Graphs

Robin Dawes

February 24, 2021

Refreshes our memories regarding some basic definitions of graph theory. We are introduced to the Breadth-First Search algorithm for graphs. We discover that the complexity of this algorithm depends on the data structure we use to represent the graph.

.

Definitions and Notation

Most of the following definitions and notation are introduced in CISC-203. This information is included here for completeness.

Graphs can be viewed as generalizations of trees (actually, trees are a subclass of graphs). A graph G consists of two sets: a set V of vertices, and a set E of edges, where each edge consists of a pair of vertices in V.

Terminology:

We use n to denote $|V|$

We use m to denote $|E|$. Note that if we disallow duplicate edges and loops, $m \leq \binom{n}{2}$.

Two vertices x and y are adjacent if $(x, y) \in E$

If x and y are adjacent, we call them neighbours.

A loop is an edge where both vertices are the same, such as (x, x). Loops are useful in a few situations but for the most part we will focus on graphs that have no loops.

Sometimes it is reasonable to have multiple edges between a pair of vertices. These are called multiple edges (surprise!) - usually we prohibit this for the sake of simplicity.

Graphs with no loops and no multiple edges are called simple graphs. Unless otherwise noted, all the graphs we discuss will be simple graphs.

If the pairs in E are unordered pairs (ie. the pair (x, y) is the same as the pair (y, x)) then we say the graph is UNDIRECTED. If the pairs in E are ordered pairs (ie. the pair (x, y) is **not** the same as the pair (y, x)) then we say the graph is DIRECTED and we say that the edges are directed from the first vertex in the pair to the second. When writing a directed edge it is traditional to place a small left-to-right arrow above it, so the directed edge from x to y is written as $(\overrightarrow{x, y})$. We call x the TAIL of the edge, and y the HEAD of the edge. When drawing a directed edge in a graph, we use an arrow that goes from the tail to the head of the edge.

It is possible for a graph to contain some undirected edges and some directed edges. Such a graph is called MIXED.

In our extremely brief study of the data structures that are suitable for graphs we will deal almost exclusively with undirected graphs.

We can say that undirected graphs are just a special case of directed graphs because an undirected edge (x, y) can be re-imagined as two directed edges $(\overrightarrow{x, y})$ and $(\overrightarrow{y, x})$. This idea was explored extensively by the Canadian graph theorist William Tutte, who was a professor at Waterloo. During World War II he played a vital role in Turing's code-breaking work. Tutte's contributions are considered to have shortened the war significantly. Look him up on Wikipedia!

The DEGREE of a vertex is the number of times the vertex appears in E. If loops are prohibited (as is usual), the degree of a vertex is the number of edges that touch the vertex. We use $d(x)$ to denote the degree of x .

A WALK is a sequence of vertices such that consecutive vertices in the sequence are adjacent in the graph (ie. each pair of consecutive vertices in the sequence are joined by an edge in the graph). A walk can contain the same edge multiple times.

A CIRCUIT is a walk where the first and last vertices are the same.

A PATH is a walk in which all the vertices and edges are distinct.

A CYCLE is a circuit in which all the vertices and edges are distinct (with the exception of the first vertex, which is also the last vertex).

An equivalent definition of a cycle is a path plus one more edge that joins the two end vertices of the path.

Unless otherwise noted, we will always limit our discussions to

- undirected graphs with no multiple edges and no loops
- paths (rather than walks)
- cycles (rather than circuits)

Edges may be weighted. The weight of an edge e is usually denoted by $w(e)$. If the edge is identified by its end vertices such as

(x, y) , we will sometimes indicate its weight by $w(x, y)$. Edge weights are sometimes referred to as costs.

In general, graphs are used to represent bilateral (ie. undirected) or unilateral (ie. directed) pairwise relationships between discrete entities. In a communication network, the edges could represent the existence of a direct link between two objects in the network. In a social network, the edges could represent "friend" links.

