Lecture 14: Graph Representation, DFS, applications of DFS
Outline

- Graph Representation
  - Edge List
  - Adjacency List
  - Adjacency Matrix

- Graph Traversals
  - Depth First Search (DFS)
  - Applications of DFS
Graph Representation: Edge List Structure

- **Vertex objects** stored in unsorted sequence
  - space $O(n)$

- **Edge Objects** stored in unsorted sequence
  - space $O(m)$

- Each edge object has reference to origin and destination vertex object

- Total space: $O(n + m)$
Each vertex object has a **back pointer** to the node that references it in the linked list of vertices.

Each edge object has a **back pointer** to the node that references it in the linked list of edges.
**Edge List Structure**

- **Operation**
  - **vertices**
    - Time: $O(n)$
  - **edges**
    - Time: $O(m)$
  - **endVertices(e), opposite(v,e)**
    - Time: $O(1)$
  - **incidentEdges(v), areAdjacent(v,w)**
    - Time: $O(m)$
  - **replace(v,o), replace(e,o), insertVertex(o)**
    - Time: $O(1)$
  - **insertEdge(u,v), removeEdge(e)**
    - Time: $O(1)$
  - **removeVertex(v)**
    - Time: $O(m)$

- Easy to implement, but inefficient incidentEdges(v), areAdjacent(v,w), removeVertex(v), have to examine the entire edge sequence
Problem with edge list
- vertex does not know which edges are incident on it

Solution
- have each vertex reference an incident edge object
  - need a container, since each vertex can have many incident edges
  - this container is called **Adjacency List**, because usually implemented as a list
### Improving Edge List Structure

- **Problem with edge list**
  - vertex does not know which edges are incident on it

- **Solution**
  - have each vertex reference an incident edge object
    - need a container, since each vertex can have many incident edges
    - this container is called **Adjacency List**, because usually implemented as a list
To remove edge \((u,v)\)
- first remove edge from the edge list, \(O(1)\)
- then remove reference to edge \((u,v)\) from adjacency list of \(u\) and adjacency list of \(v\)
  - \(O((\text{deg}(v)+\text{deg}(w)))\) time, as opposed to \(O(1)\) with Edge List Structure

Solution: put back pointer from each edge \((u,v)\) in the edge sequence to where it is referenced in the adjacency list of \(u\) and \(v\)
- removeEdge is \(O(1)\) now
Graph Representation: Adjacency List

- Space requirement:
  - \( O(n) \) for the vertex sequence, \( O(m) \) for the edge sequence
  - For vertex \( v \), \( O(\text{deg}(v)) \) for the adjacency list of \( v \).
  - For all adjacency lists, \( O(\Sigma_v \text{deg}(v)) = 2m = O(m) \)
  - For the “back links” \( O(m) \)
  - Thus total space is \( O(n + m) \)
Adjacency List Structure

- Adjacency list is more efficient than edge list, but more complex to implement

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertices</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>edges</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>endVertices, opposite</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>incidentEdges(v)</td>
<td>$O(deg(v))$</td>
</tr>
<tr>
<td>areAdjacent(v, w)</td>
<td>$O(min(deg(v), deg(w)))$</td>
</tr>
<tr>
<td>insertVertex, insertEdge</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeEdge(v, w)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeVertex</td>
<td>$O((deg(v)))$</td>
</tr>
</tbody>
</table>
Graph Representation: Traditional Adjacency Matrix

- Each vertex is associated with an integer index from 0 to \( n - 1 \)
- Two-dimensional boolean adjacency array \( M \) of size \( n \) by \( n \)
- Set \( M[v,w] = true \) if there is an edge between \( v \) and \( w \)
- Set \( M[v,w] = false \) if there is no edge between \( v \) and \( w \)
- Space requirement is \( O(n^2) \)
- Adjacency matrix is space efficient only for dense graphs
  - In a dense graph, \( m \) is \( O(n^2) \), that is many edges
  - In a sparse graph, \( m \) is \( O(n) \), that is few edges
Graph Representation: Adjacency Matrix

- Edge list structure
- Integer index is associated with each vertex
- 2D adjacency array
  - Reference to the edge object for adjacent vertices
  - Null for nonadjacent vertices
- Space requirement: $O(n^2)$
### Adjacency Matrix Efficiency

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertices</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>edges</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>endVertices($e$), opposite($e,v$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>incidentEdges($v$)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>replace, areAdjacent($v,w$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertEdge($v,w$), removeEdge($e$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertVertex($v$), removeVertex($v$)</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

