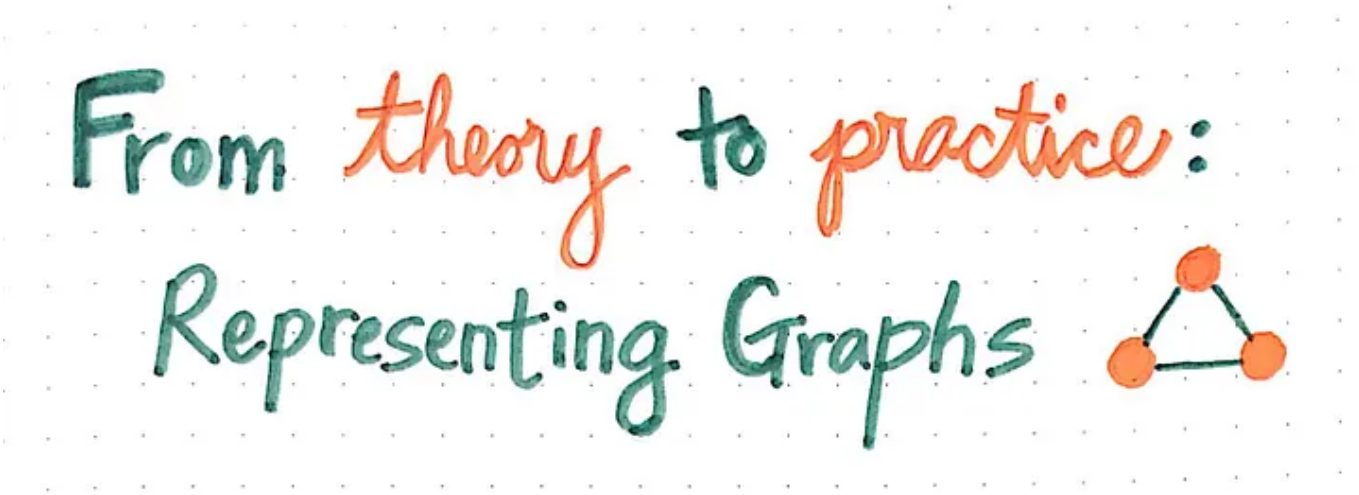
From Theory To Practice: Representing Graphs

Vaidehi Joshi · Follow
Published in basecs · 15 min read · Sep 11, 2017



From theory to practice: representing graphs

The best investment you can make in your own learning is returning back to to the things you (think) you already know, and this is particularly true in the world of computer science. Everything in computing is a layer of abstraction built upon *even more layers* of abstraction; if all the

industries in the world were onions, computing would surely be one of the few with an uncountable number of layers.

All of these abstractions are also what makes software so hard (and sometimes, so intimidating). None of us know everything there is to know about how the software or hardware that powers our world works. And the truth of the matter is that none of us will ever really know all there is to know. But that's okay! We learn what we need to learn at the moment, and we explore the new things that we are interested in exploring. Usually, this means learning more about things that we're already at least a little bit acquainted with.

This is exactly why I love returning back to topics that I think I already know, and digging a little bit deeper under the surface — you will almost always find something new. Earlier in this series, we sunk our teeth into the basics of graph theory; but there's still so much more to know about graphs! We know that they are the foundations of fundamental things like social networks, or our even our browser histories, but how do they really work, and how do we implement them? We were gently introduced to the theory behind graphs, but how can we put that theory into practice and understand graphs on a deep level so that we can actually use them to solve real problems?

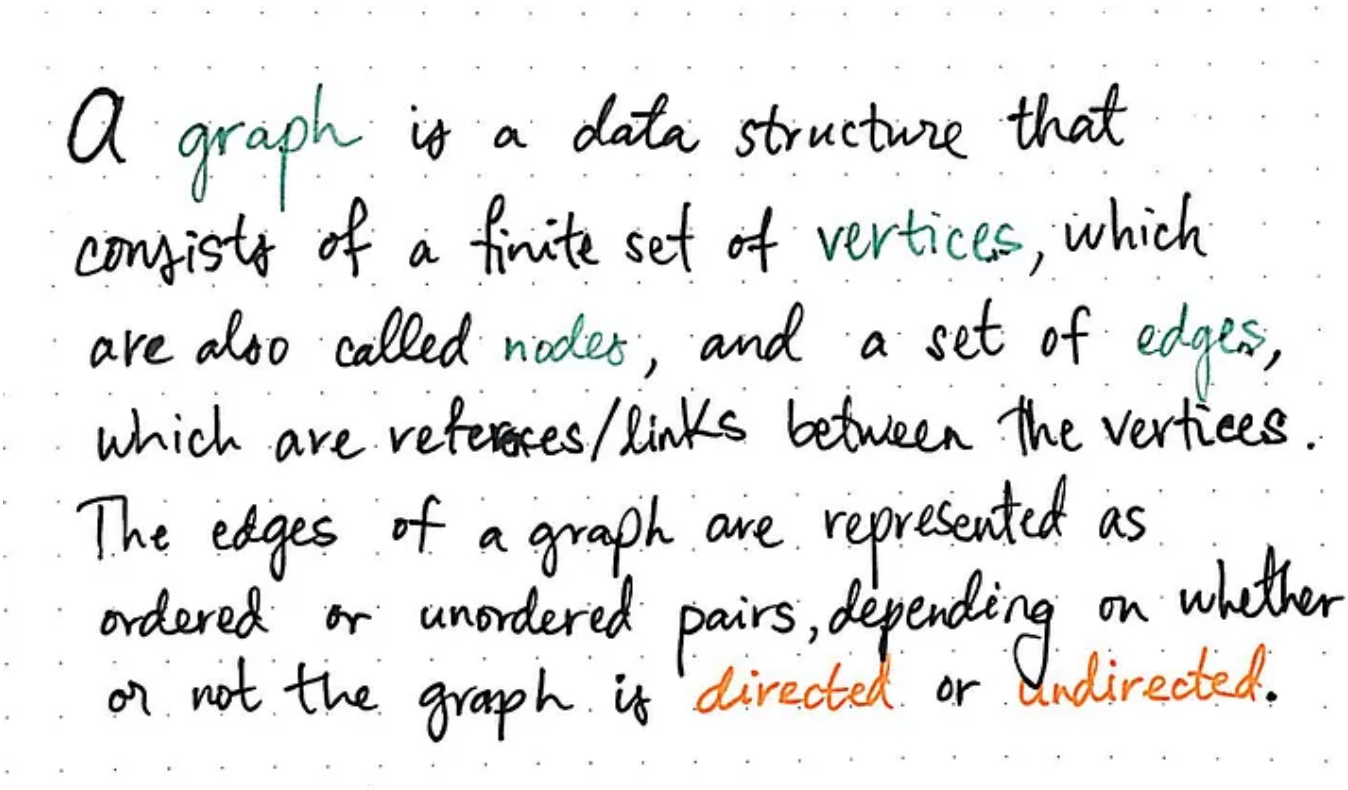
In order to do all of those things, we need to go back to the basics. We need to understand how to represent graphs and turn the theoretical into something more concrete.

