Lecture Summary

In this lecture, we will learn more about graphs, their terminology, and how to implement them. In particular, we will see the relative advantages of adjacency lists and adjacency matrices.

1 The Basics and Nomenclature

In the previous lecture, we briefly talked about some examples of graphs, and the very basics of what a graph is. As we study graphs in more depth, it is important to stress once more that graphs only capture pairwise relationships between entities. For example, suppose that you are trying to model a set of students (nodes), and whether they take classes together. There are three students, A, B, and C. If you have an edge between all three pairs (A, B), (B, C), and (A, C), that could mean that all three are in a class together, or that there are three different classes, one which A and B are taking, one which B and C are taking, and one which A and C are taking. A graph does not let you distinguish between these two real-world scenarios. If we really wanted to model the difference, there would be two different approaches.

1. One would be to move from graphs to hypergraphs: hypergraphs have hyperedges, which may contain more than two nodes. Then, a hyperedge of \{A, B, C\} could capture that all three students are in the same class together. In most of your standard college classes (including here at USC), hypergraphs are not covered in depth; the main reason is that there are a lot of very beautiful and useful results about graphs, and most extensions to hypergraphs aren’t nearly as useful or beautiful.

2. A different way would be to include in our graph two completely different types of nodes: students and classes. Then, we draw an edge between a student node and a class node if the student is enrolled in the class. We would not have any edges between students, or between classes. This would completely capture the scenario we are interested in. Such graphs (two different types of nodes, with edges only between nodes of different types) are called bipartite. They are often very useful, and there are many really interesting results about them. You may see some of them in CS170, CS270, and possibly higher-level classes.

In order to talk about graphs, we need to learn some terminology and agree on some notation:

- A graph consists of nodes (or vertices, singular: vertex) and edges between them.
- The set of all nodes is usually denoted by \(V\), and the set of all edges by \(E\). Their sizes are \(n = |V|\) and \(m = |E|\). In talking about graphs, the use of \(n\) and \(m\) in this sense is very standard, so you should avoid deviating from it. Graphs are often denoted as \(G = (V, E)\).
- Vertices are often named \(u, v, w\), edges \(e, e', f\).
- Edges can be directed or undirected. Undirected edges model relationships that are automatically reciprocated (such as being friends, siblings, married, in the same class, or connected with a 2-way street). Directed edges model relationships that are not necessarily reciprocated (such as having a crush on, being a parent of, being a supervisor of, or being connected with a one-way street). Directed edges are also frequently called arcs.

---

1 This list is of course far from complete: also read up in the textbook for some more terminology. You’ll also likely learn much more in CS170 and CS270.
A graph in which all edges are undirected is an undirected graph. A graph in which all edges are directed is a directed graph. A graph that contains both types is sometimes called a mixed graph.

To visualize graphs, we will often draw the nodes as circles (or points or rectangles), and edges as lines (not necessarily straight). When edges are directed, we draw them as arrows.

When there is an undirected edge \((u, v)\) between nodes \(u\) and \(v\), we call \(u\) and \(v\) adjacent (think: neighbors). When the edge is directed, people usually avoid the term “adjacent” to prevent confusion.

In an undirected graph, the degree of a node \(v\) (written as degree\((v)\), or \(\deg(v)\), or sometimes just \(d(v)\) or \(d_v\)) is the number of edges that \(v\) is part of. In directed graphs, we usually carefully distinguish between the out-degree (number of outgoing edges), denoted by outdegree\((v)\) (or \(d_{out}(v)\) or \(d^+(v)\)), which is the number of outgoing edges, and the in-degree (number of incoming edges), denoted by indegree\((v)\) (or \(d_{in}(v)\) or \(d^-(v)\)).

When you do some basic calculations (counting), you see that a graph with \(n\) nodes can have at most \(n^2\) (or \(n(n-1)\) or \(n(n-1)/2\)) edges, depending on whether we allow a node to have an edge to itself, and whether edges are directed or undirected. In any of these cases, the maximum number is \(\Theta(n^2)\). We call a graph with \(\Theta(n^2)\) edges dense, and a graph with much fewer than \(n^2\) edges sparse.\(^2\)

One of the most useful and central concepts in modeling the world with graphs is a path: it captures one’s ability to get from one node to another not just with one direct edge, but by following a sequence of edges. The formal definition of a path is a sequence of \(k \geq 1\) nodes \(v_1, v_2, \ldots, v_k\) such that for each \(i\), there is an edge from \(v_i\) to \(v_{i+1}\) (directed or undirected). In this sense, a path exactly captures our ability to traverse multiple edges in sequence. Notice that we explicitly allow the case \(k = 1\), i.e., a node by itself is also a path. Even though this path is often not particularly interesting, it usually helps us avoid having to deal with special cases in proofs.