![Adjacency Matrix Diagram]
Asymptotic Performance

<table>
<thead>
<tr>
<th>n vertices, m edges simple graph</th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$n + m$</td>
<td>$n + m$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>incidentEdges($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
<td>$n$</td>
</tr>
<tr>
<td>areAdjacent ($v$, $w$)</td>
<td>$m$</td>
<td>$\min(\text{deg}(v), \text{deg}(w))$</td>
<td>1</td>
</tr>
<tr>
<td>insertVertex($o$)</td>
<td>1</td>
<td>1</td>
<td>$n^2$</td>
</tr>
<tr>
<td>insertEdge($v$, $w$, $o$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>removeEdge($e$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Summary of Graph Data Structures

- **Edge list structure**
  - not efficient, but good starting point for the other graph data structures

- **Adjacency list**
  - good efficiency for any graph
  - most complex to implement

- **2D adjacency array, good choice when**
  1. graph is dense
     - \( m \) is \( O(n^2) \), that is lot of edges
  2. And the set of vertices mostly stays fixed
     - almost no vertex insertion or deletion
     - when (1,2) hold, adjacency array is the best choice
Traversing a Graph

- A basic operation on a graph is traversal.
- How traverse a graph so that each vertex and edge is visited exactly once?
  - can go through vertex and edge sequences
  - but will not learn anything useful about graph
- Basic “Algorithm” : have a current vertex and visit adjacent vertices
  - start with currVertex = v and visit adjacent w

Which vertex visit next?
Traversing a Graph

- Start with current vertex = \( v \) and visit adjacent vertex \( w \)

- Go for **breadth**
  - Explore all vertices from the current one before switching current vertex

- Go for **depth**
  - Move current vertex to an adjacent one before finishing exploring current vertex (go deep in the graph)
Depth First Search

- Systematic graph traversal, go “deep”
- Can start DFS at any vertex $v$, DFS($v$)
- Mark all vertices as unvisited, not to get stuck in infinite cycle
- Maintain $\text{currVertex}$, set to $v$ at the start
- Move away from $\text{currVertex}$ to an adjacent unvisited vertex
  - mark $\text{currVertex}$ as visited
  - $\text{currVertex} =$ unvisited adjacent vertex
Now stuck in a “dead end”, no more unvisited adjacent vertices at CurVertex
Retrace to the previous vertex
- vertex that was CurVertex previously
Depth First Search

- Continue changing **CurVertex** to the next adjacent unvisited vertex
- When in “dead end”, retrace to the previous **CurVertex**

- Keep previous “**curVertex**” in a container to know which vertex to retrace to
  - container is a stack (FILO)
DFS: Non Recursive Algorithm

DFS(G,v)
mark all vertices UNVISITED
construct a new empty stack
curVertex = v

while curVertex is not null
  mark curVertex VISITED
  if curVertex has adjacent UNVISITED vertex w
    stack.push(curVertex)
    curVertex = w
  else // Dead end, retrace
    if stack is not empty
      curVertex = stack.pop()
    else curVertex = null
Example: DFS(A)

curVertex S = {} 

1. curVertex S = { A }
2. curVertex S = { B, A }
3. curVertex S = { C, B, A }
4. curVertex S = { C, B, A }

Diagram:

- A connected to B, D, E
- B connected to E
- D connected to E
- C connected to A, B, D, E
Example Continued: DFS(A)

1. Start with vertex A, stack S = {C, B, A}
2. Move to neighbor C, stack S = {B, A}
3. Move to neighbor D, stack S = {A, B, A}
4. Move to neighbor E, stack S = {B, A}
5. Move back to C, stack S = {B, A}
6. Move to neighbor D, stack S = {B, A}
7. Move to neighbor E, stack S = {B, A}

The algorithm follows the vertices in the order A, C, D, E, and then back to C, D, and E.
Example Continued: DFS(A)

- $S = \{ B, A \}$
- $S = \{ A \}$
- $S = \{ \}$

- CurVertex visited
- Done!
  - Stack is empty
  - CurVertex visited
DFS: Recursive Algorithm

Algorithm DFS(G)
Input: graph G

for all u ∈ G.vertices()
    setLabel(u, UNVISITED)
//get first vertex in vertex sequence
v = G.vertices().next()
DFS(G,v)

Algorithm DFS(G,v)
Input: graph G, start vertex v

setLabel(v, VISITED)
Print(v)
for all e ∈ G.incidentEdges(v)
    w ← opposite(v,e)
    if getLabel(w) = UNVISITED
        DFS(G,w)

After DFS, edges divided in 2 groups

- **discovery** edges
  - in solid red
  - lead to new unmarked vertices
  - “follow” these edges

- **back** edges
  - in dashed green
  - lead to already marked vertices
  - do not “follow” these edges

