Graph Algorithms

Problems on Graphs

What kind of problems do we want to solve on/via graphs?

- is the graph fully-connected?
- can we remove an edge and keep it fully-connected?
- is one vertex reachable starting from some other vertex?
- what is the cheapest cost path from \( v \) to \( w \)?
- which vertices are reachable from \( v \) (transitive closure)
- is there a cycle that passes through all vertices? (circuit)
- is there a tree that links all vertices? (spanning tree)
- what is the minimum spanning tree?
- …
- can a graph be drawn in a plane with no crossing edges? (planar graphs)
- are two graphs "equivalent"? (isomorphism)

Graph Algorithms

In this course we examine algorithms for

- connectivity (simple graphs)
- path finding (simple/directed graphs)
- minimum spanning trees (weighted graphs)
- shortest path (weighted graphs)

and look at generic methods for traversing graphs.

We begin with one of the simplest graph traversals …

Graph Traversal

Finding a Path

Questions on paths:

- is there a path between two given vertices \( (src, dest) \)?
- what is the sequence of vertices from \( src \) to \( dest \)?

Approach to solving problem:

- examine vertices adjacent to \( src \)
- if any of them is \( dest \), then done
- otherwise try vertices two edges from \( v \)

Two strategies for graph traversal/search: depth-first, breadth-first

- DFS follows one path to completion before considering others
- BFS "fans-out" from the starting vertex (*spreading* subgraph)

Exercise #1: Traversal-induced Spanning Trees

A spanning tree of a graph

- includes all vertices
- uses a subset of edges, without cycles

Show the DFS and BFS spanning trees for the graph below, starting with 0:
Consider neighbours in ascending order

Answer:

Depth-first Search

Depth-first search can be described recursively as

```
def depthFirst(G, v):
    1. mark v as visited
    2. for each (v, w) ∈ edges(G) do
       if w has not been visited then
           depthFirst(w)
```

The recursion induces backtracking

Exercise #2: Depth-first Traversal (i)

Trace the execution of dfsPathCheck(G, 0, 5) on:

```
Consider neighbours in ascending order

Answer:

Depth-first Search

Recursive DFS path checking

```
hasPath(G, src, dest):
    Input graph G, vertices src, dest
    Output true if there is a path from src to dest in G,
           false otherwise
    return dfsPathCheck(G, src, dest)

def dfsPathCheck(G, v, dest):
    mark v as visited
    for all (v, w) ∈ edges(G) do
        if w=dest then // found edge to dest
            return true
        else if w has not been visited then
            if dfsPathCheck(G, w, dest) then
                return true // found path via w to dest
            end if
        end if
    end for
    return false // no path from v to dest
```

Cost analysis:

- each vertex visited at most once ⇒ cost = O(V)
- visit all edges incident on visited vertices ⇒ cost = O(E)
  - assuming an adjacency list representation

Time complexity of DFS: O(V+E) (adjacency list representation)

... Depth-first Search

Knowing whether a path exists can be useful

Knowing what the path is even more useful
⇒ record the previously visited node as we search through the graph (so that we can then trace path through graph)

Make use of global variable:

```
visited[] ... array to store previously visited node, for each node being visited
```
... Depth-first Search
visited[] // store previously visited node, for each vertex 0..nV-1

findPath(G, src, dest):
| Input graph G, vertices src, dest |
| for all vertices v ∈ G do |
| visited[v]=-1 |
| end for |
| visited[src]=src // starting node of the path |
| if dfsPathCheck(G, src, dest) then // show path in dest..src order |
| v=dest |
| while v≠src do |
| print v="-" |
| v=visited[v] |
| end while |
| print src |
| end if |

dfsPathCheck(G, v, dest):
| for all (v,w)∈ edges(G) do |
| if visited[w]=-1 then |
| visited[w]=v |
| if w=dest then // found edge from v to dest |
| return true |
| else if dfsPathCheck(G, w, dest) then |
| return true // found path via w to dest |
| end if |
| end for |
| return false // no path from v to dest |

Exercise #3: Depth-first Traversal (ii)

Show the DFS order in which we visit vertices in this graph when searching for a path from 0 to 6:

Consider neighbours in ascending order

Path: 6-5-1-0

... Depth-first Search
visited[] // array of visiting orders, indexed by vertex 0..nV-1

findPathDFS(G, src, dest):
| Input graph G, vertices src, dest |
| for all vertices v ∈ G do |
| visited[v]=-1 |
| end for |
| found=false |
| visited[src]=src |
| push src onto new stack s |
| while not found ∧ s is not empty do |
| pop v from s |
| for each (v,w)∈ edges(G) do |
| if visited[w]=-1 then |
| visited[w]=v |
| if w=dest then // found edge from v to dest |
| found=true |
| else |
| push w onto s |
| end if |
| end for |
| end while |
| if found then |
| display path in dest..src order |
| end if |

Uses standard stack operations (push, pop, check if empty)

Time complexity is the same: \( O(V+E) \) (each vertex added to stack once, each element in vertex's adjacency list visited once)

Exercise #4: Depth-first Traversal (iii)

Show how the stack evolves when executing findPathDFS(g, 0, 5) on:
Push neighbours in descending order … so they get popped in ascending order

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

(empty) → 0 → 5 → 5 → 5 → 5 → (empty)

---

**Breadth-first Search**

Basic approach to breadth-first search (BFS):

- visit and mark current vertex
- visit all neighbours of current vertex
- then consider neighbours of neighbours

Notes:

- tricky to describe recursively
- a minor variation on non-recursive DFS search works
  ⇒ switch the stack for a queue

---

**Exercise #5: Breadth-first Traversal**

Show the BFS order in which we visit vertices in this graph when searching for a path from 0 to 6:

Consider neighbours in ascending order

Path: 6-5-0

---

**BFS algorithm (records visiting order):**

