#h1 20 - Graphs and Networks

#h2 Nodes, Edges, Communities, and Ranks

#h3 20.1 - Introduction

#pg Graph theory deals with the structures, properties, metrics, and behaviors of interconnected systems. Graphs represent independent parts or elements of a system as nodes (also called vertices) connected to other nodes by edges. Almost anything can be a node (humans, cities, genes, Lego pieces, and words) which specifies the relationships between them, given uses edges that represent friendships, roads, interactions, relative location, and ordering). As a theory, it has been around for many years. Still, it is only recently that humans started producing enough data for the theory to show its practical applications and value. The mathematician Leonhard Euler is credited with publishing the first known scientific paper on graph theory in 1736. In it, he solved the seven bridges of the city of Konigsberg problem (interestingly, the first-ever textbook on graph theory was written by Denes König, published in 1936). At the time of Euler's publication, Konigsberg consisted of two large central islands connected and the adjacent mainland by a series of 7 bridges, as shown in Figure 20.1. The question Euler addressed is whether you plot a route through all parts of the city (the two islands and the mainland) by traversing each bridge once and only once. Euler proved this wasn't possible, and with this proof, he laid the foundation of graph theory.

#im ../assets/figures/020/020-01.png 50 256 Figure 20.1 - Historical map of Konigsberg, showing the location of the seven bridges referenced by Euler in his 1736 paper.

#h3 20.2 - Graph or network?

#pg We use the words graph and network somewhat interchangeably. One can argue that both have different meanings beyond what they have in common. Networks can refer to real physical systems, like the wires connecting hardware in an amazon data center, which we can abstract as graphs. And graphs can be things that can be entirely abstract, without any real-world counterpart. For example, we can imagine a graph of words and their relationships. Here, both nodes and edges are concepts, not physical entities.

#im ../assets/figures/020/020-02.png 50 256 Figure 20.2 - The internet was created in 1969 when the first connection between two physically remote computers was made. This connection between the computers of three universities in California is known as the ARPANET, of which the 1977 structure is shown here.

#h3 20.3 - Applications of graph theory

#pg We can find applications of graph theory everywhere. Social graphs have been at the forefront of the current interest in graph theory, especially since our online data is so heavily connected. Facebook is the graph of all their members (nodes) and whether they are 'friends' (edges). Twitter and Instagram followers are also connected through a system of followers. Exploring the graph structure can provide valuable insights into the flow of (mis)information, opinion, and polarization. It is worth noting that the application of graph theory to networks of connected people did not start with Facebook or Twitter. One of the most famous and classic datasets to first explore is Zachary's Karate Club dataset. The nodes in this small world example were members of a Karate Club, the edges between them whether they interacted outside the club. The data set is interesting because of a conflict between two individuals, an administrator, and an instructor. As a result, the club members fractioned into two groups, aligning themselves with either the administrator or instructor. See if you can see who those two individuals may have been in the data set in Figure 20.3.

#im ../assets/figures/020/020-03.png 50 256 Figure 20.3 - Zachary's Karate Club Graph. The nodes represent the Karate club members, and the links between them indicate whether the two members had any contact outside the club.

#br

#pg Even a deeper level, the internet itself is a massive graph of interconnected domains and websites through links. Early graphs described systems of actual flow, like electricity through circuits or water through a sewage system. More recently, these concepts provided by graph theory have been applied to abstract systems, like the global system of trade and logistics, and our understanding of highly complex interacting systems like our genetic code.

#im ../assets/figures/020/020-04.png 50 256 Figure 20.4 - The new MTA Subway Map (as of 2022)

#h3 20.4 - Erdos, Natalie Portman, Kevin Bacon, and six degrees of separation.

#pg You may be familiar with the Kevin Bacon game. If not, here is one of the many variations I have heard. Person A will offer two movies (or to actors) at random. Person B will try to connect the movies or actors through a sequence. Starting with a movie, we can think of an actor who played in it. That actor played in a different movie, in which other actors played as well. And those actors played in various movies themselves. The goal is to string these movies and actors together such that they connect the two movies given by Person A. But the twist is that they, at some point, must have Kevin Bacon listed as part of that sequence of movie-actor-movie. Doing it in the smallest number of steps while still including Kevin Bacon wins you the game. Here's a quick and straightforward example. The movies are Pretty Woman and The Matrix. Pretty woman had Julia Roberts in it, who starred in the movie Flatliners. Flatliners also starred Kevin Bacon, who had a part in the movie Mystic River, alongside Laurence Fishburne, who was Morpheus in the Matrix. This game is an excellent example of graph search and how, when playing the game, you might go down a path that leads nowhere, something algorithms that search through graphs will have to deal with. The idea of six degrees of separation is another graph-centric notion. It simply states that between you and any other person in this by six steps. I know person A, who knows person B, etc. Six degrees of separation states that there is a sequence of these steps starting with me that will lead me to anyone. With Facebook, this idea has been experimentally tested, and for most pairs, the average number of steps is 6.6. This number is close to the popular culture notion of about 6, although we should mention that the distribution has a long tail. For some pairs, the number of steps can be closer to 30 or more! Taking this idea a bit further is the Erdos number. Erdos was a very eccentric but prolific scientist and mathematician. He wrote hundreds of papers with many different collaborators and a good deal of them on the topic of graphs. Scientists in mathematics and physics often pride themselves on their Erdos number. Your Erdos number reflects how many steps you are separated from Erdos himself through your scientific publications. If you were Erdos yourself (there can be only one), your Erdos number is 0. If you have published with Erdos as a co-author, your number is 1. If you published a paper with someone who published an article with Erdos, your number is 2, and so on. As with the movies and actors, we know link publications with authors. There are websites where you can look this up, should you be curious. An interesting variant exists in my former world of cognitive and computational neuroscience. And we refer to it as your Natalie Portman number. Before committing to a career as an actress, Natalie Portman co-authored a paper while she was an undergraduate at Harvard, titled 'Frontal Lobe Activation during object permanence: data from near-infrared spectroscopy', published under her real name Natalie Hershlag in 2001. My Erdos number is 5, which isn't that high. My own Natalie Hershlag/Portman number is slightly better: 4.

