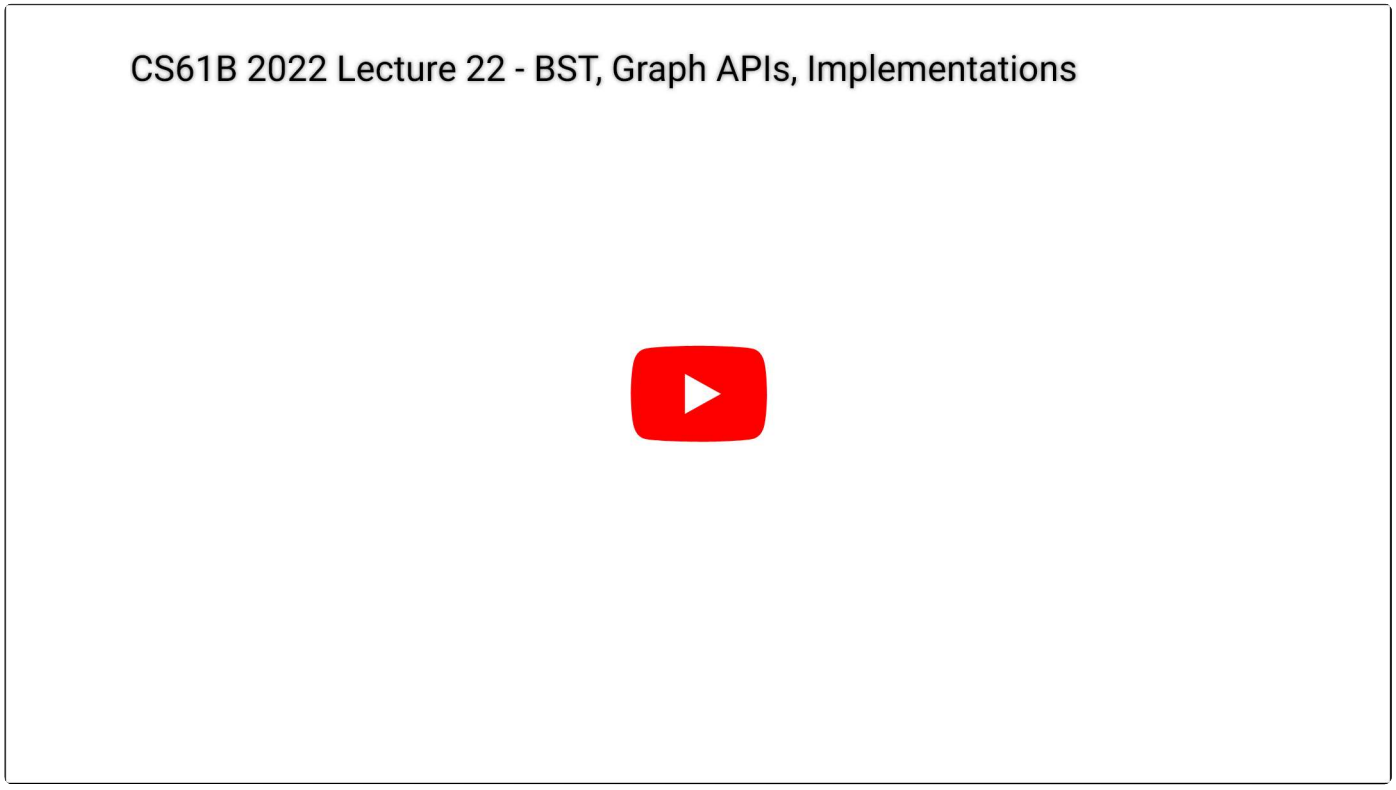
23.2 Representing Graphs

How do we create a graph in Java?



Professor Hug's Lecture on Graphs

We will discuss our choice of **API**, and also the **underlying data structures** used to represent the graph. Our decisions can have profound implications on our *runtime*, *memory usage*, and *difficulty of implementing various graph algorithms*.

Graph API

An API (Application Programming Interface) is a list of methods available to a user of our class, including the method signatures (what arguments/parameters each function accepts) and information regarding their behaviors. You have already seen APIs from the Java developers for the classes they provide, such as the Deque.

