Graph Representations

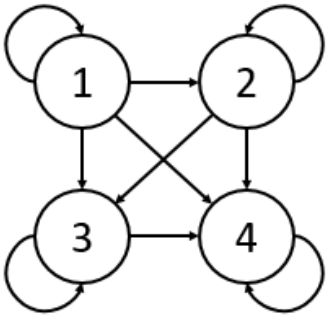
Step 1

As you saw in the last section, graphs can come in many shapes and sizes. Consequently, it is a good idea to consider the properties of the graph you are dealing with to make the best decision on how to go about representing it.

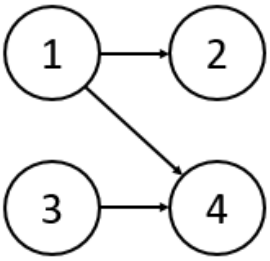
One good way to quantify the shape and size of a graph is to compare the number of edges in the graph, $|E|$, to the number of vertices in the graph, $|V|$.

Suppose a graph has $|V|$ vertices. How many possible edges could the graph have? We know that, at minimum, the graph could just look like a "forest" of isolated vertices, and as a result, it would have no edges ($|\boldsymbol{E}| = \mathbf{0}$). Can we calculate an upper-bound on the number of edges? For our purposes, as mentioned in the previous section of the text, we will disallow "multigraphs" (i.e., we are disallowing "parallel edges": multiple edges with the same start and end node). Suppose that our first vertex is connected to every other vertex in the graph (including itself). That means that we would have $|V|$ edges coming from that single vertex. Now, suppose we had the same for *all* vertices in the graph, not just the first one. All $|V|$ of our vertices would have exactly $|V|$ edges (one to each vertex, including itself), resulting in $|\boldsymbol{E}| = |\boldsymbol{V}| * |\boldsymbol{V}| = |\boldsymbol{V}|^2$ edges.

We like to use the word **dense** to describe graphs with many edges (close to our upper-bound of $|\boldsymbol{V}|^2$ edges for our purposes), such as the one below:



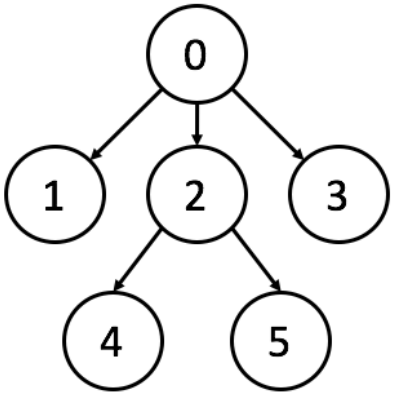
We like to use the word **sparse** to describe graphs with few edges (significantly fewer than $|\boldsymbol{V}|^2$ edges for our purposes), such as the one below:



Step 2

One way to represent a graph is by using an **adjacency matrix**. Below is an example graph with its corresponding **adjacency matrix**, *M*:

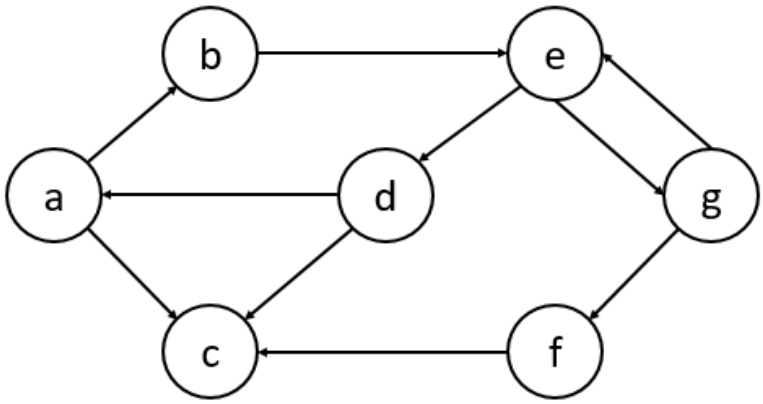
	0	1	2	3	4	5
0	0	1	1	1	0	0
1	0	0	0	0	0	0
2	0	0	0	0	1	1
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0



- Row *i* represents all of the edges *coming from* vertex *i*. Column *j* represents all of the edges *going to* vertex *j*
- A value of 1 in a given cell *M*(*i*, *j*) means that there exists an edge from vertex *i* to vertex *j*
  - For example, the 1 in the top row, located in cell *M*(0,1), means that there exists an edge *from* vertex 0 *to* vertex 1
  - See if you can match the rest of the non-zero entries in the matrix with the edges in the graph

Below is a more complex example of an **adjacency matrix** with its corresponding graph. See if you can match the non-zero entries in the matrix with the edges in the graph:

	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	0	0	0	0	1	0	0
c	0	0	0	0	0	0	0
d	1	0	1	0	0	0	0
e	0	0	0	1	0	0	1
f	0	0	1	0	0	0	0
g	0	0	0	0	1	1	0

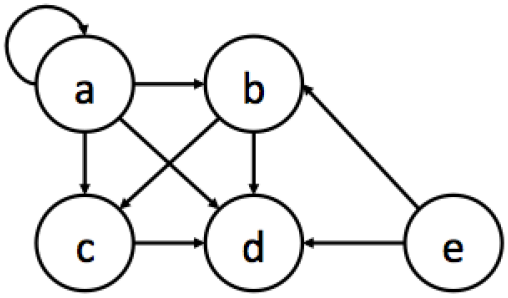


Notice that the **space complexity** (i.e., the amount of space that this matrix takes up) is **O(|*V*|²)**. As a result, this representation is not ideal for **sparse** graphs (i.e., graphs with significantly fewer than |*V*|² edges) because we would expect to have a lot of 0s (which do not really give us any information) and therefore a lot of wasted space.

STOP and Think: How would the **adjacency matrix** of an **undirected graph** look like?

Step 3

EXERCISE BREAK: Fill in the adjacency matrix for the directed graph below. Use the checkboxes to designate which entry in the matrix would have a "1" as their entry. You can assume that all un-checked boxes have a "0" for their entry.

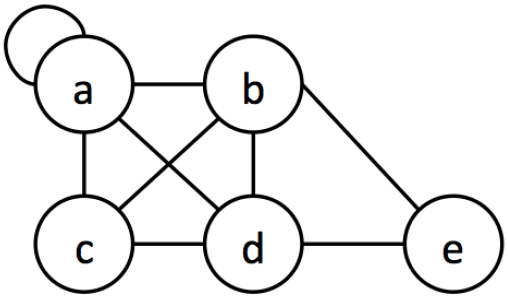


To solve this problem please visit https://stepik.org/lesson/28877/step/3

Step 4

EXERCISE BREAK: Fill in the adjacency matrix for the *undirected* graph below. Use the checkboxes to designate which entry in the matrix would have a "1" as their entry. You can assume that all un-checked boxes have a "0" for their entry.

HINT: Recall that, in an **undirected** graph, *e* = (*v*, *w*) would mean that there is a connection from node *v* to *w* **and** from *w* to *v*.



To solve this problem please visit https://stepik.org/lesson/28877/step/4

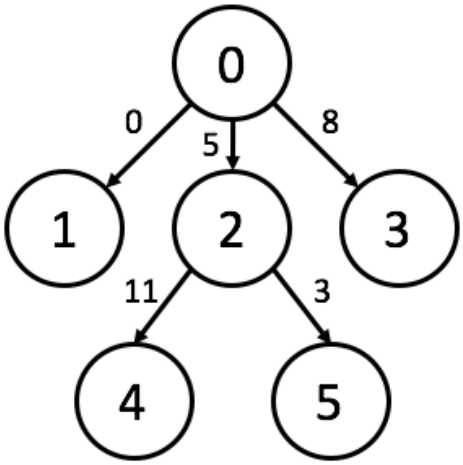
Step 5

So how would we go about using an adjacency matrix for a weighted graph (a graph where each edge has a cost associated with)?

You might have actually guessed it! We can replace the "1"s in the matrix with the actual costs of the edges. The only other thing we would also need to worry about is how to define a "no edge." At first glance, it might be tempting to just assume that we can always keep a "0" value as NULL. However, in general, it is perfectly valid for edges to have a weight of 0 and thus it is up to the person implementing the matrix to make the appropriate decision as to which NULL value to choose.

Below is an example where we took the graph and adjacency matrix from the previous step and added some costs to all the edges.

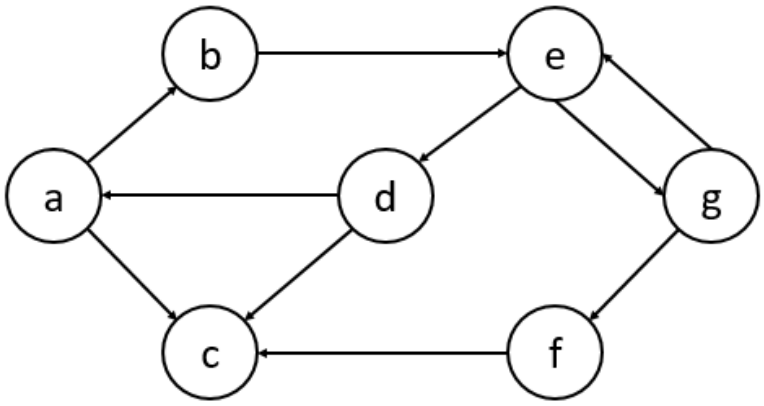
	0	1	2	3	4	5
0	NULL	0	5	8	NULL	NULL
1	NULL	NULL	NULL	NULL	NULL	NULL
2	NULL	NULL	NULL	NULL	11	3
3	NULL	NULL	NULL	NULL	NULL	NULL
4	NULL	NULL	NULL	NULL	NULL	NULL
5	NULL	NULL	NULL	NULL	NULL	NULL



