $AA^{-1} = A^{-1}A = I$, i.e., that the inverse of A^{-1} is just A, i.e., $(A^{-1})^{-1} = A$. For $A, B \in M_n(\mathbb{R})$ both invertible, we have that AB is also invertible and $(AB)^{-1} = B^{-1}A^{-1}$.

There is a function $\det : M_n(\mathbb{R}) \to \mathbb{R}$ called the determinant. It satisfies $\det(AB) = \det(A)\det(B)$, and A is invertible if and only if $\det A \neq 0$. We can define it inductively on n as follows. For $n = 1$, put $\det[a] = a$. Now $A = (a_{ij}) \in M_n(\mathbb{R})$ with $n > 1$. Let A_{ij} be the ij cofactor matrix, i.e., the matrix obtained by deleting the i-th row and j-th column. Then

$$\det(A) = a_{11} \det A_{11} - a_{12} \det A_{12} + \cdots + (-1)^{n-1} a_{1n} \det A_{1n}. \tag{3.1}$$

Hence the determinant can be computed by using a cofactor expansion along the top row. It can be computed similarly with a cofactor expansion along any row, and since $\det(A) = \det(A^t)$, it can be also computed using a cofactor expansion along any column. (Just remember to be careful of signs—the first sign is -1 if you start along an even numbered row or column.) For $n = 2$, we have

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc.$$

Example 3.1.1. *Let* $A = \begin{pmatrix} 2 & -1 & 0 \\ 0 & 2 & -1 \\ -1 & 0 & 2 \end{pmatrix}$. *Then*

$$\begin{aligned} \det A &= 2 \det \begin{pmatrix} 2 & -1 \\ 0 & 2 \end{pmatrix} - (-1) \det \begin{pmatrix} 0 & -1 \\ -1 & 2 \end{pmatrix} + 0 \det \begin{pmatrix} 0 & 2 \\ -1 & 0 \end{pmatrix} \\ &= 2 \cdot 4 + 1 \cdot (-1) + 0 = 7. \end{aligned}$$

For larger n, determinants are unpleasant to compute by hand in general, but some special cases can be worked out. For instance, if $D = \mathrm{diag}(d_1, \dots, d_n)$ is diagonal, then $\det D = d_1 d_2 \cdots d_n$. In particular, $\det I = 1$, and $\det kI_n = k^n$ for a scalar $k \in \mathbb{R}$. Also note that for invertible A, since $\det(A)\det(A^{-1}) = \det(AA^{-1}) = \det I = 1$, we have $\det(A^{-1}) = (\det A)^{-1}$. If S is invertible, then $\det(SAS^{-1}) = \det S \det A \det S^{-1} = \det A$. Hence similar matrices have the same determinant. Thus it makes sense to define the determinant $\det T$ of a linear transformation (i.e., this determinant does not depend upon a choice of basis).

Given any linear transformation $T : \mathbb{R}^n \to \mathbb{R}^n$, and any basis $\mathcal{B} = \{v_1, \dots v_n\}$, we can associate a matrix $[T]_{\mathcal{B}}$ in $M_n(\mathbb{R})$ such that if

$$T(x_1 v_1 + \cdots x_n v_n) = y_1 v_1 + \cdots + y_n v_n$$

then

$$[T]_{\mathcal{B}} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}.$$

In particular, let $\mathcal{B}_0 = \{e_1, \dots, e_n\}$ denote the standard basis for $\mathbb{R}^n$, i.e., e_i is the column vector with a 1 in the i-th position and 0's elsewhere. Then, if $Te_i = u_i$, the i-th column of $[T]_{\mathcal{B}_0}$ is u_i, i.e.,

$$[T]_{\mathcal{B}_0} = [u_1 | u_2 | \cdots | u_n].$$

Given any bases $\mathcal{B} = \{v_1, \dots, v_n\}$ and $\mathcal{B}' = \{v_1', \dots, v_n'\}$, there is an invertible change of basis matrix S, determined by requiring $Sv_i = v_i'$ for all i, such that

$$[T]_{\mathcal{B}'} = S[T]_{\mathcal{B}} S^{-1}. \tag{3.2}$$

For any matrices $A, B \in M_n(\mathbb{R})$, we say they are similar if $A = SAS^{-1}$ for an invertible $S \in M_n(\mathbb{R})$. This means A and B can be viewed as representing the same linear transformation T, just with respect to different bases.

We say a nonzero vector $v \in \mathbb{R}^n$ is a (real) eigenvector for A with (real) eigenvalue $\lambda \in \mathbb{R}$ if $Av = \lambda v$. If λ is an eigenvalue for A, the set of $v \in \mathbb{R}^n$ (including the zero vector) for which $Av = \lambda_v$ is called the λ-eigenspace for A, and it is a linear subspace of $\mathbb{R}^n$. Even if $A \in M_n(\mathbb{R})$ it may not have any real eigenvalues/vectors but it may have complex ones, so we will sometimes consider those. The dimension of the λ-eigenspace is called the geometric multiplicity of λ.

Eigenvalues and eigenvectors are crucial to understanding the geometry of a linear transformation as A having real eigenvalue λ means A simply acts as scaling by λ on the λ-eigenspace. Any complex eigenvalues must occur in pairs $re^{\pm i\theta}$, and complex eigenvalues means that A acts by scaling by r together with rotation by θ on an appropriate 2-dimensional subspace.

Eigenvalues can be computed as follows. If $Av = \lambda v = \lambda I v$ for a nonzero v, this means $(\lambda I - A)$ is not invertible, so it has determinant 0. The characteristic polynomial of A is

$$p_A(\lambda) = \det(\lambda I - A). \tag{3.3}$$

This is a polynomial of degree n in λ, and its roots are the eigenvalues. By the fundamental theorem of algebra, over $\mathbb{C}$ it always factors into linear terms

$$p_A(\lambda) = (\lambda - \lambda_1)^{m_1}(\lambda - \lambda_2)^{m_2} \cdots (\lambda - \lambda_k)^{m_k}. \tag{3.4}$$

Here $\lambda_1, \dots, \lambda_k$ denote the distinct (possibly complex) eigenvalues. The number m_i is called the (algebraic) multiplicity of λ_i. It is always greater than or equal to the (complex) dimension of the (complex) λ_i-eigenspace (the geometric multiplicity).

Now let us write the eigenvalues of A as $\lambda_1, \dots, \lambda_n$ where some of these may repeat. The spectrum of A is defined to be the (multi)set $\mathrm{Spec}(A) = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$, where we include each eigenvalue with (algebraic) multiplicity. (Multiset is like a set, except that elements are allowed to appear with some multiplicity.) If $A = (a_{ij})$ is diagonal, then $\mathrm{Spec}(A) = \{a_{11}, a_{22}, \dots, a_{nn}\}$. From (3.4), we see the spectrum determines the characteristic polynomial, and vice versa.

Once we find the eigenvalues λ_i, we can find the eigenspaces by solving the system of equations $(\lambda I - A)v = 0$.

Suppose B is similar to A, e.g., $B = SAS^{-1}$. Then

$$p_B(\lambda) = \det(\lambda I - SAS^{-1}) = \det(S(\lambda I - A)S^{-1}) = \det(S)\det(\lambda I - A)\det(S)^{-1} = p_A(\lambda), \tag{3.5}$$

so A and B have the same eigenvalues and characteristic polynomials. (Their eigenspaces will be different, but can be related by a change of basis matrix.)

One can shew that the spectrum a block diagonal matrix $\begin{pmatrix} A & \\ & B \end{pmatrix}$ is the $\mathrm{Spec}(A) \cup \mathrm{Spec}(B)$. Black matrix entries represent zeroes.

We say A is diagonalizable if A is similar to a diagonal matrix D. A matrix A is diagonalizable if and only if there is a basis of $\mathbb{C}^n$ consisting of eigenvectors for A. This is always possible if A has n distinct eigenvalues, but not always possible when you have repeated eigenvalues (cf. Exercise 3.1.8).

If there is a basis of eigenvectors $\{v_1, \dots, v_n\}$, the matrix $S = [v_1|\cdots|v_n]$ will be invertible, and doing the associated change of basis will give us a diagonal matrix $D = SAS^{-1}$. Specifically, we will get $D = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$ where λ_i is the eigenvalue associated to v_i. Doing this change of basis to get a diagonal matrix is called diagonalizing A.

One benefit of diagonalization is that it allows us to exponentiate easily. Suppose $D = SAS^{-1}$ is diagonal. Thinking of A as a linear transformation, there are many instances where we want to apply A repeatedly. We determine the resulting transformation by looking at powers of A. Note

$$A^m = (S^{-1}DS)^m = (S^{-1}DS)(S^{-1}DS)\cdots(S^{-1}DS) = S^{-1}D^mS. \tag{3.6}$$

For any two diagonal matrices $X = \mathrm{diag}(x_1, \dots, x_n)$ and $Y = \mathrm{diag}(y_1, \dots, y_n)$, we have $XY = YX = \mathrm{diag}(x_1y_1, \dots, x_ny_n)$. In particular, if $D = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$, then $D^m = \mathrm{diag}(\lambda_1^m, \dots, \lambda_n^m)$. In other words, D^m is easy to compute, so we may compute A^m easily.

There is some information we can easily get about eigenvalues without determining them exactly. First, suppose A is diagonalizable, i.e., $D = SAS^{-1}$ is diagonal for some S. Then

$D = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$ where the λ_i's are the eigenvalues of A (with multiplicity). Consequently, $\det A = \det D = \lambda_1 \lambda_2 \cdots \lambda_n$. In other words, the determinant of a matrix A is the product of the eigenvalues (with multiplicity). In fact, this is true for any A, not just diagonalizable matrices A. (Any A is similar to an upper triangular matrix, whose diagonal entries must be the eigenvalues, and now apply Exercise 3.1.1.)

The trace is another useful invariant to study eigenvalues. The trace of a matrix $\mathrm{tr}\, A$ is the sum of its diagonal entries. Note the trace map satisfies $\mathrm{tr}(A+B) = \mathrm{tr}\, A + \mathrm{tr}\, B$. Unlike the determinant, it does not satisfy $\mathrm{tr}(AB) = \mathrm{tr}\, A \mathrm{tr}\, B$ (just like $\det(A+B) \neq \det A + \det B$ in general). Nevertheless, similar matrices have identical trace. Therefore, we see if A is diagonalizable, then $\mathrm{tr}\, A$ is the sum of its eigenvalues. This statement is also true for any $A \in M_n(\mathbb{R})$.

Here is one of the main theorems on diagonalizability, often just called the spectral theorem.

Theorem 3.1.2. *Let $A \in M_n(\mathbb{R})$ be symmetric, i.e., $A^t = A$. Then A has only real eigenvalues and is diagonalizable over $\mathbb{R}$. In fact, A is diagonalizable by an orthogonal matrix S, i.e., there exists $S \in M_n(\mathbb{R})$ with $SS^t = I$ and SAS^{-1} is diagonal.*

Exercises

Exercise 3.1.1. *Let $A = (a_{ij}) \in M_n(\mathbb{R})$ be upper triangular, i.e., $a_{ij} = 0$ whenever $i > j$.*
(i) Prove $\det A = a_{11} a_{22} \cdots a_{nn}$.
(ii) Use this to show that $\mathrm{Spec}(A) = \{a_{11}, a_{22}, \dots, a_{nn}\}$.

Exercise 3.1.2. *Analyze the running time of the algorithm to compute the determinant by recursively using the cofactor expansion (3.1),*

Exercise 3.1.3. *Let $A \in M_n(\mathbb{R})$. Suppose v is an eigenvector with eigenvalue λ for A. Show, for $m = 1, 2, 3, \dots$, that v is an eigenvector with eigenvalue λ^m for A^m.*

Exercise 3.1.4. *Let $A \in M_2(\mathbb{R})$. Show $p_A(\lambda) = \lambda^2 - \mathrm{tr}(A)\lambda + \det(A)$ in two different ways: (i) use the definition $p_A(\lambda) = \det(\lambda I - A)$, and (ii) abstractly write $p_A(\lambda) = (\lambda - \lambda_1)(\lambda - \lambda_2)$.*

Exercise 3.1.5. *Let $A \in M_n(\mathbb{R})$ and write $p_A(\lambda) = \lambda^n + c_1 \lambda^{n-1} + c_2 \lambda^{n-2} + \cdots + c_{n-1}\lambda + c_n$. Show $c_1 = -\mathrm{tr} A$ and $c_n = (-1)^n \det A$.*

Exercise 3.1.6. *Show that A is diagonalizable if and only if the geometric multiplicity of each eigenvalue λ equals the algebraic multiplicity.*

Exercise 3.1.7. *Diagonalize the rotation-by-θ matrix $A = \begin{pmatrix} \cos i\theta & \sin i\theta \\ -\sin i\theta & \cos i\theta \end{pmatrix}$.*

Exercise 3.1.8. *Find a 2×2 matrix which is not diagonalizable. (Hint: you need to show it does not have a basis of eigenvectors, so it must have a repeated eigenvalue. Try looking at upper triangular matrices and use Exercise 3.1.1.)*

Exercise 3.1.9. *Diagonalize the matrix $A = \frac{1}{2}\begin{pmatrix} 0 & 2 \\ 1 & 1 \end{pmatrix}$ and use this to write down a simple expression for A^m. Can you make sense of $\lim_{m \to \infty} A^m$?*

3.2 Rudiments of spectral graph theory

▶ Here we assume graphs are undirected and simple unless stated otherwise.

Adjacency spectra

Let's start of by making some elementary observations on adjacency matrices. Let $G = (V, E)$ be a graph on the ordered set $V = \{1, 2, \dots, n\}$ and A be the adjacency matrix with repect to V. Write $A = (a_{ij})$ so $a_{ij} = 1$ if and only if $(i, j) \in E$.

Proposition 3.2.1. *The (i, j) entry of the matrix A^ℓ is the number of paths of length ℓ from i to j in G. This is still true if G is directed and/or non-simple.*

Proof. This is clearly true when $\ell = 0$ or 1. Now we prove this by induction on ℓ. Suppose it is true for $\ell = m$. We show it is true for $\ell = m + 1$. Write $B = (b_{ij}) = A^m$ and $C = (c_{ij}) = A^{m+1}$. Writing $C = AB$, we see

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

By our induction hypothesis, a_{ik} is the number of paths of length 1 from i to k (which is 0 or 1) and b_{kj} is the number of paths of length m from k to j. Hence c_{ij} is the sum over all neighbors k of vertex i of the number of paths of length ℓ from k to j. This must be the total number of paths of length $m + 1$ from i to j. □

Let σ be a permutation of V, i.e., $\sigma : V \to V$ is a bijective map. Let $\mathcal{B}_0 = \{e_1, \dots, e_n\}$ be the standard basis for $\mathbb{R}^n$. Then we can also view σ as a permutation of $\mathcal{B}_0$ given by $\sigma(e_i) = e_{\sigma(i)}$. Thus we can express σ as a matrix $S = [\sigma]_{\mathcal{B}_0}$. This is a 0–1 matrix (a matrix whose entries are all either 0 or 1) with exactly one 1 in each row and column. Such matrices are called permutation matrices. Since σ is bijective, the associated matrix S must be invertible, as S^{-1} effects the inverse permutation σ^{-1}.

For instance, suppose $V = \{1, 2, 3, 4\}$ and σ is the "cyclic shift" of V given by $\sigma(1) = 4$, $\sigma(2) = 1$, $\sigma(3) = 2$ and $\sigma(4) = 3$. Then the corresponding matrix is

$$S = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$

One readily checks $Se_1 = e_4$, $Se_2 = e_1$, $Se_3 = e_2$ and $Se_4 = e_3$. Note that S is the adjacency matrix for the directed cycle graph on 4 vertices where the directed cycle goes in the clockwise direction, and S^{-1} is the adjacency matrix for the directed cycle graph where the cycle goes in the counterclockwise direction.

Now let B be the adjacency matrix for G with respect to the ordering $\sigma(1), \sigma(2), \dots, \sigma(n)$. Then $B = SAS^{-1}$. Consequently all possible adjacency matrices for G are similar. Hence the quantities $\det A, \mathrm{tr} A$ and $\mathrm{Spec}(A)$ and $p_A(\lambda)$ are all invariants of G, i.e., they do not depend upon the labelling of vertices. Thus the following definition makes sense.