As defined so far, a path may repeat the same node or edge multiple times. If we want to rule this out, we can talk about simple paths: a path is simple if it does not contain the same node more than once. A cycle (sometimes also called a circuit) is a path that starts and ends at the same node. We say that two nodes \(u\) and \(v\) are connected if there is an undirected path between them. (There are similar definitions for directed graphs, but those terms are used less frequently.) Thus, two nodes are connected if it is possible to get from one to the other.

An undirected graph is called connected if every pair of vertices in the graph is connected. A directed graph is called strongly connected if for every pair \(u, v\) of vertices, there is a directed path from \(u\) to \(v\) and a directed path from \(v\) to \(u\).

Paths between nodes are extremely useful in many settings. In road networks, they are the main concept we need for any kind of route planning (MapQuest, Google Map driving recommendations, GPS). In computer networks, they are where packets are routed from one computer to another. In social networks, they tell us whether you are connected to someone else by a chain of friends. You’ll probably find useful applications in most other examples of graphs we talked about.

## 2 The Abstract Data Type Graph

In implementing graphs, we want to again think of them as an abstract data type: the information that is stored (internally) as well as functions to interact with the data. Internally, we will need to store the nodes and the edges. In both cases, we may want to associate some data with them. For nodes, this could be some information on a user of a social network, or a computer in a network, or an intersection on a road map. For

\(^2\)For this class, the “intuitive” description will have to suffice. The attentive student comfortable with big-O notation will notice that it does not make sense to speak for a single graph about the number of edges being \(\Theta(n^2)\), since the notation of \(O, \Omega\) are only defined for functions. We’d really be talking about a family of graphs, including arbitrarily large graphs, and whose edge sets grow as \(\Theta(n^2)\). For “sparse graphs”, we will keep it deliberately underspecified. In different contexts, different definitions come in handy. Sometimes, we may just mean that the number of edges is \(\Omega(n^2)\) (which we denote by it being \(o(n^2)\) with a little \(o\)). In other contexts, we may actually want the number of edges to be \(O(n)\) or \(O(n \log n)\) or \(O(n^{1.5})\).
edges, this may be the length or direction of a road, or the strength of a friendship (measured in number of e-mails exchanged or something else), or the amount of money one bank owes another, or the cost of an airline flight. When such data become more relevant later in the course, we will return to them; for now, we’ll just focus on the basic graph structure, and keep in mind for the future that we’d often want to store more information at nodes or edges. The functions that the abstract data type should provide us are (at least) the following:

- adding and deleting a node.
- adding and deleting an edge.
- testing if a particular edge \((u, v)\) exists.
- for a node \(u\), listing all of its outgoing/incoming edges or all adjacent nodes.

The latter functions are particularly useful in the context of finding paths between nodes (as well as a few others, as we’ll see later). A good implementation for them would be via iterators, i.e., the function returns an iterator that will eventually output all outgoing edges. (A separate function would return an iterator over all incoming edges; for undirected graphs, we may instead want an iterator over all adjacent nodes.)

By using an iterator implementation, we can continue to hide from other classes how the abstract data type stores information about the graph internally, and will only reveal the minimum amount of information necessary.

2.1 How to Implement the ADT internally?

There are two natural ways to store a graph internally. In both cases, we would store nodes themselves in an array or perhaps a linked list or List or Bag.

1. Via adjacency lists: for each node, store a list of all other nodes it is adjacent to. (This could be using the ADT Bag, the ADT List, a Linked List, or a dynamically growing array.) If the graph is directed, we would store two separate lists: one for outgoing edges and another for incoming ones. The elements of the list would be either pointers to the other nodes, or — if nodes themselves are stored in an array — integers, i.e., the indices in the array where the other node is stored.

2. Via an adjacency matrix: Store an \(n \times n\) matrix (two-dimensional array) \(A\). In position \(A[u, v]\), store whether there is an edge from \(u\) to \(v\). (This could be a boolean value, or perhaps an integer storing the length, cost, or other information.)

What we store in position \(A[u, u]\) depends on what our graph is modeling. In some cases, it makes sense to treat it as though there were an edge from \(u\) to itself, in other cases not so much. There is no rule that forces us to choose one or the other, and we should use whatever makes our computational task easiest.

