Introduction to Algorithms: Graph Traversals

Swarup Kr Ghosh

Introduction

In this part we discuss methods for representing a graph and for searching a graph. Searching a graph means systematically following the edges of the graph so as to visit the vertices of the graph. A graph-searching algorithm can discover much about the structure of a graph.

In particular, these traversals impose a type of tree structure (or generally a forest) on the graph, and trees are usually much easier to reason about than general graphs.

Many algorithms begin by searching their input graph to obtain this structural information. Several other graph algorithms elaborate on basic graph searching. Techniques for searching a graph lie at the heart of the field of graph algorithms.

Terminologies

A graph $G = (V, E)$ is a structure that represents a discrete set V of objects, called nodes or vertices, and a set of pairwise relations E between these objects, called edges. Edges may be directed from one node to another or may be undirected. The term digraph is sometimes used to indicate that directed edges are present.

Fundamental to graphs are notions such as paths, which are formed by a sequence of edges joined end to end, cycles, which are closed paths, and connectivity, which refers to a number of graph properties having to do with the nature reachability through paths.

People usually abuse notation and use V and E to denote both the set of nodes and edges as well as the numbers of nodes and edges, respectively. Sometimes, the convention is to use n for the number of nodes and m for the number of edges.

Examples of graphs are shown in the Figure 1.

Graphs are important computational structures because they can be used in such a wide variety of applications, including:

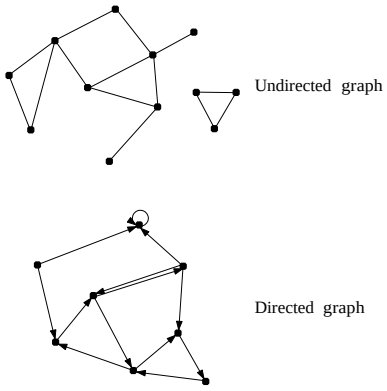

Figure 1: Example of graphs

Transportation networks Road or rail systems. For example the map of routes served by an airline carrier naturally forms a graph: the nodes are airports, and there is an edge from u to v if there is a nonstop flight that departs from u and arrives at v. Described this way, the graph is directed; but in practice when there is an edge (u, u), there is almost always an edge (u, u), so we would not lose much by treating the airline route map as an undirected graph with edges joining pairs of airports that have nonstop flights each way.

Communication networks For example, collection of computers connected via a communication network can be naturally modeled as a graph.

Information networks For example, the World Wide Web can be viewed as a directed graph, in which nodes correspond to Web pages and there is an edge from u to v if u has a hyperlink to v.

Social networks For modeling family, friend, or organizational structures, etc. For example, given any collection of people who interact (the employees of a company, the students in a high school, or the residents of a small town), we can define a network whose nodes are people, with an edge joining u and v if they are friends with one another.

Dependency networks Such as precedence relationships in scheduling. For example, given the list of courses offered by a college or university, we could have a node for each course and an edge from u to v if u is a prerequisite for v.

There are a number special classes of graphs that arise in various applications. Examples include:

- Trees, which are connected, acyclic (undirected) graphs.
- Bipartite graphs, in which the nodes a partitioned into two classes (e.g., men and women) and edges connect nodes of one class to the other.
- DAGs, an important special case of digraph. These are directed acyclic graphs. They arise often in dependency networks.

Graph Representation

There are two standard ways to represent a graph $G = (V, E)$ in computer memory: As a collection of adjacency-lists or as an adjacency-matrix. Either way applies to both directed and undirected graphs. Of

the two possibilities, adjacency-list representation provides a compact way to represent sparse graphs – those for which $|E|$ is much less than $|V|^2$ – it is usually the method of choice. We may prefer an adjacency-matrix representation, however, when the graph is dense – $|E|$ is close to $|V|^2$ – or when we need to be able to tell quickly if there is an edge connecting two given vertices.