#im ../assets/figures/020/020-05.png 50 256 Figure 20.5 - Paul Erdos, Kevin Bacon, and Natalie Portman.

#h3 20.5 - Toolbox Use

#pg Before we start, a quick note on the demo notebooks that come with this chapter. When we were writing this chapter and the accompanying demo notebooks, we had to make the same choice we had to make for most other technical chapters. Do we write our code for the algorithms described in this chapter? Or do we rely on a toolbox? For educational purposes, I would typically always opt to write completely transparent code, using a minimal set of toolboxes that hide some of the inner workings, especially when it comes to actual computations performed by the algorithm. We could have easily used the wonderful machine learning toolbox scikit-learn to demo the regression and classification algorithms we discussed in chapters 14 and 15. It would be clean code, properly refactored, efficient, optimal, peer-written, reviewed, and tested, all the things you want out of production-grade code. But you would have learned next to nothing about how these algorithms work.

#im ../assets/figures/020/020-06.png 50 64 Figure 20.6 - NetworkX is an excellent example of how a community of scientists, engineers, and developers can create and distribute excellent open-source code for anyone to use.

#br

#pg With graph algorithms, it is a little different. Graphs are a unique data structure and usually require a bit more code to operate on them. They are often recursive and, depending on the algorithm, use lists, dictionaries, sets, or arrays to store the graphs in memory. We feel that for some of the concepts we discuss in this chapter, we would spend more time on that (which is not the goal of this chapter) than on the actual inner workings of an algorithm that operates on the graphs. Therefore, we decided to implement only a few algorithms entirely from scratch. In doing so, we hope we will provide enough insight into how to work with graphs in a modern programming language while at the same time getting to the heart of the algorithms discussed. Specifically, we discuss the two original search algorithms, depth and breadth-first search and Google's famous PageRank algorithm. For all other algorithms that estimate centrality, detect communities, and predict links, we have chosen to use the python toolbox NetworkX. NetworkX is an excellent and well-documented library with graph algorithms. If you ever want to use graphs to derive insights into your data, we recommend installing and using them. In addition, several packages and applications exist that specifically aim to visualize graph structures. Furthermore, open-source and commercial software that implement graph databases are available, taking advantage of storing data in graphs rather than in table-like structures but instead as graphs. For example, consider the vast network of Facebook users and their online Facebook friends again. Querying various relationships and shared characteristics much fast and more efficiently than a traditional relational table-like database would be able to do.

#h3 20.6 - Graph essentials

#pg As stated above, at its core, a graph is a set of nodes (also called vertices), connected by a set of edges. Theoretically, a graph can have each node connected to all other nodes and even to itself (a self-edge or loop). In addition, node A can connect to node B, but node B doesn't need to be connected to node A. If the connection is always reciprocal, the graph is said to be undirected. Facebook is an example of an undirected graph: if person A is friends with person B, per definition, person B is friends with person A. Alternatively, the graph is said to be directed if the connections are not all reciprocal. The Twitter community is an example of a directed graph: person A can follow B without person B having to follow person A. Edges can represent some relationship ('friend') or quality ('is-a-subset-off') or can take on numerical values representing the strength or weight of the edge. In their most basic form, edges can be binary: simply 1 for edges present, implying that edges not present are represented by a weight of 0. Edges can also represent integer count ('number of items in common') or float, representing things like probabilities, physical units, or anything else of interest that quantifies the weight of the connection between two nodes. Numerical weights allow for some useful graph metrics and computations and offer a direct translation to algebraic and geometric approaches, of which PageRank is perhaps the most fundamental example at the time of writing. Many metrics describe the properties of single nodes, single edges, subsets of nodes, edges, subsets of graphs, and complete graphs. Below is a small sampling of the elementary ones.

#im ../assets/figures/020/020-07.png 50 256 Figure 20.7 - Core graph terminology. A graph is a set of nodes or vertices connected through edges or links.

