⋮ **Representing graphs**

🍃 Unclaimed points ▾

Graph representation

Describing graphs

Quiz: Describing graphs

Representing graphs

Quiz: Representing graphs

Challenge: Store a graph

NEXT SECTION:
Breadth-first search

There are several ways to represent graphs, each with its advantages and disadvantages. Some situations, or algorithms that we want to run with graphs as input, call for one representation, and others call for a different representation. Here, we'll see three ways to represent graphs.

We'll look at three criteria. One is how much memory, or space, we need in each representation. We'll use asymptotic notation for that. Yes, we can use asymptotic notation for purposes other than expressing running times! It's really a way to characterize *functions*, and a function can describe a running time, an amount of space required, or some other resource. The other two criteria we'll use relate to time. One is how long it takes to determine whether a given edge is in the graph. The other is how long it takes to find the neighbors of a given vertex.

It is common to identify vertices not by name (such as "Audrey," "Boston," or "sweater") but instead by a number. That is, we typically number the $|V|$ vertices from 0 to $|V| - 1$. Here's the social network graph with its 10 vertices identified by numbers rather than names:
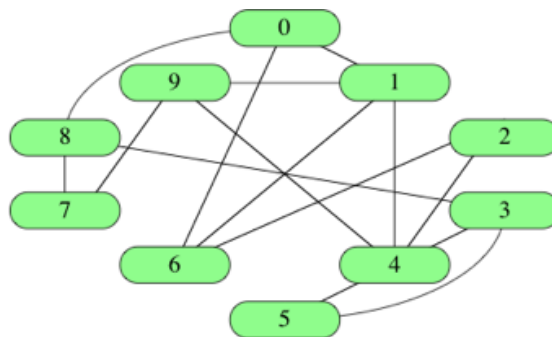
Edge lists

One simple way to represent a graph is just a list, or array, of $|E|$ edges, which we call an **edge list**. To represent an edge, we just have an array of two vertex numbers, or an array of objects containing the vertex numbers of the vertices that the edges are incident on. If edges have weights, add either a third element to the array or more

information to the object, giving the edge's weight. Since each edge contains just two or three numbers, the total space for an edge list is $\Theta(E)$. For example, here's how we represent an edge list in JavaScript for the social network graph:

```
[ [0,1], [0,6], [0,8], [1,4], [1,6], [1,9], [2,4], [2,6], [3,4], [3,5],
  [3,8], [4,5], [4,9], [7,8], [7,9] ]
```

Edge lists are simple, but if we want to find whether the graph contains a particular edge, we have to search through the edge list. If the edges appear in the edge list in no particular order, that's a linear search through $|E|$ edges. Question to think about: How can you organize an edge list to make searching for a particular edge take $O(\lg E)$ time? The answer is a little tricky.

Adjacency matrices

For a graph with $|V|$ vertices, an **adjacency matrix** is a $|V| \times |V|$ matrix of 0s and 1s, where the entry in row i and column j is 1 if and only if the edge (i, j) is in the graph. If you want to indicate an edge weight, put it in the row i, column j entry, and reserve a special value (perhaps `null`) to indicate an absent edge. Here's the adjacency matrix for the social network graph:

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

In JavaScript, we represent this matrix by:

```
[ [0, 1, 0, 0, 0, 0, 1, 0, 1, 0],
  [1, 0, 0, 0, 1, 0, 1, 0, 0, 1],
  [0, 0, 0, 0, 1, 0, 1, 0, 0, 0],
  [0, 0, 0, 0, 1, 1, 0, 0, 1, 0],
  [0, 1, 1, 1, 0, 1, 0, 0, 0, 1],
  [0, 0, 0, 1, 1, 0, 0, 0, 0, 0],
  [1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
```

```
[1, 0, 0, 1, 0, 0, 0, 1, 0, 0],
[0, 1, 0, 0, 1, 0, 0, 1, 0, 0] ]
```