The adjacency-list representation of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency-list Adj[u] contains all the vertices v such that there is an edge $(u, v) \in E$. Figure 2 and Figure 3 show examples of adjacency-list representation of undirected and directed graphs respectively.

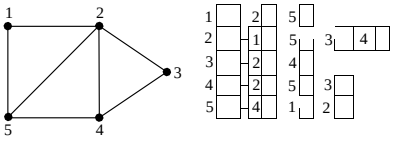

Figure 2: Adjacency-list representation

If G is a directed graph, the sum of the lengths of all the adjacency-lists in Adj[u] only. If G is 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 as well as v appears in u's adjacency-list.

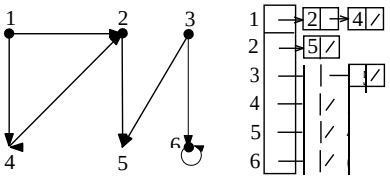

Figure 3: Adjacency-list representation of directed graphs

For both directed and undirected graphs $G = (V, E)$, the asymptotic complexity of memory requirement by its adjacency-list representation is $\Theta(V + E)$.

The adjacency-lists representation can readily be adapted to represent weighted graphs, i.e., graphs for which each edge has an associated weight, typically given by a weight function $w : E \rightarrow \mathbb{R}$. Here we store the weight w(u, v) of the edge $(u, v) \in E$ with vertex in u's adjacency-list. The adjacency-list representation is quite robust in the sense that we can modify it to support other graph variants also.

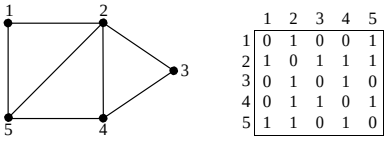

Figure 4: Adjacency-matrix representation of undirected graphs

A disadvantage of the adjacency-list representation is that it provides no quicker way to determine whether a given edge (u, v) is present in the graph than to search for in the adjacency-list Adj[u]. If this is the major operation in certain situation, then the alternative representation scheme that we might consider is adjacency-matrix representation with the cost of paying more memory requirement.

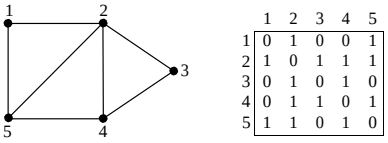

Figure 5: Adjacency-matrix representation of directed graphs

For the adjacency-matrix representation of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \ldots, |V|$ in some arbitrary manner. The adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $a = (a_{ij})$ such that

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

Figure 2 and Figure 3 show examples of adjacency-matrix representation of undirected and directed graphs respectively.

The adjacency matrix of a graph $G = (V, E)$ requires $\Theta(V^2)$ memory,

and this requirement is independent of the number of edges in the graph.

Since in an undirected graph, (u, v) and (v, u) represent the same edge, the adjacency-matrix A of an undirected graph G is its own transpose: $A = A^T$. In some applications, we can exploit this property to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half. Notice that, asymptotically, memory requirement remains the same. Like the adjacency-list representation of a graph, an adjacency-matrix also can represent a weighted graph.

For example, if $G = (V, E)$ is a weighted graph with edge weight function w, we can simply store the weight $w(u, v)$ of the edge $(u, v) \in E$ as the entry in row u and column v of the adjacency matrix. If an edge does not exist, we can store a Nil value as its corresponding matrix entry, though some times it is convenient to use a value such as 0 or ∞.

Although the adjacency-list representation is asymptotically at least as space efficient as the adjacency-matrix representation, adjacency matrices are simpler. So they may be preferable when graphs are reasonably small. Moreover, adjacency matrices carry a further advantage for unweighted graphs: they require only one bit per entry.

Breadth First Search (BFS)

Given an graph G(V, E), breadth-first search starts at some given source vertex s and "discovers" which vertices of G are reachable from s. The algorithm is so named because of the way in which it discovers vertices in a series of "layers".

