Laboratory practice No. 5: Graphs

Mariana Quintero Montoya
Universidad Eafit
Medellín, Colombia
mquintero3@eafit.edu.co

Isabella Pérez Serna
Universidad Eafit
Medellín, Colombia
iperezs2@eafit.edu.co

3) Practice for final project defense presentation

3.2 *The size of the matrix is n*n, as in this situation n is equivalent to 300.00, then 90,000,000,000 memory spaces would be consumed.*

3.3 *Solve the problem that the identifier of the point on the map does not start from zero by subtracting 1 from each identifier. This way it can be perfectly adjusted in the desired position in the matrix.*

3.5 *O(n^2)*

3.6 *n is equivalent to the number of vertices in the network.*

4) Practice for midterms

4.1

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

4.2

0 -> [3,4]
1 -> [0,2,5]
2 -> [1,6,4]

3 -> [7]
4 -> [2]
5 -> [ ]
6 -> [2]
7 -> [ ]

4.3 *O(n)*

4.4 .
4.4.1 ii)
4.4.2 i)

5) Recommended reading (optional)

Introduction to graphs Graphs are data structures quite similar to trees. In fact, in a mathematical sense, a tree is a kind of graph. However, in computer programming, graphics are used differently from trees. Often they have a shape dictated by a physical problem, for example, the nodes of a graph can represent cities, while the edges can represent airline flight paths between cities, the shape of the graph comes from the specific real-world situation.
Before we go any further, we should mention that, when talking about graphics, the nodes are called vertices, the circles represent highway interchanges, and the straight lines connecting the circles represent highway segments.
A graph is said to be connected if there is at least one path from each vertex to all other vertices, however, if "You can't get there from here" If you do not see the graphic, it is not connected. An offline graphic consists of several connected components.
To simplify, the algorithms we will discuss in this chapter are written to apply to connected graphics, or a connected component of an unconnected graphic. If appropriate, small modifications will allow you to work with unconnected graphics as well.
Targeted and weighted graphics Untargeted graphics can be moved from vertex A to vertex B, or from vertex B to vertex A, with equal ease.
In some graphics, the edges are given a weight, a number that can represent the physical distance between two vertices, or the time it takes to go from one vertex to another, or how much it costs to travel from one vertex to another. These graphs are called weighted graphs.
Representing a graph in a program, in most situations, a vertex represents some real-world object, and the object must be described using data elements. If a vertex represents a city in an airline route simulation, for example, it may be necessary to store the city's name, altitude, location, and other such information. Therefore, it is usually convenient to represent a vertex by an object of a vertex class. Vertices can also be placed in a list or some other data structure. It is not relevant how the vertices are connected by borders.
The Adjacency Matrix An adjacency matrix is a two-dimensional set in which the elements indicate whether an edge is present between two vertices. If a graph has N vertices, the adjacency matrix is an NxN set. This redundancy may seem inefficient, but there is no convenient way to create a triangle array in most computer languages, so it is simpler to accept redundancy.
Consequently, when a border is added to the graph, two entries should be made in the adjacency matrix instead of one. The adjacency list The other way to represent borders is with