With an adjacency matrix, we can find out whether an edge is present in constant time, by just looking up the corresponding entry in the matrix. For example, if the adjacency matrix is named `graph`, then we can query whether edge (i, j) is in the graph by looking at `graph[i][j]`. So what's the disadvantage of an adjacency matrix? Two things, actually. First, it takes $\Theta(V^2)$ space, even if the graph is **sparse**: relatively few edges. In other words, for a sparse graph, the adjacency matrix is mostly 0s, and we use lots of space to represent only a few edges. Second, if you want to find out which vertices are adjacent to a given vertex i, you have to look at all $|V|$ entries in row i, even if only a small number of vertices are adjacent to vertex i.

For an undirected graph, the adjacency matrix is **symmetric**: the row i, column j entry is 1 if and only if the row j, column i entry is 1. For a directed graph, the adjacency matrix need not be symmetric.

Adjacency lists

Representing a graph with **adjacency lists** combines adjacency matrices with edge lists. For each vertex i, store an array of the vertices adjacent to it. We typically have an array of $|V|$ adjacency lists, one adjacency list per vertex. Here's an adjacency-list representation of the social network graph:
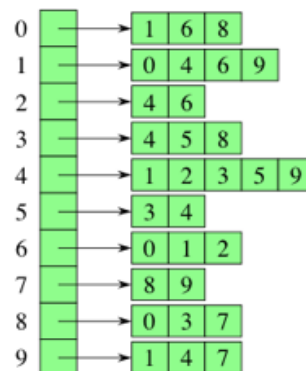
In JavaScript, we represent these adjacency lists by:

```
[ [1, 6, 8],
  [0, 4, 6, 9],
  [4, 6],
  [4, 5, 8],
```

```
    [1, 2, 3, 5, 9],
    [3, 4],
    [0, 1, 2],
    [8, 9],
    [0, 3, 7],
    [1, 4, 7] ]
```

Vertex numbers in an adjacency list are not required to appear in any particular order, though it is often convenient to list them in increasing order, as in this example.

We can get to each vertex's adjacency list in constant time, because we just have to index into an array. To find out whether an edge (i, j) is present in the graph, we go to i's adjacency list in constant time and then look for j in i's adjacency list. How long does that take in the worst case? The answer is $\Theta(d)$, where d is the degree of vertex i, because that's how long i's adjacency list is. The degree of vertex i could be as high as $|V| - 1$ (if i is adjacent to all the other $|V| - 1$ vertices) or as low as 0 (if i is isolated, with no incident edges). In an undirected graph, vertex j is in vertex i's adjacency list if and only if i is in j's adjacency list. If the graph is weighted, then each item in each adjacency list is either a two-item array or an object, giving the vertex number and the edge weight.

You can use a for-loop to iterate through the vertices in an adjacency list. For example, suppose that you have an adjacency-list representation of a graph in the variable `graph`, so that `graph[i]` is an array containing the neighbors of vertex i. Then, to call a function `doStuff` on each vertex adjacent to vertex i, you could use the following JavaScript code:

```
for (var j = 0; j < graph[i].length; j++) {
    doStuff(graph[i][j]);
}
```

If the double-subscript notation confuses you, you can think of it this way:

```
var vertex = graph[i];
for (var j = 0; j < vertex.length; j++) {
    doStuff(vertex[j]);
}
```

How much space do adjacency lists take? We have $|V|$ lists, and although each list could have as many as $|V| - 1$ vertices, in total the

adjacency lists for an undirected graph contain $2|E|$ elements. Why $2|E|$? Each edge (i, j) appears exactly twice in the adjacency lists, once in i's list and once in j's list, and there are $|E|$ edges. For a directed graph, the adjacency lists contain a total of $|E|$ elements, one element per directed edge.

This content is a collaboration of Dartmouth Computer Science professors Thomas Cormen and Devin Balkcom, plus the Khan Academy computing curriculum team. The content is licensed CC-BY-NC-SA.