Definition 3.2.2. *Let G be a (possibly directed and non-simple) graph, and A be any adjacency matrix for G. We say λ is an* **(adjacency) eigenvalue** *for G if λ is an eigenvalue for A, and the* **(adjacency) spectrum** *of G is $\mathrm{Spec}\, G = \mathrm{Spec}\, A$. The* **characteristic polynomial** *of G is $p_G(\lambda) = p_A(\lambda)$. We also define the* **determinant** *and* **trace** *of G by $\det G = \det A$ and $\mathrm{tr}\, G = \mathrm{tr}\, A$.*

However, the eigenspaces of A in general depend upon the labelling (or ordering of vertices), so eigenspaces/eigenvectors are not invariants of G. However, if $Av = \lambda v$, then

$$BSv = (SAS^{-1})(Sv) = SAv = S\lambda v = Sv. \tag{3.7}$$

That is, we can go from eigenvectors of A to eigenvectors of B by applying the permutation matrix S (which just permutes the entries of v). This will be important for us later.

While isomorphic graphs, i.e., graphs that differ only up to labelling of vertices, must have the same spectrum, knowing that $\operatorname{Spec} G_1 = \operatorname{Spec} G_2$ does not imply that G_1 and G_2 are isomorphic. Since one can compare spectra in polynomial time, if this were true it would give us a polynomial time algorithm to answer the graph isomorphism problem. However, it is conjectured that the spectrum determines the isomorphism class of the graph "most of the time." If true, this means for "most" G_1, there is a polynomial-time algorithm that can determine if any G_2 is isomorphic to G_1.* Much work is currently being done to determine which graphs are determined by their spectra.

Spectral graph theory (which includes the above conjecture) is the study of what information about graphs we can read off from the spectrum (and related data). For instance, the number of elements (with multiplicity) in $\operatorname{Spec}(G)$ determines the order n of G. Spectral graph theory is an extremely active area of research, and we will just present as sample of some basic results in this section to motivate its utility.

Proposition 3.2.3. *Spec(G) determines the number of closed paths of length ℓ for all ℓ, and this number is* $\operatorname{tr}(A^\ell)$. *Here G may be directed and non-simple.*

Proof. Note the number of closed paths of length ℓ is simply $\operatorname{tr}(A^\ell)$. If $\operatorname{Spec}(A) = \{\lambda_1, \dots, \lambda_n\}$, then $\operatorname{Spec}(A^\ell) = \{\lambda_1^\ell, \dots, \lambda_n^\ell\}$ by Exercise 3.1.3. Hence $\operatorname{tr}(A^\ell) = \sum_i \lambda_i^\ell$ is determined by $\operatorname{Spec}(G) = \operatorname{Spec}(A)$. □

In fact, one can rewrite the characteristic polynomial in terms of $\operatorname{tr}(A^\ell)$ for $1 \le \ell \le n$, and deduce the converse of this proposition, but we will not do this.

We can be more explicit about the traces $\operatorname{tr}(A^\ell)$ of the first few powers of A.

Proposition 3.2.4. *Write* $\operatorname{Spec}(G) = \{\lambda_1, \dots, \lambda_n\}$. *Then we have each $\lambda_i \in \mathbb{R}$ and*
(i) $\operatorname{tr}(A) = \lambda_1 + \lambda_2 + \cdots + \lambda_n = 0$;
(ii) $\frac{1}{2}\operatorname{tr}(A^2) = \frac{1}{2}(\lambda_1^2 + \lambda_2^2 + \cdots + \lambda_n^2)$ *is the number of edges of G; and*
(iii) $\frac{1}{6}\operatorname{tr}(A^3) = \frac{1}{6}(\lambda_1^3 + \lambda_2^3 + \cdots + \lambda_n^3)$ *is the number of triangles in G.*

Proof. Each eigenvalue is real by the spectral theorem.

(i) The trace is zero because each diagonal entry of A is 0. For non-simple graphs, $\operatorname{tr}(A)$ will just be the total number of loops.

(ii) Each closed path of length 2 is simply means traversing an edge (u, v) going back, and each edge (u, v) corresponds to 2 closed paths of length 2.

(iii) is similar to (ii). Any closed path of length 3 must be a 3-cycle, and each triangle yields 6 different closed paths of length 3. □

*Of course, given any two random graphs G_1 and G_2, most of the time they will not be isomorphic and can be easily distinguished in polynomial time by simple invariants such as degree distributions. The strength of this statement is that if G_1 lies outside of a conjecturally small class of graphs, then we get a polynomial-time algorithm that works for *any* G_2.

These arguments no longer work for higher powers of traces because the closed paths of length ℓ will include more than just the of cycles of length ℓ for $\ell \geq 4$, but include paths with repeated vertices. However, see Exercise 3.2.2 for $\ell = 4$.

Since all eigenvalues of real, we will typically order them as $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. By (ii), we see that all eigenvalues are 0 if and only if G has no edges (this is not true for directed graphs—see example below). From now on we will assume this is not the case. Then, as the sum of the eigenvalues is 0, we know $\lambda_1 > 0$ and $\lambda_n < 0$.

Now let us look at a few simple directed and undirected examples.

Example 3.2.5 (Directed path graphs)**.** *Let G be the directed path graph on n vertices. Then*

$$p_G(\lambda) = \det \begin{pmatrix} \lambda & -1 & 0 & \cdots & 0 \\ 0 & \lambda & -1 & \ddots & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \lambda & -1 \\ 0 & \cdots & \cdots & 0 & \lambda \end{pmatrix} = \lambda^n,$$

by Exercise 3.1.1. Hence all eigenvalues are 0.

Example 3.2.6 (Directed cycle graphs)**.** *Let G be the directed cycle graph on n vertices. Then*

$$p_G(\lambda) = \det \begin{pmatrix} \lambda & -1 & 0 & \cdots & 0 \\ 0 & \lambda & -1 & \ddots & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \lambda & -1 \\ -1 & 0 & \cdots & 0 & \lambda \end{pmatrix} = \lambda^n - 1.$$

This can be seen by using a cofactor expansion along the first column, and using Exercise 3.1.1 and its analogue for lower triangular matrices (true since $\det(A) = \det(A^t)$). Thus $\mathrm{Spec}(G) = \{1, \zeta, \zeta^2, \ldots, \zeta^{n-1}\}$ is the set of n-th roots of unity, where $\zeta = e^{2\pi i/n}$. Hence we may get complex eigenvalues (and also complex eigenvectors) for directed graphs.

Example 3.2.7 (Path graphs)**.** *Consider the path graph P_n on n vertices. Then*

$$p_{P_n}(\lambda) = \det \begin{pmatrix} \lambda & -1 & 0 & \cdots & 0 & 0 \\ -1 & \lambda & -1 & \ddots & 0 & 0 \\ 0 & -1 & -\lambda & \ddots & & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & & -1 & \lambda & -1 \\ 0 & 0 & \cdots & & -1 & \lambda \end{pmatrix}$$

From a cofactor expansion, we see we have the recursion

$$p_{P_n}(\lambda) = \lambda p_{P_{n-1}}(\lambda) - p_{P_{n-2}}(\lambda), \quad n \geq 4.$$

Thus we can compute the first few cases

$$\begin{aligned} p_{P_2}(\lambda) &= \lambda^2 - 1 & \mathrm{Spec}(P_2) &= \{1, -1\} \\ p_{P_3}(\lambda) &= \lambda(\lambda^2 - 2) & \mathrm{Spec}(P_3) &= \{\sqrt{2}, 0, -\sqrt{2}\} \\ p_{P_4}(\lambda) &= \lambda^4 - 3\lambda^2 + 1 & \mathrm{Spec}(P_4) &= \{\sqrt{\frac{3+\sqrt{5}}{2}}, \sqrt{\frac{3-\sqrt{5}}{2}}, -\sqrt{\frac{3-\sqrt{5}}{2}}, -\sqrt{\frac{3+\sqrt{5}}{2}}\}. \end{aligned}$$

Example 3.2.8 (Cycle graphs)**.** *Consider a cycle graph $G = C_n$. Explicit calculations give*

$$\begin{aligned} p_{C_3}(\lambda) &= \lambda^3 - 3\lambda - 2 & \mathrm{Spec}(C_3) &= \{2, -1, -1\} \\ p_{C_4}(\lambda) &= \lambda^4 - 4\lambda^2 & \mathrm{Spec}(C_4) &= \{2, 0, 0, -2\}. \end{aligned}$$

It is easy to see in general that the all ones vector $(1, 1, \ldots, 1)$ is an eigenvector with eigenvalue 2. If n is even, one can check the vector with alternating entries $(1, -1, 1, -1, \ldots, 1, -1)$ is an eigenvector with eigenvalue 2. One can show that the other eigenvalues are of the form $2\cos(2j\pi/n)$ for $1 \leq j < \lfloor \frac{n}{2} \rfloor$, each with multiplicity 2. See Exercise 3.2.3 for the case of $n = 6$.

Example 3.2.9 (Complete graphs)**.** *Take $G = K_n$. Then the adjacency matrix (with respect to any ordering of vertices!) is*

$$A = \begin{pmatrix} 0 & 1 & 1 & \cdots & 1 \\ 1 & 0 & 1 & \cdots & 1 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 1 & \cdots & 1 & 0 & 1 \\ 1 & \cdots & \cdots & 1 & 0 \end{pmatrix}.$$

In this case, due to the nature of A, the easiest way to find the eigenvalues is to guess a basis of eigenvectors. Clearly the all ones vector $v_1 = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$ is an eigenvector with eigenvalue $n - 1$. For $2 \leq i \leq n$, let v_i denote the vector with a 1 in position 1, -1 in position i, and 0's elsewhere. Then it is easy to see $Av_i = -v_i$ for $i \geq 2$. Hence -1 is an eigenvalue with multiplicity (geometric, and therefore also algebraic) $n - 1$. That is,

$$\mathrm{Spec}(G) = \{n - 1, -1, -1, \cdots, -1\},$$

and if you care

$$p_G(\lambda) = (\lambda - (n-1))(\lambda + 1)^{n-1}.$$

More generally, for a k-regular graph, the maximum eigenvalue will be k. We can also given upper bound for arbitrary (simple, undirected) graphs in terms of degree.

Proposition 3.2.10. *Let k be the maximum degree of a vertex in G. Then for any eigenvalue λ of G, $|\lambda| \leq k$. In particular, $|\lambda| \leq n - 1$. Moreover, if G is k-regular, then in fact $\lambda = k$ is an eigenvalue.*

Put another way, the maximum eigenvalue gives a lower bound for the maximum degree of G. And for regular graphs, the maximum eigenvalue determines the degree.

Proof. Let $v = (x_1, x_2, \ldots, x_n)$ be an eigenvector for G with eigenvalue λ. Say $|x_m|$ achieves the maximum of all $|x_j|$'s. By scaling, we may assume $x_m = 1$. Write $Av = \lambda v = (y_1, y_2, \ldots, y_n)$ and A_{ij} Then

$$|\lambda| = |\lambda x_m| = |y_m| = \left| \sum_{j=1}^{n} a_{mj} x_j \right| \leq k$$

since at most k of the entries a_{mj} are 1.

If G is k-regular, then all row sums of A are k, so $Av = kv$ where $v = (1, 1, \ldots, 1)$. □

In fact, for k-regular graphs G, the multiplicity of $\lambda = k$ equals number of connected components of G. More generally, to determine the spectrum of G it suffices to look at the connected components.

Lemma 3.2.11. *Let $G_1, \ldots, G_r$ denote the connected components of a graph G. Then*

$$\mathrm{Spec}(G) = \mathrm{Spec}(G_1) \cup \mathrm{Spec}(G_2) \cup \cdots \mathrm{Spec}(G_r).$$

Proof. We may order the vertices of G in such a way, that the adjacency matrix A for G is a block diagonal matrix of the form

$$A = \begin{pmatrix} A_1 & & & \\ & A_2 & & \\ & & \ddots & \\ & & & A_r \end{pmatrix}$$

where A_i is an adjacency matrix for G_i. The eigenvalues of a block diagonal matrix are just the collection of the eigenvalues of the blocks, with multiplicity. □

This implies that if G is k-regular, and G has r connected components, the multiplicity of $\lambda = k$ is at least r. To conclude it is exactly r, we need to show that this multiplicity is 1 if G is connected. In fact, a similar argument will tell us something about the minimum eigenvalue too.

Proposition 3.2.12. *If G is k-regular and connected, then the maximum eigenvalue $\lambda_1 = k$ occurs with multiplicity 1. Furthermore, the minimum eigenvalue $\lambda_n \geq -k$ with $\lambda_n = -k$ if and only if G is bipartite. If G is bipartite, then $\lambda_n = -k$ also occurs with multiplicity 1.*

In any normal graph theory course, we would have defined bipartite by now. I was hoping to avoid it all together, but it will be good to know a criterion for when $|\lambda_1| = |\lambda_n|$ later (in a more general setting).

Definition 3.2.13. *Let $G = (V, E)$ be a directed or undirected graph. We say G is* **bipartite** *if there is a partition $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$.*

In other words, no two vertices in V_1 are connected by an edge, and similarly for V_2. For instance, C_n is bipartite if and only if n is even (draw pictures), where as a path graph on n vertices is bipartite for any n. On the other hand, graphs like K_n, star graphs, and wheel graphs are never bipartite. A convenient way to think the bipartite condition in terms of colorings: we can color vertices in V_1 red, and vertices in V_2 green, so we see a bipartite graph G is one with a 2-coloring, i.e., $\chi(G) \leq 2$. (We need to allow the possibility $\chi(G) = 1$ because technically graphs

with no edges are bipartite.) In terms of matrices, bipartite means that one can order the vertices so that the adjacency matrix is a block matrix of the form $\begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}$.

Proof. Suppose $v_1 = (x_1, \dots, x_n)$ is an eigenvector for $\lambda_1 = k$. As before, scale v_1 to get a maximum coordinate $x_m = 1$ and $|x_i| \leq 1$ for $1 \leq i \leq n$. Let $(y_1, \dots, y_n) = Av_1 = \lambda v_1 = kv_1$ and $A = (a_{ij})$ Then

$$y_m = \sum_{j=1}^{n} a_{mj} x_j = k$$

implies that $x_j = 1$ for all k neighbors j of vertex m. Similarly, $x_i = 1$ for all neighbors of these x_j's, i.e., the vertices of distance 2 from m. Continuing in this manner shows $x_j = 1$ for all j in the connected component of m, which is the whole graph by assumption. Hence v_1 is determined uniquely up to a scalar, which means λ_1 has multiplicity 1. (Since A is diagonalizable, the geometric and algebraic multiplicity of an eigenvalue are the same.)

For the minimum eigenvalue, we saw already that $\lambda_n \geq -k$ in Proposition 3.2.10. So suppose $(x_1, \dots, x_n)$ is an eigenvector with eigenvalue $-k$, again satisfying $x_m = 1$ and $|x_i| \leq 1$ for $1 \leq i \leq n$. Then we have the identity

$$y_m = \sum_{j=1}^{n} a_{mj} x_j = -k,$$

so now we see $x_j = -1$ for all neighbors j of m. The same reasoning shows that if $x_j = -1$, then $x_i = +1$ for any neighbor i of j. Let $V_1 = \{i : 1 \leq i \leq n, x_i = +1\}$ and $V_2 = \{j : 1 \leq j \leq n, x_j = -1\}$. By connectedness, every vertex of G lies in either V_1 or V_2, and the reasoning above says that no two vertices in V_1 are connected by a (single) edge, and similarly for V_2. Hence G must be bipartite. In addition, this eigenvector is determined uniquely up to scaling, so if G is bipartite, $\lambda = -k$ has multiplicity 1. □

We won't prove this, but another standard fact is that the spectrum $\{\lambda_1, \dots, \lambda_n\}$ of G is symmetric about 0, i.e., $\lambda_j = \lambda_{n-j+1}$ for $1 \leq j \leq \frac{n}{2}$, if and only if G is bipartite.

Now let's prove a result about the diameter.

Proposition 3.2.14. *G connected simple undirected. If G has r distinct eigenvalues, then* $\operatorname{diam}(G) < r$.