Define the distance between a vertex v and s to be the minimum number of edges on a path from s to v. Note that we count edges, not vertices. Breadth-first search discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths. At any given time there is a "frontier" of vertices that have been discovered, but not yet processed. Breadth-first search is named because it visits vertices across the entire breadth of this frontier, thus extending from one layer to the next.

In order to implement BFS we need some way to ascertain which vertices have been visited and which haven't. This is done by suitable color coding of the vertices. Initially all vertices (except the source vertex) are colored "white", meaning that they are undiscovered. When

a vertex has first been discovered, it is colored "gray" (which means that it is part of the frontier). When processing of a gray vertex is complete it becomes "black". A black vertex is no longer considered by the algorithm.

The search makes use of a queue, a first-in first-out list, where elements are removed in the same order they are inserted. It is typically represented as a linked list or a circular array with a head and a tail pointers. The first item in the queue (the next to be removed) is called the head of the queue.

We also maintain arrays color[u] which holds the color of vertex u (either white, gray or black), pred[u] which points to the predecessor (or parent) of u (i.e. the vertex who first discovered u, and d[u], the distance from s to u. Notice that only the color is really needed for the search, which is the primary purpose of the algorithm. In fact, it is only necessary to know whether a node is nonwhite.

All the other information can be gathered during the search. These information are generally included because applications use BFS for these other information. Only visiting the vertices and the edges serves no useful purpose. We can now present the algorithm.

BFS(G, s)

```
 1  for each u ∈ V ** initialization
 2      color[u] = White; d[u] = ∞; pred[u] = Nil;
 3  color[s] = Gray ** mark source as discovered
 4  d[s] = 0 ** set distance of source
 5  Q = {s} ** put s in the queue
 6  while Q is nonempty
 7      u = dequeue from head of Q ** u is the next to visit
 8      for each v ∈ Adj[u] ** look at all neighbors of u
 9          if color[v] == White ** if v is undiscovered
10              color[v] = Gray ** mark it discovered
11              d[v] = d[u] + 1 ** set its distance from source
12              pred[v] = u ** set its predecessor
13              insert v at tail of Q ** put it in the queue
14      color[u] = Black
```

An example of BFS algorithm is shown in the Figyre 6.

As the above example execution shows, there is a natural physical interpretation of the algorithm. Essentially, we start at s and "flood" the graph with an expanding wave that grows to visit all nodes that it can reach. The layer containing a node represents the point in time at which the node is reached.

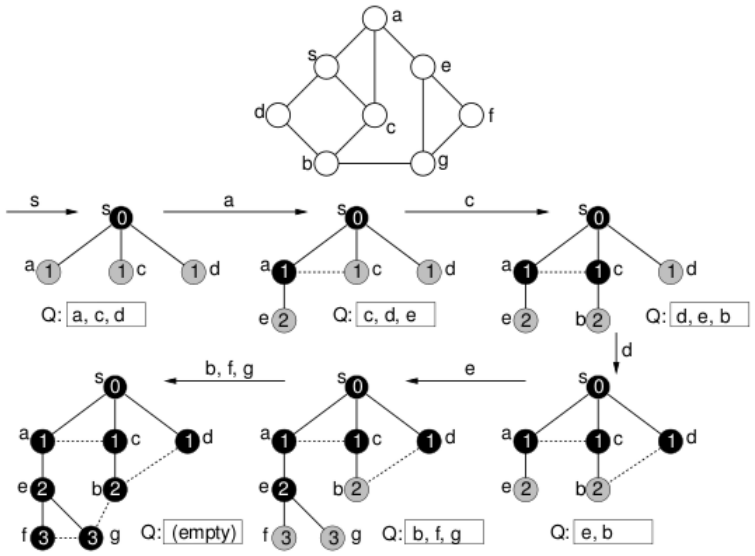

Figure 6: Example execution of the BFS algorithm

We can define the layers $L_1, L_2, L_3, \ldots$, constructed by the BFS algorithm more precisely as follows:

- Layer L_1 consists of all nodes that are neighbors of s. Sometimes the notation L_0 is used to denote the set consisting just of s.
- Assuming that we have defined layers $L_1, \ldots, L_j$, then layer L_{j+1} consists of all nodes that do not belong to an earlier layer and that have an edge to a node in layer L_j.

