Graph

quarta-feira, 16 de outubro de 2024 13:50

→ Graphs Algorithms

Searching in graph means systematically following the edges of the graph to visit all the vertices

GRAPH → Way of encoding pairwise relationship among a set of objectives

↳ Collection of V nodes and E edges, which joins the nodes

$u, v \in V$ $e \in E$ $> e = \{u, v\}$

↳ an EDGES connects two other nodes!

node ≡ vertex

↳ Indicates a symmetric relationship between nodes

• DIRECTED GRAPH (G')

↳ When we want to encode asymmetric relations

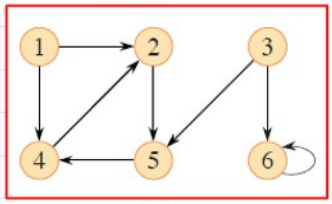
↳ Consist in a set of nodes (V) and a set of directed edges (E') based on an ordered pair (u, v)

$u \rightarrow$ tail
$v \rightarrow$ head

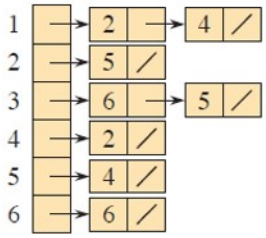
• edge e' leaves node u and enter node v!

↳ has a direction going from one node to the next

Direction does matter



(a)



(b)

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

(c)

Figure 20.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

• UNDIRECTED GRAPH (G')

↳ By default → when called a graph → it means that it's undirected

↳ It can go both ways ($A \rightarrow B$) ($B \rightarrow A$)

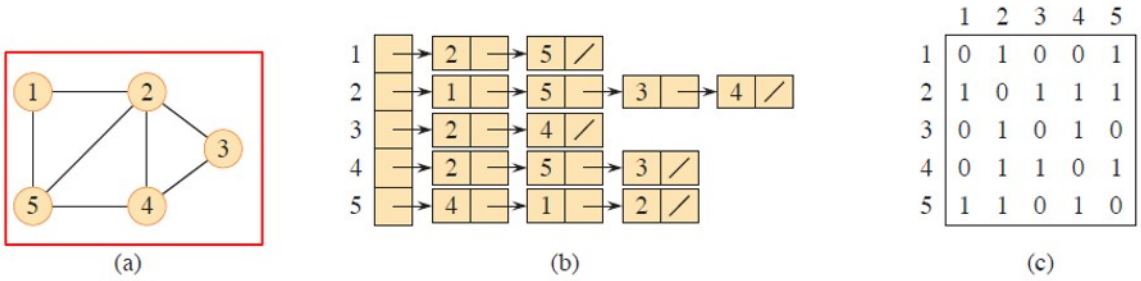


Figure 20.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Note that the edges can flow both ways

- PATH
  - Used to represent a traversing sequence of edges (find paths)
  - Are operation on graphs

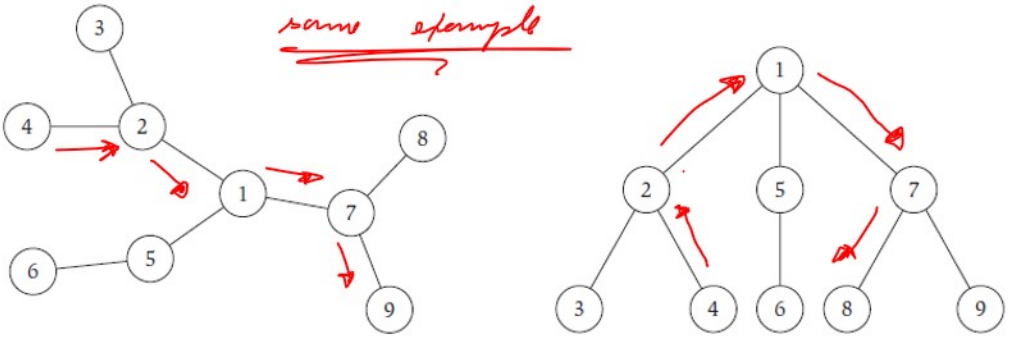


Figure 3.1 Two drawings of the same tree. On the right, the tree is rooted at node 1.