Proof. Let $\lambda_1, \dots, \lambda_r$ denote the distinct eigenvalues of G, and m_i the multiplicity of λ_i. Then we may diagonalize A as $D = SAS^{-1}$ where D is a block diagonal matrix of the form

$$D = \begin{pmatrix} \lambda_1 I_{m_1} & & \\ & \ddots & \\ & & \lambda_r I_{m_r} \end{pmatrix}.$$

Consider the polynomial in A given by

$$m(A) = (\lambda_1 I - A)(\lambda_2 I - A) \cdots (\lambda_r I - A).$$

Note that

$$\begin{aligned} m(D) &= (\lambda_1 I - A)(\lambda_2 I - A) \cdots (\lambda_r I - A) \\ &= S(\lambda_1 I - A)S^{-1} S(\lambda_2 I - A)S^{-1} \cdots S(\lambda_r I - A)S^{-1} = Sm(A)S^{-1}. \end{aligned}$$

Each $\lambda_i I - D$ is a block diagonal matrix where the i-th diagonal block is all zeroes, so when we multiply them all together, we see $m(D) = 0$. For example, if $r = 3$, we have

$$m(D) = \begin{pmatrix} 0 \cdot I_{m_1} & & \\ & (\lambda_1 - \lambda_2) I_{m_2} & \\ & & (\lambda_1 - \lambda_3) I_{m_3} \end{pmatrix} \begin{pmatrix} (\lambda_2 - \lambda_1) I_{m_1} & & \\ & 0 \cdot I_{m_2} & \\ & & (\lambda_2 - \lambda_3) I_{m_3} \end{pmatrix} \times \begin{pmatrix} (\lambda_3 - \lambda_1) I_{m_1} & & \\ & (\lambda_3 - \lambda_2) I_{m_2} & \\ & & 0 \cdot I_{m_3} \end{pmatrix} = 0.$$

(Block diagonal matrices multiply like diagonal matrices: $\begin{pmatrix} A_1 & \\ & A_2 \end{pmatrix} \begin{pmatrix} B_1 & \\ & B_2 \end{pmatrix} = \begin{pmatrix} A_1 B_1 & \\ & A_2 B_2 \end{pmatrix}$.)
Consequently, $m(A) = 0$, so we have a nontrivial linear dependence relation among the first r powers of A and $I = A^0$:

$$m(A) = c_r A^r + c_{r-1} A^{r-1} + \cdots + c_1 A + c_0 = 0.$$

(This is nontrivial as $c_r = (-1)^r \neq 0$.)

Now let d be the diameter of G. Say vertices i and j are distance d apart. By Proposition 3.2.1, the (i,j)-th entry of A^d is nonzero but the (i,j) entry of A^ℓ is 0 for all $0 \leq \ell < d$. Hence, any nontrivial linear dependence relation among $I, A, A^2, \ldots, A^{d-1}, A^d$ must not involve A^d. Similarly, there must exist a vertex j' such that $d(i,j') = d - 1$, and therefore the (i,j') entry of $A^\ell = 0$ for all $0 \leq \ell < d-1$, but the (i,j') entry of A^{d-1} is nonzero. Thus any nontrivial linear dependence relation among $I, A, A^2, \ldots, A^{d-1}, A^d$ cannot involve either A^d or A^{d-1}. Continuing in this way, we see there is no nontrivial linear dependence relation among $I, A, A^2, \ldots, A^{d-1}, A^d$. Consequently, $d < r$. □

The polynomial $m(A)$ in the proof is known as the minimal polynomial of A—it is the minimum degree polynomial such that $m(A) = 0$, the $n \times n$ zero matrix.

For instance, K_n has only 2 distinct eigenvalues, so this result says $\operatorname{diam}(K_n) = 1$. Go back to our examples of (undirected) path and cycle graphs and see what this result tells you about their diameters.

Before we move on, let us just mention a few other simple yet interesting results about adjacency spectra.

- The maximum eigenvalue λ_1 is at least the average degree of G.
- We can bound the chromatic number in terms of the maximum and minimum eigenvalues λ_1 and λ_n by
$$1 + \frac{\lambda_1}{|\lambda_n|} \leq \chi(G) \leq 1 + \lambda_1(G).$$
- If G is regular and not complete, then the clique number of G is at most $n - 1$ minus the number of eigenvalues in $(-1, 1)$.

Before we move on to random walks, let us say a word about Laplacian eigenvalues.

Definition 3.2.15. *Let $G = (V, E)$ be a directed or undirected graph on the ordered set $V = \{1, 2, \dots, n\}$. Let A be the adjacency matrix with respect to V and D be the* **degree matrix** *given by $D = \mathrm{diag}(\deg(1), \deg(2), \dots, \deg(n))$. Here $\deg(i)$ denotes the degree of i if G is undirected and the out degree of i if G is directed. The* **Laplacian** *of G (with respect to the ordering on V) is*

$$L = D - A.$$

The **Laplacian eigenvalues** *are the eigenvalues of L, and the* **Laplacian spectrum** $\mathrm{Spec}_L(G)$ *is* $\mathrm{Spec}(L)$.

As with the adjacency matrix, the Laplacian matrix depends upon the ordering of vertices in general, but if L is the Laplacian with respect to an ordered set V and L' is the Laplacian with respect to an ordered set V', then $L' = SLS^{-1}$ for a suitable permutation matrix S. Consequently, the Laplacian spectrum is also an invariant for G, and in some ways it is nicer than the adjacency spectrum.

For instance, we saw some nice results about adjacency spectra of regular graphs. One nice thing here is that the all-one vector is always an eigenvector for regular graphs, and the eigenvalue is the degree. This was because for regular graphs the adjacency matrix has constant row sums.

By definition of the Laplacian matrix, each row sum is zero—the i-th diagonal entry $\deg(i)$ of D is precisely the sum of the entries of the i-th row of A. Consequently the all-one vector $v = (1, 1, \dots, 1)$ is an eigenvector for L with eigenvalue 0. Assuming G is undirected, L is also a symmetric matrix, so it is diagonalizable and has real eigenvalues by the spectral theorem. We write the eigenvalues of L in increasing order as $\lambda_{L,1} \le \lambda_{L,2} \le \cdots \le \lambda_{L,n}$, where as the adjacency eigenvalues are arranged in decreasing order $\lambda_1 \ge \lambda_2 \ge \cdots \ge \lambda_n$. This is because for regular graphs, the adjacency and Laplacian eigenvalues correspond, but with reverse ordering.

Proposition 3.2.16. *Let G be a k-regular graph. Then $\lambda_{L,i} = k - \lambda_i$ for $1 \le i \le n$, and we have $0 = \lambda_{L,1} \le \lambda_{L,2} \le \cdots \le \lambda_{L,n} \le 2k$.*

Proof. Since G is k-regular, the degree matrix $d = kI$. Thus

$$\begin{aligned} p_L(\lambda) &= \det(\lambda I - L) = \det(\lambda I - (D - A)) = \det(\lambda I - kI + A) \\ &= \det(-((k - \lambda)I - A)) = (-1)^n p_A(k - \lambda). \end{aligned}$$

Since the eigenvalues of A are the roots λ_i of $p_A(\lambda)$, the Laplacian eigenvalues, are the roots of $p_A(k - \lambda)$, which are just the numbers $k - \lambda_i$. It is clear this map $\lambda_i \mapsto k - \lambda_i$ reverses inequalities $\lambda_i \ge \lambda_j$, so the increasing ordering of $\lambda_{L,i}$'s is justified.

The final statement then follows from Proposition 3.2.10. □

In general, it is still the case that 0 is the minimum eigenvalue $\lambda_{L,1}$, so all other (distinct) eigenvalues of L are positive. We also know $\mathrm{tr} L = \sum \deg(i) = \sum \lambda_{L,i}$ is twice the number of edges of G. (This is equal to the number of edges when G is directed, but some eigenvalues might be complex.)

In fact, it's often even better to work with the **normalized Laplacian** $\mathcal{L} = D^{-1/2} L D^{-1/2}$. (Since D is diagonal with nonnegative entries, $D^{-1/2}$ makes sense—just replace each nonzero entry of D with the reciprocal of the square root of that entry. E.g., if $D = \mathrm{diag}(2, 1, 1, 0)$, let $D^{-1/2} = \mathrm{diag}(1/\sqrt{2}, 1, 1, 0)$.) Again all eigenvalues $\lambda_{\mathcal{L},1} \le \cdots \lambda_{\mathcal{L},n}$ of $\mathcal{L}$ are real since $\mathcal{L}$ is still symmetric,

but now they lie in the range $[0, 2]$, with the minimum one being $\lambda_{\mathcal{L},1} = 0$. If G is connected, 0 occurs with multiplicity 1. It turns out that the sizes of the smallest positive eigenvalue $\lambda_{\mathcal{L},2}$ and the largest eigenvalue $\lambda_{\mathcal{L},n}$ give us a lot of information about the network flow, that is not easily accessible from adjacency eigenvalues in general. We will see this by working with the closely related transition matrices in the next section, which have eigenvalues in the range $[-1, 1]$.

Note: you can numerically compute eigenvalues in Sage as with the command `spectrum()`. If you set the optional parameter to `True`, it will return the (non-normalized) Laplacian eigenvalues. You can also compute the characteristic polynomial $p_G(\lambda)$ exactly with the `charpoly()` function. For example:

Sage 6.1

```
sage: G = graphs.CycleGraph(5)
sage: G.charpoly()
x^5 - 5*x^3 + 5*x - 2
sage: G.spectrum()
[2, 0.618033988749895?, 0.618033988749895?, -1.618033988749895?,
-1.618033988749895?]
sage: G.spectrum(True)
[3.618033988749895?, 3.618033988749895?, 1.381966011250106?,
1.381966011250106?, 0]
```

Exercises

Exercise 3.2.1. *Write a function in Sage* `countpaths(G,u,v,l)` *which counts the number of paths from u to v of length l in G by exponentiating the adjacency matrix of G.*

Exercise 3.2.2. *Let G be a k-regular graph. Prove an exact relation between $\mathrm{tr}(A^4)$ and the number of 4-cycles in G.*

Exercise 3.2.3. *Determine (by hand) the spectrum of C_6. Verify Propositions 3.2.10 and 3.2.14 hold for this graph.*

Exercise 3.2.4. *Determine (by hand) the spectrum of the star graph on 5 vertices. Verify Propositions 3.2.10 and 3.2.14 hold for this graph.*

Exercise 3.2.5. *Let G be a connected graph with maximum degree k. Show that k is an eigenvalue of G if and only if G is regular.*

Exercise 3.2.6. *Let G be a directed graph. Show that all eigenvalues of G are 0 if and only if G has no cycles (including cycles of length 1 or 2).*

Exercise 3.2.7. *Suppose G is regular. Express the eigenvalues of the normalized Laplacian $\mathcal{L}$ in terms of the adjacency eigenvalues.*

3.3 Random walks on graphs

▶ Here our graphs may be directed or undirected, simple or non-simple.

Since the works of Erdös, Rényi and Gilbert on random graphs, it was realized that probabilistic methods provide a powerful tool for studying graph theory. One revelation was that you can learn

a lot about the graph by studying random walks. Random walks are simply walks or paths along a graph, where at each stage you choose a direction (edge) to travel along at random. Here is a somewhat more formal definition.

Definition 3.3.1. *A* **random walk** *on a graph $G = (V, E)$ starting at $v_0 \in V$ is a random process which generates an infinite path $(v_0, v_1, v_2, \ldots)$ on G subject to the following rule:*

- *At any time $t = 1, 2, 3, \ldots$, v_t is chosen at random from among the neighbors of v_{t-1}. Each neighbor is equally likely to be chosen. If v_{t-1} has no neighbors, then $v_t = v_{t-1}$.*

Example 3.3.2 (Gambler's Ruin)**.** *For this example only, we will allow G to be infinite. Namely, let $G = (V, E)$ where $V = \mathbb{Z}$ and $E = \{\{i, i+1\} : i \in \mathbb{Z}\}$.*

$$\cdots \text{——} -2 \text{——} -1 \text{——} 0 \text{——} 1 \text{——} 2 \text{——} \cdots$$

Let's consider a random walk on G starting at $v_0 = 0$. Then, we either go left or right with equal probability. Hence at our next step, $t = 1$, we go to $v_1 = 1$ with probability $1/2$ and $v_1 = -1$ with probability $1/2$. We can write this as $P(v_1 = 1) = P(v_1 = -1) = 1/2$. At our the next step, $t = 2$, we see our probabilities depend on what v_1 was. For instance, if $v_1 = 1$, then we are at 0 with probability $1/2$ and 2 with probability $1/2$.

If we want to be systematic, we can list out all possibilities for the walk up to $t = 2$ as

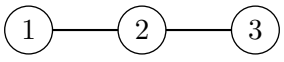
$$(0, -1, -2, \ldots)$$
$$(0, -1, 0, \ldots)$$
$$(0, 1, 0, \ldots)$$
$$(0, 1, 2, \ldots)$$

which must each occur with the same probabilities, namely $1/4$. Hence $P(v_2 = -2) = P(v_2 = 2) = 1/4$ and $P(v_2 = 0) = 1/2$.

This example is called Gambler's Ruin for the following reason. Imagine playing a fair game of chance repeatedly, where your odds of winning are 50–50. Each time you win you gain \$1, and each time you lose, you lose \$1. Then this random walk models your winnings/losings—namely v_t models the number of dollars you have won (or lost if it is negative) after t rounds of the game. One can prove that with probability 1, v_t will get arbitrarily large and arbitrarily small (meaning arbitrarily negative) as $t \to \infty$. (In fact with probability 1, v_t will make large positive and negative swings infinitely often.) Consequently, if you keep playing, you are guaranteed to go bankrupt at some point.

But I say, go for it anyway! It's fun, right! And if you lose all your money, don't get down on yourself—you just verified a mathematical theorem![*]

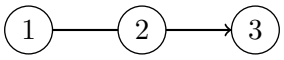
Example 3.3.3. *Suppose G is a path graph on 3 vertices.*



[*]Legal disclaimer: While I may find mathematical satisfaction in your gambling debts, I do not claim any responsibility. Do not tell anyone to break my legs instead of yours.

Take a random walk starting, say, at $v_0 = 1$. Then at the first step, $t = 1$, we have to walk to vertex $v_1 = 2$. At $t = 2$, we go to either vertex $v_2 = 1$ or $v_2 = 3$, with equal probability. Then, regardless of this choice, at $t = 3$, we see we must be back to vertex 2, and therefore this process repeats. Hence we can describe the random walk as $v_{2i+1} = 2$ for all $i \geq 0$ and v_{2i} is either 1 or 3, each with probability $1/2$, for any $i \geq 1$.

We note the condition about v_{t+1} having no neighbors in the definition of random walk is mostly to deal with the situation of nodes of out degree 0 in directed graphs. It can never happen that we reach a node of degree 0 for undirected graphs (unless we start at one). But consider this directed example:



Again, if we start at vertex 1, we must go to vertex 2 next, and from there we either go to 1 or 3. If we go to 1, we must go back to 2 and repeat the process, but if we go to vertex 3, the random walk rule says $v_t = 3$ from this point out. (One could alternatively terminate the random walk here, but this definition will give us a more uniform treatment of random walks.) One can easily check that, as $t \to \infty$, the probability of only bouncing back and forth between vertices 1 and 2 goes to 0—thus with probability 1 we will eventually go to vertex 3, where we will be stuck for eternity. How *ad infinitum*.

As you might expect, these elementary logical arguments for understanding random walks will get cumbersome very quickly with more complicated graphs. There's a better way to look at this problem using probability and linear algebra. In fact, this is a more precise way to think about random walks. At each time t, let us think of the state v_t of the random walk as a *probability distribution* on the graph G (or more precisely, on the set of vertices V). Namely, regard v_t as a *probability vector* (a vector with non-negative entries which sum to 1) $v_t = (p_1(t), p_2(t), \ldots, p_n(t))$, where $p_i(t)$ is the probability that we are at vertex i at time t. (We think of these vectors as column vectors, despite me just having written all the entries in a row.) Then to describe the random walk, we just need to explain how to get from the probability vector v_t to the next one v_{t+1}. This will be done by means of the transition matrix T (which will independent of both our time t and the initial state v_0).

