Graph Search

Adnan Aziz

Based on CLRS, Ch 22.
Recall—encountered graphs several weeks ago (CLRS B.4)

- restricted our attention to definitions, terminology, properties

Now we’ll see how to perform basic operations on graphs.

Implementation

Assumption—vertex set $V = \{1, 2, 3, \ldots, n\}$.
Two standard ways of implementing a graph $G = (V, E)$:

1. collection of adjacency lists
2. adjacency matrix

Usually, we prefer adjacency lists, since it uses considerably less memory when $|E| \ll |V|^2$.
A couple of situations where adjacency matrix is better:

- graph is “dense,” i.e., $|E| \approx |V|^2$
- frequently need to check whether an edge exists from $u$ to $v$

Adjacency list representation

Given a graph $G = (V, E)$, represent it by an array $Adj$ of $|V|$ lists, one for each vertex in $V$. For each $u \in V$ the entry $Adj[u]$ is a list of vertices $v$ such that there is an edge from $u$ to $v$.

- order within list can be arbitrary
Graph directed ⇒ sum of all adjacency list lengths is $|E|$; undirected ⇒ sum of all adjacency list lengths is $2 : |E|$.
Hence total memory needed to represent is $O(|V| + |E|)$.

“Weighted graph”—each edge has an associated weight; formally, there is a function $w : E \mapsto \mathbb{R}$.

- can easily store $w(u, v)$ in an adjacency list representation

**Adjacency matrix representation**

Can’t efficiently check if there is an edge from $u$ to $v$

- use adjacency matrix representation (possibly takes more memory)

Represent using a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

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

- needs $\Theta(|V|^2)$ memory, regardless of the number of edges

Recall—$A^T$ is the transpose of $A$, i.e., $a^T_{ij} = a_{ji}$. For an undirected graph, we have $a_{ij} = a_{ji}$; hence can save half the entries in the $A$ matrix.

Obvious—can use adjacency matrix representation with weighted graphs.

**Breadth first search**

Formally, given a graph $G = (V, E)$, and a designated “source” vertex $s$, what is the set of vertices that can be reached from $s$?

Various applications—geometric interpretation obvious (search maze), later will see less obvious applications (e.g., task scheduling).

BFS—a very simple algorithm for “searching a graph”

- prototype for many important graph algorithms

Idea—systematically explore the edges of $G$ to find (“discover”) each vertex reachable from $s$

- works on both directed and undirected graphs
• yields shortest paths, BFS tree

Conceptually—BFS uniformly expands the frontier between discovered and undiscovered vertices.

Details:

• vertices are colored white, gray, or black; initially all are colored white
  – a vertex is “discovered” the first time it’s encountered in the search, at which point we change its color to gray
    * the set of gray vertices represent the frontier

• process gray vertices: examine each outgoing edge, see if connected to any white vertex; if so, color that vertex gray
  – crucial point: insert newly discovered vertices at end of queue (“BFS queue”)
  – when done with all outgoing edges, remove vertex from the queue and change its color to black; process next vertex in queue

// CLRS page 532 - not showing depths, parents

BFS(G,s)
    for each vertex u in V[G] - {s}
        do color[u] <- WHITE
        color[s] <- GRAY
        enqueue(Q,s)
    while Q != 0
        do u <- head[Q]
           for each v in Adj[u]
               do if color[v] = WHITE
                      then color[v] = GRAY
                         enqueue(Q,v)
        dequeue(Q)
        color[u] = BLACK

Properties of BFS

• Correctness—after BFS terminates, all black vertices are reachable from s and all white vertices are colored white.
– easy part—black vertices are reachable
– slightly more involved—white vertices are not reachable

• **Runtime**—\(O(|V| + |E|)\)
  – each vertex enters \(Q\) at most once—\(O(|V|)\)
  – scan adjacency list of each vertex at most once—\(O(|E|)\)

• **Shortest distances**—define the *shortest-path distance* from \(s\) to \(v\) to be minimum number of edges in any path from \(s\) to \(v\) (set to \(\infty\) if there is no path).
  Can compute the shortest distances as follows:
  – keep an integer valued array \(d[x], x \in V\)
    * initially, \(d[s] = 0, d[x] = \infty, x \neq s\)
    * when discover a vertex \(v\) for the first time, say through edge \((u, v)\), then set \(d[v] = d[u] + 1\)
  Nontrivial fact—when BFS terminates, the \(d\) array holds the shortest distances.

• **BFS Tree**—initially, it’s just the source vertex.
  – whenever a white vertex \(v\) is discovered in the course of examining the outgoing edges of a vertex \(u\), add the vertex \(v\) and edge \((v, u)\) to the BFS tree.
    * set \(u\) to be the parent of \(v\) in the BFS tree
  Since a vertex is discovered at most once, it has at most one parent. Can talk about ancestor, descendant relative to source vertex.

  Fact—the BFS tree can be used to find shortest paths
  – follow the parent links all the way up to \(s\)

**Depth First Search**

Another way of exploring a graph

• keep exploring from most recently discovered vertex
  – nothing else left—“backtrack”
More natural way of exploring a graph.

// simplified version of algorithm in CLRS, page 541
DFS(G)
    for each vertex u in V[G]
        do color[u] <- WHITE
            p[u] <- NIL
        time <- 0
    for each vertex u in V[G]
        do if color[u] = WHITE
            then DFS_visit(u)

DFS_visit(u)
    color[u] = GRAY
    time <- time + 1
    d[u] <- time
    for each v in Adj[u]
        do if color[v] = WHITE
            then p[v] <- u
                DFS_visit(v)
    color[u] <- BLACK
    time <- time + 1
    f[u] <- time

Keep track of additional data in search: “timestamps” and “parents”

1. first time vertex encountered, time when vertex finished processing
2. vertex from which first discovered

Factoids:

- Can use DFS to “topologically sort” an acyclic graph $G = (V, E)$
  
  - linear ordering of all vertices such that if $(u, v) \in E$, then $u$ appears before $v$ in the ordering

- compute the “strongly connected components of a graph"
– strongly connected component—maximal set of vertices $U \subset V$ having the property that for every pair of vertices $x, y$ such that $x \in U$ and $y \in U$, there is a path from $x$ to $y$ and vice versa

• computation of “articulation points,” “bridges,” “biconnected components”

• widespread use in Artificial Intelligence