An ordered pair by any other name

Since we're already familiar with the theory behind graphs, we won't dive too much into the history or applications of them here. The short version of the

story is that graphs come from mathematics, and are nothing more than a way to formally represent a network, which is a collection of objects that are all interconnected.

In order to tackle the task of *representing* a graph structure, which is so often present in computer science and across the web, we should make sure we're comfortable with the parts of a graph, as well as its definition. A ***graph*** is a data structure with two distinct parts: a finite set of vertices, which are also called nodes, and a finite set of edges, which are references/links/pointers from one vertex to another.



The graph data structure: a (refresher of a) definition

A graph is defined by these two distinct parts, vertices and edges. Some graphs have many edges as compared to nodes, and are defined as ***dense*** graphs, whereas a graph a smaller edge-to-node ratio is called a ***sparse*** graph. Similarly, if the edges of a graph have a directional flow, the graph is defined

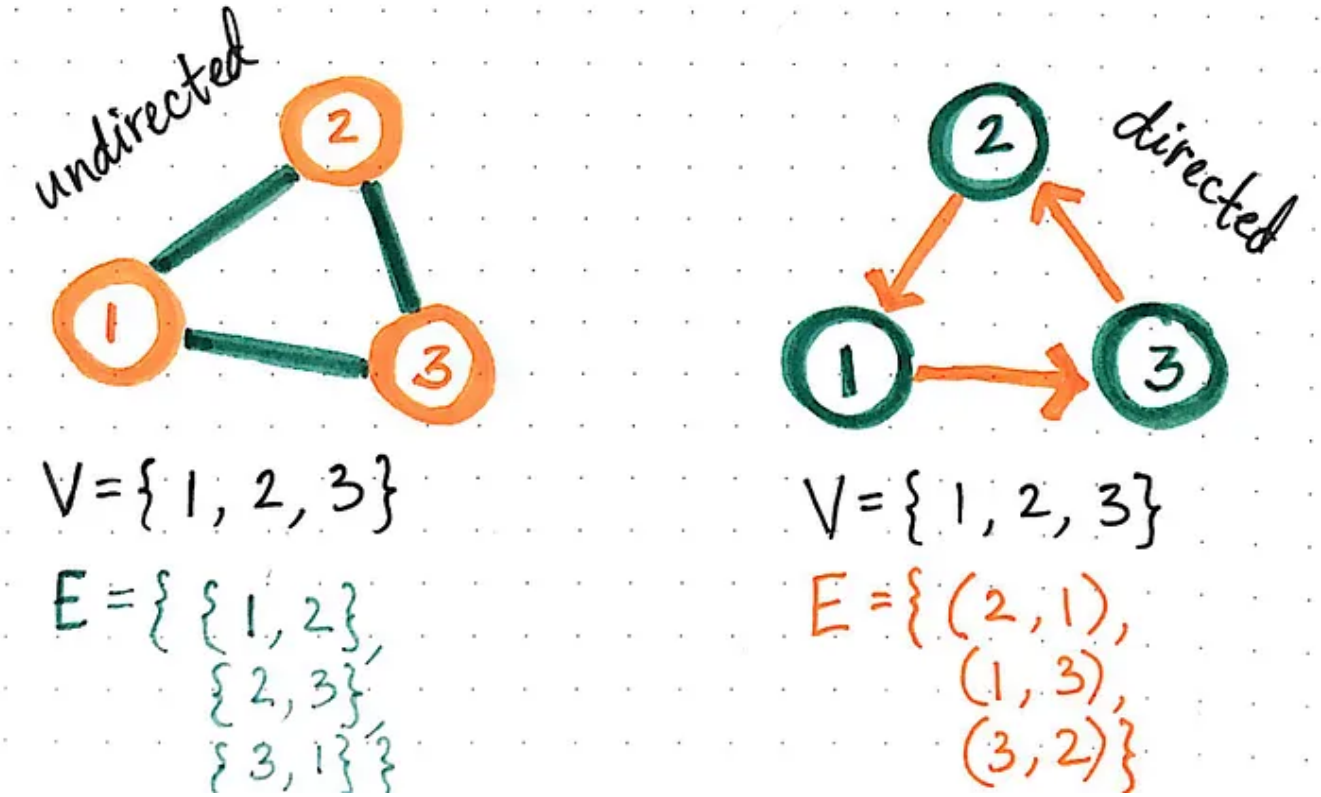
to be ***directed,*** while a graph with edges that have *no* directional flow is ***undirected.***

> The characteristics of graph are strongly tied to what its vertices and edges look like.

Being able to identify these characteristics is very important since this directly affects how we go about representing a graph. But wait — how *do* we represent a graph! That's what we set out to do, but we haven't actually gotten to that part yet, have we? Since we have some knowledge about graphs already, we can build upon that! (See what I meant about returning to topics that we *think* we already know?)

We'll recall that, in mathematics, graphs are represented as $G = (V, E)$, where V is the set of vertices, and E is the set of edges. When we learned about graph theory, we saw that there are two ways of representing this set of edges — either as unordered pairs, or ordered pairs — based on whether the graph's edges are directed or undirected.

Let's quickly refresh what that means in practice.

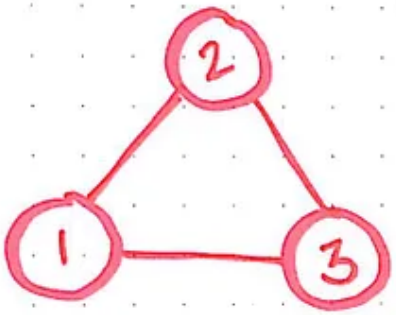


Representing edges in directed vs. undirected graphs

In this example, we'll see that the undirected graph's edges, represented by *E*, have no order to them, since it's possible to travel from one vertex to the other. However, in the edge set *E* for the directed graph, there is directional flow that's important to how the graph is structure, and therefore, the edge set is represented as ordered pairs.

This representation of a graph's edges is the simplest possible way to represent a graph — and we already kind of knew it! But how do these simple ordered pairs actually look in code and in memory? Well, that's something we don't know about yet (although we might be able to guess!). The answer is probably as easy as you expect it to be: the quickest way to turn this group of ordered pairs is to use a list or an array, depending on which language we're using.