#h3 20.7 - Degree

#pg The degree represents the number of edges a single node has. This combines incoming and outcoming edges if the graph is directed. Degree can be normalized by other metrics, like the number of possible connections a node can have or the mean number of edges across all nodes. If the graph is directed, we can compute both the indegree and outdegree of a node which count the number of incoming and outgoing edges of a node separately. We can also normalize per-node in and outdegrees and compute some metric that computes the difference or ratio between a node's in and outdegree. For example, in PageRank, it is assumed that a website of considerable importance probably has a high in-degree (many websites link to it) compared to a low outdegree (the site itself has only a moderate number of links going out from it unless it is a search engine like Google).

#h3 20.8 - Centrality

#pg Centrality is a measure of the importance of a node in the graph. The phrase ‘Importance’ here is loosely defined and depends somewhat on the context of the graph. But in general, we can imagine a node's centrality as measuring its usefulness in getting from any node to any other node in the graph. For example, the bigger subway stations in New York City have multiple subway lines that intersect them. Without these intersections, you can imagine getting from anywhere on the map to anywhere else becomes more difficult. In contrast, an isolated subway stop serving a single subway line does little towards this, except for the subway stops beyond it at the very end of the line. In the extreme, a subway stops with no active subway lines making stop makes absolutely no contribution (a centrality of 0). Defining the correct metric for centrality is not straightforward, any many of them exist with varying degrees of complexity and varying results. Here we list three common ones.

#im ../assets/figures/020/020-08.png 50 256 Figure 20.8 - Degree centrality computed for two synthetic graphs. Although the node at (0,0) in the right graph is essential to connecting any node to the left of it to the nodes on the right, and vice versa, it is nevertheless given a low centrality since it only has a low number of incoming and outgoing connections, compared to the graph on the left with a far more random structure.

#h3 20.9 - Degree centrality

#pg The most straightforward form of centrality uses the absolute degree count. In other words, the more a node is connected to other nodes, the higher its centrality.

#h3 20.10 - Betweenness centrality

#pg Like the MTA New York Subway system example, betweenness centrality is a metric of how often a node is part of any journey along the paths of the graph. More specifically, betweenness centrality measures the fraction of times a node is part of the shortest path between each possible pair of nodes. Note that this measure typically needs to be an approximation. Even for relatively small graphs, the number of possible pairs of nodes is large. Mathematical, the number of possible paths between any two nodes in a graph is factorial function of the number of nodes. This combinational explosion and the resulting computing challenges are true for this metric specifically and one of the more challenging aspects of graphs. By their very nature, the number of edges or paths increases extremely quickly when we add only a few nodes. The algorithms that need to operate on those edges thus often grow in computational complexity at a similar rate.

#im ../assets/figures/020/020-09.png 50 256 Figure 20.9 – In the above example, both degree- and betweenness-centrality measure rank the node in the center of the left synthetic graph highest. However, in contrast to the degree centrality shown in Figure 20.8, the betweenness centrality of the node at (0,0) in the right graph is the highest observed, mimicking our intuition that between centrality is more about node acting to bring disparate parts of larger nodes to each other. Graph topology can influence centrality metrics substantially.

#h3 20.11 – PageRank

#pg To be honest, Google's PageRank is much less of a scientific breakthrough than Silicon Valley wants you to believe. Its innovation was the application to an entirely new domain: the internet. Although undoubtedly, the current generation of PageRank is different than even most Google Engineers know, in its original form, it was a very straightforward algorithm to propose. However, in the original paper, it was outlined as follows:

#ns

#ne

#np Initialize each node in the graph to have a rank of 1

#np On each iteration, the PageRank of each node is updated as the summed proportional #bp PageRank of all its parents projecting onto it.

#np A small dampening factor is applied to introduce some randomness.

#np Repeat steps 2 and 3 until convergence: the change in PageRank across all nodes is deemed too small.

#ne

#pg Two things here require some unpacking. First, the concept of summed proportional PageRank. Each node can distribute its PageRank across its outgoing edges. For example, if node A connects to nodes B, C, and D, all receive 1/3 of the PageRank of A.

#im ../assets/equations/020/png/020-01.png 64 Equation 20.1 - PageRank formula. The updated PageRank PR of node i is some small dampening factor d normalizing the number of nodes n plus the complement of this dampening factor (1 - d) times the summed weighted PageRank PRj of all nodes n that have edges connect to PRi, excluding self-links. Weighted PRj indicates that the PageRank of node j is distributed across all its outbound edges, so node i receives a fraction of the PageRank at node j.

#br

#pg Therefore, we compute the PageRank of B, which only has an incoming edge from A, and the PageRank of B is thus the PageRank of A, divided by the number of outgoing edges A has. Second, there is the dampening factor. If a node has incoming edges but no outgoing edges, it will become a sink. It receives PageRank from other nodes but cannot redistribute it. Gradually, it will swallow up the available PageRank (remember that we started with all page ranks set to 1). To prevent this, a small portion of the available PageRank is redistributed equally across all nodes, independently of each node's PageRank. This dampening factor is typically set around 0.10-0.15. Equation 20.1 shows how to update the PageRank of node i on each iteration.

#im ../assets/figures/020/020-10.png 50 256 Figure 20.10 - Converge of PageRank algorithm. As an iterative algorithm, we must decide at what point the update in PageRank across nodes no longer outweighs the cost of running more iterations. In this example, the algorithm has sufficiently converged relatively early, around four iterations.