Let's see how this works first by redoing Example 3.3.3. Consider the matrix

$$T = \begin{pmatrix} 0 & \frac{1}{2} & 0 \\ 1 & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{pmatrix}.$$

Then

$$Tv_t = \begin{pmatrix} 0 & \frac{1}{2} & 0 \\ 1 & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} p_1(t) \\ p_2(t) \\ p_3(t) \end{pmatrix} = \begin{pmatrix} p_2(t)/2 \\ p_1(t) + p_3(t) \\ p_2(t)/2 \end{pmatrix}.$$

First observe that as long as v_t is a probability vector, i.e. $p_1(t) + p_2(t) + p_3(t) = 1$ and all entries are non-negative, then so is Tv_t. Now the key point is that multiplication by T perfectly describes this random walk process: the probability that we are at vertex 1 (or 3) at time $t + 1$ must be $1/2$ the probability we were at vertex 2 at time t, and the probability that we are at vertex 2 at time

$t+1$ must be the probability that we were at either vertex 1 or 3 at time t. Thus we can compute the random walk as

$$v_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad v_1 = Tv_0 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad v_2 = Tv_1 = \begin{pmatrix} \frac{1}{2} \\ 0 \\ \frac{1}{2} \end{pmatrix}, \quad v_3 = Tv_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = v_1, \quad \dots$$

Here the state vector v_0 means that (with probability 1) we are at vertex 1 at time 0, v_1 means (with probability 1) we are at vertex 2 at time 1, and so on.

Here is the general way to compute this.

Definition 3.3.4. *Let $G = (V, E)$ be a graph (possibly directed and non-simple) with the ordered set of vertices $V = \{1, 2, \dots, n\}$. Let $A = (a_{ij})$ be the adjacency matrix with respect to V. Then the* **transition matrix** *for G with respect to V is the matrix $T = (t_{ij})$ where*

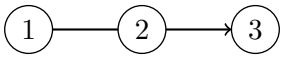
$$t_{ij} = \frac{a_{ji}}{\deg(j)}$$

if $\deg(j) \neq 0$ *and*

$$t_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

if $\deg(j) = 0$. *Here* $\deg(j)$ *denotes the out degree of j in the case G is directed.*

Put more plainly, for undirected graphs $a_{ij} = a_{ji}$, so the transition matrix T is the matrix obtained by dividing each entry a_{ij} of A by the j-th column sum (unless the j-th column is all 0's, in which case we just change the diagonal entry a_{jj} to 1). For directed graphs, we need to first take the transpose of A and then divide by column sums (except for zero columns, where we set the diagonal entry to 1). For instance, for the graph



we have

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \quad A^T = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad T = \begin{pmatrix} 0 & \frac{1}{2} & 0 \\ 1 & 0 & 0 \\ 0 & \frac{1}{2} & 1 \end{pmatrix}.$$

Consequently, the matrix T will be a matrix of non-negative entries whose columns all sum to 1. Note that T need not be symmetric even if A is (we computed the transition matrix T for the path graph on 3 vertices above, and it was not symmetric). However if G is a k-regular graph, then we simply have $T = \frac{1}{k}A$, so T is symmetric in this case.

Now let's see why this transition matrix does what we want. Consider a probability vector $v_t = (p_1(t), p_2(t), \dots, p_n(t))$. (Again this is a column vector—you just can't tell because your head is sideways.) Then

$$Tv_t = \begin{pmatrix} \sum_j t_{1j} p_j(t) \\ \sum_j t_{2j} p_j(t) \\ \vdots \\ \sum_j t_{nj} p_j(t) \end{pmatrix}.$$

Again, we can first note that each entry of the vector on the right must be non-negative, and the entries sum to 1:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} t_{ij}p_j(t) = \sum_{j=1}^{n}\left(p_j(t)\sum_{i=1}^{n} t_{ij}\right) = \sum_{j=1}^{n} p_j(t)\cdot 1 = 1, \tag{3.8}$$

as each column of T and the vector v_t does. (Whenever you see a double sum, you should always try to change the order of summation to get what you want. Even if you don't know what you want, interchange the order of summation and whatever you get is undoubtedly what your soul was desperately longing for.)

What about the i-th entry of Tv_t? This is $\sum_j t_{ij}p_j(t)$. The only contributions to this sum are from j with $t_{ij} \neq 0$, i.e., those j which have an edge to i. For such a j, $t_{ij} = \frac{1}{\deg(j)}$ (or $t_{ij} = 1$ if $i = j$ and $\deg(j) = 0$), but this is precisely the probability that we will proceed from vertex j to vertex i. Thus $t_{ij}p_j(t)$ is the probability—STOP THE PRESSES!!! I HAVE JUST REALIZED I'M USING t FOR TOO MANY THINGS—that you go from vertex j at time t to vertex i at time $t+1$. Hence summing this up over all i must yield $p_i(t+1)$, the probability that you went to vertex i (from some vertex) at time $t+1$.

This shows that the transition matrix can always be used to express the state of the random walk. That is, for any state v_t

$$v_{t+1} = Tv.$$

In particular, for any initial state v_0, we see

$$v_1 = Tv_0, \quad v_2 = Tv_1 = T^2v_0, \quad v_3 = Tv_2 = T^3v_0,$$

and in general

$$v_t = T^tv_0.$$

Example 3.3.5. *Consider $G = C_3$ and a random walk starting at vertex 1. Here the transition matrix is*

$$T = \frac{1}{2}A = \frac{1}{2}\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

The first few steps of the random walk (as probability distributions) is then given by

$$v_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad v_1 = Tv_0 = \begin{pmatrix} 0 \\ 0.5 \\ 0.5 \end{pmatrix}, \quad v_2 = Tv_1 = \begin{pmatrix} 0.5 \\ 0.25 \\ 0.25 \end{pmatrix}, \quad v_3 = Tv_2 = \begin{pmatrix} 0.25 \\ 0.375 \\ 0.375 \end{pmatrix}.$$

Continuing in this manner we see

$$v_4 = \begin{pmatrix} 0.375 \\ 0.3125 \\ 0.3125 \end{pmatrix}, \quad v_5 = \begin{pmatrix} 0.3125 \\ 0.34375 \\ 0.34375 \end{pmatrix}, \quad v_6 = \begin{pmatrix} 0.34375 \\ 0.328125 \\ 0.328125 \end{pmatrix}.$$

If we do some more calculations, it will be apparent that $v_t \to (1/3, 1/3, 1/3)$ as $t \to \infty$.

However, there's a more elegant way to see this. We saw in Example 3.2.9 that a basis of eigenvectors for A (and therefore T) is $(1,1,1)$, $(1,-1,0)$, $(1,0,-1)$. These have, respectively, eigenvalues $2,-1,-1$ for A and therefore eigenvalues $1,-\frac{1}{2},-\frac{1}{2}$ for T. Hence if we take

$$S = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \end{pmatrix}, \quad S^{-1} = \frac{1}{3}\begin{pmatrix} 1 & 1 & 1 \\ 1 & -2 & 1 \\ 1 & 1 & -2 \end{pmatrix}$$

then

$$D = STS^{-1} = \begin{pmatrix} 1 & & \\ & -\frac{1}{2} & \\ & & -\frac{1}{2} \end{pmatrix}.$$

(Recall T can be diagonalized by taking S the matrix whose columns are a basis of eigenvectors—in this example S happens to be symmetric, so the rows are also eigenvectors, but this is not true in general.) Therefore

$$T^t = S^{-1}D^tS = S^{-1}\begin{pmatrix} 1 & & \\ & (\frac{-1}{2})^t & \\ & & (\frac{-1}{2})^t \end{pmatrix} S.$$

In particular,

$$v_t = T^t v_0 = S^{-1}D^tSv_0 = S^{-1}D^t\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = S^{-1}\begin{pmatrix} 1 \\ (-2)^{-t} \\ (-2)^{-t} \end{pmatrix} = \frac{1}{3}\begin{pmatrix} 1+2(-2)^{-t} \\ 1-(-2)^{-t} \\ 1-(-2)^{-t} \end{pmatrix}.$$

In particular, we see that as $t \to \infty$, $v_t \to (1/3, 1/3, 1/3)$.

Note that from the symmetry of C_3, we see we get the same result no matter where we start the random walk. This means that if we do a random walk, starting anywhere on this graph, after a reasonable number of steps, we are (essentially) as likely to be at any one vertex i as at any other vertex j. Put another way, in any random walk, we will (with proabability 1) be spending an equal amount of time at each vertex.

Contrast this to the case of the path graph on 3 vertices from Example 3.3.3. Even though the distribution for the random walk doesn't "stabilize," i.e. v_t does not have a well-defined limit, we can still say that at any time $t > 0$, we are twice as likely to be at vertex 2 as at vertex 1 or 3 (if we do not know if t is even or odd). This should be fairly evident, but we can formally do this analysis by averaging the v_t's (see Exercise 3.3.3). Said differently, for a random walk on the path graph on 3 vertices (starting at any vertex in fact), we expect to be spending twice as much time at vertex 2 as at vertices 1 or 3. This indicates that vertex 2 is more central than vertices 1 or 3, unlike the case of C_3 where every vertex is equally central.

As you might have guessed now, this idea can be used to define another notion of centrality: eigenvector centrality. To see how this limiting behavior of random walks is related to eigenvectors, suppose $v^* = \lim_{t\to\infty} v_t$ exists. Then

$$Tv^* = T\lim_{t\to\infty} v_t = \lim_{t\to\infty} Tv_t = \lim_{t\to\infty} v_{t+1} = v^*,$$

i.e., any limiting distribution v^* for the random walk (there can be more than one—think about disconnected graphs) must be an eigenvector for T with eigenvalue 1.

Some graphs, like the path graph of length 3, may not have limits for random walks starting at any vertex. This will clearly be the case for any bipartite graph or something like a directed cycle graph. However, one can still often make sense of the percentage of time p_i spent at vertex i in a random walk, and $(p_1, \dots, p_n)$ will then be an eigenvector with eigenvalue 1. We just illustrate this with a couple of examples.

Example 3.3.6. *Consider a star graph G on n vertices—say the hub is vertex n. (For $n = 3$ this is the path graph on 3 vertices, just with a different labeling of vertices.) This is a bipartite graphs. Take a random walk starting at vertex 1, say. I.e., $v_0 = (1, 0, \dots, 0)$. Then at time $t = 1$, we must travel to the hub. From there, we travel to one of the $n - 1$ "satellite" vertices, then back again to the hub, and so forth. Hence, for $t \geq 1$, we have $v_t = (0, \dots, 0, 1)$ if t is odd and $v_t = (1/(n-1), \dots, 1/(n-1), 0)$ if t is even. So on average, we expect to spend $p_n = 1/2$ of the time at the hub, and $p_i = 1/2(n-1)$ percent of our time at any other given vertex $i < n$.*

For the star graph, the transition matrix is

$$T = \begin{pmatrix} 0 & \cdots & 0 & \frac{1}{n-1} \\ \vdots & & \vdots & \vdots \\ 0 & \cdots & 0 & \frac{1}{n-1} \\ 1 & \cdots & 1 & 0 \end{pmatrix},$$

and one easily sees that

$$v = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} = \begin{pmatrix} 1/2(n-1) \\ \vdots \\ 1/2(n-1) \\ 1/2 \end{pmatrix}$$

is an eigenvector for T with eigenvalue 1.

Example 3.3.7. *Consider a directed cycle graph G on n vertices, and a random walk starting at $v_0 = 1$. Then the random walk really has no randomness at all—it must be the infinite repeating path $(1, 2, 3, \dots, n, 1, 2, 3, \dots, n, 1, 2, 3, \dots)$. Consequently we spend $p_i = 1/n$ percent of our time at vertex i. Here the transition matrix equals the identity matrix*

$$T = A^T = \begin{pmatrix} 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & - & \ddots & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}$$

and we see

$$v = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} = \begin{pmatrix} 1/n \\ \vdots \\ 1/n \end{pmatrix}$$

is an eigenvector for T with eigenvalue 1.

Hence it seems reasonable to use the values of an eigenvector with eigenvalue 1 to define a centrality measure. For this we would like to know if 1 is always an eigenvalue, and if so, if the eigenspace is 1-dimensional. The following well-known theorem from linear algebra addresses this (though this is not typically covered in a first course on linear algebra).

Theorem 3.3.8 (Perron–Frobenius)**.** *Let G be a graph with transition matrix T. Then any (real or complex) eigenvalue λ of T satisfies $|\lambda| \leq 1$. In addition, $\lambda = 1$ is an eigenvalue and it has an eigenvector with all positive entries. The Furthermore, if G is undirected and connected, or directed and strongly connected, the eigenvalue $\lambda = 1$ has multiplicity 1.*

For the adjacency matrix A, there is a number $\rho > 0$, called the **spectral radius** *of A, such that ρ is an eigenvalue of A and any λ of A satisfies $|\lambda| \leq \rho$. The eigenvalue ρ has an eigenvector with all positive entries. If G is undirected and connected, or directed and strongly connected, the eigenvalue $\lambda = \rho$ has multiplicity 1.*

Note we have already proved this in the case that G is k-regular, as then $T = \frac{1}{k}A$, so $\mathrm{Spec}(T) = \{\frac{\lambda}{k} : \lambda \in \mathrm{Spec}(A)\}$ (cf. Proposition 3.2.10 and 3.2.12).

Due to time constraints, we won't prove this in general, but we'll just sketch out the ideas for T.

Suppose v is an eigenvector for T with eigenvalue λ, and the sum s of the entries of v is nonzero. We may scale v so $s = 1$. Then the calculation in (3.8) showing that T takes probability vectors to probability vectors means that the sum of the entries of $Tv = \lambda v$, which is λ, must also be 1.

To complete the first part of the theorem, it remains to show $\lambda = 1$ if $s = 0$. Let me just explain what to do if λ and v are real. Then some of entries are positive and some are negative. So we can write $v = v^+ + v^-$ where all entries of v^+ are positive and all entries of v^- are ≤ 0. Now one can use the fact that the sums of entries of Tv^+ and Tv^- are the same as the sums of the entries of v^+ and v^- (this follows as in (3.8)) to deduce that $|\lambda| \leq 1$.

If all eigenvalues $|\lambda| < 1$, then T must shrink the "size" of any v, i.e., $T^t v \to 0$ as $t \to \infty$. However this contradicts the fact that left multiplication by T preserves column sums. Hence we must have some $|\lambda| = 1$, and looking at v's with positive column sums will tell us in fact some $\lambda = 1$.

One can show that any eigenvector v for $\lambda = 1$ must have all nonnegative entries. Hence we can scale v to be a probability vector. Then v must essentially be the limiting distribution of some random walk. If G is (strongly) connected, then in any random walk we will spend a nonzero percentage p_i of our time at vertex i. One can prove then that $v = (p_1, p_2, \ldots, p_n)$.

Note that if G is not connected, e.g., a disjoint union of two copies of the cycle graph C_3, then $v_1 = (1/3, 1/3, 1/3, 0, 0, 0)$ and $v_2 = (0, 0, 0, 1/3, 1/3, 1/3)$ are both eigenvectors with eigenvalue 1. Consequently, so is any linear combination of v_1 and v_2. So $\lambda = 1$ will have multiplicity > 1 (it will be the number of connected components if G is undirected).

Eigenvector centralities

Definition 3.3.9. *Let G be a (strongly) connected graph and $v = (p_1, \ldots, p_n)$ the unique probability vector such that $Tv = v$. Then the* **random walk eigenvector centrality** *of vertex i is p_i.*

In other words, the eigenvector centrality of vertex i is the percentage of the time you spend at vertex i if you just wander around the graph randomly. We can explain how this reflects "centrality" as follows.

Let's think of a social network, like a Twitter graph, or a graph of webpages. Pretend you just start out somewhere random in the graph, and randomly visit other Twitter users or webpages on this network by randomly clicking users followed or webpage links. This is a random walk. The more popular a Twitter user or webpage is, the more it will be followed by or linked to from other nodes in the graph. Consequently, the more often you will visit it on this random walk.

However, this does not just give a measure of degree centrality, because the amount of time you spend at a given node will depend on the amount of time you spend at nodes that link to it. In other words the centrality of a given node is determined by not only the number of nodes linking to it, but by their centrality also. If you only have a few links to your webpage, but they're from Youtube, Wikipedia, the New York Times and Black People Love Us, you will get a lot more traffic than if you are linked to from 100 webpages the Internet has never heard of.

Another way to think of this is that each nodes links in the graph count as a sort of "popularity vote" for the nodes they link to. But these votes aren't democratic or limited—you can vote as many times as you want, however the more votes you make, the less each individual vote is worth. On the other hand, your votes count for a lot more if you personally are popular, i.e., if you got a lot of votes yourself.