An edge list is a list (or array) of all of the |E| edges in a graph. Edge lists are one of the simplest representations of a graph.



list

index →

0	1	2
1	2	3
2	3	1

array

```
[
  [1, 2]
  [2, 3]
  [3, 1]
]
```

To represent a graph with three nodes (1, 2, and 3), we could either use a list format or an array to represent the graph as an edge list.

* Note that the edges of this graph are all in the edge list, but not in any particular order!

→ This graph is fairly small, but what if it was much larger? It would take O(E) to find one particular edge. The space of representing an edge list is also going to be O(E)!

The edge list as a way to represent a graph

This list (or array) is called an ***edge list,*** and is a representation of all the edges ($|E|$) in the graph.

In the example shown here, we have a small graph, with just three nodes: 1, 2, and 3. Each edge is given an index, and represents a reference from one node to another. We'll notice that there isn't any particular order to the edges as they appear in the edge list, but every single edge must be represented.

For this particular graph, the edge list would look like this:

```
[
  [1, 2],
  [2, 3],
  [3, 1]
]
```

Because an edge list is really just an array, the only way to find something in this array is by iterating through it. For example, if we wanted to see if vertex `1` was connected to vertex `2`, we'd need to iterate through this array and look for the existence of a pair `[1, 2]` or `[2, 1]`. Now, this is fine for our graph, which only has three vertices and three edges; iterating through this array isn't really a big deal. But realistically speaking, most graphs are going to be *much* bigger than this one!

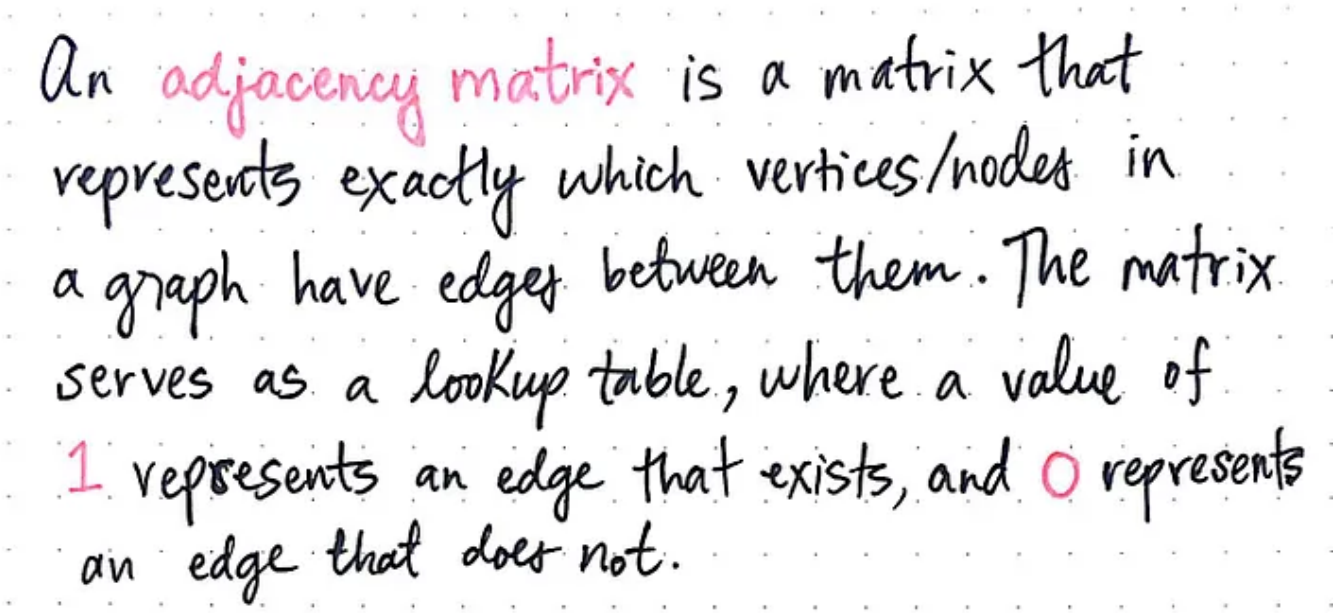
Imagine having to iterate through a massive array to see if *one* particular edge existed in the array; since the list of edges doesn't have to follow any particular order, the edge could be at the very end of the list! Or, it could not be there at all, and we'd *still* have to iterate through the whole thing to check

for it. Not only would doing this work would take *linear*, *O(E)* time, where *E* represents all the edges in the graph — an edge list also requires *O(E)* amount of space, and is yet fairly limiting in what we can do with it, despite needing that space.

Let's remember what we started off with: this is the *simplest* representation of a graph. Sometimes, the simplest version isn't quite enough for us to work with. Some, let's complicate it a little bit!

When lists just won't cut it

For most graphs, an edge list won't end up being the most efficient choice. So, we can kick it up a notch and go from a list to a matrix — an adjacency matrix, that is!