#br

#pg The first part is the equal redistribution of the PageRank across all available nodes. The second part sums the proportional PageRank of all incoming edges. Mathematically, it is easier to write we consider all nodes in the step while summing incoming PageRank, with the understanding that nodes that are not connected contribute 0 PageRank. In addition, a node does not receive its PageRank, as nodes cannot be connected to themselves. Figures 20.10 show two edge-case scenarios and a more realistic but small graph. The first edge case is a completely circular network. Here, we expect the PageRank to be distributed equally, as all nodes are equal. Computing the PageRank for these two edge cases validates our assumptions. Similarly, intuitively we feel that in the graph on the right, nodes 4 and 2 are of greater importance given their relative centrality in the network. The PageRank metric agrees with this observation. In the second case, the network is completely directed in a sequence of nodes. As a result, we expect most of the PageRank will end up at the end of the sequence at node 5. As a matter of fact, without a dampening factor, all PageRank would have ended up in this node. The dampening factor prevents this by redistributing a portion of it (determined by the dampening factor) back to earlier nodes.

#im ../assets/figures/020/020-11.png 50 256 Figure 20.11 - PageRank edge cases. (A) In a perfectly circular graph, each node is equally important. Indeed, the PageRank algorithm does converge on each node having the same value. (B) In a perfectly directed graph, the nodes are of increasing levels of importance. Where Node 2 only received input from Node 1, Node 5 can receive inputs from all nodes routed through the chained connections. This should and does give it the highest PageRank. (C) In an elementary 7-node graph, our intuition points to Node 2 as the most important and central, as it is the only node connected to all other nodes. PageRank confirms this intuition.

#h3 20.12 – PageRank Revisited

#pg Up to now, we have implemented the PageRank algorithm with no specific data structure for the graph in mind. We imagined we would have some access to all nodes, their neighbors, and the strength of their connection if applicable. With those assumptions, it was easy to compute PageRank iteratively using equation 20.1. However, when you want to use it to compute PageRank for individual nodes in a particularly large graph (something the original algorithm was supposed to do), the method becomes very computationally expensive since it requires having to constantly look up neighbors on nodes each time an individual node's PageRank is updated. The code becomes more prone to errors, and the computations are optimized for efficiency.

#im ../assets/figures/020/020-12.png 50 256 Figure 20.12 - Whether we compute PageRank using equation 20.1 or use the algebraic matrix approach, we will obtain (almost) the same result. The algebraic method is preferred as we can code it more concisely. More importantly, the matrix multiplication operations it uses can be optimized by specialized software and hardware, with dramatic increases in speed and efficiency.

#br

#pg Luckily, we can reframe the entire algorithm using our good friend algebra. The first step in this process is representing our graph as a matrix. An adjacency matrix, to be precise. An adjacency matrix of a graph G is a square matrix of n x n, where n is equal to the number of nodes n in the graph G. Each (i,j) entry reflects the existence of a (directed) edge between (i,j) if that value is >0. It can be 1, which merely indicates the presence of an edge, or some value representing the weights associated with the edge. The matrix is symmetric around its diagonal if the graph is undirected because, in this case, any edge (i,j) that is positive (an edge exists from node i to node j), there is an edge (j,i) (an edge from node j to node i).

#im ../assets/figures/020/020-13.png 50 256 Figure 20.13 - In the movie Good Will Hunting, a mathematical problem written on a blackboard is provided to students to solve, suggesting that it is a challenging problem that even stumps mathematicians. It is not. It is more or less the proof of equivalence between the iterative and algebraic implementations of PageRank we discussed. It asks the question of the probability of ending up on any node within a hypothetical graph, given some starting point on the graph and knowledge of the edges between nodes.

#br

#pg To find the PageRank for each node using the algebraic approach, we take advantage of a particularly interesting property of adjacency matrices. An adjacency matrix multiplied with itself yields a third matrix (also n x n) in which the original mass of the adjacency matrix has been spread according to the edges found in the adjacency matrix. This statement needs a little unpacking. Imagine that I start with my adjacency matrix, but I multiply it instead with a vector V of (n x 1) where all entries are zero, except for one entry where it is non-zero and large (say, 1000). Whereas the adjacency matrix A represents the structure of the edges of the graph, the vector V represents the 'activity' at each node of the graph. Imagine that that activity represents the behavior of hypothetical users on a website. In other words, our vector V creates 1000 hypothetical users on one website and none on any other. This website has many links, represented by the edges in adjacency matrix A. And here comes the clever bit: by multiplying matrix A with vector V, we create a new matrix A where all these 1000 hypothetical users have clicked on one of the links leading out to another website in our graph. If the probability of clicking on any outgoing link is the same across all links, we thus evenly spread out these 1000 users over the nodes that users can visit by clicking a link. But we don't have to stop here. We now iterate over multiplying A with itself to see whether all the 1000 users end up (statistically speaking) after two clicks, four clicks, and one thousand clicks. Suppose all nodes are of equal importance (i.e., should have equal PageRank assigned to them). In that case, this process will quickly settle into a pattern where the 1000 users are just distributed evenly across all nodes (i.e., websites). If some nodes are more central and have more than an average fair share of incoming links, users will find themselves more likely to navigate to these nodes. And this is, in the end, precisely what PageRank computes: if I let loose some arbitrary number of hypothetical users, where do they most often end up if they just click on links on the webpages and move in between them as a result? Formulated in this way, the users are called random walkers, and the model is a random walk model. We can simulate it using Agent-Based Modelling techniques we will come across in a few chapters. However, for this problem, we can use the algebraic approach of iteratively multiplying an adjacency matrix A of a graph G with itself using the resulting matrix as an estimate of the PageRank of the nodes in graph G. The notebook that comes with this paragraph will offer some additional and necessary detail on how to compute PageRank from these matrices, and as to incorporate the needed dampening factor as well. But in the end, this paragraph will hopefully give you a new way of looking at graphs, like Will Hunting when he solved the mathematical problems left on the hallway blackboard.

