CSE 421: Introduction to Algorithms

Graph Traversal

Paul Beame

Directed Graph $G = (V,E)$

Generic Graph Traversal Algorithm

Find: set $R$ of vertices reachable from $s \in V$

Reachable($s$):

$R \leftarrow \{s\}$

While there is a $(u,v) \in E$ where $u \in R$ and $v \notin R$

Add $v$ to $R$

Graph Traversal

- Learn the basic structure of a graph
- Walk from a fixed starting vertex $s$ to find all vertices reachable from $s$
- Three states of vertices
  - unvisited
  - visited/discovered
  - fully-explored

Generic Traversal Always Works

Claim: At termination $R$ is the set of nodes reachable from $s$

Proof

- For every node $v \in R$ there is a path from $s$ to $v$
- Suppose there is a node $v \in R$ reachable from $s$ via a path $P$
  - Take first node $v$ on $P$ such that $v \notin R$
  - Predecessor $u$ of $v$ in $P$ satisfies
    - $u \in R$
    - $(u,v) \in E$
  - But this contradicts the fact that the algorithm exited the while loop.
**Breadth-First Search**

- Completely explore the vertices in order of their distance from $s$.
- Naturally implemented using a queue.

**BFS(s)**

Global initialization: mark all vertices "unvisited".

- **BFS(s)**
  - mark $s$ "visited"; $R ← \{s\}$; layer $L_0 ← \{s\}$
  - while $L_i$ not empty
    - $L_{i+1} ← ∅$
    - for each $u ∈ L_i$
      - for each edge $\{u, v\}$
        - if $v$ is "unvisited"
          - mark $v$ "visited"
          - Add $v$ to set $R$ and to layer $L_{i+1}$
          - mark $u$ "fully-explored"

**Properties of BFS(v)**

- BFS(s) visits $x$ if and only if there is a path in $G$ from $s$ to $x$.
- Edges followed to undiscovered vertices define a "breadth first spanning tree" of $G$.
- Layer $i$ in this tree, $L_i$ is those vertices $u$ such that the shortest path in $G$ from the root $s$ is of length $i$.
- On undirected graphs, all non-tree edges join vertices on the same or adjacent layers.

**Properties of BFS**

- On undirected graphs, all non-tree edges join vertices on the same or adjacent layers.
- Suppose not.
  - Then there would be vertices $(x, y)$ such that $x ∈ L_i$ and $y ∈ L_j$ and $j > i + 1$.
  - Then, when vertices incident to $x$ are considered in BFS, $y$ would be added to $L_{i+1}$ and not to $L_j$.

**BFS Application: Shortest Paths**

- Tree gives shortest paths from start vertex.
- Can label by distances from start.

**Graph Search Application: Connected Components**

- Want to answer questions of the form:
  - **Given**: vertices $u$ and $v$ in $G$
  - Is there a path from $u$ to $v$?
- **Idea**: create array $A$ such that $A[u] = \text{smallest numbered vertex that is connected to } u$

Q: Why not create an array $\text{Path}(u,v)$?
Graph Search Application: Connected Components

- initial state: all $v$ unvisited
- for $s = 1$ to $n$ do
  - if state($s$) ≠ "fully-explored" then
    - BFS($s$): setting $A[u] = s$ for each $u$ found
    - (and marking $u$ visited/fully-explored)
  - endif
- endfor
- Total cost: $O(n+m)$
  - each vertex is touched once in this outer procedure and the edges examined in the different BFS runs are disjoint
  - works also with Depth First Search

DFS(u) – Recursive version

Global Initialization: mark all vertices "unvisited"

- DFS(u)
  - mark $u$ "visited" and add $u$ to $R$
  - for each edge $(u,v)$
    - if (v is "unvisited")
      - DFS(v)
    - end for
  - mark $u$ "fully-explored"

Properties of DFS(s)

- Like BFS(s):
  - DFS(s) visits $x$ if and only if there is a path in $G$
  - Edges into undiscovered vertices define a "depth first spanning tree" of $G$
- Unlike the BFS tree:
  - the DFS spanning tree isn't minimum depth
  - its levels don't reflect min distance from the root
  - non-tree edges never join vertices on the same or adjacent levels
- BUT...

Non-tree edges

- All non-tree edges join a vertex and one of its descendents/ancestors in the DFS tree
- No cross edges.

No cross edges in DFS on undirected graphs

- Claim: During DFS(x) every vertex marked visited is a descendant of $x$ in the DFS tree $T$
- Claim: For every $x,y$ in the DFS tree $T$, if $(x,y)$ is an edge not in $T$ then one of $x$ or $y$ is an ancestor of the other in $T$
- Proof:
  - One of $x$ or $y$ is visited first, suppose WLOG that $x$ is visited first and therefore DFS(x) was called before DFS(y)
  - During DFS(x), the edge $(x,y)$ is examined
  - Since $(x,y)$ is a not an edge of $T$, $y$ was visited when the edge $(x,y)$ was examined during DFS(x)
  - Therefore $y$ was visited during the call to DFS(x) so $y$ is a descendant of $x$.

Applications of Graph Traversal: Bipartiteness Testing

- Easy: A graph $G$ is not bipartite if it contains an odd length cycle
- WLOG: $G$ is connected
  - Otherwise run on each component
- Simple idea: start coloring nodes starting at a given node $s$
  - Color $s$ red
  - Color all neighbors of $s$ blue
  - Color all their neighbors red
  - If you ever hit a node that was already colored
    - the same color as you want to color it, ignore it
    - the opposite color, output error
BFS gives Bipartiteness

- Run BFS assigning all vertices from layer $L_i$ the color $i \mod 2$
  - i.e. red if they are in an even layer, blue if in an odd layer
- If there is an edge joining two vertices from the same layer then output “Not Bipartite”

Why does it work?

- $u$ and $v$ have a common ancestor
- Cycle length $2(j-i)+1$

DFS($v$) for a directed graph

DFS($v$)

Properties of Directed DFS

- Before DFS($s$) returns, it visits all previously unvisited vertices reachable via directed paths from $s$
- Every cycle contains a back edge in the DFS tree

Directed Acyclic Graphs

- A directed graph $G=(V,E)$ is acyclic if it has no directed cycles
- Terminology: A directed acyclic graph is also called a DAG
Topological Sort

- **Given:** a directed acyclic graph (DAG) $G = (V, E)$
- **Output:** numbering of the vertices of $G$ with distinct numbers from 1 to $n$ so edges only go from lower number to higher numbered vertices

- **Applications**
  - nodes represent tasks
  - edges represent precedence between tasks
  - topological sort gives a sequential schedule for solving them

In-degree 0 vertices

- Every DAG has a vertex of in-degree 0
- **Proof:** By contradiction
  - Suppose every vertex has some incoming edge
  - Consider following procedure:
    
    ```
    while (true) do
      v ← some predecessor of v
    
    After $n+1$ steps where $n = |V|$ there will be a repeated vertex
    - This yields a cycle, contradicting that it is a DAG
    ```

Topological Sort

- Can do using DFS
- Alternative simpler idea:
  - Any vertex of in-degree 0 can be given number 1 to start
  - Remove it from the graph and then give a vertex of in-degree 0 number 2, etc.
Implementing Topological Sort

- Go through all edges, computing in-degree for each vertex \(O(m+n)\)
- Maintain a queue (or stack) of vertices of in-degree 0
- Remove any vertex in queue and number it
- When a vertex is removed, decrease in-degree of each of its neighbors by 1 and add them to the queue if their degree drops to 0
- Total cost \(O(m+n)\)