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

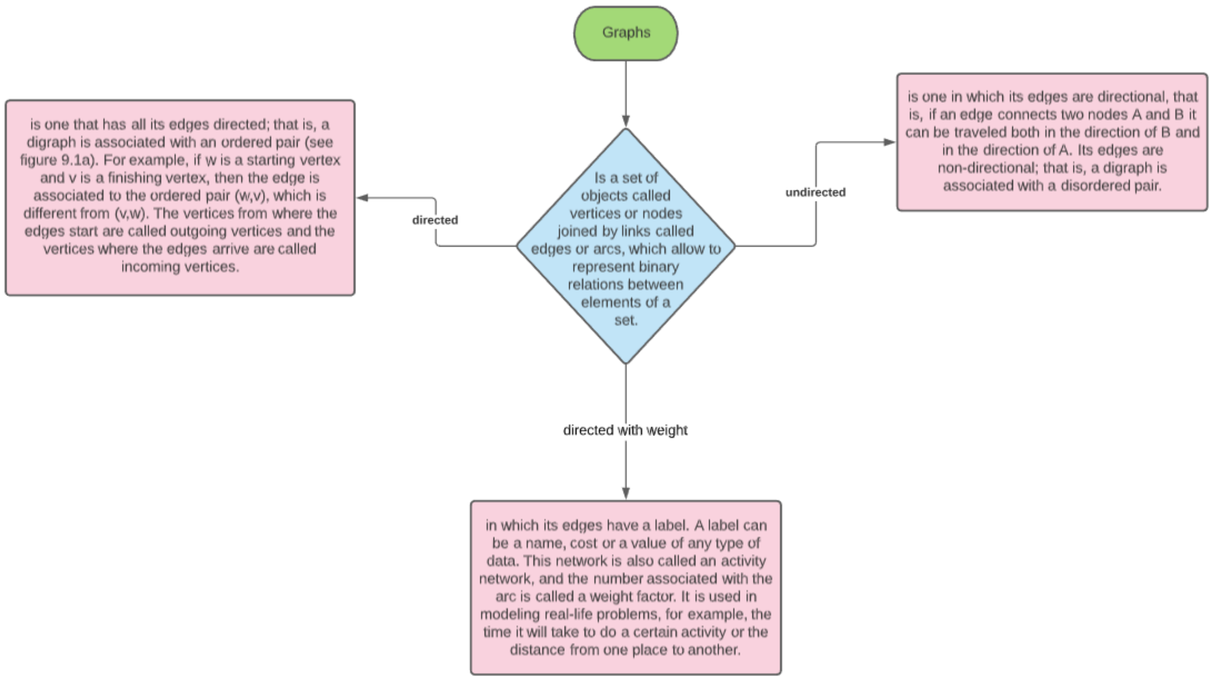
an adjacency list. The list in the adjacency list refers to a linked list, in fact, an adjacency list is a set of lists.
Each individual list shows which vertex a given vertex is adjacent to, each link in the list is a vertex.
Adding vertices and edges to a graphic To add a vertex to a graphic, a new vertex object is made with new vertex and inserted into its vertex matrix, vertexList. In a real world program a vertex can contain many data elements, but for simplicity we will assume that it contains only one character. Let's say you are using an adjacency array and you want to add a border between vertices 1 and 3. These numbers correspond to the array indexes in the vertex list where the vertices are stored.
The Graph class Let's see a Graph class that contains methods to create a list of vertices and an adjacency matrix, and to add vertices and edges to a Graph object. Within the Graph class, vertices are identified by their index number in vertexList. To display a vertex, we simply print its label of a character. The adjacency matrix provides information that is local to a given vertex, specifically, it tells you which vertices are connected by a single edge to a given vertex. To answer more global questions about the arrangement of vertices, we must resort to several algorithms.
Searches One of the most fundamental operations to perform on a graph is to find which vertices can be reached from a specific vertex. Here is another situation where you might need to find all the vertices that can be reached from a specific vertex. On a graph, each pin can be represented by a vertex, and each stroke by an edge. Now you need an algorithm that provides a systematic way to start at a specific vertex, and then move along the edges to other vertices, so that when it is done you are guaranteed to have visited every vertex that is connected to the starting vertex.
The depth-first search The depth-first search uses a stack to remind you where to go when you hit a dead end. The numbers in this figure show the order in which the vertices are visited. To carry out the depth-first search, you choose a starting point in this case, vertex A. Then you go to any vertex adjacent to A that has not yet been visited. We will assume that the vertices are selected in alphabetical order, so that brings us to B. You visit B, mark it, and push it over the stack.
Rule 1. Applying Rule 1 again brings you to H. At this point, however, you need to do something else, because there are no unvisited vertices adjacent to H. This is where Rule 2 comes into play. Following this rule, you take H out of the pile, which brings you back to F. F has no unvisited adjacent vertices, so you take it out. Now only A is left in the pile.
A, however, has adjacent unvisited vertices, so you visit the next one, C. This time, however, A has no unvisited neighbors, so you pop it out of the stack. But now there is nothing more to blow up, which brings us to rule 3. The content of the stack is the route you took from the initial vertex to get to where you are.
As you move away from the initial vertex, you push the vertices as you go. As you move back towards the initial vertex, you blow them up. The order in which you visit the vertices is ABFHCDGIE. At the beginning there are no vertices or edges, just an empty rectangle.
If you use the algorithm on an unconnected graph, it will only find the vertices that are connected to the starting vertex. Java code A key to the DFS algorithm is being able to find the vertices that are not visited and are adjacent to a specific vertex. Then you can check if this vertex is not visited. If so, you have found what you want the next vertex to visit. If no vertex in the row is simultaneously 1 and also not visited, then there are no unvisited vertices adjacent to the specified vertex. Now we are ready for the dfs method of the Graph class, that actually

