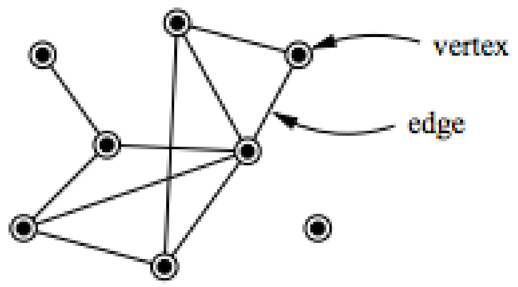
Notes from CCC prep with Troy Vasiga

C. Woodford

Feb 12, 2018

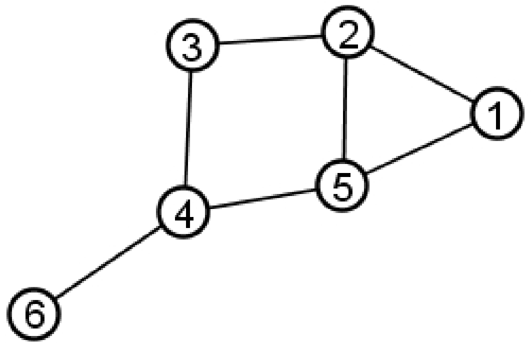
1 Definitions

Think of the different kinds of graphs that we've used and seen before: scatter plots, histograms, pie charts, line graphs, etc. These are useful for conveying information, but their not the typical computer science and math way of defining a graph. A graph can be defined as a series of vertices (points) and edges, like in the graphic below.



For these graphs the idea is that the vertices represent points and the edges are links between those points. Notice that not all vertices have edges between them, some are separated by more than one edge, and so on.

Define V = set of vertices, and E = set of edges. Then we can define the graph $G = (V, E)$, and the pairs (v, w), where v is a vertex and w is an edge. Some examples of graphs like this include, airways, car information, resources (electricity, gas, water). Maybe we want to find the neighbours in our graph. Where can I go next from a vertex? Typically you have to check every edge, but that's inefficient. What else can we do? We can store them differently so that it's not just a list of vertices and edges. Let's take the example graph shown in the below graphic:



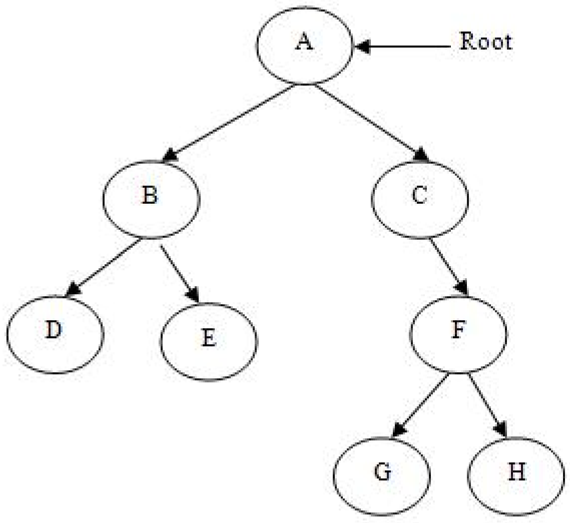
We can see that some of the vertices are connected by edges, but not all of the vertices are connected to each other. We can create an *adjacency* list that contains all of the connected vertices:

$$\begin{pmatrix}1\\2\\3\\4\\5\\6\end{pmatrix} \to \begin{pmatrix}2,3\\1,3,5\\2,4\\3,5,6\\1,2,4\\4\end{pmatrix} \tag{1}$$

This is one way to represent the information, but what we really want is a 2D array, a matrix of booleans that tell us which nodes are connected to which other nodes. ie each row and column relate to a point and the matrix is filled with 0's and 1's, like in the matrix below. You may notice that this is a symmetric matrix. Now checking for connections is 1 step. You can also check for ALL connections by looking at a single row or a single column. This is an *adjacency* matrix. Lists are more memory efficient but take longer to use, matrices are less memory efficient but are faster to use.

$$\begin{array}{c c} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Lastly, a tree is a connected graph without cycles. This means exactly one path from each vertex. An example is shown below.



2 Searches

Now that we know how to store information in adjancy matrices, we need a way of searching through them. There are 2 main search methods:

2.1 Breadth-first search, BFS

Visit all nodes reachable from node v in breadth-first order.
We can make a function for this search:

BFS(v):

1. initialize a queue Q: (FIFO, first-in first-out structures). How do we make one?
    - Make a list, as you move through move "first" down the list and as other items come in, put them as "last".
    - What if we make a linked list? Say, instead of having a "line" (0 to n-1 items), we think of it as a circle? as in, the last item in the list (n-1) is "linked" to the first item (item 0)[1]. How do we keep track?

[1]This is called "periodic", and is the type of storing style used for solving differential equations

- Use a modulo. Dividing the item numbers by n will give you a remainder. The only one with 0 remainder is the first element.