For instance, let's say I tweet about graph theory, and my arch-nemesis Steve tweets about differential equations. I only have 3 followers, and Steve has 10, but none of our followers are too popular. Then, one day, Kanye West gets interested in social network analysis and decides to start following me. If Kanye only follows me and a few other people, I will get a lot of kickback traffic from Kanye's followers (though people tracing followers or from Kanye retweeting my brilliant graph theory tidbits), and I may get some more direct followers this way. Thus my popularity will blow Steve's to bits!

However, if Kanye follows 10000 people, almost no one will notice Kanye's following me, and he's not likely to often retweet my pith. So maybe Steve and I will stay at around the same level of popularity, until someone finds out his last name is Hawking, and he becomes super popular because people confuse him with a genius who can't walk. I don't know why everyone thinks Stephen Hawking is so smart anyway. His wheelchair does all the talking for him. Anyone could be a genius with that wheelchair, even Steve! Twitter's stupid, with stupid followers anyway. I don't need Twitter. I can make my own social network—a social network of one—and it won't have followers, just leaders. And no Steves allowed.

The probability vector $(p_1, \dots, p_n)$ with eigenvalue 1 (assuming the graph is stongly connected) can be interpreted as p_i representing the percentage of votes vertex i gets. This is for the following reason. Lets say we will vote in rounds for the centrality of a vertex, and a really large number of ballots can be cast. Start with some initial distribution $(p_1(0), \dots, p_n(0))$ of how important the nodes are—namely $p_i(0)$ represents the percentage of total votes (ballots) vertex i has to start with. At each round of voting, we revise the centrality (number of total votes) vertex v_i by giving i the percentage $p_i(t+1)$ of votes it just received. This is given by the rule

$$\begin{pmatrix} p_1(t+1) \\ \vdots \\ p_n(t) \end{pmatrix} = T \begin{pmatrix} p_1(t) \\ \vdots \\ p_n(t) \end{pmatrix}.$$

If this converges, it must converge to $(p_1, \dots, p_n)$. Even if it does not converge for our intial choice of $(p_1(0), \dots, p_n(0))$, if we *were* to take $(p_1(0), \dots, p_n(0)) = (p_1, \dots, p_n)$, then

$$\begin{pmatrix} p_1(t) \\ \vdots \\ p_n(t) \end{pmatrix} = T^t \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix}$$

for all t since this is an eigenvector with eigenvalue 1 for T. In other words, with this judicious choice of initial centralities $(p_1, \dots, p_n)$, voting does not change any nodes popularity at any step.

This means $(p_1, \dots, p_n)$ must be a solution to the recursively-defined problem: find a measure of centrality where each the centrality of each node is proportional to the centrality of the nodes linking to (or voting for) it.

This notion of eigenvector centrality is the basis for Google's PageRank algorithm, however we will need to address the issue that hyperlink graphs are not strongly connected.

This is all rather clever, but for undirected graphs, this turns out to be a little too clever, and we might want to modify this measure of centrality (it's still very useful for directed graphs, as evidenced by PageRank). The issue is that the value of node j voting for node i diminishes with the number of nodes vertex j votes for. Suppose we're in an undirected social network. Having a higher degree (e.g., Kanye) should make a node more important. And being connected to important people should make you more important. However, if you're connected to Kanye, but he's connected to 1,000,000 other people, this connection doesn't transfer much influence to you. In fact, this measure simply reduces to a measure of degree centrality. (This is less of an issue for directed graphs, where, say, Kanye probably has a lot more followers than followees.)

So we might like to allow nodes to be able to vote in such a way that their voting power does not get divided. We can do this by just using the adjacency matrix rather than the transition matrix, but now we will need to scale the values since Av does not typically have the same sum as v. Start with some initial probability vector $v_0 = (p_1(0), \dots, p_n(0))$ and iterate this by

$$v_{t+1} = \frac{1}{|Av_t|} A v_t,$$

where $|v|$ denotes the sum of entries in the vector v. This way v_{t+1} will still be a probability vector, and this process does not dilute the votes of nodes of high degree—each of Kanye's votes are worth more than each of mine, which was not true under the random walk process using T. Under some mild hypotheses, $v^* = \lim_{t\to\infty} v_t$ exists, and is an eigenvector for A with eigenvalue ρ, the largest positive eigenvalue. We'll see this below, but let's first state the definition and see a couple of examples.

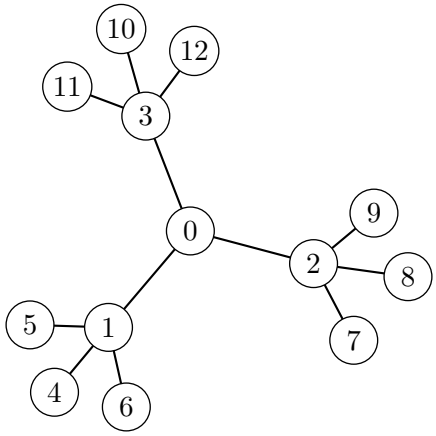
Definition 3.3.10. *Let G be a connected undirected graph with adjacency matrix A. Say ρ is the maximum positive eigenvalue of A, and $v = (p_1, \dots, p_n)$ is a probability vector which is an eigenvector for eigenvalue ρ. Then p_i is the* **(adjacency) eigenvector centrality** *of vertex i.*

The Perron–Frobenius theorem guarantees the existence and uniqueness of such a v. One can similarly define adjacency eigenvector centrality for directed graphs, but then it makes more sense to use the transpose A^T. Some authors normalize the eigenvector v so that the maximum entry is 1.

Example 3.3.11. *Let G be a k-regular graph. Then each vertex has both random walk and adjacency eigenvector centrality $\frac{1}{n}$. This is because $v = (1/n, \dots, 1/n)$ is an eigenvector for A, whence for $T = \frac{1}{k}A$.*

So for regular graphs, no nodes are more central than any other nodes—that is in any random walk we will *eventually* spend an equal amount of time at any vertex. This will happen sooner, say for complete graphs than for cycle graphs, and we will come back to this point as a way to measure how good network flow is.

Example 3.3.12. *Consider the tree G given by*



By iterating a random walk (and averaging), we can numerically find an eigenvector $v = (p_0, \dots, p_{12})$ of T with eigenvalue 1 given by

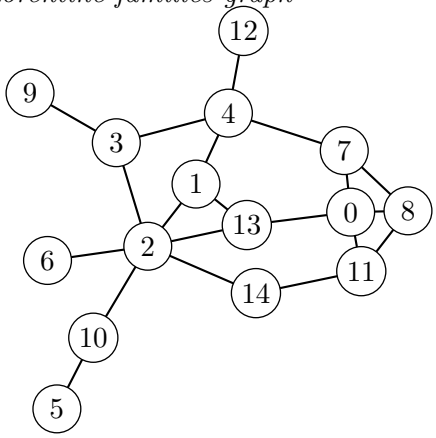
$$p_i = \begin{cases} \frac{1}{8} & i = 0 \\ \frac{1}{6} & i = 1, 2, 3 \\ \frac{1}{24} & else \end{cases}$$

and verify it is an eigenvector by matrix multiplication. In other words, the random walk eigenvector centrality of vertex i is just $\frac{deg(i)}{24}$. Note 24 is twice the size of G, i.e., the sum of all vertex degrees.

By iterating the process $v \mapsto \frac{1}{|Av|}Av$ (and averaging), we can numerically find an eigenvector $v = (p_0, \dots, p_{12})$ (which again we can verify afterwards) with maximium eigenvalue $\rho = \sqrt{6}$ to get the eigenvector centralities

$$p_i = \begin{cases} \frac{1}{10} & i = 0 \\ \frac{1}{5} & i = 1, 2, 3 \\ \frac{1}{30} & else. \end{cases}$$

Example 3.3.13. *Recall the Florentine families graph*



By iterating a random walk, we numerically compute the random walk eigenvector centralities in Sage as

```
[ 0.10000000000?]
[ 0.07500000000?]
[ 0.15000000000?]
[ 0.07500000000?]
```

```
[ 0.10000000000?]
[0.025000000000?]
[0.025000000000?]
[ 0.07500000000?]
[ 0.07500000000?]
[0.025000000000?]
[ 0.05000000000?]
[ 0.07500000000?]
[0.025000000000?]
[ 0.07500000000?]
[ 0.05000000000?]
```

Again, these numbers are simply the degrees of the vertices divided by the sum of all vertex degrees.

Similarly, by iterating $v \mapsto \frac{1}{|Av|}Av$*, we can estimate the adjacency eigenvector centralities in Sage as*

```
[ 0.1016327190110338?]
[ 0.0930282524780531?]
[ 0.1228533246839824?]
[ 0.0696496754352986?]
[ 0.0825427479718137?]
[0.0127942738390679 4?]
[0.0377301628845139 9?]
[ 0.0807396638222649?]
[ 0.0787212547936576?]
[0.0213904963971233 4?]
[ 0.0416594829674572?]
[ 0.0739521897453467?]
[ 0.0253501590935615?]
[ 0.0975135686688800?]
[ 0.0604420282079456?]
```

These are much more interesting. We see the most central nodes are the vertex 2 and vertex 0. We also note that vertex 1 (0.093...) has considerably higher eigenvector centrality than vertex 11 (0.073...), even though they both have degree 3, because the neighbors of 13 are more central than the neighbors of 11.

Computing stable eigenvectors

Now let's consider the problem of computing random walk and eigenvector centralities. We already seen that we can sometimes do this by looking at limits of random walks. Let explain this in more detail. Suppose G is graph with transition matrix T. Let $\{\lambda_1, \dots, \lambda_n\}$ be the (possibly complex) eigenvalues of T (with multiplicity). Assume for simplicity that (at least over $\mathbb{C}$), we have a basis of eigenvectors $v_1, \dots, v_n$ for T. (This is always the case if G is T-regular, and is true "most" of the time. When it is not, the conclusions we we make will still be valid, but one needs to modify the arguments below—e.g., by putting T in upper triangular form—see Exercise 3.3.4 for the case of $n = 3$.) Hence any $v \in \mathbb{C}^n$ can be written as a linear combination

$$v_0 = c_1 v_1 + \cdots + c_n v_n.$$

Then

$$T^t v_0 = c_1 T^t v_1 + \cdots + c_n T^t v_n = c_1 \lambda_1^t v_1 + \cdots + c_n \lambda_n^t v_n. \tag{3.9}$$

Since each $|\lambda_i| \leq 1$, these terms are bounded, and in fact $\lambda_i^t \to 0$ if $|\lambda_i| < 1$.

We know 1 is an eigenvalue of T—say $\lambda_1 = 1$. If all other eigenvalues are strictly less than 1 in absolute value (real or complex), then we in fact have

$$\lim_{t\to\infty} T^t v_0 = \lim_{t\to\infty} (c_1 \lambda_1^t v_1 + \cdots + c_n \lambda_n^t v_n) = c_1 v_1.$$

If v is a probability vector then, using the fact that multiplication by T preserves column sums, we must have $c_1 = 1$ if we scale v_1 so it's sum is 1. (The above limit implies that we can scale v_1 to get a probability vector.) In other words, $|\lambda_i| < 1$ for all $2 \leq i \leq n$, then $T^t v_0 \to v_1$ for *any* initial state probability vector v_0. That is, no matter where you start the random walk, the long term behavior is exactly the same. Furthermore, this method is typically very fast, even for really large networks, so iterating $T^t v_0$ allows us to numerically compute v_1 (and thus eigenvector centralities) very quickly. In this case v_1 is called a *dominant* or *principal eigenvector* because it dominates the limiting behavior of the random walk.* The existence of a dominant eigenvector is what make random walks an extremely useful tool for analyzing networks.

If the eigenvalue $\lambda_1 = 1$ occurs with multiplicity $m > 1$, say $\lambda_1 = \cdots = \lambda_m = 1$ but $|\lambda_i| < 1$ for $i > m$, then it is still true that $T^t v_0$ always has a limit, but it is no longer uniquely determined. Namely,

$$T^t v_0 = c_1 v_1 + \cdots + c_m v_m.$$

Hence the behavior of random walks can depend upon the initial starting position. The Perron–Frobenius theorem says this doesn't happen for (strongly) connected graphs. In fact the multiplicity m of $\lambda = 1$ will, for undirected graphs, be the number of connected components of G, and one obviously will get a different limit for random walks starting in different components.

Now assume G is (strongly) connected, so $\lambda_1 = 1$ occurs with multiplicity 1. Then the only issue in investigating these limits is that there may be other eigenvalues of absolute value 1. First, suppose for simplicity that G is undirected, in which case all eigenvalues of T are real. Then any other eigenvalue of absolute value 1 is -1—this happens if G is bipartite, but only with multiplicity 1. Assume the eigenvalues are ordered so that $\lambda_1 = -1$. Then we have

$$T^t v_0 \approx c_1 v_1 + (-1)^t c_n v_n$$

for large t as all other eigenvalue powers $\lambda_i^t \to 0$. This only converges if $c_n = 0$, otherwise it oscillates between 2 values: $c_1 v_1 + c_n v_n$ and $c_1 v_1 - c_n v_n$. However, averaging these values gives $c_1 v_1$, and if v_1 is scaled to have sum 1, then $c_1 = 1$ if v_0 is a probability vector. Thus we can still quickly numerically evaluate the unique normalized (so the sum of entries is 1) eigenvector v_1 of multiplicity 1 for bipartite graphs.

Now suppose G is directed. Then there could be many eigenvalues of absolute value 1. Recall the directed cycle graph from 3.2.6. Here $T = A$ and all eigenvalues have absolute value 1—e.g., for $n = 4$ they are ± 1 and $\pm i$. However it is the case that any eigenvalue of absolute value 1 must be a root of unity, i.e., $\lambda^m = 1$ for some m. This means there is a single m such that $\lambda^m = 1$ for all λ of absolute value 1. Consequently, when we try to take the limit of $T^t v_0$ it will (approximately, for t large) oscillate periodically amongst a finite number of states. Again, one can numerically recover the unique normalized eigenvector by averaging these periodic states.

*This is the basis of an important technique is statistics and financial mathematics called principal component analysis.

PageRank

The original version of Google's PageRank algorithm was published in a paper titled "The PageRank Citation Ranking: Bringing Order to the Web" in 1999 by Larry Page, Sergey Brin, Rajeev Motwani and Terry Winograd. It was originally designed as a way to find the most significant webpages linking to your webpage.

To do a web search, first one "crawls" the web. This is done by following links from known pages to find new ones, and indexing each webpage. (And in olden days, people and business manually submitted their pages for categorized listings in Yahoo like an online Yellow Pages. Let me explain—in historic times, there were things called books. They were made out of primitive substances known as ink and paper. Papers were thin rectangles cut out of trees. Ink is a liquid that comes out of sea creatures private parts, and I think also elephants in India. This ink gets sprayed on the paper in the shape of words, and then you glue a bunch of paper together to get something called a book. Glue is a sticky substance made out of dead horses that children squirt in their noses to keep their brains from falling out. Books were what contained all the information we knew before the Internet. A phone book was a kind of book, not made out of cell phones as one might first think, but a book with a list of phone numbers of every person and business in your village. The Yellow Pages were the part of the phone book where business were listed by categories—e.g., Blacksmiths or Cobblers. No one knows how these categories were arranged.)

Once a search engine has a reasonable index of the web, it can accept searches from users, compare those searches to the pages it knows, and return a list of the ones it thinks most relevant, hopefully with pages your more likely to be interested in at the top of the list. Prior to Google, search engines (e.g., Altavista and Lycos) would just compare the text of the webpages indexed to the text you entered in the search box. These would return results in an ordering based on similarities of the text entered and text on the webpage (e.g., if all of your search words were in the title or at the top of the page, or if your search words appears on the page many time or close to each other and in order, the webpage gets ranked higher).

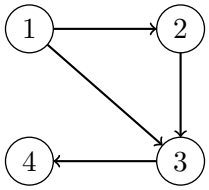
The idea behind PageRank is what the authors called the "random surfer model." It is a variant of the notion of a random walk. Consider a (directed) hyperlink graph. We want to model a "random surf" of the internet. In a random walk, we will start with a random webpage, and randomly visit pages by randomly clicking on links. There are a couple of issues with this. One, we might get stuck on a page with no links. This we can resolve by picking another page at random and starting over again. The more serious issue is that, as our graph is almost surely not strongly connected, we will get stuck in a small portion of the web. This means we won't converge to a meaningful distribution (as in random walk eigenvector centrality) on the nodes—it will be highly dependent on not just our starting node, but the actual choices we make in our random walk as well. Again, this can be resolve by randomly restarting our walk from time to time.