Useful to mark these edge types during
**DFS with Discovery and Back Edges**

**Algorithm DFS(G)**

**Input:** graph G

for all \( e \in G\text{.edges}() \)

setLabel\((e, \text{UNVISITED})\)

for all \( u \in G\text{.vertices}() \)

setLabel\((u, \text{UNVISITED})\)

//get first vertex in vertex sequence

\( v = G\text{.vertices}()\text{.next}() \)

DFS\((G,v)\)

**Algorithm DFS(G, v)**

**Input:** graph G, start vertex v

**Output:** labeling of discovery and back edges

setLabel\((v, \text{VISITED})\)

print\((v)\)

for all \( e \in G\text{.incidentEdges}(v) \)

\( w \leftarrow \text{opposite}(v,e) \)

if getLabel\((w) = \text{UNEXPLORED} \)

setLabel\((e, \text{DISCOVERY})\)

DFS\((G,w)\)

else if getLabel\((e) = \text{UNEXPLORED} \)

setLabel\((e, \text{BACK})\)
Example with Edge Labeling

- A: unexplored vertex
- •: visited vertex
- - - -: unexplored edge
- ➤ ➤ ➤ ➤: discovery edge
- ➣ ➣ ➣ ➣: back edge
Example (cont.)
DFS Continued

Algorithm DFS(G)
Input: graph G

for all \( e \in G.\text{edges()} \)
    setLabel\((e, \text{UNVISITED})\)

for all \( u \in G.\text{vertices()} \)
    setLabel\((u, \text{UNVISITED})\)

//get first vertex in vertex sequence
\( v = G.\text{vertices()}.\text{next()} \)
DFS(G,v)

Algorithm DFS(G, v)
Input: graph G, start vertex v
Output: labeling of discovery and back edges

setLabel\((v, \text{VISITED})\)
print\((v)\)

for all \( e \in G.\text{incidentEdges(v)} \)
    \( w \leftarrow \text{opposite}(v, e) \)
    if getLabel\((w) = \text{UNEXPLORED} \)
        setLabel\((e, \text{DISCOVERY})\)
        DFS(G,w)
    else if getLabel\((e) = \text{UNEXPLORED} \)
        setLabel\((e, \text{BACK})\)

- Will not visit the whole graph if disconnected graph
- Example: DFS(A)
DFS For Unconnected Graphs

Algorithm DFS(G)

Input: graph G

for all e ∈ G.edges()
    setLabel(e, UNVISITED)
for all u ∈ G.vertices()
    setLabel(u, UNVISITED)

//call DFS(G,v) on any unmarked v
for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        DFS(G, v)

Algorithm DFS(G, v)

Input: graph G, start vertex v

Output: labeling of discovery and back edges

setLabel(v, VISITED)
print(v)

for all e ∈ G.incidentEdges(v)
    w ← opposite(v, e)
    if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
    else if getLabel(e) = UNEXPLORED
        setLabel(e, BACK)
Properties of DFS

Property 1

\textit{DFS}(G, v) \textit{visits all the vertices and edges in the connected component of } v

Property 2

The discovery edges labeled by \textit{DFS}(G, v) form a spanning tree of the connected component of v
Analysis of DFS

- Assume adjacency list structure
- Assume setting/getting a vertex/edge label takes $O(1)$ time

**DFS($G$)**
- count time spend in $DFS(G,v)$ separately
- go over all graph vertices twice and over all edges ones
- takes $O(m+n)$ time

**DFS($G,v$)**
- called 1 time for each vertex
- one call to $DFS(G,v)$ takes $1+\deg(v)$
- all calls to $DFS(G,v)$ take time $\sum_v [1+\deg(v)] = \sum_v 1+\sum_v \deg(v) = n+2m$
- DFS is $O(n + m)$ time

---

### Algorithm DFS($G$)

```plaintext
for all $e \in G$'s edges()
    setLabel($e$, UNVISITED)
for all $u \in G$'s vertices()
    setLabel($u$, UNEXPLORED)
for all $v \in G$'s vertices()
    if getLabel($v$) = UNEXPLORED
        DFS($G,v$)
```

---

### Algorithm DFS($G,v$)

```plaintext
setLabel($v$, VISITED)
for all $e \in G$'s incidentEdges($v$)
    $w \leftarrow$ opposite($v,e$)
    if getLabel($w$) = UNEXPLORED
        setLabel($e$, DISCOVERY)
        DFS($G,w$)
    else if getLabel($e$) = UNEXPLORED
        setLabel($e$, BACK)
```
vertex $v$ is **active** if $v$ is marked VISITED but call to $DFS(G,v)$ did not finish yet

Active vertices form a single path
- in non-recursive DFS, active vertices are in the **stack**

If we keep track of this path, we can find a path between any two vertices, provided there is path between them
Path Finding