A graph G is CONNECTED if for each pair of vertices x and y, there is a path in G that starts at x and ends at y. Connectivity is a fundamentally important property of graphs and we will look at methods of determining if a graph is connected.

One extremely important problem in applied graph theory is the problem of finding shortest paths in a graph (applications include internet routing and road trip planning). In an unweighted graph, the shortest path between vertex x and vertex y is a path with one end at x, the other end at y, and the least possible number of edges of all such paths.

In a weighted graph, the term "shortest path" between x and y is often used to mean a path with one end at x and the other end at y that has the least sum of the weights of its edges of all such paths. This is more properly called a "least weight path" but the two terms are often used interchangeably.

If we think of an unweighted graph as actually having $w(e) = 1$ for all edges, the two versions of the problem become the same.

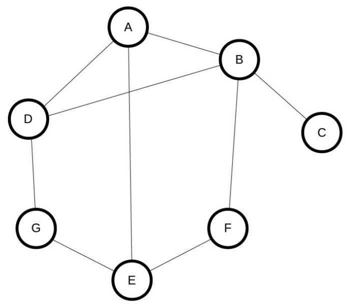
We will examine an algorithm that, given a starting vertex v in an unweighted graph, will find shortest paths from v to all other vertices in the graph. We actually mentioned this algorithm elsewhere in these notes when we looked at traversals of binary trees - the algorithm is Breadth-First Search.

Breadth-First Search

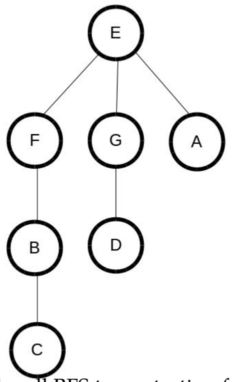
THE IDEA OF BFS is to explore the graph in levels. The first level consists of the start vertex, then all of its neighbours are on the next level, then all of their neighbours are on the next level, etc. We keep track of which vertices we have seen so that we don't go around in circles.

... or cycles, ha ha.

Consider this graph. If we start at vertex E, the set of vertices on the first level is just E. its neighbours are G, A and F - we do not specify an order for these vertices, and the set of vertices on the next level is $\{F, G, A\}$ To get the next level, we take one of these vertices ... say F ... and add all of its neighbours that we have not already seen to the set for the next level. This gives us B. Then we take another vertex from the current level ... say G ... and add its neighbours (again, only the ones we haven't seen) to the new set, which is now {B,D}. And then we take the next (and in this case last) vertex from the current set, which is A. A has no new neighbours to contribute, so the next level consists of {B,D}. The final level of the exploration consists of just {C}.



We often keep track of the "neighbour" relations that we are using - this lets us build a BFS tree for the graph. The exploration above would give this tree:



It is important to see that this will always generate a tree - the rule that we only go to each vertex once guarantees this. It is also important to be aware that the tree is not unique.

Even though we can get several different trees by applying BFS and starting at E, you will see that they have something in common: the sets of vertices at each level are always the same. Make sure you can explain why this is true.

To see why all BFS trees starting from E have the same level sets, draw the tree that starts at E but then chooses A as the first vertex on the next level to explore further - then do it again and choose G first.

It is also important to see that the paths with one end at E in a BFS tree that starts at E are in fact shortest paths from E to all the other vertices. In fact this is easy to see: if there were a shorter path from E to some vertex X, then X would be on a higher level of the BFS tree. Each vertex is added to the tree as soon as it is reachable.

We should also recognize that the number of levels in the BFS tree can vary depending on the vertex where we start. To convince yourself of this, find a vertex in the graph shown above which gives BFS trees with a different number of levels than the BFS trees that start at E.