For our Graph API, let's use the common convention of assigning each unique node to an integer number. This can be done by maintaining a map which can tell us the integer assigned to each original node label. Doing so allows us to define our API to work with integers specifically, rather than introducing the need for generic types.

We can then define our API to look something like this perhaps:

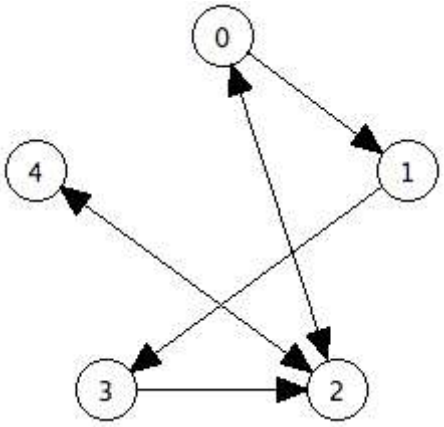
```
public class Graph {
  public Graph(int V):                // Create empty graph with v vertices
  public void addEdge(int v, int w): // add an edge v-w
  Iterable<Integer> adj(int v):      // vertices adjacent to v
  int V():                           // number of vertices
  int E():                           // number of edges
...
```

Clients (people who wish to use our Graph data structure), can then use any of the functions we provide to implement their own algorithms. The methods we provide can have a significant impact on how easy/difficult it may be for our clients to implement particular algorithms.

Now that we know how to draw a graph on paper and understand the basic concepts and definitions, we can now consider how a graph should be represented inside of a computer. We want to be able to get quick answers for the following questions about a graph:

- Are given vertices `u` and `v` adjacent?
- Is vertex `v` incident to a particular edge `e`?
- What vertices are adjacent to `v`?
- What edges are incident to `v`?

Imagine that we want to represent a graph that looks like this:

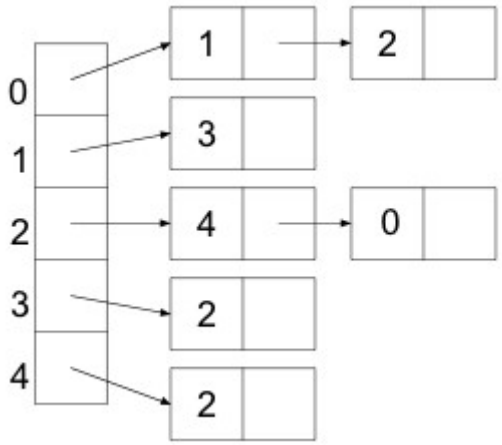


One data structure we could use to implement this graph is called an *array of adjacency lists*.

The Adjacency List

In an adjacency list, an array is created that has the same size as the number of vertices in the graph. Each position in the array represents one of the vertices in the graph. Each of these positions point to a list. These lists are called adjacency lists, as each element in the list represents a neighbor of the vertex.

The array of adjacency lists that represents the above graph looks like this:



Another data structure we could use to represent the edges in a graph is called an *adjacency matrix*.

The Adjacency Matrix

In this data structure, we have a two dimensional array of size N×N (where N is the number of vertices) which contains boolean values. The (*i*, *j*)th entry of this matrix is true when there is an edge from *i* to *j* and false when no edge exists. Thus, each vertex has a row and a column in the matrix, and the value in that table says true or false whether or not that edge exists.

The adjacency matrix that represents the above graph looks like this:

	0	1	2	3	4
0	F	T	T	F	F
1	F	F	F	T	F
2	T	F	F	F	T
3	F	F	T	F	F
4	F	F	T	F	F

Efficiency

Your choice of underlying data structure can impact the runtime and memory usage of your graph. This table from the slides summarizes the efficiencies of each representation for various operations. It is strongly not recommended to directly just copy this on to your cheatsheet for the exams without taking the time to first understand where and how these bounds fundamentally came to be. The lecture contains walkthroughs explaining the rationale in detail behind several of these cells.

Further, DFS/BFS on a graph backed by adjacency lists runs in O(V+E), while on a graph backed by an adjacency matrix runs in O(V^2). See the slides for help in understanding why.

Previous
23.1 BFS & DFS

Next
23.3 Summary

Last updated 1 year ago