Step 6

As we saw previously, an **adjacency matrix** is not ideal for graphs with significantly fewer than $|V|^2$ edges because of the wasted space used up by unneeded 0s. As a result, we turn to an alternative data structure to represent **sparse** graphs: the **adjacency list**. We will use the example below to describe how an adjacency list works.

```
a: {b, c}
b: {e}
c: {}
d: {a, c}
e: {d, g}
f: {c}
g: {e, f}
```



- The *i*-th "row" is a list representing the edges coming from vertex *i*
- The *j*-th entry in a given row *i* corresponds to the *j*-th edge coming out of vertex *i*

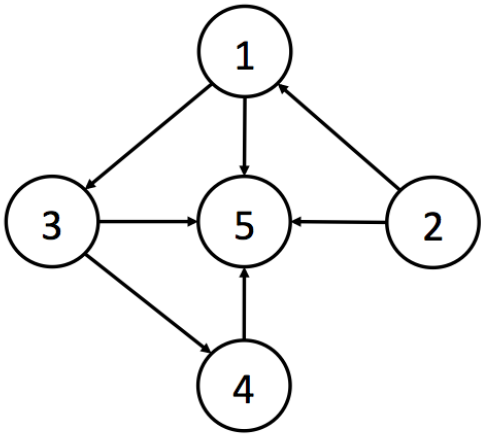
As you can probably see, if $|E|$ is relatively small, the **adjacency list** has a much smaller storage requirement than an equivalent **adjacency matrix**. Since we are storing a single list for each of our $|V|$ vertices (and each list has some constant-space overhead) and each of our $|E|$ edges is represented by a single element in one of the lists (so each edge is only accounted for once), our space complexity becomes **$O(|V|+|E|)$**.

It is important to note that, as mentioned before, $|E|$ can be as large as $|V|^2$. If this is the case, then the space complexity for storing a dense graph becomes $|V| + |V|^2$ and therefore will take up more space than an adjacency matrix (which would have only taken up $|V|^2$). As a result, we can conclude that an **adjacency list** is not ideal for **dense** graphs.

Step 7

EXERCISE BREAK: Fill in the adjacency list for the graph below by typing the corresponding vertices, separated by a space, for each list entry. If there are no corresponding vertices, just leave the entry blank.

For example, if *vertex a* had an edge directed to *vertex b* and *vertex c*, then for *vertex a*'s entry you would type "b c" (without the quotation marks).



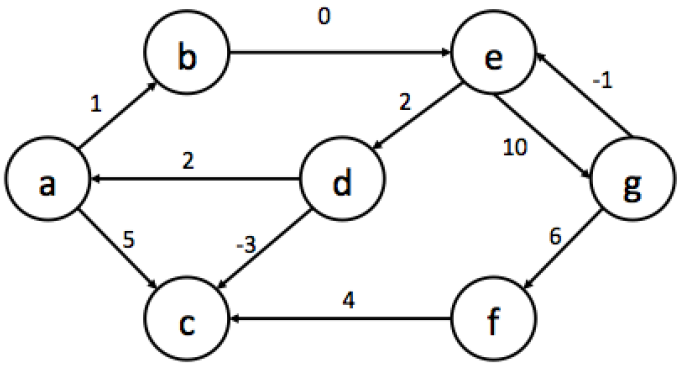
To solve this problem please visit https://stepik.org/lesson/28877/step/7

Step 8

So how would we go about making an adjacency list for a *weighted* graph? This might be a bit less intuitive than it was for the adjacency matrix since we no longer have a space where we could just write in the weight of each edge. As a result, one of the ways to represent a weighted graph would be to *make* space to store the weights of each edge. For example, we could store each entry of our adjacency list as a pair (***v***, ***c***), where ***v*** would be the destination vertex (just as before) and ***c*** would now be the cost of the edge.

Take a look at the example below:

```
a: {(b,1), (c,5)}
b: {(e,0)}
c: {}
d: {(a,2), (c,-3)}
e: {(d,2), (g,10)}
f: {(c,4)}
g: {(e,-1), (f,6)}
```



Now that we have learned about different ways to store different types of graphs, it is time to discover how we can go about using these representations to not just *store* a graph, but to also be able to *efficiently traverse* it.