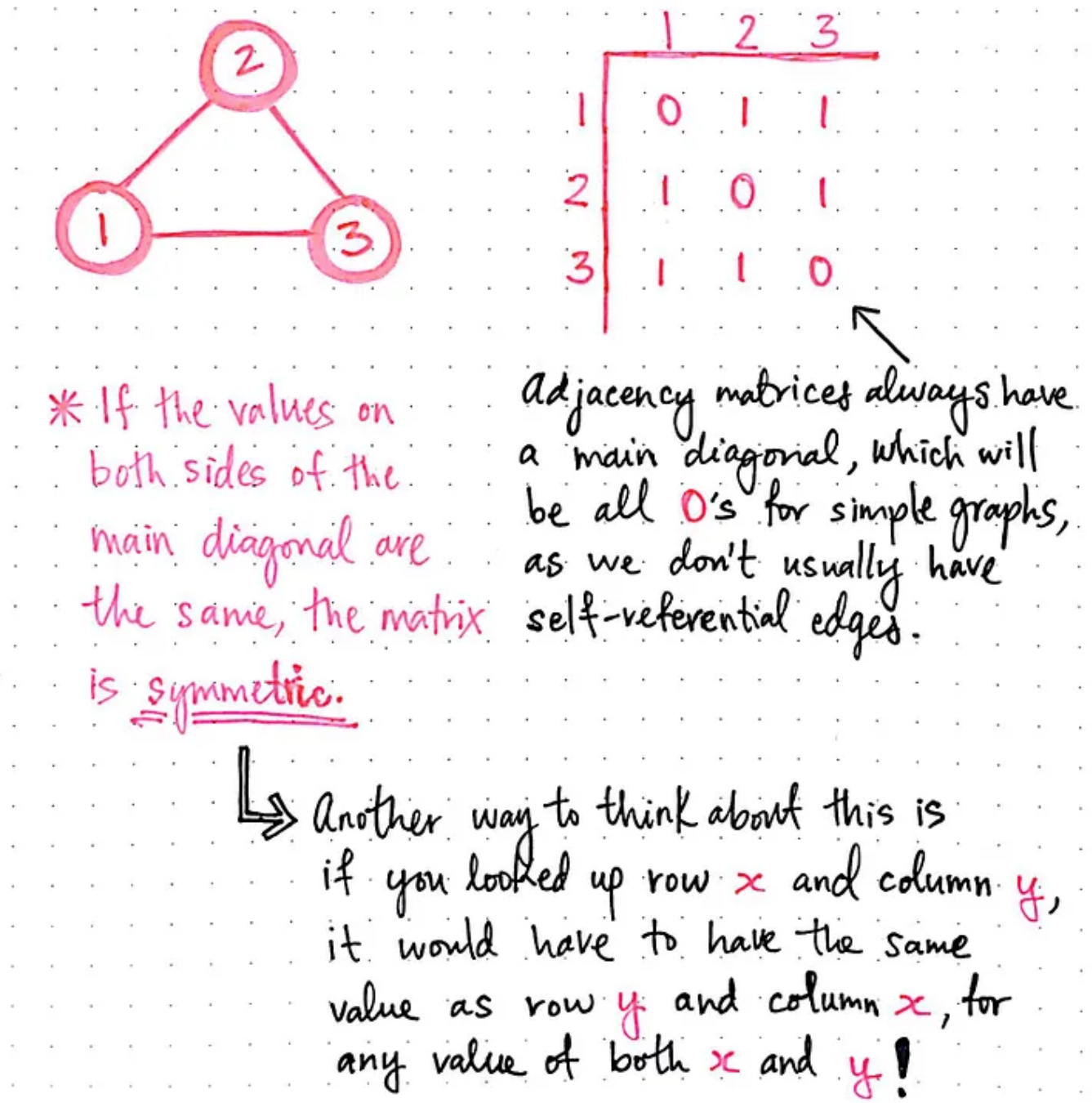


An adjacency matrix: a defintion

An ***adjacency matrix*** is a matrix representation of exactly *which* nodes in a graph contain edges between them. The matrix is kind of like a lookup table: once we've determined the two nodes that we want to find an edge between,

we look at the value at the intersection of those two nodes. The values in the adjacency matrix are like boolean flag indicators; they are either present or not present. If the value is `1`, that means that there is an edge between the two nodes; if the value is `0`, that means an edge does not exist between them.

Let's look at an example to make this a little clearer. We'll work with the same graph we used earlier, with just three nodes (1, 2, and 3) again.



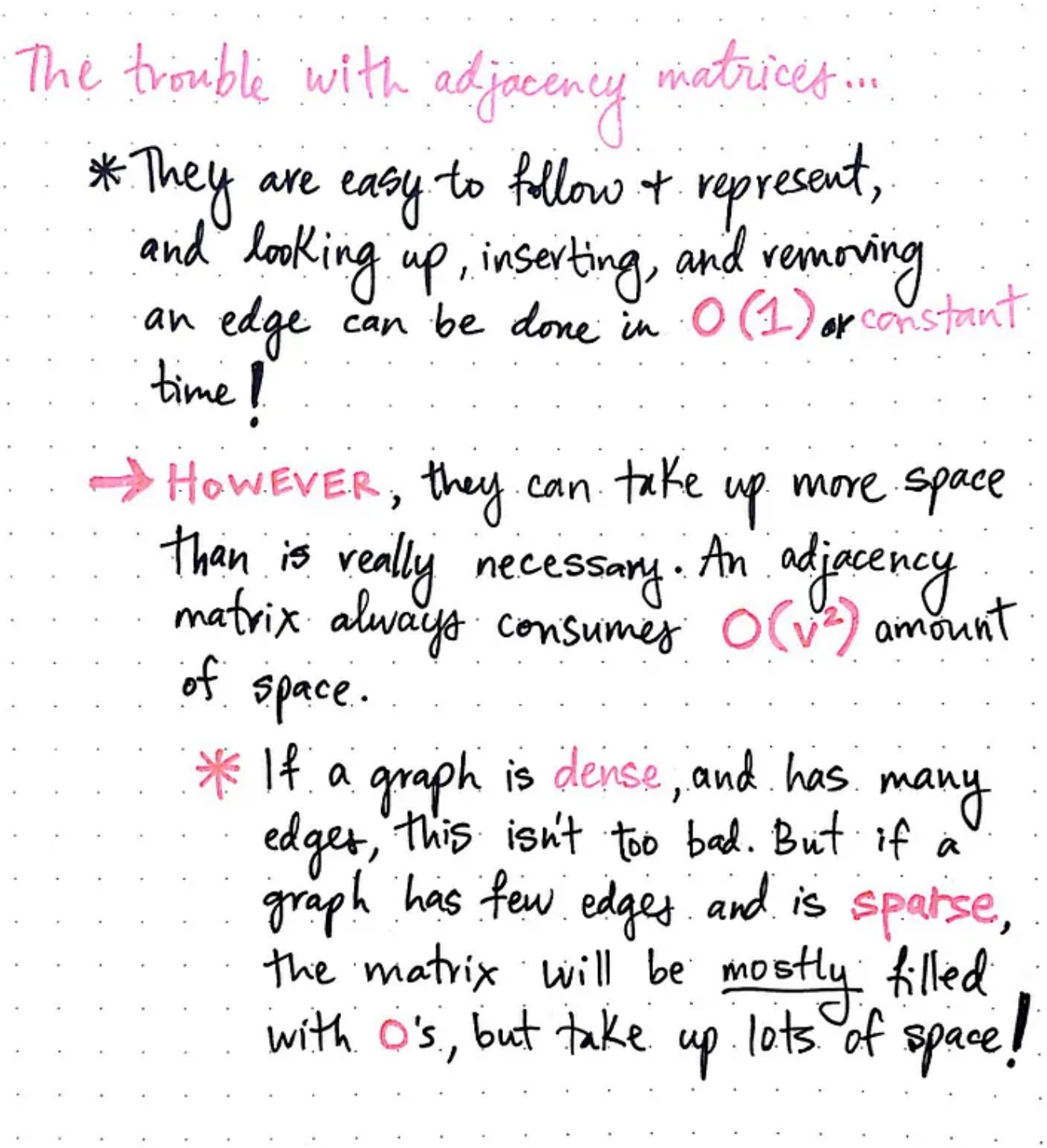
Symmetry within adjacency matrices

In this illustration, we can see that the adjacency matrix always has a main diagonal that will have a value of `0` down the diagonal, since most graphs that we're dealing with won't be referential. In other words, since node `2` cannot have a link to itself, if we draw a line from column `2` to row `2`, the value will be `0`. However, if we wanted to see if node `3` was connected to

node `1`, we could find column `3`, row `1` and see that the value is `1`, which means that there is indeed an edge between these two nodes.