Ask a question...

Questions   Tips & Thanks   Report a mistake

                                                   Guidelines

Top   Recent

"How long does that [checking for membership in the adjacency list] take? The answer is $\Theta(d)$." If you store the adjacency lists in sorted order, though, then wouldn't it be $\Theta(\lg d)$, using binary search?

6 Votes ▲ ▼ • Comment • Flag   *about a year ago* by William Chargin

> Agreed, if the list of neighbors is implemented as an array, and that array is sorted then determining membership is Θ(log(d)), using binary search.
>
> However, if the array is unsorted, or if the list of neighbors is implemented as a linked list (both are typical implementations) then determining membership is Θ(d), as we need to look at every item in the list before deciding that something is not in the list.
>
> In the article they mention "Vertex numbers in an adjacency list are not required to... (more)
>
> 7 Votes ▲ ▼ • Comment • Flag   *about a year ago* by Cameron

> Answer this question...

How to show edges and vertices in the quiz there is no hint to solve

4 Votes ▲ ▼ • Comment • Flag   *about a year ago* by shruts2907

> In representing graphs there is a question to create an edge list. It uses the image:
> https://ka-exercise-screenshots.s3.amazonaws.com/quiz--

representing-graphs_256.png
There are 6 answer slots, but the 6th vertex doesn't have a out...
3 Votes ▲ ▼ • 1 Comment • Flag   *about a year ago* by 👤 Ron Jensen

Show all 4 answers • **Answer this question**

Can someone please explain the algorithm of how to reverse a linked list with algorithm written down here??
1 Vote ▲ ▼ • Comment • Flag   *2 months ago* by 🍃 Chanda Yasaswini

> I don't see any linked lists here but here's how you do it:
>
> Assume:
> - start is the first node in your linked list to be reversed
> - each node has a next pointer pointing to the next node in the list
> - the last nodes next pointer points to null
>
> ```
> current = start
> //make sure the new end points to a null
> last = null
>
> while(current != null) {
>         nextnode = current.next;
>         //reverse the arrow we just went across
>         current.next = last;
>
>         last = current;
>         //jump across the next arrow
>         current = nextnode;
> }
>
> //make start
> ```
> ... (more)
> 1 Vote ▲ ▼ • 1 Comment • Flag   *2 months ago* by 👤 Cameron

> *Answer this question...*

What's the answer to the "Question to think about: How can you organize an edge list to make searching for a particular edge take O(lgE) time? The answer is a little tricky." ?
1 Vote ▲ ▼ • Comment • Flag   *3 months ago* by 🟢 haylinwang2009

> Sort the list by the 1st vertex in each edge, and then by 2nd vertex in each edge.
> The list is sorted, so we can now use a binary search on it.
> Our comparison is a little bit more complicated now.
> [v1,v2] < [v3,v4] if (v1 < v3) OR ((v1 === v3) AND v2 < v4)
>
> If we have an edge [v1,v2] we do a binary search for [v1,v2] on the list. Time O(log E)
>
> If your edges are not directed i.e. [v1,v2] is equivalent to [v2,v1] then perform a second search on the equivalent edge [v2,v1] if your first... (more)
> 1 Vote ▲ ▼ • Comment • Flag   *3 months ago* by 👤 Cameron

> *Answer this question...*

ABOUT

Our Mission

You Can Learn Anything

Our Team

Our Interns

Our Content Specialists

Our Board

SUPPORT

Help center

CONTACT US

Contact

Press

COACHING

Coach Reports

Coach Resources

Case Studies

Common Core

CAREERS

Full Time

Internships

CONTRIBUTE

Donate

Volunteer

Our Supporters

INTERNATIONAL 🌐

English

Translate our content

SOCIAL

Facebook

Twitter

Blog

Life at KA

Terms of Use   Privacy Notice

Free classes and courses available for online learning at every level: Elementary, High school, and College lessons.