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

performs the depth search, it examines the vertex at the top of the stack, using peek. It tries to find an unvisited neighbor of this vertex. If it finds such a vertex, it visits it and pushes it into the stack.
As we saw in the depth-first search, the algorithm acts as if it wanted to get as far away from the starting point as possible. In the amplitude-first search, on the other hand, the algorithm prefers to stay as close as possible to the starting point. It visits all the vertices adjacent to the starting vertex, and only then moves further away, this type of search is implemented using a queue instead of a stack. Now the queue is HI, but when you delete each of these and you don't find any unvisited adjacent vertex, the queue is empty, so you are done.
You can think of the search for amplitude-first as the waves widen when you drop a stone in the water or, for those of you who enjoy epidemiology, as the flu virus that air travelers carry from one city to another. The outer loop waits until the queue is empty, while the inner loop looks in turn at each unvisited neighbor of the current apex. It can be based on both depth and width searches.
Directed graphics In a directed graphic you can only proceed in one way along an edge. In a program, the difference between an un-directed graph and a directed graph is that an edge on a directed graph has only one entry in the adjacency matrix.
Thus, the border from A to B is represented by a single 1 in row A of column B. If the directed edge were reversed so that it went from B to A, there would be a 1 in row B of column A instead. For a non-directed graph, as noted above, half of the adjacency matrix reflects the other half, so half of the cells are redundant. When you use an algorithm to generate a topological sort, the approach you take and the details of the code determine which of the various valid sortings is generated. Although not shown here, a weighted graph can be used, which allows the graph to include the time needed to complete the different tasks of a project. The chart can then indicate things like the minimum time needed to complete the project. This applet works much like GraphN, but provides a point near one end of each border to show in which direction the border is pointing. The idea behind the topological classification algorithm is unusual but simple. The successors of a vertex are those vertices that are directly "downstream" of it², that is, connected to it by an edge pointing in its direction. If there is an edge that points from A to B, then B is a successor of A. Steps 1 and 2 are repeated until all vertices disappear. At this point, the list shows the vertices arranged in topological order. The algorithm cannot determine the second vertex to be eliminated until the first vertex disappears. If necessary, you can save the chart data to another location and restore it when the classification is completed. The algorithm works because if a vertex has no successors, it must be the last one in the topological order. As soon as it is deleted, one of the remaining vertices must have no successors, so it will be the penultimate in the ordering, and so on.
The topological classification algorithm works for both unconnected and connected plots. Cycles and trees One type of chart that the topological classification algorithm cannot handle is a chart with cycles. A chart without cycles is called a tree. In a graph, one vertex of a tree can be connected to any number of other vertices, as long as no cycles are created.
A topological classification is carried out in a directed graphic without cycles. Such a graph is called a directed, acyclic, often abbreviated DAG graph. Java Code The work is done in the while loop, which continues until the number of vertices is reduced to 0. Call noSuccessors to find any vertex without successors. If such a vertex is found, put the vertex label at the end of sortedArray and delete the vertex from the graph. If no such vertex is found, the chart must be cycled. The last vertex to be deleted appears first in the list, so the vertex label is placed on sortedArray starting at the end and working towards the beginning, as nVerts is reduced. If the

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473

vertices remain on the graph but all of them have successors, the graph must have a cycle, and the algorithm displays a message and closes.
Normally, the while loop exits, and the sortedArray list is displayed, with the vertices in topological order. The noSuccessors method uses the adjacency matrix to find a vertex without successors. In the for loop, go down the rows, looking at each vertex. For each vertex, scan the columns on the inside for the loop, looking for a 1.
If you find one, you know that that vertex has a successor, because there is an edge from that vertex to another. When it finds a 1, it leaves the inner loop so that the next vertex can be investigated. If no such vertex is found, 1 is returned. The vertex is removed from the vertexList array, and the vertices above it are moved down to fill the vacant position. Also, the row and column of the vertex is removed from the adjacent matrix, and the rows and columns above and to the right are moved down and to the left to fill the vacant position. It is actually more efficient to use the graph's adjacency list representation for this algorithm, but that would take us too far. The addEdge method inserts a single number in the adjacency matrix because it is a directed graph. Once the graph is created, main calls mole to sort the graph and display the result. The search algorithms allow to visit each vertex of a graph in a systematic way. The two main search algorithms are the depth search first and the width search first.
A minimum extension tree consists of the minimum number of edges needed to connect all the vertices of a graph. A slight modification of the depth-first search algorithm on an unweighted chart results in its minimum extension tree. In a directed graph, the edges have a direction. A topological classification can only be carried out on a DAG, a directional, acyclic plot.


Graphs
Is a set of objects called vertices or nodes joined by links called edges or arcs, which allow to represent binary relations between elements of a set.
directed
is one that has all its edges directed; that is, a digraph is associated with an ordered pair (see figure 9.1a). For example, if w is a starting vertex and v is a finishing vertex, then the edge is associated to the ordered pair (w,v), which is different from (v,w). The vertices from where the edges start are called outgoing vertices and the vertices where the edges arrive are called incoming vertices.
undirected
is one in which its edges are directional, that is, if an edge connects two nodes A and B it can be traveled both in the direction of B and in the direction of A. Its edges are non-directional; that is, a digraph is associated with a disordered pair.
directed with weight
in which its edges have a label. A label can be a name, cost or a value of any type of data. This network is also called an activity network, and the number associated with the arc is called a weight factor. It is used in modeling real-life problems, for example, the time it will take to do a certain activity or the distance from one place to another.

PhD. Mauricio Toro Bermúdez
Professor | School of Engineering | Informatics and Systems
Email: mtorobe@eafit.edu.co | Office: Building 19 – 627
Phone: (+57) (4) 261 95 00 Ext. 9473