This is all accomplished with the following elegant conceptual idea. When people actually surf the web, they don't search forever (well, maybe in some sense they do, but they don't follow links without restarting forever). People sometimes click on links to continue visiting pages, but sometimes they will start a new "surfing session" at another page. Let's say α is the "restart probability"—the probability that a person starts a new surfing session at any given time. So $1-\alpha$ is the "click-through" probability—the probability of clicking on some link on a given page (i.e., continuing your surfing session).

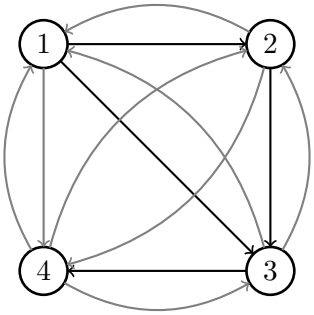
Our default restart probability will be $\alpha = 0.15$, so the click-through probability is $1 - \alpha = 0.85$. This is the value Google originally used—see the 1998 paper "The anatomy of a large-scale

hypertextual Web search engine" by Brin and Page, where they explain their original version of the Google search engine." There $1 - \alpha = 0.85$ is called the damping factor d.

Note we can model this random surfing with restarts as a random walk on a *weighted* directed graph. Here the weights will denote relative probabilities of taking an edge. For instance, let's consider the directed graph



Now we form a complete weighted graph, where the new edges added are in gray.



At each step, with probability $1 - \alpha$, we will travel along a black edge. With probability α, we randomly restart, which means with probability α we take any edge. (Technically, I should have included loops, because there is also the possibility we restart at the vertex we are already at.)

Definition 3.3.14. *Let J denote the all 1 matrix. The* **PageRank matrix** *R_α for G with restart probability α is*

$$R_\alpha = (1 - \alpha)T + \frac{\alpha}{n}J,$$

where T is the transition matrix.

Note the matrix $\frac{1}{n}J$ is the transition matrix for the complete graph K_n with a loop at each vertex. That is, using $\frac{1}{n}J$ for a random walk just means we pick a new vertex at random at each time t—there is no dependence on where we were at time $t-1$—the probability of being at vertex i at time t is $\frac{1}{n}$ for all i and all t. So R_α is a weighted combination two random processes: (i) doing a random walk on G and (ii) just picking a node completely at random at each step. We do (i) with probability $1 - \alpha$, and (ii) with probability α.

Example 3.3.15. *For the directed graph on 4 vertices pictured above, we have*

$$T = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0 & 0 \\ 0.5 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

so with $\alpha = 0.15$,

$$R_\alpha = (1-\alpha)T + \frac{\alpha}{4}J = \frac{1}{4}\begin{pmatrix} 0.15 & 0.15 & 0.15 & 0.15 \\ 1.85 & 0.15 & 0.15 & 0.15 \\ 1.85 & 3.55 & 0.15 & 0.15 \\ 0.15 & 0.15 & 3.55 & 3.55 \end{pmatrix}.$$

One can think of each (j,i) *entry of this matrix as the weight of the edge (or loop if* $i = j$*) from vertex* i *to* j*.*

One key advantage of working with R_α is that now all entries are positive (for $\alpha > 0$). We stated earlier the Perron–Frobenius theorem in our context—this was the version for matrices with non-negative entries. However the Perron–Frobenius theorem for matrices with only positive entries is actually quite a bit stronger—in our case it says the following.

Theorem 3.3.16 (Perron–Frobenius, positive matrix version)**.** *Let* G *be any graph (directed or not, simple or not, connected or not), and* $0 < \alpha \leq 1$*. The PageRank matrix* R_α *has one eigenvalue 1, and all other eigenvalues* λ *satisfy* $|\lambda| < 1$*. Furthermore,* $\lambda = 1$ *has an eigenvector with all positive entries.*

This means there is a unique probability vector $v*$ for such that $R_\alpha v^* = v^*$.

Definition 3.3.17. *Let* G *be any graph on* $V = \{1, \dots, n\}$ *and* $0 < \alpha \leq 1$*, and* $v^* = (p_1, \dots, p_n)$ *the probability vector such that* $R_\alpha v^* = v^*$*. Then the* **PageRank centrality** *(with respect to* α*) of vertex* i *is* p_i*.*

In other words, the PageRank matrix lets us define a centrality measure analogous to random walk eigenvector centrality for *any* graph, not just strongly connected ones.

For Google, what is key is that v^* can be computed efficiently. Since every other eigenvalue $|\lambda| < 1$ by this stronger version of Perron–Frobenius, as we argued above, v^* is a *dominant* eigenvector, in the sense that

$$v^* = \lim_{t \to \infty} R_\alpha^t v_0$$

for any initial state probability vector v_0.

Algorithm 3.3.18. (PageRank)

1. *Start with the initial state* $v_0 = (1/n, \dots, 1/n)$ *and a threshold* $\epsilon > 0$*.*
2. *Compute* $R_\alpha^t v_0$ *successively for* $t = 1, 2, \dots$ *until each entries do not change more than* ϵ*. Record the result as the (approximate) eigenvector* v^**.*
3. *Output* v^* *as the vector of (approximate) PageRank centralities.*

Some remarks are in order:

- In practice, for large graphs, one does not literally need to represent R_α as a matrix to do these calculations. Brin and Page originally considered 24 million webpages and Google indexes many many more now. Instead, one does the calculation with adjacency lists.

- Since the web graph is *sparse*, in that the number of edges grows roughly linearly in the number of nodes. (With about 24 million webpages, there were around 500 million links.) As long as α is not too small, the convergence of $R_\alpha^t v_0$ to v^* is quite fast. Google could take the maximum t to be between 50 to 100. (We'll discuss rate of converge of random walks in the next section.) In 1998, Google could perform do their PageRank in a few hours on a personal computer (and of course faster with more advanced hardware). Note: Google does not need to recompute PageRanks all the time, but just recompute them periodically as they update their index of the web and store the result for use in their search algorithm.

- There are some technical differences with actual original PageRank I have suppressed. For example, if one considers dynamic webpages (e.g., a Google search page), there is no limit to the possible number of webpages. These dynamic pages are essentially ignored when indexing the web. Also, Google dealt with nodes with out-degree 0 ("dangling links") separately. The actual version of PageRank used by Google constantly gets tinkered with to combat people trying to game the system, and the details of the version of PageRank actually in use are private (to make it harder for people to game the system). Companies created vast "farms" of essentially contentless webpages with links to businesses who paid money to these companies to get them increased web traffic, but now this strategy is not so effective. For instance, I heard that at some point (and perhaps still), getting a link from a page which was deemed unrelated to your page would actually lessen your PageRank score.

- PageRank is just one component of Google's search algorithm. In the original version, Google would index the words on each page and count their occurrence. In addition, Google would take in to account "anchor text"—the words on the pages linking to a given page (this allows one to meaningfully index pages without much text—e.g., images or collections of data—and in general provides better context for each webpage). Based on this, Google would give each page a similarlity ranking to your search query, and combine this ranking with the PageRank to give you an ordered set of results. Google started outperforming the popular search engines like Altavista right away (verified by the fact that you probably never heard Altavista, though I used to use it a lot). Note that in practice, one does not need to compute these similarity rankings for all webpages dynamically—Google can get away first with just checking the webpages with high PageRanks against your query, and also storing results for common queries. (I do not know how this is actually implemented, these are just my speculations on how Google returns search results so quickly—I'm fairly certain Google must implement some form of the first speedup I suggest, but I don't know about the second.)

- The term "link analysis" is used for web search algorithms that use the structure of the network (i.e., what pages link to what pages). Google seemed to be the first use of link analysis for web searching, but there are other algorithms. For example, around the same time another link analysis algorithm called HITS by Jon Kleinberg was proposed, where the idea is to classify webpages on a given topic as either hubs (jumping points for more information) or authorities (providing detailed information). This algorithm is the basis for the search engine Ask.com.

- PageRank can be used for ranking other things as well—e.g., sports teams within a conference. One forms a win–loss matrix and applies PageRank to the associated graph. One can vary this by weighting wins with appropriated factors—for instance, by score differentials or how

recent the win was. These algorithms have done quite well in betting tournaments, such as March Madness.

Exercises

Exercise 3.3.1. *In Example 3.3.2, determine the set of possibilities for v_t for $t \geq 1$. When $t = 3$, compute the probability of each of these vertices.*

Exercise 3.3.2. *Fix $m \geq 0$. In Example 3.3.2, prove that the probability v_t lies in the fixed finite interval $[-m, m]$ goes to 0 as $t \to \infty$.*

Exercise 3.3.3. *For a random walk $(v_0, v_1, v_2, \ldots)$, let $\bar{v}_t = (v_0 + v_1 + \cdots + v_t)/(t+1)$. Compute the limit $\lim_{t\to\infty} \bar{v}_t$ (and show it exists) in the following cases (the answer does not depend on which vertex you start at, but you may start at vertex 1 for simplicity):*

(i) G is the path graph on 3 vertices;

(ii) $G = C_4$;

(iii) G is a star graph on n vertices (say vertex n is the hub)

Exercise 3.3.4. *Let T be a 3×3 matrix with column sums 1. Suppose $T = S^{-1}BS$ for 3×3 (possibly complex) matrices B, S where B is of the form*

$$B = \begin{pmatrix} 1 & a & b \\ & \lambda & c \\ & & \lambda \end{pmatrix},$$

and $|\lambda| < 1$.

(i) For any vector $v \in \mathbb{C}^3$ show $v^* = \lim_{t\to\infty} B^t v$ exists and satisfies $Bv^* = v^*$.

(ii) Deduce that for any probability vector v_0, $\lim_{t\to\infty} T^t v_0$ exists and equals the unique probability vector v^* such that $Tv^* = v^*$.

3.4 The spectral gap and network flow

In computing random walk/adjancency eigenvector or PageRank centralities, we approximated a dominant eigenvector by looking at the convergence of some kind of random walk. This is only an efficient procedure if the convergence is fast.

Let's think about two simple examples: C_n and K_n. Take a random walk starting at vertex 1. Then, as long as n is odd for C_n (otherwise C_n is bipartite), the random walk converges to the "flat" distribution $(1/n, \ldots, 1/n)$. How fast will these things converge?

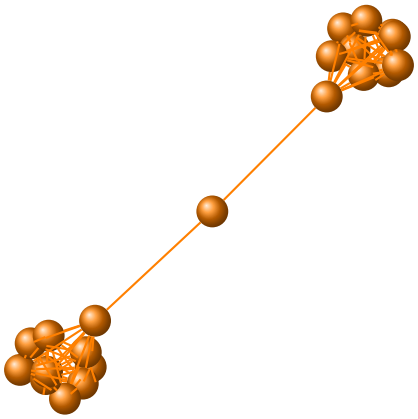
For C_n, it is not even possible to get to every vertex before time $t = (n-1)/2$. However at this point, you are much more likely to still be closer to vertex 1 than one of the vertices diametrically opposite it. So it will take even more time for this distribution to even out.

On the other hand, for K_n, already at $t = 1$ you are equally likely at any vertex besides 1, i.e., each vertex besides 1 had probability $1/(n-1)$. At $t = 2$, the probability of going back to 1 is simply $1/(n-1)$. For any other vertex i, the probability of being there at $t = 2$ is the probability you weren't there at $t = 1$ (which is $1 - 1/(n-1)$) times $1/(n-1)$, i.e., $1/(n-1) - 1/(n-1)^2$. Thus, already at $t = 2$ all of these probabilities are close to the limiting probability of $1/n$, at least for n reasonably large.

In other words, the time required for convergence (say for each coordinate to get within some threshold $\epsilon > 0$ of the limiting value) grows (at least) linearly for C_n as $n \to \infty$, whereas it is bounded for K_n. (In fact, for K_n, convergence is *faster* for larger values n.) Remark: the fast convergence for K_n is what ensures the PageRank algorithm converges quickly if α is not too small—since with probability α we are essentially doing a random walk on K_n (with loops).

We can think of the meaning of this time required for convergence as the amount of time required for a random walk to even out—or, put another way, the time required for the random walk to "forget" it's initial state. The random walk will even out faster, the faster you can get from the initial vertex to every other vertex in the graph. However, the speed of converges is not just determined by distance/diameter. In order for this convergence to be fast, there should be many short ways to get from one vertex to another.

For instance, consider a graph G on $2n+1$ vertices formed as follows: take two disjoint copies of K_n's and a vertex v_0, and connect v_0 to each K_n with a single link.



This graph has diameter 4, no matter how large n is. Now do a random walk, say starting at some vertex in the first copy of K_n. If n is large, it will take a long time to get to the "bridge vertex" v_0. First one has to get to the vertex in K_n connected to v_0, which on average happens one every n steps. From this vertex, the chance of going to v_0 is $1/n$, so we expect it will take about n^2 steps to get to the hub. From the hub, it will now go to each copy of K_n with equal probability. But the point is this argument shows it will take on the order of n^2 steps for the random walk to get reasonably close to the limiting distribution.

The issue is that if there aren't many paths from one part of your graph to another, then it will take a long time for a random walk to spread out evenly over these parts. In fact, one can also do a similar argument with a diameter 2 graph—connect v_0 to every other vertex in the example above. Then a similar argument says it will take on the order of n steps for a random walk starting in one of the K_n's to approach it's limit. In this graph, there are in some sense many paths from one K_n to the other, but they all travel through the "hub" v_0, and the chances of getting to the hub are small (around $1/n$) at each step. So a better conclusion to draw is that if there're aren't many *disjoint paths* from one part of a graph to another, then the random walk will be slow to converge.

Turning this idea around, we see that if the random walk converges quickly, there should be many disjoint ways to get from any one part of the graph to any other part of the graph. In addition, these should also be short paths. (Think of the cycle graph. Or, in our example of 2 K_n's connected by v_0, think about extending the distance between the K_n's by inserting many extra vertices on the links from each K_n to v_0. Now it will take much longer to get to v_0—this time is not just extended by the number of vertices added to this path because with high probability there

will be considerable backtracking, and possibly one will first return to the K_n one came from. Cf. Exercise 3.4.1.)

These were precisely the criteria we wanted our communication/transportation networks to have—having small diameter means the network is efficient. Having many short paths between two points means that the network is still efficient even if some nodes/links fail (which implies good vertex/edge connectivity), and rerouting can be done to handle traffic issues. In short, we can summarize this as saying: (strongly/connected) networks where random walks converge quickly have good *network flow*.

Now let's think about we can measure this notion of network flow—in other words, the rate of convergence of the random walk—more precisely. For simplicity, let us assume we are working with a graph G such that the transition matrix T has a basis of eigenvectors $\{v_1, \dots, v_n\}$ and v_1 is a dominant eigenvector v_1—i.e., if $\lambda_1, \dots, \lambda_n$ are the eigenvalues, then $\lambda_1 = 1$ and $|\lambda_i| < 1$ for all $i \geq 2$. This happens "most of the time" (with high probability for random (strongly) connected graphs). In particular, we know it happens for regular, non-bipartite graphs (cf. Section 3.2). Let v_0 be the initial state vector for a random walk. Write

$$v_0 = c_1 v_1 + \cdots c_n v_n.$$

Then

$$v_t = T^t v_0 = c_1 v_1 + c_2 \lambda_2^t v_2 + \cdots + c_n \lambda_n^t v_n.$$

Writing $v^* = c_1 v_1$ (the limiting value of v_t's), we know

$$\lim_{t \to \infty} (v_t - v*) = \lim_{t \to \infty} c_2 \lambda_2^t v_2 + \cdots + c_n \lambda_n^t v_n = 0. \tag{3.10}$$

Asking for how fast v_t converges to v^* is the same as asking how fast this quantity goes to 0. This is equivalent to asking the rate at which the λ_i^t's go to 0 (for $i \geq 2$). This is simply determined by $|\lambda_i$. The smaller each $|\lambda_i$ is, the faster the term $c_i \lambda_i^t v_i \to 0$. If we order the λ_i's so that $1 = \lambda_1 > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n| \geq 0$, then the rate at which the expression in (3.10) goes to zero is determined by the most dominant term, $c_2 \lambda_2^t v_2$, i.e., by $|\lambda_2|$. (It might be that $|\lambda_i| = |\lambda_2|$ for some other i, but then term i still go to zero at the same rate as term 2, and all other terms go to zero faster, so it suffices to look at $|\lambda_2|$. Also note that ordering of eigenvalues is not necessarily unique, but this again is not important for us.) Hence the smaller the second largest (in absolute value) eigenvalue $|\lambda_2|$ is, the faster the random walk converges, and the better the network flow.