The interesting thing about adjacency matrix representations of a graph is that, just by looking at them, we can tell whether the graph is *directed* or *undirected*. We can determine this characteristic of a graph based on whether its adjacency matrix is *symmetric* or not. If the values on both sides of the main diagonal of an adjacency matrix are the same, the matrix is ***symmetric.*** In other words, if we looked up row `x`, column `y`, it would contain the same value as whatever was in row `y`, column `x`. If this was true for all the rows and columns of the matrix, the matrix would be symmetric, which our particular graph representation happens to be.

It will become more clear how the symmetry of a matrix is tied to the "directness" of a graph in the next section.



Adjacency matrices: the good, the bad, the ugly

Adjacency matrices are definitely a step up from an edge list, for sure. For one thing, they are fairly easy to represent; our adjacency matrix would be look like this:

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

Adding or deleting an edge, from a graph is also easy to do with this representation, since looking up a potential node between two edges only requires knowing the two nodes, finding the appropriate index for each, and finding the value at the intersection between the two. Most operations on an adjacency matrix can be done in *constant,* or *O(1)* time.

However, what would happen if we had a *sparse* graph? What would our adjacency matrix look like if our graph had very few edges as compared to nodes? Well, we can probably imagine it without having to even draw it out: it would mostly be filled with values of `0`. But, because of the way that a matrix is structured, we'd still need to build out the entire thing!

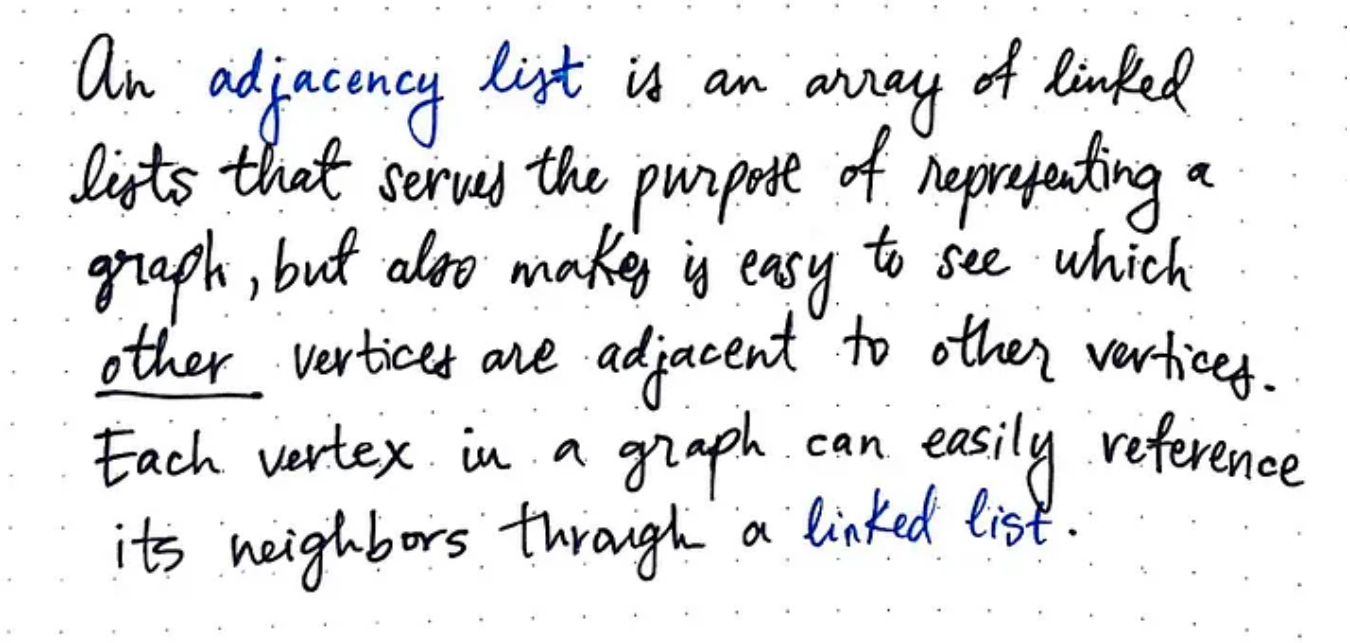
The trouble with adjacency matrices is that they will always require $O(V^2)$ amount of space, which, depending on what our graph looks like, can mean a lot of wasted space! So, as great as an adjacency matrix can be, it isn't always the right representation for a graph (particularly if that graph is sparse).

Well, not to worry. There's yet another option when it comes to graph representation — and it's the most fun one of all!

Adjacency lists: the hybrid choice

When both edge lists and adjacency matrices seem to fail us, what are we to do? Why, combine them both together, of course! And that's exactly what an