```plaintext
visited[] // array of visiting orders, indexed by vertex 0..nV-1

findPathBFS(G,src,dest):
    | Input graph G, vertices src,dest
    | for all vertices v∈G do
    |     visited[v]=-1
    | end for
    | found=false
    | visited[src]=src
    | enqueue src into new queue q
    | while ¬found ∧ q is not empty do
    |     dequeue v from q
    |     for each neighbour w of v do
    |         if visited[w]=-1 then
    |             visited[w]=v
    |             if w=dest then
    |                 found=true
    |             else
    |                 enqueue w into q
    |             end if
    |         end if
    |     end for
    | end while
    | if found then
    |     display path in dest..src order
    | end if
```

Uses standard queue operations (enqueue, dequeue, check if empty)

---

**Time complexity of BFS:**

O(V+E) (same as DFS)

BFS finds a "shortest" path

- based on minimum # edges between src and dest.
- stops with first-found path, if there are multiple ones

In many applications, edges are weighted and we want path
Other DFS Examples

Other problems to solve via DFS graph search

- checking for the existence of a cycle
- determining which connected component each vertex is in

Exercise #6: Buggy Cycle Check

A graph has a cycle if

- it has a path of length > 1
- with start vertex src = end vertex dest
- and without using any edge more than once

We are not required to give the path, just indicate its presence.

The following DFS cycle check has two bugs. Find them.

```python
hasCycle(G):

Input graph G
Output true if G has a cycle, false otherwise

choose any vertex v∈G
return dfsCycleCheck(G,v)

dfsCycleCheck(G,v):
mark v as visited
for each (v,w)∈edges(G) do
  if w has been visited then // found cycle
    return true
  else if dfsCycleCheck(G,w) then
    return true
end for
return false // no cycle at v
```

Computing Connected Components

Problems:

- how many connected subgraphs are there?
- are two vertices in the same connected subgraph?

Both of the above can be solved if we can

- build an array, one element for each vertex V
- indicating which connected component V is in

- componentOf[] … array [0..nV-1] of component IDs

Exercise #7: Connected components

```python
components(G):

Input graph G
Output connected components

for all vertices v∈G do
  componentOf[v]=-1
end for

compID=0
for all vertices v∈G do
  if componentOf[v]=-1 then
    dfsComponents(G,v,compID)
    compID=compID+1
  end if
end for

dfsComponents(G,v,id):
mark v as visited
for all vertices w adjacent to v do
  if componentOf[w]=-1 then
    dfsComponents(G,w,id)
  end if
end for
```

Exercise #7: Connected components
Trace the execution of the algorithm
1. on the graph shown below
2. on the same graph but with the dotted edges added

Consider neighbours in ascending order

1.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

2.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

---

... Computing Connected Components

With this structure, the above tasks become trivial:

```c
int nConnected(Graph g) {
    return g->nC;
}
```

```c
bool inSameComponent(Graph g, Vertex v, Vertex w) {
    return (g->cc[v] == g->cc[w]);
}
```

Consider maintenance of such a graph representation:

- Initially, `nC = nV` (because no edges)
- Adding an edge may reduce `nC`
- Removing an edge may increase `nC`
- `cc[]` can simplify path checking
  (ensure `v, w` are in same component before starting search)

Additional maintenance cost amortised by reduced cost for `nConnected()` and `inSameComponent()`

---

Hamiltonian and Euler Paths

Hamiltonian Path and Circuit

Hamiltonian path problem:

- find a simple path connecting two vertices `v, w` in graph `G`
- such that the path includes each `vertex` exactly once

If `v = w`, then we have a Hamiltonian circuit

Simple to state, but difficult to solve (NP-complete)

Many real-world applications require you to visit all vertices of a graph:

- Travelling salesman
Bus routes

... Problem named after Irish mathematician, physicist and astronomer Sir William Rowan Hamilton (1805 - 1865)

... Hamiltonian Path and Circuit

Graph and two possible Hamiltonian paths:

... Hamiltonian Path and Circuit

Approach:
- generate all possible simple paths (using e.g. DFS)
- keep a counter of vertices visited in current path
- stop when find a path containing V vertices

Can be expressed via a recursive DFS algorithm
- similar to simple path finding approach, except
  - keeps track of path length; succeeds if length = v
  - resets "visited" marker after unsuccessful path

Exercise #8: Hamiltonian Path

Trace the execution of the algorithm when searching for a Hamiltonian path from 1 to 6:

Consider neighbours in ascending order

1-0-2-3-4-5-6  
d ≠ 0
1-0-2-3-4-5-7-8-9  
no unvisited neighbour
1-0-2-3-4-5-7-9-8  
no unvisited neighbour
1-0-2-3-4-7-5-6  
d ≠ 0
1-0-2-3-4-7-8-9  
no unvisited neighbour
1-0-2-3-4-7-9-8  
no unvisited neighbour
1-0-2-3-4-8-7-5-6  
d ≠ 0
1-0-2-3-4-8-7-9  
no unvisited neighbour
1-0-2-3-4-8-9-7-5-6  ✓

Repeat on your own with src=0 and dest=6

... Hamiltonian Path and Circuit

Algorithm for finding Hamiltonian path:

visited[] // array [0..nV-1] to keep track of visited vertices

hasHamiltonPath(G,src,dest):
|
| for all vertices v∈G do
|     visited[v]=false
| end for
| return hamiltonR(G,src,dest,#vertices(G)-1)

hamiltonR(G,v,dest,d):
|
| Input G    graph
|     v current vertex considered
| dest destination vertex
|     d distance "remaining" until path found
|
| if v=dest then

... Hamiltonian Path and Circuit

Analysis: worst case requires (V-1)! paths to be examined

Consider a graph with isolated vertex and the rest fully-connected
Checking `hasHamiltonianPath(g, 0, x)` for any `x` requires us to consider every possible path:

- e.g. 1-2-3-4, 1-2-4-3, 1-3-2-4, 1-3-4-2, 1-4-2-3, ...
- starting from any `x`, there are `3!` paths ⇒ `4!` total paths
- there is no path of length 5 in these `(V-1)!` possibilities

There is no known simpler algorithm for this task ⇒ NP-hard.

Note, however, that the above case could be solved in constant time if we had a fast check for 0 and `x` being in the same connected component.

---

### Euler Path and Circuit

**Euler path** problem:

- find a path connecting two vertices `v, w` in graph `G`
- such that the path includes each edge exactly once
  (note: the path does not have to be simple ⇒ can visit vertices more than once)

If `v = w`, the we have an **Euler circuit**

---

Many real-world applications require you to visit all edges of a graph:

- Postman
- Garbage pickup
- ...

---

### Exercise #9: Euler Path

Is there a way to cross all the bridges of Konigsberg exactly once on a walk through the town?

- treat land as nodes; bridges as edges

---

No

---

### Euler Path and Circuit

Problem named after Swiss mathematician, physicist, astronomer, logician and engineer Leonhard Euler (1707 - 1783)

Based on a circuitous route via bridges in Konigsberg

---

### Exercise #10: Eulerian Graphs

Graphs with an Euler path are often called Eulerian Graphs

Which of these two graphs have an Euler path? an Euler circuit?
No Euler circuit

Only the second graph has an Euler path, e.g. 2-0-1-3-2-4-5

... Euler Path and Circuit

Assume the existence of \( \text{degree}(g,v) \) (degree of a vertex, cf. problem set week 6)

Algorithm to check whether a graph has an Euler path:

```plaintext
hasEulerPath(G, src, dest):