Definition 3.4.1. *Let G be a graph (directed or undirected, weighted or unweighted, simple or not) with transition matrix T. Order the eigenvalues of T so that $1 = \lambda_1 \geq |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n| \geq 0$. The number $1 - |\lambda_2|$ the* **(random walk) spectral gap**.

The quantity $1 - |\lambda_2|$ is called the spectral gap, as it is the size of the gap between the largest and second largest (absolute value of) eigenvalues of T, and from the point of network flow/random walks, λ_2 is the most important eigenvalue. Note if the spectral gap is 0, i.e., $|\lambda_2| = 1$ (which happens, say, if G is not (strongly) connected or if G is bipartite) this means a typical random walk will not converge to a unique distribution. As long as the spectral gap is > 0, i.e., $|\lambda_2| < 1$, then the random walk converges to a unique dominant eigenvector, and the spectral gap measure the rate of convergence—the larger the spectral gap (i.e., the smaller $|\lambda_2|$), the better the network flow.

Example 3.4.2. *Let $G = K_n$. From Example 3.2.9, we know that the eigenvalues of $T = \frac{1}{n-1}A$ are $\{1, -\frac{1}{n-1}, \dots, -\frac{1}{n-1}\}$. Hence the spectral gap for K_n is $1 - |\frac{1}{n-1}| = \frac{n-2}{n-1}$. This is large (getting closer to 1 for large n), and of course this has the best network flow possible for simple, undirected graphs.*

Example 3.4.3. *Let $G = C_n$. From Example 3.2.8, we know the second largest (absolute) eigenvalue of $T = \frac{1}{2}A$ is $|\lambda_2| = 1$ if n is even and $|\lambda_2| = \cos(2\pi/n)$ if n is odd. Hence the spectral gap is 0 when n is even or $1 - \cos(2\pi/n)$, which is close to 0 for n large, when n is odd. This agrees with our observations that small spectral gap should mean poor network flow.*

Thus we can compare the network flow in different graphs by looking at the spectral gap (for large graphs, one way to estimate is by looking at how fast random walks converge). But if we want to actually construct a network with as good of a flow as possible, it would be nice to have a good theoretical upper bound on the spectral gap. One is at least known in the case of regular graphs.

Theorem 3.4.4 (Alon–Bopanna)**.** *The spectral gap for* any *k-regular graph G on n nodes satisfies*

$$|\lambda_2| \geq \frac{2\sqrt{k-1}}{k} - \epsilon(n,k)$$

i.e., the spectral gap

$$gap(G) \leq 1 - \frac{2\sqrt{k-1}}{k} + \epsilon(n,k)$$

where for a fixed k, the error $\epsilon(n,k) \to 0$ as $n \to \infty$.

Fix $k \geq 2$. In other words, for n large, the best we can do for the spectral gap is about $1 - \frac{2\sqrt{k-1}}{k}$. (In fact a more precise bound is given by Nilli involving the diameter of G.) This motivates the following definition

Definition 3.4.5. *We say a k-regular graph G on n nodes is* **Ramanujan** *if the second largest (absolute) eigenvalue λ_2 satisfies*

$$|\lambda_2| \leq \frac{2\sqrt{k-1}}{k}$$

i.e., if the spectral gap is

$$gap(G) \geq 1 - \frac{2\sqrt{k-1}}{k}.$$

The Alon–Bopanna theorem says that this is optimal for n large. This terminology was introduced by Lubotzky–Phillips–Sarnak in 1988, who explicitly constructed infinite families of Ramanujan graphs for $k = p+1$ where p is a prime of the form $4x+1$ by making use of the something in number theory known as the Ramanujan Conjecture (which has been already proven). Their construction is somewhat technical so we will not describe it here. Since, other constructions has been made, but it is not known if infinitely many Ramanujan graphs exist for all k. (When $k = 2$, the bound above is simply that $|\lambda_2| \geq 0$, which is automatic, so cycle graphs, and unions thereof—these are the only 2-regular graphs—are automatically Ramanujan.)

One might wonder if random regular graphs are likely to be Ramanujan, or "almost Ramanujan" in the sense that the spectral gap is not too far from optimal. This seems reasonable from the point

of view of the Alon–Bopanna theorem, which says that as n gets large, k-regular graphs get closer and closer to being Ramanujan. Indeed, Alon conjectured that "most" random regular graphs are almost Ramanujan (for any $\epsilon > 0$, most random k-regular graphs satisfy $|\lambda_2| \leq \frac{2\sqrt{k-1}}{k} + \epsilon$), and this has been proved by Friedman.

Let me just mention another notion of graphs which make good networks and has been well studied in mathematics and computer science. This is the notion of **expander graphs** (whose definition makes sense for non-regular graphs as well). Technically, individual graphs are not defined to be expander graphs, but a family of graphs is. (Similar to the way it doesn't make sense to define a single graph to be a random graph.) I won't give the precise definition, but the idea is that an expander family is a sequence of graphs G_n with increasing orders n (these graphs need not be subgraphs of each other) such that the network flow is good as $n \to \infty$. Here the notion of network flow is defined by a geometric parameter, called expansion, rather than a spectral one in terms of eigenvalues. Roughly, the expansion parameter measures the percentage of nodes that are neighbors of vertices in S as S ranges over different subsets of vertices. For instance, suppose you know that for any pair of vertices in G, the union of their neighbors contains ever other vertex in the graph. This implies the diameter is at most 2. A consequence of the definition is that in a sequence of expanders G_n, the number of edges grows at most linearly in n while the diameter grows very slowly—logarithmically in n. In other words, the networks maintain low cost yet are efficient.

Here is the first explicit construction of expanders.

Example 3.4.6 (Margulis, 1973)**.** *Let $V_n = \{(i,j) : 0 \leq i,j < n\}$. Define G_n on V_n as by the rule: the neighbors of (i,j) are defined to be $(i+j,j)$, $(i-j,j)$, $(i,i+j)$, $(i,i-j)$, $(i+j+1,j)$, $(i-j+1,j)$, $(i,i+j+1)$, $(i,j-i+1)$. Here these operations are taken mod n so the results always lie between 0 and $n-1$—e.g., if $i+j > n$ we subtract n, or if $i-j < 0$ we add n. This gives us a family G_n of 8-regular graphs, which is a family of expander graphs.*

If a family of graphs G_n has spectral gaps that don't get too small (e.g., don't go to 0), it will be a family of expanders. In particular, any sequence of Ramanujan graphs (with growing orders) will be a sequence of expanders. However it is possible that in a family of expanders the spectral gap goes to 0. It is known that if one takes a sequence of random regular graphs, then with very high probability, it will be a family of expanders (in fact with good expansion).

Expanders have many applications in mathematics, computer science and engineering. Detailed surveys can be found in the *Bulletin of the AMS*, "Expander graphs and their applications" by Hoory, Linial and Wigderson (2006), or the 2003 course notes by Linial and Wigderson of the same name (available online), which have a strong computer science flavor. Applications to circuit design, error-correcting codes and analysis of randomized algorithms are discussed. There is also a more recent 2012 *Bulletin* article by Lubotzky, "Expander graphs in pure and applied mathematics" with more of a bent towards pure mathematics (e.g., number theory and group theory). These surveys are all quite long and technical, and require more mathematical sophistication than we have assumed in these notes (sorry, these are the main introductory references I'm familiar with, and the subject nature itself is quite technical), but may be worth a glance if you're interested in getting a sense of some non-obvious applications, and want more meat than the Wikipedia entry.

Exercises

Exercise 3.4.1. *Let P_n be the path graph on $V = \{1, 2, \dots, n\}$. Consider a random walk on P_n starting at vertex 1.*

(i) Let p_t denote the probability that we reach vertex n for the first time at time t. Compute this (for all t) for $n = 3, 4, 5$.

(ii) For $n = 3, 4, 5$, compute the expected amount of time it will take to reach vertex n— this is given by $\sum_{t=1}^{\infty} p_t t$. (If you can't evaluate this sum exactly, use a computer to estimate it numerically.)

(iii) How do you think the expected amount of time to reach vertex n grows with n (e.g., linearly, quadratically)?

Exercise 3.4.2. *Verify that K_n is a Ramanujan graph.*

Chapter 4

This is not the chapter you're looking for [handwave]

The plan of this course was to have 4 parts, and the fourth part would be half on analyzing random graphs, and looking at different models, and half on other issues in social networks such as diffusion (disease spread, or information) and graph partitioning (as with the Karate club). Well, we just finished the third part with one lecture left, so needless to say we won't get that far. I hope to reteach this course again in the future as a year-long course, and at that point I can expand and revise these notes further (and maybe actually include a proper reference section).

Here's just a little taste of things.

4.1 Two random results

To analyze random graphs, one typically proves (or at least heurstically argues) that some property is true for most graphs, possibly by proving it in the limit.

Proposition 4.1.1. *For $n \geq 4$, most graphs on n vertices are connected.*

This is a cute result motivated by a MathOverflow post I came across recently. Here "most" just means more than half, and it is valid if we are counting labelled graphs or unlabelled graphs. For $n = 1$ it is all graphs, and for $n = 2$ or $n = 3$ it is exactly half the graphs.

Proof. Suppose $G = (V, E)$ is disconnected. Then I claim the complement, $\bar{G} = (V, \bar{E})$, is connected. Here $\bar{E} = \{(u, v) : u \neq v, (u, v) \notin E\}$ is the set complement of E inside $V \times V - \Delta V = \{(u, v) : u, v \in V, u \neq v\}$. In other words, all edges of G are not edges off $\bar{G}$, and vice versa. Put another way, the adjacency matrix $\bar{A}$ of $\bar{G}$ is formed by changing all non-diagonal entries of A from 0's to 1's or 1's to 0's.

To prove the claim, we want to show any $u, v \in V$ lie in the same component in $\bar{G}$. If u, v are in different components in G, then they must be connected by an edge in $\bar{G}$, hence in the same component in $\bar{G}$. If they are in the same component in G, take $w \in V$ in a different component in G, so u and v are connected to w in $\bar{G}$, and again in the same component. This shows the claim.

If there are m disconnected graphs on n vertices (labelled or unlabelled, the argument is the same), taking complements gives m connected graphs (whose complement is disconnected, as the complement of the complement of G is simply G). Now we just observe that for $n \geq 4$, there

are other connected graphs, i.e., there are connected graphs whose complements are connected. We can, for example, just take the path graph on n vertices (exercise—check its complement is connected for $n \geq 4$). □

Proposition 4.1.2. *Fix $0 < p \leq 1$. In the random $G(n,p)$ model, the probability of having an isolated node goes to 0 as $n \to \infty$.*

Proof. Let G be a $G(n,p)$ graph. The probability a given vertex i is isolated is simply $(1-p)^{n-1}$. (Vertex i pairs with $n-1$ other vertices, and all of these pairs are non-edges with independent probability $1-p$.) If A_i denotes the event that vertex i is isolated, the probability that G has at least one isolated node is

$$P(A_1 \cup \cdots \cup A_n) \leq P(A_1) + \cdots + P(A_n) = n(1-p)^{n-1}.$$

But this goes to 0 as $n \to \infty$. □

This says that for large n, we don't expect isolated nodes.

These results hint at the idea that random graphs tend to be fairly well connected, at least if n is large. Indeed, one can prove under certain conditions on n, p statements like: almost all random $G(n,p)$ graphs have a *giant component*, or almost all random $G(n,p)$ graphs are connected, or almost all random $G(n,p)$ graphs have a clique of size m.

4.2 Spectral graph partitioning

Take a graph G with transition matrix T. Assume for simplicity that T has a basis of eigenvectors $v_1, \ldots, v_n$ with corresponding eigenvalues $\lambda_1, \ldots, \lambda_n$. As before, we order these so that $1 = \lambda_1 \geq |\lambda_2| \geq \cdots \geq |\lambda_n|$. Further assume $1 > |\lambda|_2 > |\lambda_3|$.

Take a random walk on G with initial state v_0. Write

$$v_0 = c_1 v_1 + \cdots c_n v_n.$$

Then the t-th state of the random walk is

$$\begin{aligned} v_t = T^t v_0 &= c_1 \lambda_1^t v_1 + c_2 \lambda_2^t v_2 + \cdots + c_n \lambda_n^t v_n \\ &= c_1 v_1 + c_2 \lambda_2^t v_2 + \cdots + c_n \lambda_n^t v_n. \end{aligned}$$

For t of moderate size, λ_i^t are really small for $i \geq 3$, so

$$v_t \approx c_1 v_1 + c_2 \lambda_2^t v_2. \tag{4.1}$$

There is a lot packed into this approximation. We've seen already that v_1 gives a centrality ranking of vertices and λ_2 measures the rate of convergence of the random walk, i.e., network flow. We might call λ_2 the second dominant eigenvector and v_2 the second dominant eigenvector. This is because λ_2 and v_2 are the second most important things (after $\lambda_1 = 1$ and v_1) in determining the behavior of the random walk. Namely $|\lambda_2|$ measures how fast the random walk converges, and v_2 tells us *how* it actually converges.

One way to see this is to look at

$$v_{t+1} - v_t \approx c_2 \lambda_2^t (\lambda_2 - 1) v_2. \tag{4.2}$$

So in some sense, v_2 is essentially a "directional derivative" of this random walk. (Apologies again for the conflicting notation of v_2 and v_t—v_2 is not v_t with $t = 2$. Remind me to change this in the next version of these notes.)

Example 4.2.1. *Let G be the path graph on 2 vertices. Then we can take $\lambda_1 = 1$, $\lambda_2 = -1$, $v_1 = (1, 1)$ and $v_2 = (1, -1)$. (Okay, here $\lambda_1 \not> \lambda_2$, but this is the simplest example possible, and I think it is somewhat illustrative anyway.) Any random walk will just flip back and forth between vertices 1 and 2. From the derivative point of view, we can think of*

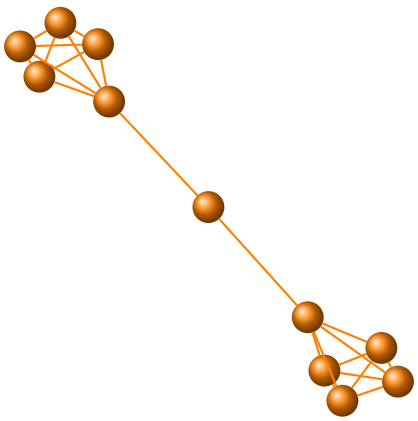
$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = c\lambda_2^t v_2 = (-1)^t c \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

as representing the approxmate change in v_t at time t. If $y_1 > 0$ then this means we're more likely to be at vertex 1 at the next step, and if $y_1 < 0$ this means we're less likely to be at vertex 1 at the next step. Similarly the sign of y_2 tells us if we're more or less likely to be at vertex 2 at the next step.

Now let's engage in a though experiment with the above example: suppose we had $1 > \lambda_2 > 0$. Then the signs of y_i's don't change at each step, and the random walk won't keep flipping back and forth between vertices. Say $y_1 < 0$ and $y_2 > 0$. This would mean that at every step we're getting more and more likely to be at vertex 2 and less likely to be at vertex 1. In other words, the random walk is getting "pulled" towards vertex 2.

This suggests that the signs of the entries of v_2 can tell us if a random walk is converging towards or away from from certain vertices. Note that the signs of the entries of v_2 are not determined uniquely, since $-v_2$ is also an eigenvector with eigenvalue λ. What will be important for us is which vertices have the same sign (assuming v_2 is real)—this is uniquely determined.

Consider the following graph, which is two copies of K_5 connected by a single vertex with two edges.



If we start a random walk on the left copy of K_5, it will wander around there for awhile, but it will gradually get pulled to the other side.