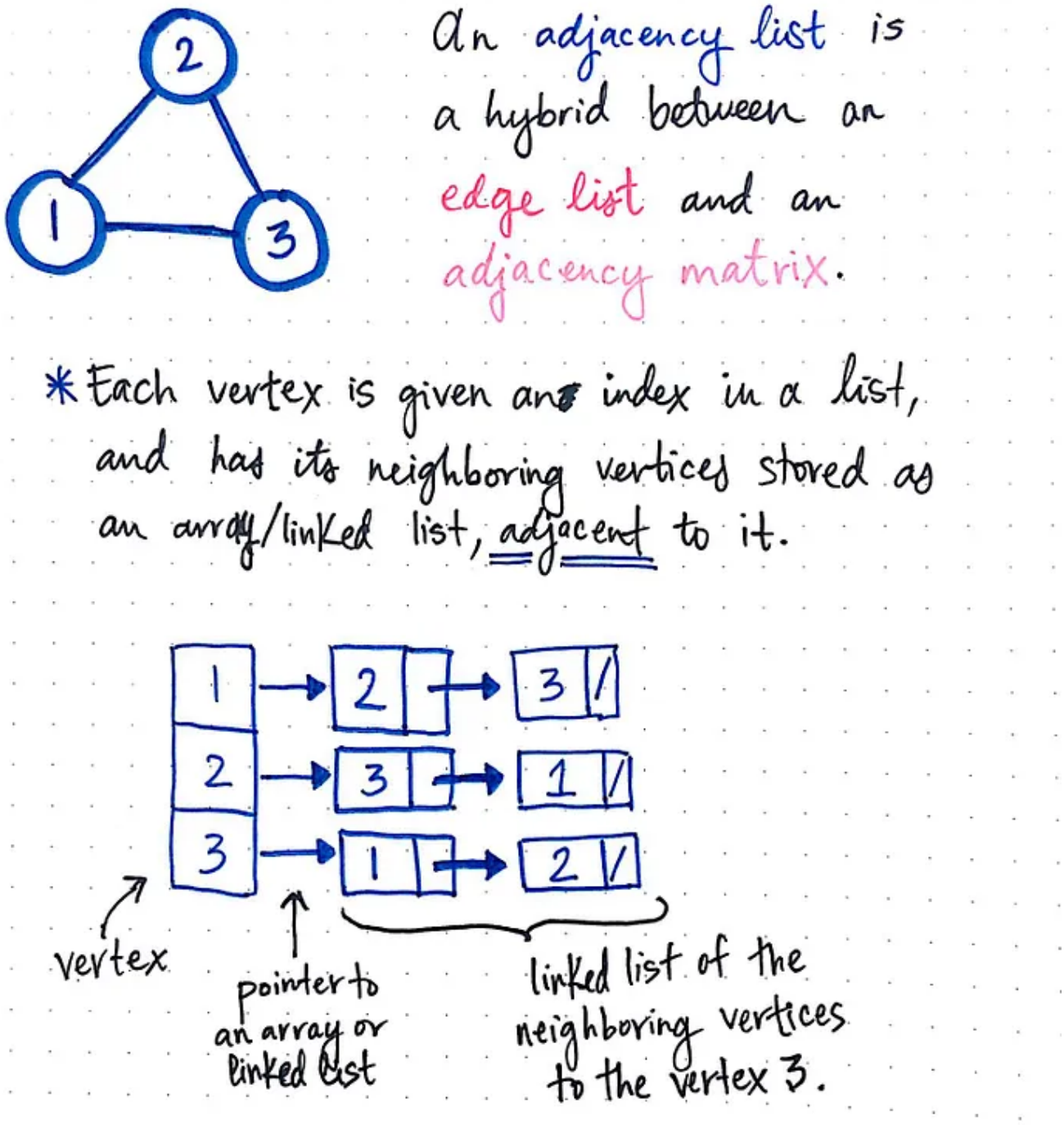
adjacency list is — a hybrid between an *edge list* and an *adjacency matrix*. An adjacency list is an array of linked lists that serves as a representation of a graph, but also makes it easy to see which other vertices are adjacent to *other* vertices.



An adjacency list: a definition

An adjacency list is the most popular and commonly-used representation of a graph since most graph traversal problems (which we'll see a whole lot more of, later on in this series!) require us being able to easily figure out which nodes are the neighbors of another node. In most graph traversal scenarios, we don't even really need to *build* the entire graph out; we just need to be able to know where we can travel — in other words, who the neighbors of a node are. This is fairly easy to determine when using an adjacency list, since every single node/vertex has a reference to all of its neighbors through *an adjacent linked list*.

Let's look an example to see what this looks like in practice.



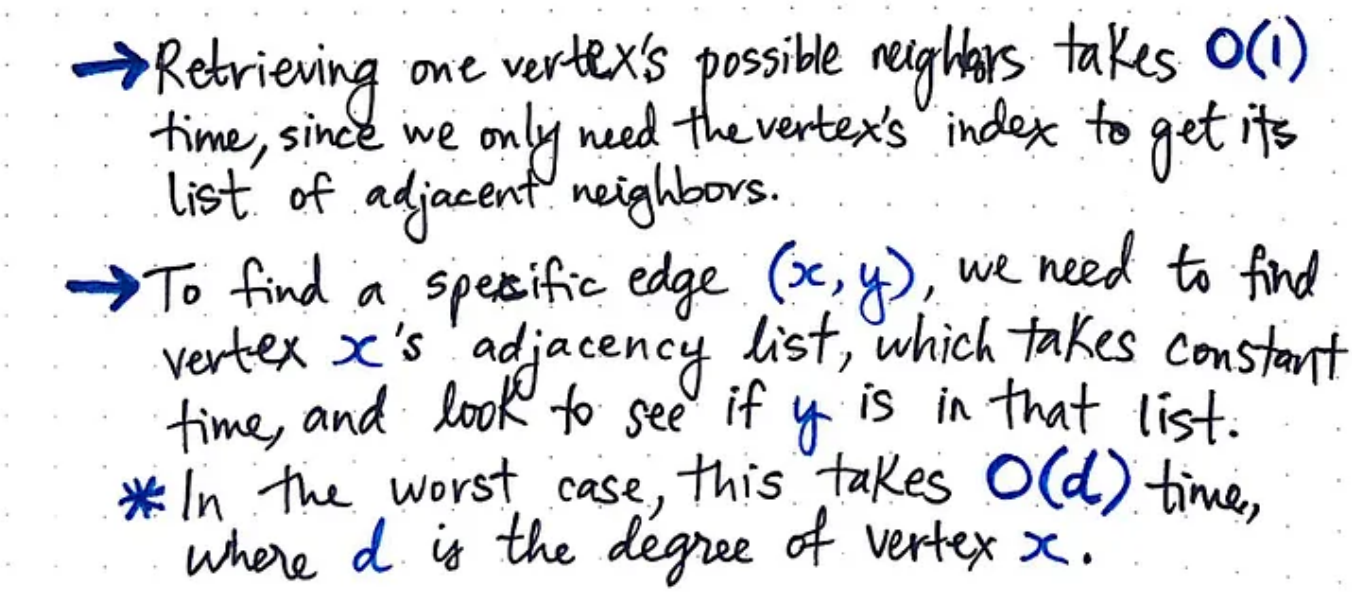
An adjacency list is a hybrid between an edge list and an adjacency matrix

In the illustration drawn here, each vertex is given an index in its list, and has all of its neighboring vertices stored as an linked list (which could also be an array), adjacent to it. For example, the last element in the list is the vertex `3`, which has pointer to a linked list of its neighbors. The linked list that is adjacent to vertex `3` contains references to two other vertices: `1` and

`2`, which are the two nodes that are connected to the node `3`. Thus, just by looking up the node `3`, we can determine who it's neighbors are and, by proxy, quickly identify that it has two edges connected to it.

We can see that, because of the *structure* of an adjacency list, it's very easy to determine all the neighbors of one particular vertex. In fact, retrieving one node's neighbors takes *constant*, *O(1)* time, since all we really need to do is find the index of the node we're looking for, and pull out its list of adjacent vertices.