Input: graph G, vertices src, dest
Output: true if G has Euler path from src to dest
false otherwise

if src ≠ dest then
    if degree(G, src) is even \( \lor \) degree(G, dest) is even then
        return false
    else
        return true
end if
for all vertices \( v \in G \) do
    if \( v \neq src \land v \neq dest \land \) degree(G, v) is odd then
        return false
    end if
end for
```

If degree requires iteration over vertices

- cost to compute degree of a single vertex is \( O(V) \)
- overall cost is \( O(V^2) \)

⇒ problem tractable, even for large graphs (unlike Hamiltonian path problem)

For the keen, a linear-time (in the number of edges, \( E \)) algorithm to compute an Euler path is described in [Sedgewick] Ch.17.7.

---

**Directed Graphs**

**Directed Graphs (Digraphs)**

In our previous discussion of graphs:

- an edge indicates a relationship between two vertices
- an edge indicates nothing more than a relationship

In many real-world applications of graphs:

- edges are directional \( (v \rightarrow w \neq w \rightarrow v) \)
- edges have a weight (cost to go from \( v \rightarrow w \))

---

### Directed Graphs (Digraphs)

Example digraph and adjacency matrix representation:

```
Undirectional ⇒ symmetric matrix
Directional ⇒ non-symmetric matrix
```

Maximum #edges in a digraph with \( V \) vertices: \( V^2 \)

---

**Directed Graphs (Digraphs)**

Terminology for digraphs …

\( \text{Directed path: } \) sequence of \( n \geq 2 \) vertices \( v_f \rightarrow v_2 \rightarrow \ldots \rightarrow v_n \)
where \((v_i,v_{i+1}) \in \text{edges}(G)\) for all \(v_i,v_{i+1}\) in sequence
if \(v_1 = v_n\), we have a directed cycle

Degree of vertex: \(\text{deg}(v) = \text{number of edges of the form } (v,\_ ) \in \text{edges}(G)\)
- Indegree of vertex: \(\text{deg}^{-1}(v) = \text{number of edges of the form } (_,v) \in \text{edges}(G)\)

Reachability: \(w\) is reachable from \(v\) if \(\exists\) directed path \(v,\ldots,w\)

Strong connectivity: every vertex is reachable from every other vertex

Directed acyclic graph (DAG): graph containing no directed cycles

### Digraph Applications

Potential application areas:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Vertex</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence</td>
</tr>
<tr>
<td>chess</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>science</td>
<td>journal article</td>
<td>citation</td>
</tr>
<tr>
<td>dynamic data</td>
<td>malloc'd object</td>
<td>pointer</td>
</tr>
<tr>
<td>programs</td>
<td>function</td>
<td>function call</td>
</tr>
<tr>
<td>make</td>
<td>file</td>
<td>dependency</td>
</tr>
</tbody>
</table>

### Digraph Representation

Similar set of choices as for undirectional graphs:

- array of edges (directed)
- vertex-indexed adjacency matrix (non-symmetric)
- vertex-indexed adjacency lists

\(V\) vertices identified by \(0 .. V-1\)

---

### Costs of representations:

<table>
<thead>
<tr>
<th></th>
<th>array of edges</th>
<th>adjacency matrix</th>
<th>adjacency list</th>
</tr>
</thead>
<tbody>
<tr>
<td>space usage</td>
<td>(E)</td>
<td>(V^2)</td>
<td>(V+E)</td>
</tr>
<tr>
<td>insert edge</td>
<td>(E)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>exists edge ((v,w))?</td>
<td>(E)</td>
<td>(1)</td>
<td>(\text{deg}(v))</td>
</tr>
<tr>
<td>get edges leaving (v)</td>
<td>(E)</td>
<td>(V)</td>
<td>(\text{deg}(v))</td>
</tr>
</tbody>
</table>

Overall, adjacency list representation is best
- real graphs tend to be sparse (large number of vertices, small average degree \(\text{deg}(v)\))
- algorithms frequently iterate over edges from \(v\)

### Reachability

**Transitive Closure**

Given a digraph \(G\) it is potentially useful to know
- is vertex \(t\) reachable from vertex \(s\)?

Example applications:
- can I complete a schedule from the current state?
- is a malloc’d object being referenced by any pointer?

How to compute transitive closure?
One possibility:

- implement it via `hasPath(G, s, t)` (itself implemented by DFS or BFS algorithm)
- feasible if `reachable(G, s, t)` is infrequent operation

What if we have an algorithm that frequently needs to check reachability?

Would be very convenient/efficient to have:

```plaintext
reachable(G, s, t):
    return G.tc[s][t] // transitive closure matrix
```

Of course, if V is very large, then this is not feasible.

---

**Exercise #11: Transitive Closure Matrix**

Which reachable `s .. t` exist in the following graph?

![Graph](image)

---

**Transitive closure of example graph:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

---

**Exercise #12: Transitive Closure**

Trace Warshall’s algorithm on the following graph:

1st iteration $i=0$:

- `tc[s][t]=1` if there is a path from `s` to `t` of length 2 ($s \rightarrow i \rightarrow t$)

---

**Transitive Closure**

If we implement the above as:

```plaintext
make tc[] a copy of edges[]
for all i vertices(G) do
    for all s vertices(G) do
        for all t vertices(G) do
            if tc[s][i]=1 ∧ tc[i][t]=1 then
                tc[s][t]=1
            end if
        end for
    end for
end for
```

then we get an algorithm to convert `edges` into a `tc`

This is known as *Warshall’s algorithm*

---

**Exercise #12: Transitive Closure**

Trace Warshall’s algorithm on the following graph:

1st iteration $i=0$:
### Example: Web Crawling

**Goal:** visit every page on the web

**Solution:** breadth-first search with "implicit" graph

```plaintext
webCrawl(startingURL):
  mark startingURL as alreadySeen
  enqueue(Q, startingURL)
  while isEmpty(Q) do
    nextPage = dequeue(Q)
    visit nextPage
    for each hyperlink on nextPage do
      if hyperlink not alreadySeen then
        mark hyperlink as alreadySeen
        enqueue(Q, hyperlink)
    end if
  end while
```

Visit scans page and collects e.g. keywords and links

### PageRank

**Goal:** determine which Web pages are "important"

**Approach:** ignore page contents; focus on hyperlinks

- treat Web as graph: page = vertex, hyperlink = di-edge
- pages with many incoming hyperlinks are important
- need to computing "incoming degree" for vertices

**Problem:** the Web is a very large graph

- approx. $10^{14}$ pages, $10^{15}$ hyperlinks

Assume for the moment that we could build a graph ...

Most frequent operation in algorithm "Does edge (v,w) exist?"

### Digraph Traversal

Same algorithms as for undirected graphs:

**depthFirst(v):**

1. mark v as visited
2. for each (v,w) ∈ edges(G) do
   if w has not been visited then
     depthFirst(w)

**breadth-first(v):**

1. enqueue v
2. while queue not empty do
   dequeue v
   if v not already visited then
     mark v as visited
     enqueue each vertex w adjacent to v

---

**Cost analysis:**

- storage: additional $V^2$ items (each item may be 1 bit)
- computation of transitive closure: $V^3$
- computation of reachable(): $O(I)$ after having generated $tc[][]$

Amortization: would need many calls to reachable() to justify other costs

Alternative: use DFS in each call to reachable()

**Cost analysis:**

- storage: cost of queue and set during reachable
- computation of reachable(): cost of DFS = $O(V^2)$ (for adjacency matrix)
PageRank(myPage):

| rank=0
| for each page in the Web do
|   if linkExists(page,myPage) then
|     rank=rank+1
| end if
| end for

Note: requires inbound link check (not outbound as assumed above for cost of representation)

**PageRank**

\( V = \# \text{ pages in Web}, \ E = \# \text{ hyperlinks in Web} \)

Costs for computing PageRank for each representation:

<table>
<thead>
<tr>
<th>Representation</th>
<th>linkExists(v,w)</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacency matrix</td>
<td>( \text{edge}[v][w] )</td>
<td>( l )</td>
</tr>
<tr>
<td>Adjacency lists</td>
<td>( \text{inLL}(\text{list}[v],w) )</td>
<td>( # E/V )</td>
</tr>
</tbody>
</table>

Not feasible ...

- adjacency matrix ... \( V \approx 10^{14} \Rightarrow \) matrix has \( 10^{28} \) cells
- adjacency list ... \( V \) lists, each with \( \approx 10 \) hyperlinks \( \Rightarrow 10^{15} \) list nodes

So how to really do it?

**Exercise #13: Implementing Facebook**

Facebook could be considered as a giant "social graph"

- what are the vertices?
- what are the edges?
- are edges directional?

What kind of algorithm would …

- help us find people that you might like to "befriend"?

**Summary**

- Graph traversal
  - depth-first search (DFS)
  - breadth-first search (BFS)
- applications: path finding, connected components
- Hamiltonian paths/circuits, Euler paths/circuits
- Digraphs: representations, applications, reachability

Suggested reading (Sedgewick):

- Hamiltonian/Euler paths … Ch.17.7
- Graph search … Ch.18.1-18.3,18.7
- Digraphs … Ch.19.1-19.3