In comparing the two approaches, we should always take into account which operations we expect to be performed more frequently. Let us compare which operations are more or less expensive in the two implementations.

1. For adding or removing nodes, an adjacency matrix has a bit more overhead, since changing the size of a matrix involves a bunch of reallocation and copying. For many applications that a computer scientist is called upon to solve, changes in the set of nodes tend to be a bit rarer while other processing is happening.

2. Testing whether the edge \((u, v)\) exists in the graph is the specialty of adjacency matrices. It only involves one lookup in a known position of a 2-dimensional array, and thus takes \(O(1)\) time.

For an adjacency list, we don’t have that information readily available. Instead, we need to go through the list of all outgoing edges of \(u\), and see if \(v\) is in that list. That will take \(\Theta(\text{outdegree}(u))\), which can be much larger.
3. Listing all outgoing/incoming edges of $u$ is the specialty of adjacency lists. Since they explicitly store this information, they just have to go through their Bag/List, which takes time $\Theta(\text{outdegree}(u))$ or $\Theta(\text{indegree}(u))$.

For an adjacency matrix, we don’t have this information stored, so we have to loop through the entire row (or column) of the node $u$, and find all the entries that are 1 or true. This will take $\Theta(n)$, which can be much worse if the degrees are pretty small.

4. We can also compare the memory requirements of the two implementations. For the adjacency matrix, we need to store an $n \times n$ array, which obviously requires $\Theta(n^2)$ memory.

For adjacency lists, we store all incoming and all outgoing edges, which gives us a total memory requirement of $\Theta(\sum_v (\text{indegree}(v) + \text{outdegree}(v)))$. To evaluate this sum, notice that when we sum all in-degrees, we count each edge exactly once, since it contributes 1 to the indegree of exactly one node. So the sum of all indegrees is $m$, and similarly for the sum of all outdegrees. Thus, the sum above is equal to $2m$, which is $\Theta(m)$. Thus, the total storage requirement is $\Theta(n + m)$ (since we also need to store information for all nodes).

Thus, for sparse graphs, adjacency list are much more economical with space than adjacency matrices, whereas for dense graphs, it doesn’t really matter too much in terms of memory.

We saw above that each way of storing a graph has its own advantages and disadvantages. If memory isn’t too big of a concern (it rarely is — computation time is the bottleneck much more often than memory nowadays), one possible approach is to store the entire graph twice internally. When a user queries whether an edge $(u, v)$ exists, we answer it by using the adjacency matrix, while when the user queries all neighbors of a node $u$, we use adjacency lists.

The downside of this approach (besides using twice as much memory) is that it requires us to execute each update to the graph twice: once on the adjacency matrix and once on the list. Thus, update operations (such as adding or removing nodes or edges) will take twice as long as before. However, this may well be worth it: for many graphs, updates are much less frequent than queries. For instance, when building a GPS, it is much rarer to have roads or intersections added to a road network than to look for routes between two locations. Similarly, in a social network, users add/remove friends much less frequently than reading their friends’ updates. In those cases, the extra investment in updates is well worth it to have faster responses in queries.

As an aside for the interested student, this idea of doing more “pre-processing” during update time to speed up later operations at “query time” is an important and active direction in data structures. Many papers are trying to find useful internal representations to answer queries about connectivity or short routes faster than with the “default” representations we considered above.

As another aside, students familiar with linear algebra (hopefully most of you) will remember “matrices” as the central concept in that field. There are a lot of interesting things one can do with matrices, such as adding, multiplying, finding eigenvectors and eigenvalues, and more. It turns out that some of these have really useful meanings in computing stuff on graphs, and some of the most exciting research in graph theory these days is based on interesting computations with adjacency matrices (and slight variations of them). The topic is significantly beyond the content of this class, even though some easy examples are pretty directly accessible, and may even be covered in CSCI 170.

3 Trees: A brief look at next lecture

Trees are a special type of graph, and while graphs are really useful to model many phenomena in the real world that we want to solve computationally, trees are at the heart of many non-trivial data structures. There are two natural ways to think about trees:

1. We can think of trees as a special case of graphs, namely undirected connected graphs that do not have any cycles.
2. As a way to store data structures, a different viewpoint is often more useful. We saw linked lists as a structure in which each node of a list was pointed to by at most one other node, and pointed to at most one other node. In trees, a node is still pointed to by at most one other node, but it may point to multiple other nodes. We will see more next lecture.