But what if we wanted to find a particular edge — or check whether an edge between two nodes exists? This was easy to do with an adjacency matrix... but how easy is it with an adjacency list?



The practical points behind using adjacency lists

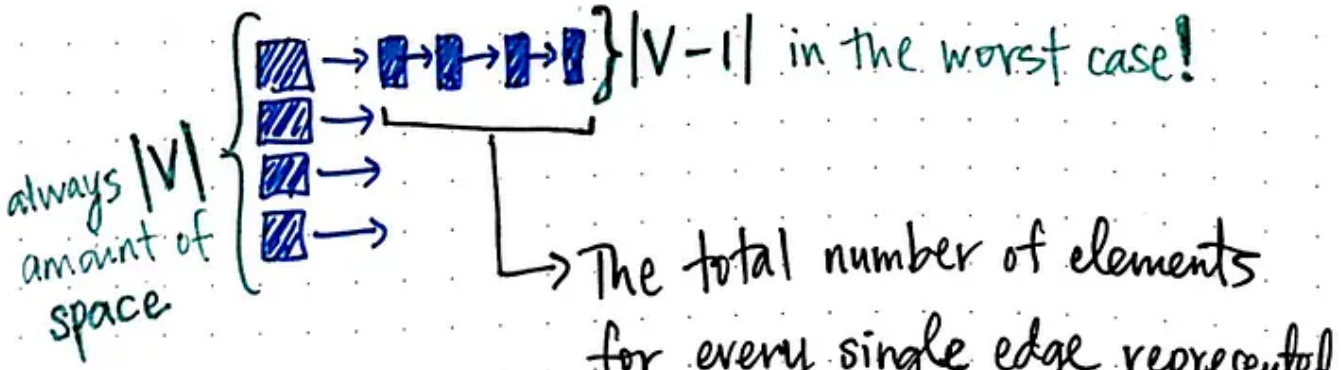
Well, to find a specific edge — for example, `(x, y)` —we need to find vertex `x` in the adjacency list, which we already know takes constant, *O(1)* time to look up as an index. The second step is checking to see whether `y` is in the

adjacency list for node `x`. This could be fairly quick to do, particularly if `y` is first in the list, or if it's the *only* item in the list.

But, what about the worst-case scenario? Well, `y` could potentially be at the *very end* of the linked list. Or, it might not even exist! In that case, we'd end up iterating through the whole linked list to check for it, which will take us ***O(d)*** time, where *d* is the degree of vertex `x`. The ***degree*** of a vertex is the number of edges that it has, which is also known as the number of neighboring nodes that it has.

The maximum possible value for d, the degree of any vertex in a graph is $(|V|-1)$, in the case that a vertex has edges to every other vertex in the graph (except for itself). The minimum value for d is 0, if the graph has one isolated vertex with no other vertices.

→ An adjacency list will require $|V|$ amount of space for the list itself, as every vertex must be represented in the list.



→ The total number of elements for every single edge represented in an adjacency list depends on whether the graph is directed or undirected.

Space complexity of an adjacency list

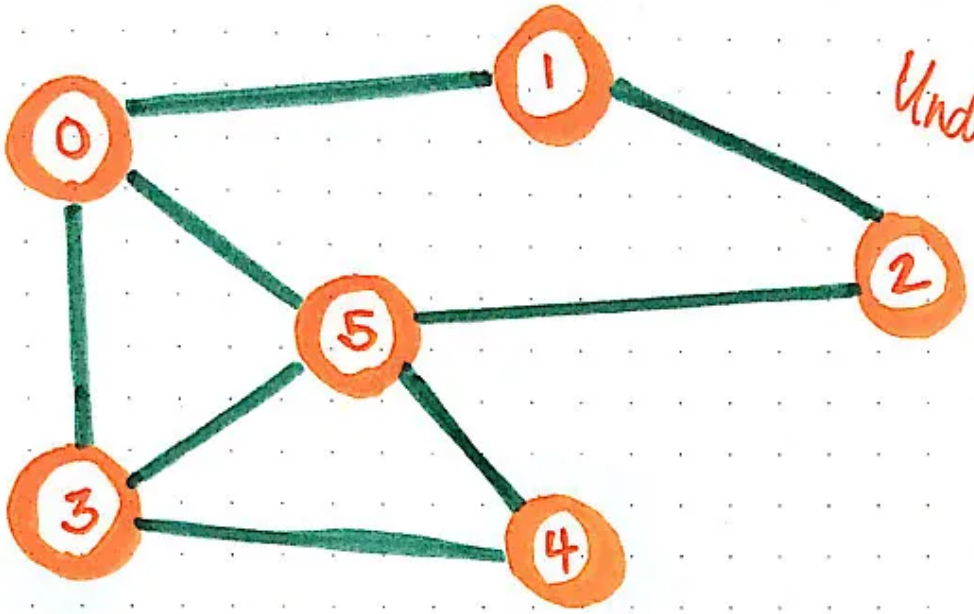
So, how big could our degree possibly end up being? Well, the maximum possible value for the degree (*d*) of any vertex in a graph can never be more than `(|V| – 1)`, where *V* is the total number of vertices in the graph. If we think about it, this makes sense; the highest number of potential neighbors that node could ever have is *every single node* in the graph, except for itself! The minimum possible value for *d* is `0`, since we could always have a graph that has only one vertex, which would mean that it has 0 edges and 0 neighboring nodes.

Okay, so now that we know how much time it could possibly take us to iterate through an adjacency list, what about how much space it takes up?

An adjacency list itself will require *|V|* amount of space for the list, since every single vertex is going to require it's own index and spot in the list. The vertex will also need a pointer reference to its linked list/array of neighbors, but a pointer takes a negligible amount of space in memory. So what about the linked list itself? We have already determined that the longest that a linked list of neighboring nodes could ever possibly be is *(|V|-1)*, in the worst case.

But there's one additional caveat here: the total number of elements for every single edge represented in an adjacency list depends on whether the graph is *directed* or *undirected*.

This starts to become more obvious when we compare an undirected graph representation with a directed graph representation! So let's take a look and see how they compare.