#h3 20.13 – Pathfinding and search

#pg A common question for any graph is whether any two nodes in the graph are connected, and if they are, how long the path is between them. And what is the shortest path length that exists between them? Many of the graph algorithms that implement this are recursive in nature and use queues or stacks to keep track of nodes they have visited, nodes still to visit, and in what order. The way that how nodes are being prioritized has a profound effect on how the search is carried out and what nodes we visit in what order. The two classic examples of these are breadth and depth-first search. Both algorithms will traverse all the nodes in the graph (taken for granted here that all nodes are reachable from the starting node. But both visit nodes are at very different times during their search. This difference can be an important consideration for large graphs, as the time to find a particular node can be a factor. Knowledge of the nature of connections can guide the approach taken if we use either breadth or depth-first search.

#h3 20.14 – Depth First Search

#pg In-Depth First Search, we initialize an array of Booleans, all set to False, one for each node in our data. This array keeps track of which node has been visited. We initialize our search placing our starting node on our stack. While there are still nodes in our stack, we take a node from our stack. If we are looking for a specific node, we can terminate the search early if we do come across it. But other use cases, such as determining the degrees of separation between our starting node and all other nodes: the number of steps each node is away from the starting node. Since we are using a stack, this is the last node added to it (LIFO - last in, first out). We set it to TRUE in our visited array and add all its neighbors to the stack as well, provided they have not been visited (set to False in the array). If we do not add new nodes, we will need to backtrack and take a node out from a previous depth (number of nodes between the current node and our starting node. From that point, we have another path forward to a greater depth, or we are forced to backtrack once again. As you can imagine, this process will eventually get us to backtrack all the way to the starting node. And once we remove that from our stack, the stack is empty, and the search stops. The figure below gives a better intuition, animating and color coding a depth-first search running. Depth-first search essentially tries to get as deep as possible as soon as possible along a single pathway. When that doesn't yield the node we are looking for, it backtracks just far enough to move forward again, hopefully finding the node there. Therefore, this search is useful if we suspect our target node is a reasonable distance (close to the maximum depth) away from our starting node.

#im ../assets/figures/020/020-14.png 50 256 Figure 20.14 - Depth (left) and Breadth (right) First Search. The numbers and color coding of the nodes represent in which order each node was visited by the Depth First Search Algorithm. The algorithms take different strategies traversing the graph while looking for a target node. What search method works best is a function of your graph topology and problem statement.

#h3 20.15 – Breadth-First Search

#pg In contrast, Breadth First Search (BFS) aims to exhaust the immediate surroundings before venturing further away from the starting node in terms of depth. As with DFS, BFS starts with an array of Booleans for keeping track of visited nods. But rather than a stack, the BFS uses a queue (FIFO - first in, first out). We initialize the algorithm by placing the starting node on the. While we haven't found the target node and there are still nodes in the queue, we take a node out of the queue, and set its status to visited in the array. We also add all its neighbors to the queue, provided they have not been visited yet. Since we use a queue, they are put at the back. When all neighbors have been added to the queue, we repeat the process of taking out another node. This process repeats until the target node is found or the queue is empty.

#h3 20.16 – Dijkstra's Shortest Path Algorithm

#pg While both BFS and DFS will find a path between two nodes if one exists, it is not guaranteed to be the shortest path. For this, we need a clever bit of innovation on the breadth-first search algorithm proposed by Dutch computer scientist Dijkstra. By picking a starting node, we can compute the shortest distance from that node to every other node. Note that Dijkstra's algorithm does expect edges to be weighted in some metric of distance between the two nodes it connects. Otherwise, it will yield results identical to an ordinary breadth-first search. In the initialization, we create an array of Booleans, but some numeric type identical to the numeric type used to specify the weights (integer, float, double). Initially, these are all set to Infinity. As we run the algorithm, we replace those values, likely more than once, with the latest shortest distance we have found. An additional array is created that will hold the ancestor of the node at the position. By ancestor, we mean the node visited before that node if we were to follow the shortest path between the current node and the starting node. We initialize this node with NONE (or NILL) values. As with BFS, we start by placing a starting node in the queue. At the same time, we set the value in the distance matrix to 0 (the shortest path from the starting node to the starting node is 0. Iterating until the queue is empty, we pop the first node in it and find the shortest edge leading out of it. This process is followed by the crucial part of the algorithm. We compare the distance of our current node plus the distance to the new node to the distance we might already have for that node stored in the shortest distance array. We are traversing a particular path that has led us to this new node. For our path, we know that distance: it is the distance we have already traveled plus whatever is still needed to get to this new node. If that distance is smaller than a previous path imping on this node, we have found a shorter path to this new node. In this case, we update the shortest distance and update its ancestor to our current node. If a new node has never been visited, that distance is still set to Infinity, guaranteeing an update. This process continues until all nodes have been visited, which is the same as saying that no more nodes are left with an initial distance of Infinity. How do we then get the actual path between two nodes? We do this by using the two arrays that are now fully specified. Starting with the target node, we look up what its ancestor is and take note of the distance between them. We then do the same for its pacesetter and add the distance, and so on, until we find the start node. We now have a total distance and a sequence of nodes that specify the shortest path.