In Sage, I computed the transition matrix (the single "bottleneck" or "bridge" vertex in the middle is the last vertex in this ordering), the eigenvalues and dominant and second dominant eigenvectors.

Sage 6.1

```
sage: T
[  0 1/4 1/4 1/4 1/4   0   0   0   0   0 1/2]
```

```
[1/5   0 1/4 1/4 1/4   0   0   0   0   0   0]
[1/5 1/4   0 1/4 1/4   0   0   0   0   0   0]
[1/5 1/4 1/4   0 1/4   0   0   0   0   0   0]
[1/5 1/4 1/4 1/4   0   0   0   0   0   0   0]
[  0   0   0   0   0   0 1/4 1/4 1/4 1/4 1/2]
[  0   0   0   0   0 1/5   0 1/4 1/4 1/4   0]
[  0   0   0   0   0 1/5 1/4   0 1/4 1/4   0]
[  0   0   0   0   0 1/5 1/4 1/4   0 1/4   0]
[  0   0   0   0   0 1/5 1/4 1/4 1/4   0   0]
[1/5   0   0   0   0 1/5   0   0   0   0   0]
sage: T.eigenvalues()
[1, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, -0.2086308764964376?, 0.9586308764964376?,
-0.5319705149024926?, 0.2819705149024927?]
sage: ev = T.eigenvectors_right()
sage: ev[0][1] # dominant eigenvector
[
(1, 4/5, 4/5, 4/5, 4/5, 1, 4/5, 4/5, 4/5, 4/5, 2/5)
]
sage: ev[3][1] # next dominant eigenvector
[(1, 0.9586308764964376?, 0.9586308764964376?, 0.9586308764964376?,
0.9586308764964376?, -1, -0.9586308764964376?, -0.9586308764964376?,
-0.9586308764964376?, -0.9586308764964376?, 0)]
```

Here we see $\lambda_2 \approx 0.9586$ is large, i.e., the spectral gap is small, i.e., the network flow is bad, and this is due to the bottleneck as discussed in the last chapter. In the second dominant eigenvector v_2, all signs for the left hand K_5 are positive, while all signs for the right hand K_5 are negative, and the vertex in the middle has value 0.

This means in a random walk, when t is of moderate size so the behavior is dominated by v_2, we have $v_t \approx c_1 v_1 + \lambda_2^t c_2 v_2$. If we started our random walk on the left hand K_5, then c_2 will be negative, so the signs of the entries of v_2 will be negative for the left hand K_5 and positive for the right-hand K_5. This means the random walk is getting "pulled" towards the right hand K_5. That is, if we start on the left, for quite some time we are much more likely to be on the left, but we gradually become more likely to be on the right hand K_5. On the other hand, if we started on the right hand K_5, then c_2 would be positive and the random walk would get pulled to the left.

We can use this idea to partition graphs. Let's just think about the problem of bipartitioning, i.e., splitting partitioning a graph into two subsets V_1 and V_2. Roughly the idea is to partition so that the subgraphs corresponding to V_1 and V_2 should be well connected, but there shouldn't be too many connections between V_1 in V_2. Hence if we start a random walk in V_1, it will stay in V_1 for awhile, then eventually get pulled towards V_2, and vice versa. Thus if we look at the second dominant eigenvector v_2, the signs of the entries for the V_1 vertices should be (say) positive, while for V_2 will be (say) negative. If the value of the entry is 0 (or maybe close to 0), we can say that vertex is "on the fence." Note if we apply this algorithm to the above example of two K_5's connected to a single vertex, it does perfectly—the signs of the entries of v_2 partition the graph into 3 pieces: the two K_5's and the middle vertex which has value 0.

To close the semester, let's revisit the Karate club graph.

Sage's innate linear algebra functions are not so fast (Sage's focus is on precision, not speed), so we'll use `numpy`, a numerical method package for Python.

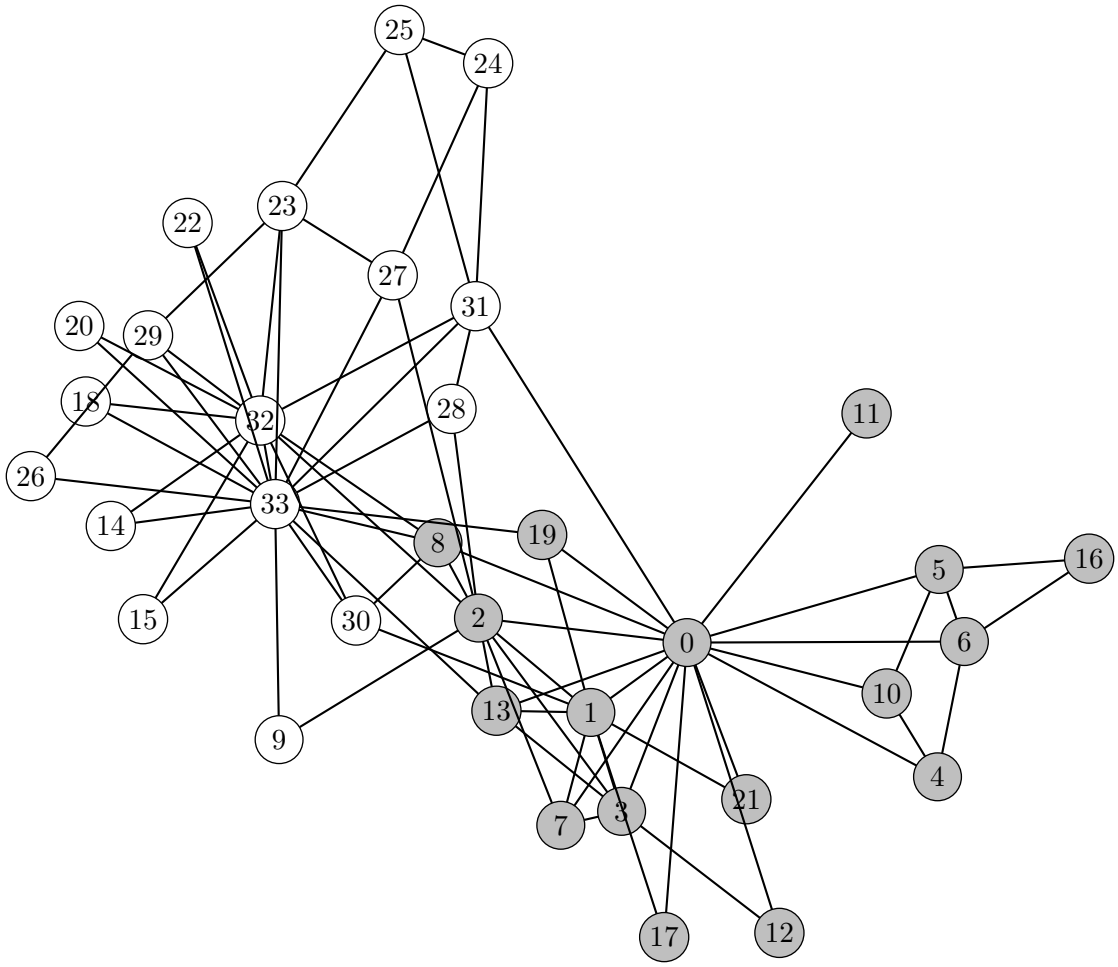


Figure 4.1: The Karate Club, actual split

Sage 6.1

```
sage: from numpy.linalg import eig
sage: w, v = eig(T)
sage: w # eigenvalues
array([  1.00000000e+00 +0.00000000e+00j,
         8.67727671e-01 +0.00000000e+00j,
         7.12951015e-01 +0.00000000e+00j,
         6.12686767e-01 +0.00000000e+00j,
        -7.14611347e-01 +0.00000000e+00j,
         3.87769460e-01 +0.00000000e+00j,
         3.51007053e-01 +0.00000000e+00j,
         2.92791798e-01 +0.00000000e+00j,
         2.60042011e-01 +0.00000000e+00j,
         2.29089383e-01 +0.00000000e+00j,
         1.77057148e-01 +0.00000000e+00j,
         1.35167055e-01 +0.00000000e+00j,
         9.31839984e-02 +0.00000000e+00j,
        -1.05380839e-01 +0.00000000e+00j,
```

```
        -1.59299956e-01 +0.00000000e+00j,
        -2.68023547e-01 +0.00000000e+00j,
        -6.11909588e-01 +0.00000000e+00j,
        -5.69506603e-01 +0.00000000e+00j,
        -3.51778259e-01 +0.00000000e+00j,
        -3.93104541e-01 +0.00000000e+00j,
        -4.16915851e-01 +0.00000000e+00j,
        -4.48579382e-01 +0.00000000e+00j,
        -4.97030113e-01 +0.00000000e+00j,
        -5.83333333e-01 +0.00000000e+00j,
        -8.49622483e-17 +0.00000000e+00j,
        -1.17901012e-16 +0.00000000e+00j,
        -5.32777279e-17 +0.00000000e+00j,
         3.97274886e-17 +1.75808521e-17j,
         3.97274886e-17 -1.75808521e-17j,
         3.63910315e-17 +0.00000000e+00j,
        -5.74878926e-18 +0.00000000e+00j,
         2.32016470e-18 +1.32924556e-17j,
         2.32016470e-18 -1.32924556e-17j,   1.52951722e-17 +0.00000000e+00j])

sage: v[:,0] # dominant eigenvector v_1
array([ 0.45958799+0.j,  0.25851825+0.j,  0.28724249+0.j,  0.17234550+0.j,
        0.08617275+0.j,  0.11489700+0.j,  0.11489700+0.j,  0.11489700+0.j,
        0.14362125+0.j,  0.05744850+0.j,  0.08617275+0.j,  0.02872425+0.j,
        0.05744850+0.j,  0.14362125+0.j,  0.05744850+0.j,  0.05744850+0.j,
        0.05744850+0.j,  0.05744850+0.j,  0.05744850+0.j,  0.08617275+0.j,
        0.05744850+0.j,  0.05744850+0.j,  0.05744850+0.j,  0.14362125+0.j,
        0.08617275+0.j,  0.08617275+0.j,  0.05744850+0.j,  0.11489700+0.j,
        0.08617275+0.j,  0.11489700+0.j,  0.11489700+0.j,  0.17234550+0.j,
        0.34469099+0.j,  0.48831224+0.j])

sage: v[:,1] # second dominant eigenvector v_2
array([ 0.48059815+0.j,  0.13792141+0.j, -0.01149982+0.j,  0.11431394+0.j,
        0.18758378+0.j,  0.28082530+0.j,  0.28082530+0.j,  0.07290804+0.j,
       -0.04788581+0.j, -0.03189334+0.j,  0.18758378+0.j,  0.03461614+0.j,
        0.05657271+0.j,  0.04233999+0.j, -0.06450152+0.j, -0.06450152+0.j,
        0.16181649+0.j,  0.05227675+0.j, -0.06450152+0.j,  0.02170870+0.j,
       -0.06450152+0.j,  0.05227675+0.j, -0.06450152+0.j, -0.17767894+0.j,
       -0.09509451+0.j, -0.10191495+0.j, -0.07308313+0.j, -0.10937613+0.j,
       -0.05632550+0.j, -0.14756600+0.j, -0.05787797+0.j, -0.12720276+0.j,
       -0.35334004+0.j, -0.45092074+0.j])
```

This says the second dominant eigenvalue $\lambda_2 \approx 0.8677$. By looking at the signs of the entries of v_2, we are led to the partition of the Karate club graph in Figure 4.2.

Note vertices 8 and 2 are classified as going with 32 and 33, rather than with 0 as actually happened. However this makes sense from an algorithmic point of view: 8 is friends with 30, 32 and 33 which, so will more likely go with those; then 2 is friends with 32, 28, 8 and 9 and may go with them. It in fact makes more sense for at least 8 to go with 33 than with 0, based solely on the information in the graph. (Factors that may have played a role, such as strength of ties, convenience and beliefs are not accounted for in this graph.)

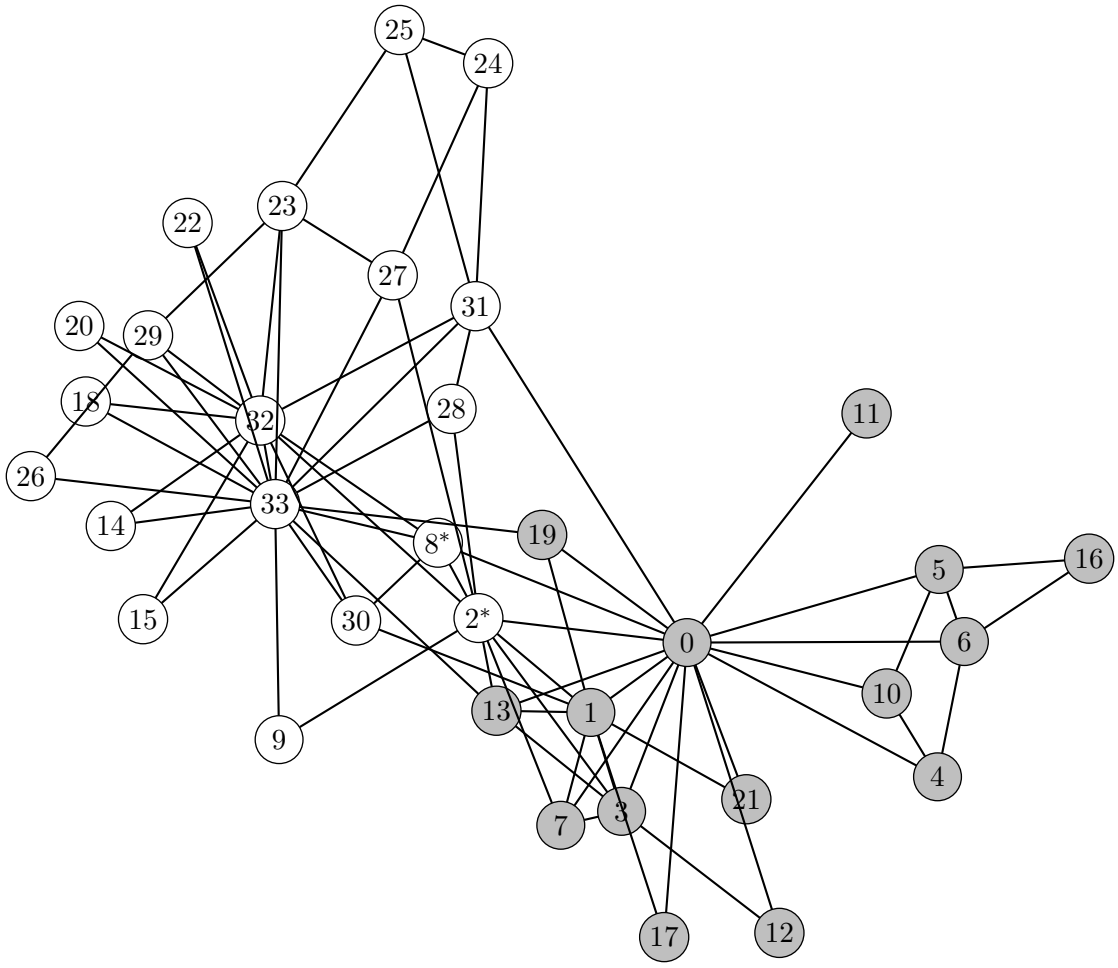


Figure 4.2: The Karate Club, spectral partition (starred vertices mispartitioned)

However, it is true that the values of v_2 for vertices 2 and 8, -0.0114... and -0.04788... are quite close to 0, so they're "on the fence" from this spectral partitioning point of view (along with some other vertices like 9 and 13).

This technique also falls under the heading of spectral clustering, as it divides the graph into 2 "clusters" (or more with repeated application). However, if $\lambda_2 < 0$, one won't get a partition into 2 clusters but into 2 whatever-the-opposite-of-clusters-are. Namely if your graph is close to bipartite, e.g., one can partition the vertices into V_1 and V_2 such that most connections are from V_1 to V_2, rather than within V_1 and within V_2, this spectral partitioning method will separate V_1 and V_2. If G actually is bipartite (and connected) with partition $V_1 \cup V_2$, then $\lambda_2 = -1$ and all entries of v_2 for V_1 will have the same sign, while V_2 will have the opposite sign, so this method will split V_1 from V_2, as in Example 4.2.1.

Well, that's all. Have a good life. If there's one thing you take away from this course, make it this: random walks are f**king awesome.