- Specialize the DFS to find a path between vertices \( u \) and \( z \)
- Mark all vertices unexplored before call to `pathDFS`
- Call `pathDFS(G, u, z)`
- Stack \( S \)
  - is an instance variable initialized to empty before `pathDFS` is called
  - holds active vertices
  - keeps track of the path between the start vertex \( u \) and the current vertex \( v \)
- As soon as destination vertex \( z \) is found, return the path as the contents of the stack

Algorithm `pathDFS(G, v, z)`

```plaintext
setLabel(v, VISITED)
S.push(v)
if v = z then  // found path, return it
    return S.elements()
for all e ∈ G.incidentEdges(v)
    w ← opposite(v, e)
    if getLabel(w) = UNEXPLORED
        result = pathDFS(G, w, z)
        if result != null then
            return result
s.pop()
return null  //did not find path yet
```
Algorithm pathDFS(G,v,z)

setLabel(v,VISITED)

S.push(v)

if v = z then // found path, return it
    return S.elements()

for all e ∈ G.incidentEdges(v)
    w ← opposite(v,e)
    if getLabel(w) = UNEXPLORED
        result = pathDFS(G,w,z)
        if result != null then
            return result

S.pop()

return null //did not find path yet
Example: Path between A and E

Algorithm pathDFS(G,v,z)

setLabel(v,VISITED)
S.push(v)
if v = z then // found path, return it
    return S.elements()
for all e ∈ G.incidentEdges(v)
    w ← opposite(v,e)
    if getLabel(w) = UNEXPLORED
        result = pathDFS(G,w,z)
        if result != null then
            return result
S.pop()
return null // did not find path yet
Algorithm pathDFS(G,v,z)
setLabel(v, VISITED)
S.push(v)
if v = z then // found path, return it
    return S.elements()
for all e ∈ G.incidentEdges(v)
    w ← opposite(v, e)
    if getLabel(w) = UNEXPLORED
        result = pathDFS(G, w, z)
        if result != null then
            return result
    S.pop()
return null // did not find path yet
Cycle Finding

- Specialize DFS to find simple cycle
- Mark all edges and vertices UNEXPLORED before call to `cycleDFS(G,v)`
- Stack $S$ stores path between start and current vertex $v$
  - stack holds active vertices
  - initialized to empty before `cycleDFS` is called
- When a back edge $(v, w)$ is encountered, return cycle as the portion of the stack from the top to vertex $w$
- `cycleDFS(G,v)` finds cycle in the connected component of $v$

```
Algorithm cycleDFS(G,v)
  setLabel(v,VISITED)
  S.push(v)
  for all $e \in G$.incidentEdges(v)
    \[ w \leftarrow \text{opposite}(v,e) \]
    \[ \text{if getLabel}(w) = \text{UNEXPLORED} \]
    \[ \text{setLabel}(e,\text{DISCOVERY}) \]
    \[ \text{result} = \text{cycleDFS}(G,w) \]
    \[ \text{if result} \neq \text{null} \text{ then} \]
    \[ \text{return result} \]
  \[ \text{else if getLabel}(e)\text{=}\text{UNEXPLORED} \]
  \[ \text{// found cycle} \]
  \[ \text{setLabel}(e,\text{BACK}) \]
  \[ T \leftarrow \text{new empty stack} \]
  \[ T\text{.push}(w) \]
  \[ \text{repeat} \]
  \[ o \leftarrow S\text{.pop()} \]
  \[ T\text{.push}(o) \]
  \[ \text{until} o = w \]
  \[ \text{return T.elements()} \]
  \[ S\text{.pop()} \]
  \[ \text{return null} \]
```
Cycle Finding

$S = E$

$S = A, C, E$

$S = B, A, C, E$

$\text{"w"} = C$
Algorithm cycleDFS(G,v)

setLabel(v,VISITED)
S.push(v)
for all e ∈ G.incidentEdges(v)
  w ← opposite(v,e)
  if getLabel(w) = UNEXPLORED
     setLabel(e,DISCOVERY)
     result = cycleDFS(G,w)
  if result != null then
     return result
else if getLabel(e)=UNEXPLORED
  // found cycle
  setLabel(e,BACK)
  T ← new empty stack
  T.push(w)
  repeat
     o ← S.pop()
     T.push(o)
  until o = w
  return T.elements()
S.pop()
return null
## Depth-First Search Summary

- Depth-first search (DFS) is a general technique for traversing a graph.
  - A DFS traversal of graph G visits all the vertices and edges of G.
  - Determines whether G is connected.
  - Computes the connected components of G.
  - Computes a spanning forest of G.

- DFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time with adjacency list implementation.

- DFS can be further extended to solve other graph problems:
  - Find and report a path between two given vertices.
  - Find a cycle in the graph.