Directed Graphs DFS 1.3 1

DFS on Directed Graphs

Directed Graphs DFS 1.3 2

Outline and Reading (§6.4)
- Reachability (§6.4.1)
  - Directed DFS
  - Strong connectivity
- Directed Acyclic Graphs (DAG’s) (§6.4.4)
  - Topological Sorting

Directed Graphs DFS 1.3 3

Digraphs
- A digraph is a graph whose edges are all directed
  - Short for “directed graph”
- Applications
  - one-way streets
  - flights
  - task scheduling

A digraph is a graph whose edges are all directed
- Short for “directed graph”
- Applications
  - one-way streets
  - flights
  - task scheduling

Outline and Reading (§6.4)
- Reachability (§6.4.1)
  - Directed DFS
  - Strong connectivity
- Directed Acyclic Graphs (DAG’s) (§6.4.4)
  - Topological Sorting

A digraph is a graph whose edges are all directed
- Short for “directed graph”
- Applications
  - one-way streets
  - flights
  - task scheduling
**Digraph Application**

- **Scheduling:** edge \((a, b)\) means task \(a\) must be completed before \(b\) can be started.

**Directed Graphs DFS**

- DFS on digraphs traverses edges only along their proper 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 \(x\) determines the vertices reachable from \(x\).

**Directed DFS example**

- DFS_Sweep starts at \(A\), then \(B\),...
A directed acyclic graph (DAG) is a digraph that has no directed cycles. A topological ordering of a digraph is a numbering $v_1, \ldots, v_n$ 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.

**Directed Graphs DFS 1.3**

- Shortest Path
- DAGs and Topological Ordering
- Directed Graphs DFS 1.3
- Topological Sorting
- Algorithm for Topological Sorting

**Topological Sorting**

Number vertices, so that $(u, v)$ in $E$ implies $u < v$.

A typical student day:

1. Wake up
2. Eat
3. Study computer science
4. Play
5. More c.s.
6. Work out
7. Study computer science
8. Write c.s.
9. Make cookies for professors
10. Dream about graphs
11. Dream about graphs

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

Running time: $O(n + m)$ [with smart implementation]. How...?
Topological Sorting
Algorithm using DFS

- Simulate the algorithm by using depth-first search

**Algorithm topologicalDFS(G, v)**

- Input: graph G and a start vertex v of G
- Output: a labeling of the vertices of G in the connected component of v in topological order

```plaintext
setLabel(v, VISITED)
for all e ∈ G.outIncidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← opposite(v, e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            topologicalDFS(G, w)
        else
            e is a forward or cross edge
    else
        setLabel(e, VISITED)
```

**O(n+m) time.**

---

Topological Sorting Example

---

Topological Sorting Example

---
Topological Sorting Example

Strong Connectivity

*Each vertex can reach all other vertices*
**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 be computed in $O(n+m)$ time using DFS.

**SCC algorithm**
- Using DFS_Sweep for directed graphs, construct list $L$ of reverse finish order of the vertices in the traversal.
  - Node is finished when traversal leaves it permanently.
- Do another DFS_Sweep on $G^R$, ($G$ with edges reversed), with the following modification: in DFS_Sweep outer loop, start DFS calls on vertices according to the order in list $L$.
- Each spanning tree produced by DFS_Sweep on $G^R$ will contain all nodes from exactly one SCC of $G$. 
**Strongly Connected Components**

1. **Phase 1**
   - Run DFS_Sweep on G, returning a list L of nodes in reverse finish order. Done by adding vertex v to the front of L after traversal on v is finished in DFS_Sweep.

2. **Phase 2a**
   - Construct $G^r$ from G by copying the vertices, and then adding the reverse of every edge from G to $G^r$.

3. **Phase 2b**
   - Do a modified DFS_Sweep traversal on $G^r$, where list L is used to order the DFS calls. Each DFS call labels vertices traversed with a different SCC number.

4. **Final Phase:**
   - Label vertices and edges of G.
DFS Phase 1

- Construct list L.
- Similar to topological sort.