- SIMPLE PATH → if all vertices are distinct from one another
- Cycle → Path that comes back to the start point

a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $k > 2$, and $v_1 = v_k$—in other words, the seque

- for the direct graph, it must respect the edge connectivity
  - ↳ Strongly connected when able to move both ways ($A \to B$, $B \to A$)
- DISTANCE → Between 2 nodes → shortest distance
  - ↳ minimum number of edges in the path

Trees We say that an undirected graph is a *tree* if it is connected and does not contain a cycle. For example, the two graphs pictured in Figure 3.1 are trees. In a strong sense, trees are the simplest kind of connected graph: deleting any edge from a tree will disconnect it.

→ Every (N) node tree has exactly (N-1) edges

Representation of graphs

$G = (V, E)$ → Collection of adjacent lists (+) or adj matrix

→ Apply to direct and indirect graphs

→ prefer when the graph is dense

$G = (V, E)$ → $E \ll V^2$ → Sparse Graphs → Adjacent-list

$E \approx V^2$ → Dense Graphs → Adjacent-Matrix

- Adjacent-list representation

$G = (V, E)$ → array with adj $|V|$ list → One for each vertex

→ Its adj list contains all the vertices v connected by edges from the original vertex

an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa. For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is $\Theta(V + E)$. Finding each edge in the graph also takes $\Theta(V + E)$ time, rather than just $\Theta(E)$, since each of the $|V|$ adjacency lists must be examined. Of course, if $|E| = \Omega(V)$ —such as in a connected, undirected graph or a strongly connected, directed graph—we can say that finding each edge takes $\Theta(E)$ time.

Adjacency lists can also represent ***weighted graphs*** that is, graphs for which each edge has an associated ***weight*** given by a ***weight function*** $w : E \to \mathbb{R}$. For example, let $G = (V, E)$ be a weighted graph with weight function w. Then you can simply store the weight $w(u, v)$ of the edge $(u, v) \in E$ with vertex v in u's adjacency list. The adjacency-list representation is quite robust in that you can modify it to support many other graph variants.

DISADVANTAGE

→ No guarantee way to determine if a given edge (u, v) is present in the graph → Have to search (v) in the $Adj[u]$ list

present in the graph → then to search (v) in the Adj[u] list

• Adjacent - Matrix representation

$G = (V, E)$ → Vertices are numbered $\{1, 2, \ldots, |V|\}$ so that the representation consists on a $|V| \times |V|$ matrix

$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$

Disadvantage → Requires more memory to store it $\Theta(V^2)$

↳ Independent of the edge

But it's simple to verify if edge (u,v) exists!

↳ Advantage from the list representation

Like the adjacency-list representation of a graph, an adjacency matrix can represent a weighted graph. For example, if $G = (V, E)$ is a weighted graph with edge-weight function w, you can store the weight $w(u, v)$ of the edge $(u, v) \in E$

Although the adjacency-list representation is asymptotically at least as space-efficient as the adjacency-matrix representation, adjacency matrices are simpler, and so you might prefer them when graphs are reasonably small. Moreover, adjacency matrices carry a further advantage for unweighted graphs: they require only one bit per entry.

↳ Representations of vertices to store information

Representing attributes

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. We indicate these attributes using our usual notation, such as $v.d$ for an attribute d of a vertex v. When we indicate edges as pairs of vertices, we use the same style of notation. For example, if edges have an attribute f, then we denote this attribute for edge (u, v) by $(u, v).f$. For the purpose of presenting and understanding algorithms, our attribute notation suffices.

Implementing vertex and edge attributes in real programs can be another story entirely. There is no one best way to store and access vertex and edge attributes. For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph. If you represent a graph using adjacency lists, one design choice is to represent vertex attributes in additional arrays, such as an array $d[1 : |V|]$ that parallels the Adj array. If the vertices adjacent to u belong to $Adj[u]$, then the attribute $u.d$ can actually be stored in the array entry $d[u]$. Many other ways of implementing attributes are possible. For example, in an object-oriented programming language, vertex attributes might be represented as instance variables within a subclass of a `Vertex` class.