#im ../assets/figures/020/020-15.png 50 256 Figure 20.15 - Dijkstra's algorithm to find the shortest path between LA and New York. It is obvious this isn't the shortest path, but instead, the way we created this graph from geographical data. We only allowed cities to be connected to a maximum of 6 of their closest neighbors. This mimics with some accuracy how US roads connect cities.

#br

#pg Dijkstra is an important algorithm that often finds itself embedded in larger search and optimization algorithms, most notably the wayfinding algorithms offered by our smart devices. However, it is also core to some other graph metrics, like betweenness centrality, where we count how often a node i is on the shortest path between two other nodes.

#h3 20.17 – Community Detection - Theory

#pg Communities in graphs and networks are nodes that show some level of interconnectedness amongst each other that is more pronounced than their connections with other nodes outside their community. However, their boundaries can be fuzzy and overlap to some degree. This makes them different from the components in graphs we discussed earlier, which, although connected, are cohesive. Online communities typically share some type of viewpoint or interest, and this is reflected in the graph. We might find that some websites share many outgoing and ingoing links. They primarily link to each other. A closer look at these websites might reveal they are all related to some human interest, like baseball, cosplay, or flower arranging. Community boundaries and membership usually are not all that well-defined, strict, and absolute. For example, even though politically, the USA is very much bipartisan, plenty of people consider themselves and their views not perfectly aligned with one side or another. Therefore, strictly speaking, they are not exclusively a member of either party, even though US politics often leads to 'voting down party lines'. In addition, community boundaries are usually fuzzy as well. Consider all the different post-war generations we are supposedly a part of, culturally speaking. People are labeled into generational boxes, such as boomers, Gen X, millennials, etc. Strictly speaking, your generation can only refer to your own next of kin. You are the third generation from your biological grandparents, and your children will be the fourth. The behavior of your peers is a much more significant influence on a child, especially from puberty onward into adolescence, and our lives are not neatly bucketed or segmented into generations before or after the smartphone, before or after MTV, or before or after TikTok, to give a few zeitgeisty examples. You could turn out to be a better example of a certain generation if you were born somehow right in the middle of some stretch in time during which a certain cultural aspect was very much present and had been for some time and continued to be present for a while after your birth. In that way, most of your peers would have similar cultural experiences. But for every person like that, another person finds themselves in a period of cultural flux and has peers on either side of the changes happening to their world. Finally, communities can exist and coexist at different scales. Ask anyone to define their identity ranked from most to least important, and you'll get a sense of what they consider to be communities they are part of and to what extent. People use to start with things like gender, race, nationality, religious belief, and political stance before they will offer what they see as their memberships to communities at much smaller scales, like liking pineapple on pizza, thinking Harry Potter is overhyped, and not caring for avocado on toast, not even for brunch Sunday.

#im ../assets/figures/020/020-16.png 50 256 Figure 20.16 - What communities do the different Muppets belong to?

#br

#pg All of this makes community detection a bit of a moving target. Similar to clustering, there aren't a lot of objective criteria we can use for community detection. How many different communities should we try to detect? How do we quantify membership? And how much do you allow communities to overlap? Or when they do, we should consider the union between the two as simply a third community. And ultimately, community as indicated by what aspect of affinity? Remember that in clustering, we were looking at how points would collapse together based on similarities between data points in some subset of the input variables. Let's take a dataset of demographic data on a group of individuals, like gender, race, age, income level, education level, life expectancy, etc. We will not be surprised if these combined variables show some clustering: your gender, race, and age are not independent of income level, education level, and life expectancy. In this city alone, some group together based on being predominantly white, affluent, highly educated with above average income and education levels on one side, and those who group because they are predominantly Black or Asian and are poorly educated, have low incomes and education levels—a sad but true statistic. But in graphs, communities are ultimately detected from preferentially connected nodes: nodes that 1) have a large probability of being connected in the first place, and 2) if they are connected, show a connection that is stronger with some nodes compared other nodes, and that this connection probability and strength is reciprocal. But that still begs the question of the probability that they are connected in the first place. Often, the connections are formed by easily observed signals that can be recorded at scale. For example, the number of words used in online conversation between two nodes representing two individuals. The leap of scientific faith is that this easily measured and recorded signal (number of words) can be a proxy for a more interesting construct, in our case of ‘friendship'. You can't measure friendship directly without in-depth psychological testing of any two individuals you consider potential friends. But perhaps the number of words is a good-enough stand-in signal: a proxy. Finding a proxy and making a case for it being a proxy is usually as much part of a community detection effort as is the computational work that goes into the algorithm.