Undirected Graph

EDGE LIST

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

ADJACENCY MATRIX

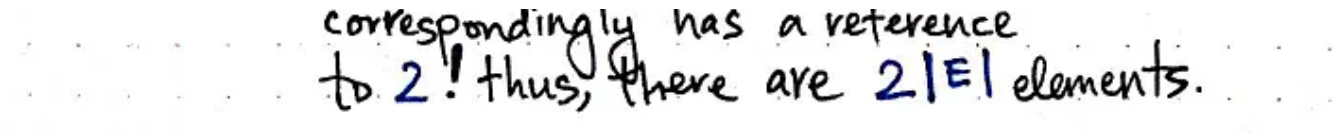
	0	1	2	3	4	5
0	0	1	0	1	0	1
1	1	0	1	0	0	0
2	0	1	0	0	0	1
3	1	0	0	0	1	1
4	0	0	0	1	0	1
5	1	0	1	1	1	0

notice that an undirected graph is symmetric

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

ADJACENCY LIST

notice that each edge appears twice; 2's adjacency list contains a reference to 5, and 5's list



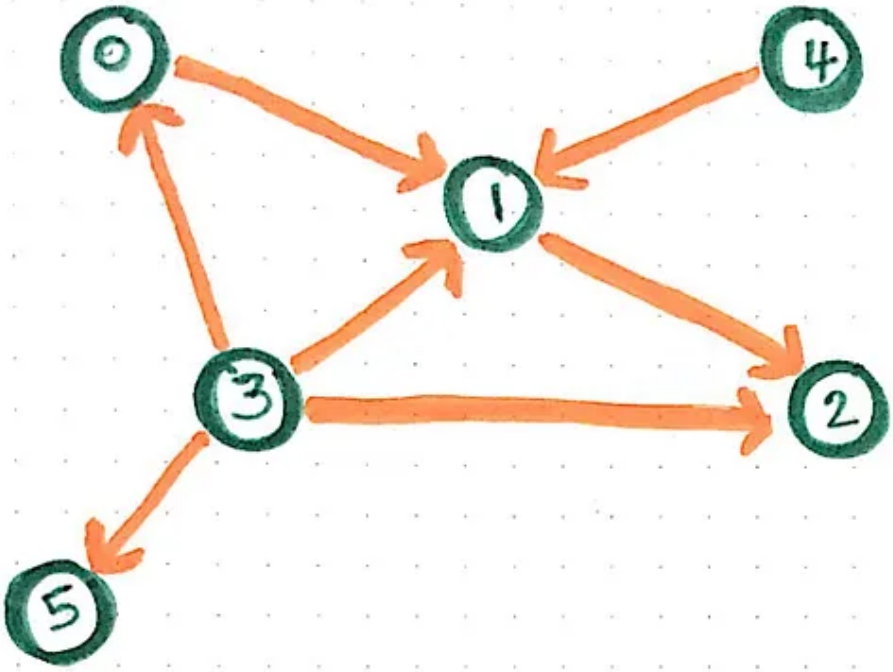
An undirected graph, represented three ways

In this ***undirected*** graph example, we have five nodes, and eight undirected edges.

Since we know that there are three different ways to represent this graph, let's look at all of these formats with this one example. The *edge list* is the most straightforward of the three: it's an array of the eight edges, which explains why its indices range from `0-7`. If we look at the *adjacency matrix*

undirected graph will always have a symmetric adjacency matrix.

Now, to answer our final question: how does our *adjacency list* change in size based on the directness of our graph? Well, if we look at this adjacency list example, we'll notice that each edge appears twice. For example, vertex `2`'s adjacency list contains a reference to vertex `5`; similarly, vertex `5`'s adjacency list contains a reference to vertex `2`. Each edge is represented twice, so the total amount of elements that we will need to allocate space for in an adjacency list is actually *2(|E|)* elements, where *E* is the total number of edges in an undirected graph.



Directed Graph

EDGE LIST

0	0	1
1	1	2
2	3	0
3	3	1
4	3	2
5	3	5
6	4	1

ADJACENCY MATRIX

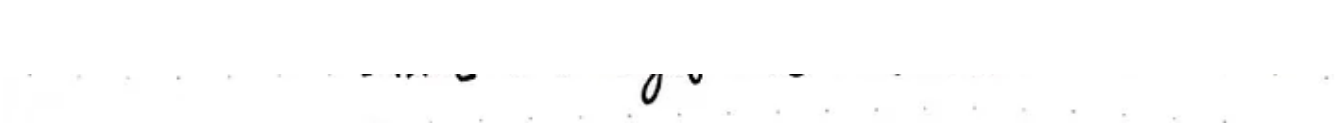
	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	0	0	0	0
3	1	1	1	0	0	1
4	0	1	0	0	0	0
5	0	0	0	0	0	0

notice that this matrix is not symmetrical!

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

ADJACENCY LIST

notice that this adjacency list contains a total of $|E|$ elements, one for each edge, since the edges are directed.



A directed graph, represented three ways

Now that we've looked at all three representations of an undirected graph, we might already be able to guess how a ***directed*** graph compares.

In this illustration, we have five nodes, and seven edges. The *edge list* representation of this graph has, as we might expect, seven elements, with indices that range from `0-6`, one for each of the directed edges.

We'll notice that *adjacency matrix* for a directed graph looks a little different from any that we've seen before! That's because it is *not symmetric*. If we look up `(0, 1)`, we'll see that the value is `0`, but if we look up `(1, 0)`, we'll see that the value is `1`! This is because of the directional flow of the edges in this graph; not every node has a bi-directional link to a neighboring node, and the layout of the adjacency matrix displays that perfectly.

Finally, the *adjacency list* representation of this directed graph also looks different from what we saw in undirected one earlier. We'll notice that the list contains only $|E|$ number of elements, where E is the total number of edges. If we think about it a bit more, it makes sense; the edges aren't bidirectional in this graph, so we don't need to represent them twice for each node; rather, they are one-way connections between nodes, so we only need to represent them once, for whichever node is linked to its neighbor.

Thus, the representation of a graph (and how much space it takes up) all depends on what the graph looks like, and what we're trying to do with! As is the case with most things in computer science, the answer to "which representation should we use?" is quite literally: *it depends on what you want to do!*

For the purposes of this series and understanding graph traversal algorithms and graph coloring problems, we'll most likely be using adjacency lists. But, as always, they are just one tool for the job. The most important thing, however — far more important that what representation we choose to use — is the fact that we *know* how to transform theory into practice!

Resources

Understanding the basics of the graph theory is pretty fundamental to unpacking some of the most complicated and well-known computer science problems. But knowing all that theory isn't helpful if you can't apply it! Thankfully, there are a lot of good resources that show how to represent a graph in programmatic terms. If you're looking to understand even more, these are some good places to get started.

1. Graph and its representations, Geeksforgeeks
2. Representing graphs, Khan Academy
3. Representing Graphs — Algorithms On Graphs, Coursera
4. Graph Representation — Adjacency List, mycodeschool
5. Graph Visualizations, VisuAlgo

Programming | Data Structures | Computer Science | Tech | Software Development