**Algorithm SCCDFS_Sweep(G)**
- Input: dag G.
- Output: list L of vertices of G in reverse finish order.
- L = empty list.
  - for all \( u \in G\).vertices
    - setLabel(\( u \), VISITED)
  - for all \( e \in G\).edges
    - setLabel(\( e \), UNEXPLORED)
  - for all \( v \in G\).vertices (in reverse L in order)
    - if getLabel(\( v \)) = UNEXPLORED
      - SCCDFS(G, v)
      - L.insertFirst(\( v \))
    - else
      - e is a forward or cross edge

**Algorithm SCCDFS(G, v)**
- Input: graph G and a start vertex v of G.
- Output: vertices of G in the connected component of v added to L, according to reverse finish order.
- setLabel(v, VISITED)
  - for all \( e \in G\).outIncidentEdges(\( v \))
    - if getLabel(\( e \)) = UNEXPLORED
      - w = opposite(\( v, e \))
      - if getLabel(\( w \)) = UNEXPLORED
        - setLabel(\( e \), DISCOVERY)
      - else
        - e is a forward or cross edge
  - L.

DFS Phase 2b

- Similar to Connected Components.

**Algorithm SCCDFS_Sweep(G, L)**
- Input: dag G, list L.
- Output: labeling of vertices in G by scc component number.
- sccNum = 1
  - for all \( u \in G\).vertices
    - setLabel(\( u \), UNEXPLORED)
  - for all \( e \in G\).edges
    - setLabel(\( e \), UNEXPLORED)
  - for all \( v \in L \) (traverse L in order)
    - if getLabel(\( v \)) = UNEXPLORED
      - SCCDFS(G, v, sccNum)
      - sccNum++

Correctness of SCC algorithm

- **Lemma 1:** In terms of vertices, SCC's of G are the same as the SCC's of \( G^R \).
- **Lemma 2:** For graph G, let F be a forest generated by DFS_Sweep on G. Let S be a tree of F. Then S contains one or more complete SCC's of G. (No partial SCC's).
- **Lemma 3a:** Let F be the forest generated by SCC phase 2b, and S be a spanning tree in F. Let x be the root of S, and \( v \) be a descendent of x. Then there is a path from \( v \) to x in \( G^R \).
- **Lemma 3b:** Let S be as in Lemma 3a. S combined with other edges in \( G^R \) form a strongly connected subgraph of \( G^R \).
Outline and Reading

- Breadth-first search (§6.3.3)
  - Algorithm
  - Example
  - Properties
  - Analysis
  - Applications
- DFS vs. BFS (§6.3.3)
  - Comparison of applications
  - Comparison of edge labels

Breadth-First Search

- Breadth-first search (BFS) is:
  - general graph traversal technique
  - visits all the vertices and edges
  - with a vertices and m edges takes O(n + m) time
  - like searching a binary tree level by level
- A BFS can:
  - Determine whether G is connected
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one
BFS Algorithm

The algorithm uses a queue to keep track of vertices.

**Algorithm BFS(G, s)**

- **Input**: graph G
- **Output**: labeling of the edges and the vertices of G
- **for all** $v \in G$.vertices()
  - setLabel(s, UNEXPLORED)
- **for all** $v \in G$.vertices()
  - if getLabel($v$) = UNEXPLORED
    - Q.enqueue($v$)

Properties

**Notation**
- $G_i$: connected component of s
- $L_i$: nodes at depth $i$ in BFS tree.

**Property 1**
- BFS$(G, s)$ visits all the vertices and edges of $G_i$.

**Property 2**
The discovery edges labeled by BFS$(G, s)$ form a spanning tree $T_i$ of $G_i$, $T_i$ called BFS tree.

**Property 3**
- For each vertex $v \in L_i$
  - The path of $T_i$ from $s$ to $v$ has $i$ edges.
  - Every path from $s$ to $v$ in $G_i$ has at least $i$ edges.

Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time.
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into $Q$.
- Inner loop of BFS runs in $O(\deg(v))$ time.
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure.
  - Recall that $\Sigma \deg(v) = 2m$.

Directed Graphs DFS 1.3

Directed Graphs DFS 1.3 37

Directed Graphs DFS 1.3 38

Directed Graphs DFS 1.3 39
Applications

- Can specialize the BFS traversal of a graph $G$ to solve the following problems in $O(n + m)$ time:
  - Compute the connected components of $G$
  - Compute a spanning forest of $G$
  - Find a simple cycle in $G$, or report that $G$ is a forest
  - Given two vertices of $G$, find a minimum length path in $G$ (if it exists)

DFS vs. BFS

<table>
<thead>
<tr>
<th>Applications</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanning forest, connected components, paths, cycles</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shortest paths</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Topological sort, Biconnected components, SCC</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Back edge $(v, w)$

- $w$ is an ancestor of $v$ in the tree of discovery edges

Cross edge $(v, w)$

- $w$ is in the same level as $v$ or in the next level in the tree of discovery edges