Let's look at pseudo-code for the BFS algorithm. The first thing we need to do is decide how to keep track of the vertices at the "current" level while we build the set of vertices at the "next" level. Fortunately this is very easy - we can just use a queue. Each time we find new neighbours for a vertex at the current level, we add them to the end of the queue. This ensures that we will work through all vertices at the current level before we start on the next level.

"New" in the sense that we haven't seen them before.

We need to make sure each vertex is only added to the queue once - we can do this by adding a Boolean attribute to the Vertex object, or we can simply create a Boolean array of length n with one element for each vertex. Either way, we set the Boolean for vertex X to True when we add X to the queue.

```
def BFS(v):
    Q = new queue
    Q.add(v)
    set the "added_to_queue" Boolean for v to True
    while Q is not empty:
        x = Q.remove_first()
        process x            # This is actually just a place-holder operation.
                             # In practice we would probably print x's value,
                             # or add x to the BFS tree, or record x's distance
                             # from v, or modify x's value, or do some other
                             # specific processing
        for each neighbour y of x:
            if (y not marked "added_to_queue"):
                Q.add(y)
                set the "added_to_queue" Boolean for y to True
```

You should work through this for the graph in our examples to see how it works. You will observe that each vertex is added to Q exactly once and removed from Q exactly once.

If we want to build the tree, every time we add a neighbour y of x to the queue we need to include the edge from x to y in our tree. We can represent the tree by the list of edges we have used.

The tree that we generate through BFS has a useful property: the graph is connected if and only if a BFS tree of it contains all n vertices. This gives us a very practical method to test a graph for connectivity. When we look at other graph algorithms we often start with "Assume G is connected". We can make that assumption because now we have a simple way to test it - and if the graph is not connected we can either deal with each piece separately, or (if appropriate) add some more edges until it is connected.

Now we can address the crucial question: what is the complexity of this algorithm? First, we can assume that all queue operations are in $O(1)$. Second, we can see that (assuming the graph is connected), all n vertices go onto Q and come off Q again. Thus the **`while`** loop executes n times. The tricky part of the analysis is the **`for`** loop. We need to find or build a list of the neighbours of vertex x. **The time required to do this depends on the graph representation we are using.**

There are two candidates: an Adjacency Matrix, or a set of Adjacency Lists. We now interrupt our discussion of the complexity of the BFS algorithm to describe these structures. Regularly scheduled programming will resume shortly.

Adjacency Matrix Representation

An adjacency matrix for a graph G with n vertices is an $n * n$ matrix A where

- $A[i,j] = 1$ if vertex i is adjacent to vertex j
- $A[i,j] = 0$ if vertex i is not adjacent to vertex j

If the edges have weights, then

- $A[i,j] = w(i,j)$ if vertex i is adjacent to vertex j
- $A[i,j] = 0$ if vertex i is not adjacent to vertex j

Note that the size of A is n^2 regardless of the number of edges in E

If the graph is undirected, $A[i,j] = A[j,i]$. The lower left diagonal half of the matrix is the mirror image of the upper right diagonal half. In practice we often don't bother filling in the lower left diagonal half because all of its information is already in the upper right diagonal half.

In the dark ages of computing when memory was critically expensive and usually in extremely short supply, we used to use those empty array elements to store other data.

However if the edges are directed, we set $A[i,j] = 1$ (or $w(i,j)$ for a weighted graph) if there is a directed edge from vertex i to vertex j, and $A[i,j] = 0$ otherwise. In the adjacency matrix for a directed graph, it is not necessarily true that $A[i,j] = A[j,i]$.

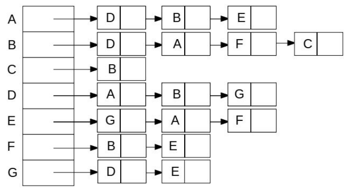
Here is the adjacency matrix for the graph shown above. Note that the ordering of the rows and columns is arbitrary - as long as we know which row and which column corresponds to each vertex, we are ok.

