Directed Graphs


Digraphs

- A **digraph** is a graph whose edges are all directed
  - Short for “directed graph”

- Applications
  - One-way streets
  - Flights
  - Task scheduling
Digraph Properties

- A graph $G=(V,E)$ such that
  - Each edge goes in one direction:
  - Edge $(a,b)$ goes from $a$ to $b$, but not $b$ to $a$
- If $G$ is simple, $m \leq n(n-1)$
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size

Digraph Application

- **Scheduling**: edge $(a,b)$ means task $a$ must be completed before $b$ can be started
Directed DFS

- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction.
- In the directed DFS algorithm, we have four types of edges:
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex \( s \) determines the vertices reachable from \( s \).

The Directed DFS Algorithm

```
Algorithm DirectedDFS(G, v):
    Label v as active  // Every vertex is initially unexplored
    for each outgoing edge, e, that is incident to v in G do
        if e is unexplored then
            Let w be the destination vertex for e
            if w is unexplored and not active then
                Label e as a discovery edge
                DirectedDFS(G, w)
            else if w is active then
                Label e as a back edge
            else
                Label e as a forward/cross edge
    Label v as explored
```

© 2015 Goodrich and Tamassia

Directed Graphs
Reachability

- DFS tree rooted at v: vertices reachable from v via directed paths

```
A -> B
  |    |
  v    v
C ---- D
|
E <-- F
```

Strong Connectivity

- Each vertex can reach all other vertices

```
a ---------- c ---------- g
|                       |
|                       |
|                       |
d ---------- e ---------- h
|                       |
|                       |
|                       |

A graph where every vertex can reach every other vertex.
Strong Connectivity Algorithm

- Pick a vertex v in G
- Perform a DFS from v in G
  - If there’s a w not visited, print “no”
- Let G’ be G with edges reversed
- Perform a DFS from v in G’
  - If there’s a w not visited, print “no”
  - Else, print “yes”
- Running time: O(n+m)

Strongly Connected Components

- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in O(n+m) time using DFS, but is more complicated (similar to biconnectivity).

\[
\begin{align*}
\text{Strong Connectivity Algorithm} & \\
\{a, c, g\} & \\
\{f, d, e, b\} & 
\end{align*}
\]
Directed Graphs

Transitive Closure

- Given a digraph $G$, the transitive closure of $G$ is the digraph $G^*$ such that
  - $G^*$ has the same vertices as $G$
  - if $G$ has a directed path from $u$ to $v$ ($u \neq v$), $G^*$ has a directed edge from $u$ to $v$
- The transitive closure provides reachability information about a digraph

![Graph G and G*]

Computing the Transitive Closure

- We can perform DFS starting at each vertex
  - $O(n(n+m))$
- Alternatively ... Use dynamic programming: The Floyd-Warshall Algorithm

If there's a way to get from A to B and from B to C, then there's a way to get from A to C.
Floyd-Warshall Transitive Closure

- Idea #1: Number the vertices 1, 2, ..., n.
- Idea #2: Consider paths that use only vertices numbered 1, 2, ..., k, as intermediate vertices:

```
Uses only vertices numbered 1,...,k-1

Uses only vertices numbered 1,...,k
(add this edge if it’s not already in)
```

Floyd-Warshall’s Algorithm: High-Level View

- Number vertices \( v_1, \ldots, v_n \)
- Compute digraphs \( G_0, \ldots, G_n \)
  - \( G_0 = G \)
  - \( G_k \) has directed edge \((v_i, v_j)\) if \( G \) has a directed path from \( v_i \) to \( v_j \) with intermediate vertices in \( \{v_1, \ldots, v_k\} \)
- We have that \( G_n = G^* \)
- In phase \( k \), digraph \( G_k \) is computed from \( G_{k-1} \)
- Running time: \( O(n^3) \), assuming \( \text{areAdjacent} \) is \( O(1) \) (e.g., adjacency matrix)
The Floyd-Warshall Algorithm

Algorithm FloydWarshall(G):
  Input: A digraph G with n vertices
  Output: The transitive closure G\textsuperscript{*} of G
  Let v\textsubscript{1}, v\textsubscript{2}, \ldots, v\textsubscript{n} be an arbitrary numbering of the vertices of G
  \[ G\textsubscript{0} \leftarrow G \]
  for k ← 1 to n do
    \[ G\textsubscript{k} \leftarrow G\textsubscript{k-1} \]
    for i ← 1 to n, i ≠ k do
      for j ← 1 to n, j ≠ i, k do
        if both edges (v\textsubscript{i}, v\textsubscript{k}) and (v\textsubscript{k}, v\textsubscript{j}) are in G\textsubscript{k-1} then
          if G\textsubscript{k} does not contain directed edge (v\textsubscript{i}, v\textsubscript{j}) then
            add directed edge (v\textsubscript{i}, v\textsubscript{j}) to G\textsubscript{k}

return G\textsubscript{n}

- The running time is clearly O(n\textsuperscript{3}).

Floyd-Warshall Example

© 2015 Goodrich and Tamassia
Floyd-Warshall, Iteration 5

Floyd-Warshall, Iteration 6
Floyd-Warshall, Conclusion

DAGs and Topological Ordering

- A directed acyclic graph (DAG) is a digraph that has no directed cycles.
- A topological ordering of a digraph is a numbering of the vertices such that for every edge \((v_i, v_j)\), we have \(i < j\).

**Example**: in a task scheduling digraph, a topological ordering is a task sequence that satisfies the precedence constraints.

**Theorem**
A digraph admits a topological ordering if and only if it is a DAG.
Topological Sorting

- Number vertices, so that \((u,v)\) in \(E\) implies \(u < v\)

A typical student day

1. Wake up
2. Study computer sci.
3. Eat
4. Nap
5. More c.s.
6. Work out
7. Play
8. Write c.s. program
9. Bake cookies
10. Sleep
11. Dream about graphs

Algorithm for Topological Sorting

- Note: This algorithm is different than the one in the book

```plaintext
Algorithm TopologicalSort(G)
    \(H \leftarrow G\) // Temporary copy of \(G\)
    \(n \leftarrow G.numVertices()\)
    while \(H\) is not empty do
        Let \(v\) be a vertex with no outgoing edges
        Label \(v \leftarrow n\)
        \(n \leftarrow n - 1\)
        Remove \(v\) from \(H\)
```

Running time: \(O(n + m)\)
Implementation with DFS

- Simulate the algorithm by using depth-first search
- $O(n+m)$ time.

Algorithm `topologicalDFS(G, v)`

**Input** graph $G$ and a start vertex $v$ of $G$

**Output** labeling of the vertices of $G$
in the connected component of $v$

```
setLabel(v, VISITED)
```

for all $e \in G$.outEdges($v$)

```
setLabel(w, UNEXPLORED)
```

if `getLabel(w) = UNEXPLORED`

```
topologicalDFS(G, w)
```

else

```
e is a forward or cross edge
```

Label $v$ with topological number $n$

```
n = n - 1
```

Algorithm `topologicalDFS(G)`

**Input** dag $G$

**Output** topological ordering of $G$

```
n = G.numVertices()
```

for all $u \in G$.vertices()

```
setLabel(u, UNEXPLORED)
```

for all $v \in G$.vertices()

```
if `getLabel(v) = UNEXPLORED`

```
topologicalDFS(G, v)
```

Topological Sorting Example
Topological Sorting Example

8 → 9

© 2015 Goodrich and Tamassia
Topological Sorting Example

Diagram showing a directed graph with nodes labeled 6, 7, 8, 9, and 7, 8, 9, with edges connecting the nodes in a specific order.
Topological Sorting Example

1. 2. 3. 4. 5. 6. 7. 8. 9.

© 2015 Goodrich and Tamassia
Directed Graphs
Topological Sorting Example

Directed Graphs

© 2015 Goodrich and Tamassia