2. Mark v as visited and add v to Q
3. while Q is not empty: remove the front element from. Then for each edge w, check u: if u is not visited, mark as visited and add to Q.

Say we start at 2 in our graph diagram on the previous page. Then Q:2, visited:2. Then remove 2 from Q, check edge to 1. Then Q: 1, visited: 2,1. Then we also have edges to 5 and 3, so Q loses 1, gains 3, loses 3, gains 5, and so Q: {2,1,3},5, visited: 2,1,3,5. Then we go through the edge to 4, and then to 6 leaving us with Q:{2,1,3,5,4,6}, visited: 2,1,3,5,4,6.
Notice that we're moving out in a ripple pattern. If we numbered then according to how many "ripples" it took, we start at 2 (ripple 0), then hit 3,5,1 (ripple 1), then 4 (ripple 2), and finally 6 (ripple 3). So we had a total of 3 searches (starting point doesn't count). If we were to write this more explicitly according to the function defined above for BFS, then we would input BFS(2) and pass through parts 1,2. When we hit part 3, Q is NOT empty (it has 2 in it), and so cycle through the edges for 1,3,5. Then we check if Q is empty, see that it still has 5, cycle through the remaining edges to 4, check if Q is empty again and see that 4 is still in there, and so cycle through the edges to 6. Q is then checked a final time, but all of the vertices u along the edges from 6 have been visited, so we're done.

2.2 Depth-first search (DFS)

BFS doesn't always work best, especially for a lot of data as it may take too long to look at each node and edge. DFS is best when you may be looking at something like a chess game tree. ie. You have a bunch of opening moves, and then from the "branch" that you use, your opponent then has a number of moves that follow-up. And so on along the branches - but we don't want to look at each possible move (BFS) we just want to find the assortment of moves that will lead to winning the game.
DFS(v)

1. mark v as visited
2. for each edge w connecting to a vertex u
   (a) if u is not visited, call DFS(u)

If you don't use a recursive format, then you'll need to push it onto a stack. This is a LIFO structure (Last-in first-out). Think of a stack of plates - you wash one and put it on top, and then use that same one on top first.

Using the same graph and starting at 2: Current: 2, visited: 2. Now look at each edge. Go to 1. Call DFS(1). So now Current: 2,1, visited: 2,1. We've already looked at the edge between 1 and 2, so look at the edge to 5. Call DFS(5). Current: 2,1,5, visited: 2,1,5. Already used the edge from 5 to 1, so look at the edge to 4. Call DFS(4), Current: 2,1,5,4, visited: 2,1,5,4. Call DFS(3), Current: 2,1,5,4,3, visited: 2,1,5,4,3. But we've already visited 2 and 4, so that's it for 3, so take it out of the stack and go back to 4: Current: 2,1,5,4 {3}, visited: 2,1,5,4,3. Then we can travel to 6, call DFS(6), current: 2,1,5,4,{3},6, visited: 2,1,5,4,3,6. But we've visited everything, so then our current stack does: 2,1,5,4,{3,6}, goes to 4 but we've visited everything so 2,1,5 {4,3,6}, then go to five and so on until everything in the stack has been checked.

Note that this is NOT the fasted route to any of the nodes. BUT it does give you a tree. A spanning tree - which if you have a specific criteria that you're checking the nodes for, you can incorporate into this search and stop it once you've found what you're looking for (say, checkmate).

2.3 Minimum spanning tree

Say instead of assuming that all edges are the same, we assign "weights". Think of flight paths - even though 2 flights are only 1 connection, they may be different distances and different prices. This then leads to different search patterns:

2.3.1 Kruskal's

Add the smallest weighted edge which doesn't form a cycle. Can put in weights in the adjacency matrix - instead of booleans you put in the value for the weight. 0 can still be the "false", or no weight, option.

2.3.2 Prim's

From the vertices seen, what is the least weighted edge (v,x) where v are visited nodes and x are not yet visited nodes.

3 Additional Material

As another resource for BFS and DFS, you may find the following link helpful:
`http://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/`

Not much was said about Kruskal's method and Prim's method, but these are very common and very powerful searching algorithms. Here is a link that describes, walks through an example, and shows sample code for Kruskal's method:
`https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/`
Note that the python code uses items that we haven't covered yet, like classes. There are simpler ways to define a Kruskal minimum spanning function, like here:
`https://gist.github.com/hayderimran7/09960ca438a65a9bd10d0254b792f48f`
and here:
`http://blog.hguochen.com/programming/greedy%20algorithm/2014/06/17/Kruskal's-Algorithm/`.
Prim's algorithm can also be explained fairly easily, as seen here:
`https://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/`
which again uses classes. You can see other, simpler versions here:
`http://interactivepython.org/runestone/static/pythonds/Graphs/PrimsSpanningTreeAlgorithm.html`
and here (which also has a Kruskal example):
`https://pythonexample.com/code/prim-minimum-spanning-tree-algorithm-python/`