#h3 20.18 – Community Detection - Algorithms

#pg After we are satisfied that the numerical value representing the strength of the connection between any two nodes is of some interest to us, we can apply a few community detection algorithms, all of which use slightly different criteria. Hopefully, they segment the graph into a set of (non)-overlapping communities based on various optimization and minimization methods. The very useful python module NetworkX offers a few of them, and a demo notebook is provided to apply these to a real data set and contrast their result. Although the algorithms use different approaches, they do have a similar optimization objective: modularity. The highest modularity we can get is when our graph perfectly segments into n number of communities, like in Figure 20.17.

#im ../assets/figures/020/020-17.png 50 256 Figure 20.17 - Community detection algorithms often optimize for modularity. Modularity captures how well our labeling of nodes captures the pattern of their connectivity. In this figure, we increase the separation between two communities of nodes. In the first graph, they overlap completely, and the nodes are not preferentially attached to nodes we designated as belonging to a single community. As we separate the two communities, their preferential attachment to each other aligns increasingly with their community labeling. Computed modularity for graphs A to C are 0, 0.22, and 0.5, respectively. 0.5 is the theoretical upper limit: maximum modularity = 1 / number of communities.

#br

#pg Here, the graph has two communities, A and B, which are interconnected but have absolutely no connection between each other. At the other extreme, our graph is fully connected, so no matter how we label each node as either A or B, there is no sense of a preferential attachment between the nodes (that is they have an edge between them). Here, the modularity is zero. Of course, we aim to find communities s in datasets where things aren't as clear-cut. In these cases, most community detection algorithms will have some method and a starting point to label and change the labels of nodes to increase the total amount of modularity in the graph based on the current labeling. The early methods followed a greedy approach, essentially iterating over every conceivable labeling and finding the best one. This, of course, becomes impractical for even a moderately sized graph, as the number of different ways we can label the nodes is not even exponential. It is factorial. Therefore, newer methods, like the Louvain community detection algorithm, use heuristics to guide their labeling to increase modularity as best they can. Figure 20.18 shows the results of a notebook in which we use both methods, as implemented by the NetworkX toolbox.

#im ../assets/figures/020/020-18.png 50 256 Figure 20.18 - Community detection for the Karate Club data set. The greedy algorithm only finds three separate communities, and the Louvain method finds 4.

#h3 20.19 – Link Prediction - Theory

#pg Join any social media platform and connect to a few people, and within mere moments the app will suggest other individuals you may want to consider connecting with. This is for a good reason: a message spreads more quickly if everyone has more connections they can share it with. We will leave it up to the reader to imagine what kind of messages certain key stakeholders would like to see spread. Outside of the why, how do we predict which two or more nodes in a graph should be connected if they are not? Predicting links in graphs can be useful beyond suggesting whose timeline we want to subscribe to on some social media platform. We can think of the relationship between concepts, ideas, and findings in a body of scientific theory and or literature as taking on the form of a graph. The nodes can be entities, like specific genes, or processes, while the edges can represent some relationship between these nodes. For example, in a graph representing the medical literature of breast cancer, we are likely to find the genes BRA1 and BRA2 having a link to a specific type of breast cancer, with the link representing something like 'can cause' (in one direction) or 'can be caused by' (in the other direction). Or, if we feel less certain about the causal relationship, we can at the very least have the link indicate 'correlates with' or 'statistically covary in certain populations. As you can imagine, these graphs, even in small areas of scientific inquiry, are vast and complicated. There are likely to be many nodes, not just many edges, but also many types of edges. Think of all the ways two entities can be linked. Like before, links can indicate scientific insight of two things being linked experimentally. Or they can indicate membership to some superclass. Imagine our graph already consists of 10 different genes that are 1) all associated with increased for cancer as well as 2) showing some epigenetic effect of gene suppression through methylation during early pregnancy. We observe an increased probably in a new gene for the same cancer. A link prediction algorithm can suggest this overarching common denominator of epigenetic effects to be verified, which will strengthen our understanding of the interplay between these three observables greatly. We could ask ourselves the opposite question of the above, which in some use cases can be preferred: of the links between nodes that currently exist, which can remove to optimize for some effect or quantity. For example, during infectious disease outbreaks, we can selectively close off certain roads between cities. We can model how we can best (temporarily) remove links (roads) between nodes (cities) to 1) reduce the spread of the disease itself while 2) not disrupt other important flows, like food and other resources. Similarly, in a highly connected network of computers, connections can be removed that do not affect the average speed of transferring data between computers yet still removes enough links (and the hardware needed to establish and maintain those links) to cut operational costs dramatically. Similarly, in brains, connections between nodes (read: neurons) are often pruned quite extensively during development. The neural machinery seems to come fully connected, and the pruning of unused or noninformative connections helps optimize the brain by reducing the metabolic cost (fewer connections require fewer recourses) and differentiating and thereby specializing groups of neurons.