Shortest path

Recalling our definition of the distance between two nodes as the minimum number of edges on a path joining them, we see that layer L_1 is the set of all nodes at distance 1 from s. More generally layer L_j is the set of all nodes at distance exactly j from s.

A node fails to appear in any of the layers if and only if there is no path to it. Thus, BFS is not only determining the nodes that s can reach, it is also computing shortest paths to them. We can sum this up in the following fact:

Fact 1 For each $j \geq 1$, layer L_j produced by BFS consists of all nodes at distance exactly j from s. There is a path from s to t if and only if t appears in some layer.

We now claim that, on termination, d[v] is equal to the distance from s to v. Below is a sketch of the proof (see CLRS for a detail):

Theorem 1 Let $\delta(s, v)$ denote the length (number of edges) of the shortest path from s to v. Then, on termination of the BFS procedure, $d[v] = \delta(s, v)$.

Proof The proof is by induction on the length of the shortest path.
Let u be the predecessor of v on some shortest path from s to v, and among all such vertices the first to be processed by the BFS.
Thus, $\delta(s, v) = \delta(s, u) + 1$.
When u is processed, we have (by induction) $d[u] = \delta(s, u)$.
Since v is a neighbor of u, we set $d[v] = d[u] + 1$.
Thus we have

$$d[v] = d[u] + 1 = \delta(s, u) + 1 = \delta(s, v)$$

as desired. ∎

Assuming that BFS is already run on a given graph G w.r.t a source vertex s, the following procedure prints out the vertices on a shortest path from s to a given vertex v in the order v . . . s:

```
Print-Path(G, s, v)
1  if v == s
2        print s
3  elseif pred[v] == Nil
4        print "no path from" s "to" v "exists"
5  else
6        Print-Path(G, s, pred[v])
7  print v
```

BFS tree

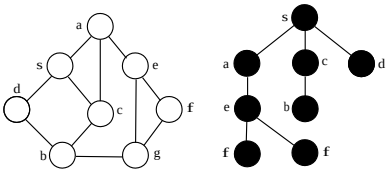

Figure 7: BFS tree

Observe that the predecessor pointers of the BFS search define an inverted tree: An acyclic directed graph in which the source s is the root, and every other node has a unique path to the root.

If we reverse these edges we get a normal rooted unordered tree called a breadth-first search tree or BFS tree for G. To see how the tree is constructed implicitly, consider the moment when v is first "discovered" by the BFS algorithm. This happens when some node in layer L_j is being examined, and we find that it has an edge to the previously unseen node v. At this moment, by setting pred[v] = u, we implicitly add the edge (u, v) to the tree.

Note that there are many potential BFS trees for a given graph, depending on where the search starts, and in what order vertices are placed on the queue. These edges of G that becomes part of the BFS tree are called tree edges and the remaining edges of G are called cross edges.

It is not hard to prove that if G is an undirected graph, then cross edges always go between two nodes that are at most one level apart in the BFS tree. (Prove it!).

Analysis

Observe that the initialization portion requires $\Theta(V)$ time. In the traversal loop, since we never visit a vertex twice, the number of times we go through the while loop is at most V. It would be exactly V when each vertex is reachable from the source. (Remember that the input graph is not required to be connected).

The number of iterations through the inner for loop is proportional to $\deg(u) + 1$. The $+1$ is because even if $\deg(u) = 0$, we need to spend a constant amount of time to set up the loop.

Summing up over all vertices we have the running time

$$\begin{aligned} T(V) &= V + \sum_{u \in V} (\deg(u) + 1) \\ &= V + \sum_{u \in V} \deg(u) + V \\ &= 2V + 2E \\ &\in \Theta(V + E) \end{aligned}$$