	A	B	C	D	E	F	G
A	0	1	0	1	1	0	0
B	1	0	1	1	0	1	0
C	0	1	0	0	0	0	0
D	1	1	0	0	0	0	1
E	1	0	0	0	0	1	1
F	0	1	0	0	1	0	0
G	0	0	0	1	1	0	0

Adjacency Lists Representation

The idea behind the adjacency lists representation of a graph is to store a list of all the neighbours of each vertex. We use a 1-dimensional array for the vertices. To the array element for each vertex we attach a linked list of the neighbours of that vertex - the list of neighbours is in no particular order. If the graph is undirected, each edge will create an entry in two of the adjacency lists. If the edges are directed, an edge from vertex i to vertex j would be represented by including j in the adjacency list for vertex i but not vice versa.

If the edges are weighted the weights of the edges can be stored as auxiliary data in the adjacency lists. Here is an adjacency list representation of the graph shown above:



Back to BFS

We now return to the discussion of the complexity of the BFS algorithm. As you will recall, we were trying to determine the complexity of the *for* loop in which we need to find all neighbours of a vertex x.

If we are using an adjacency matrix this will take $O(n)$ time, even if vertex x has very few neighbours ... because we have to look at an entire row of the matrix A. This gives our BFS algorithm $O(n^2)$ complexity.

But if we are using adjacency lists we don't need to do any work to obtain the neighbours of x. The structure simply gives us the list of neighbours and all we need to do is iterate through it.

This means that the inner loop (the *for* loop) will potentially execute a different number of times for each vertex - this may make it seem impossible to compute the complexity. But here is one idea (not a very good one as it turns out): each vertex has at most $n-1$ neighbours, so this inner loop will execute at most $n-1$ times for each vertex ... this gives us $O(n^2)$ complexity again.

A bit more thought lets us obtain a more precise bound. Once we have the list of neighbours of x, the "if (y not marked "added_to_queue")" test executes once for each neighbour of x. And since each vertex gets to take the role of x exactly once, the total number of times this test is executed is exactly the sum of the lengths of all the adjacency lists.

And we know what that is! The length of each vertex's adjacency list is the degree of the vertex, so we just need to sum the degrees. We know from basic graph theory that for any graph,

$$\sum d(v) = 2m$$

where $d(v)$ is the degree of vertex v and m is the number of edges in the graph.

So this test (which is the most frequently executed line in the algorithm) executes $O(m)$ times.

In summary, when we use adjacency lists the complexity of BFS is in $O(m)$.

Some authors write this as $O(n+m)$ and this is technically correct ... but I am assuming the graph is at least potentially connected, which means $m \geq n-1$. This makes it unnecessary to include the n in the complexity for our analysis.

So we have two different complexities, depending on which data structure we use. Which one is better? Does it make a difference?

Recall that $m \leq \binom{n}{2} = \dfrac{n*(n-1)}{2}$, so $O(m)$ can never be worse than $O(n^2)$.

But in many "real-world" graphs, most (or all) of the vertices have very low degree. It is not uncommon to have a restriction such as "all vertices have degree $\leq k$" where k is a small constant. For example, the graph might represent a network of micro-processors, each of which can only connect to at most 4 other processors. If all degrees are $\leq k$, the number of edges is $\leq \dfrac{k*n}{2}$ ..... which is in $O(n)$ because k is constant.

This fact leads to the depressing truth that it is probably true that on average, your friends have more friends than you do! This is known as the Friendship Paradox, but it's not really a paradox, just a surprising result of the typical structure of social networks.

So in this situation BFS runs in $O(n)$ time using adjacency lists, and in $O(n^2)$ time using an adjacency matrix.

The bottom line is that for this algorithm, adjacency lists are never worse than the adjacency matrix, and they are sometimes much better.

There are other problems and algorithms for which the adjacency matrix is preferable. For example, certain structural properties of the graph can be determined by performing algebraic computations on the adjacency matrix.