#h3 20.20 - Link prediction - CCPA and Soundarajan-Hopcroft Algorithm

#pg Like community detection, clustering, and other unsupervised algorithms, the choice of link prediction algorithm isn't clear-cut, and many different methods have been proposed and used. We will present two algorithms that are both included in the NetworkX python toolbox, leaving exploration of other link prediction algorithms to you, the reader, should you have an interesting use case to pursue. The first algorithm is perhaps the most intuitive. It uses the Common neighbor and Centrality-based Parameterized Algorithm (CCPA) score, which is, in fact, pretty much self-explanatory. It combined a measure of the number of neighbors that two nodes have in common with a centrality score computed for the pair. Having common neighbors should be a good reason to assume two nodes are or should be connected. A closer look at equation 20.2 shows that the centrality score is a sort of betweenness centrality, as it factors in the distance (in the number of nodes) between the pair of nodes relative to the total number of nodes. Two nodes with high centrality will be connected through only a small number of intermediate nodes. In contrast, nodes with low centrality will take a larger number of intermediate nodes to get to. By taking the distance between the pair of nodes, we, in effect, estimate the weighted average of each node's centrality. If both a non-central, the resulting centrality between them is low. If both are central, the resulting centrality between them is high. And with one of them being highly central and one not at all, their combined centrality is somewhere in the middle. Figure 20.17 shows this algorithm applies to a synthetically created graph, with the edge thickness representing the likelihood of each edge existing, according to the algorithm.

#im ../assets/figures/020/020-19.png 50 256 Figure 20.19 - Link prediction. Two link prediction algorithms (left: Soundarajan-Hopcroft; right: CCPA) were used on the Karate club data set. They produce very similar results. The line width of the edges indicates the probability or score of each link, as predicted by each algorithm.

#br

#pg The second algorithm uses the same metric that, for obvious reasons, is central to link prediction: the number of neighbors two nodes have in common. But whereas the CCPA algorithm adjusted this using a measure of centrality, the algorithm put forward by Soundarajan and Hopcroft uses a prior estimate of community membership to refine the prediction of links. In practice, one can think of this further constraining the affinity of two nodes and their neighbors. Having similar neighbors but not belonging to the same community should make the link between two nodes less likely, and this is exactly with the Soundarajan and Hopcroft algorithm does. Figure 20.18 shows this algorithm applies to a synthetically created graph, identical to the one in Figure 20.17, with the edge thickness representing the likelihood of each edge existing, according to the algorithm.

#h3 20.21 - Conclusion

#pg Graphs have become increasingly widespread as ways of representing data. In no small part, this is because the internet, itself a massively connected network, generates data that is often best understood and analyzed as graph data. We have touched on a few algorithms central to graph theory, but many more performing similar or completely different tasks can be found in textbooks and online repositories. The field is still relatively young, and there are many ways to define and compute important aspects of graphs, like node centrality, similarity, or community membership. No real consensus exists or is probably even sought, and researchers will tailor their algorithms and metrics to the problem they are exploring. Graphs are unique data structures and require algorithms that are usually quite different than found elsewhere in data science. In addition, our modern tech stack isn't necessarily built with graphs in mind. Databases are still mostly row-based stores, and distributed algorithms to work on large graphs are still in their infancy. The connected nature of graphs makes them exciting and computationally a bit daunting. Every new node brings the possibility of many new edges, and these combinations do not scale gently.

#h3 20.22 - Notebooks

#pg Each section discussing an actual graph algorithm or metric has a notebook accompanying it, working through one or more examples. As such, you will find notebooks demoing Dijkstra, breadth and depth-first search, community detection, link prediction, and PageRank.

#h3 20.23 - References

#bs

#be

#bp Soundarajan, S. and Hopcroft, J (2012). Using community information to improve the precision of link prediction methods. In Proceedings of the 21st international conference companion on World Wide Web. ACM, New York, NY, USA, 607-608

#bp Ahmad, I., Akhtar, M.U., Noor, S. (2020). Missing Link Prediction using Common Neighbor and Centrality based Parameterized Algorithm. Sci Rep 10, 364.

#bp Zachary, W.W. (1977). An Information Flow Model for Conflict and Fission in Small Groups." J. Anthro. Research 33(4), 452-473 (1977).

#bp Springer International Publishing. (2016). Social network analysis - community detection and evolution.

#bp Page, Lawrence, Brin, Sergey and Motwani, Rajeev and Winograd, Terry (1999) The PageRank Citation Ranking: Bringing Order to the Web. Technical Report. Stanford InfoLab.

#bp Traag, V.A., Waltman, L. & van Eck, N.J. (2019). From Louvain to Leiden: guaranteeing well-connected communities. Sci Rep 9, 5233.

#bp Trudeau, R., 2015. Introduction to graph theory. New York: Dover Publications.

#bp Newman, M., 2010. Networks. Oxford University Press

#bp Blondel, Vincent D; Guillaume, Jean-Loup; Lambiotte, Renaud; Lefebvre, Etienne (2008). Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment. 2008 (10): P10008

#be