Lecture 9

Graphs, BFS and DFS
Announcements!

• HW4 due Friday

• **MIDTERM** in class, Monday 10/30.
  • That’s 1 week from today. **Please show up.**
  • During class, 1:30-2:50
    • If your last name is A-M: 370-370 (here)
    • If your last name is N-V: 160-124
    • If your last name is W-Z: 160-323
  • You may bring one double-sided letter-size page of notes, that *you have prepared yourself*.

• Any material through Hashing (Lecture 8) is fair game.
• Practice exams on the website
• Review Session tomorrow in Section
Outline

• Part 0: Graphs and terminology

• Part 1: Depth-first search
  • Application: topological sorting
  • Application: in-order traversal of BSTs

• Part 2: Breadth-first search
  • Application: shortest paths
  • Application (if time): is a graph bipartite?
Part 0: Graphs
Graphs

Graph of the internet (circa 1999...it’s a lot bigger now...)
Graphs

Citation graph of literary theory academic papers
Graphs

Theoretical Computer Science academic communities

Example from DBLP: Communities within the co-authors of Christos H. Papadimitriou
Graphs

Game of Thrones Character Interaction Network
Graphs

jetblue flights
Graphs

Complexity Zoo containment graph
Graphs
Graphs

The bilateral flows between 196 countries are estimated from sequential stock tables (see overleaf for details). They are comparable across countries and capture the number of people who changed their country of residence between mid-2005 and mid-2010.

The circular plot shows the estimates of directional flows between the 50 countries that send and/or receive at least 0.5% of the world’s migrants in 2005–10. Tick marks indicate gross migration (in + out) in 100,000s.

Immigration flows
Graphs

World trade in fresh potatoes, flows over 0.1 m US$ average 2005-2009
Graphs

Soybeans

Water
Graphs

Graphical models
What eats what in the Atlantic ocean?
Graphs

Neural connections in the brain
Graphs

• There are a lot of graphs.

• We want to answer questions about them.
  • Efficient routing?
  • Community detection/clustering?
  • From pre-lecture exercise:
    • Computing Bacon numbers
    • Signing up for classes without violating pre-req constraints
    • How to distribute fish in tanks so that none of them will fight.

• This is what we’ll do for the next several lectures.
Undirected Graphs

- Has vertices and edges
  - \( V \) is the set of vertices
  - \( E \) is the set of edges
  - Formally, a graph is \( G = (V,E) \)

- Example
  - \( V = \{1,2,3,4\} \)
  - \( E = \{\{1,3\}, \{2,4\}, \{3,4\}, \{2,3\}\} \)

- The degree of vertex 4 is 2.
  - There are 2 edges coming out
  - Vertex 4’s neighbors are 2 and 3
Directed Graphs

• Has **vertices** and **edges**
  • V is the set of vertices
  • E is the set of **DIRECTED** edges
  • Formally, a graph is G = (V,E)

• Example
  • V = \{1,2,3,4\}
  • E = \{ (1,3), (2,4), (3,4), (4,3), (3,2) \}

• The **in-degree** of vertex 4 is 2.
• The **out-degree** of vertex 4 is 1.
• Vertex 4’s **incoming neighbors** are 2,3
• Vertex 4’s **outgoing neighbor** is 3.
How do we represent graphs?

• Option 1: adjacency matrix

\[
\begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
0 & 1 & 1 & 0
\end{bmatrix}
\]
How do we represent graphs?

• Option 1: adjacency matrix

\[
\begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 \\
\end{bmatrix}
\]
How do we represent graphs?

• Option 1: adjacency matrix

\[
\begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
\end{bmatrix}
\]
How do we represent graphs?

- Option 2: linked lists.

1 2 3 4

4’s neighbors are 2 and 3

How would you modify this for directed graphs?
In either case

- Vertices can store other information
  - Attributes (name, IP address, ...)
  - helper info for algorithms that we will perform on the graph

- Want to be able to do the following operations:
  - **Edge Membership**: Is edge $e$ in $E$?
  - **Neighbor Query**: What are the neighbors of vertex $v$?
Trade-offs

Say there are $n$ vertices and $m$ edges.

<table>
<thead>
<tr>
<th></th>
<th>Edge membership</th>
<th>Neighbor query</th>
<th>Space requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e = {v,w}$ in $E$?</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td>$O(deg(v))$ or $O(deg(w))$</td>
<td>$O(deg(v))$</td>
<td>$O(n + m)$</td>
</tr>
</tbody>
</table>

See Lecture 9 IPython notebook for the actual data structure that we will be using!
Part 1: Depth-first search
How do we explore a graph?

At each node, you can get a list of neighbors, and choose to go there if you want.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet
Been there, haven’t explored all the paths out.
Been there, have explored all the paths out.

start
Depth First Search

Exploring a labyrinth with chalk and a piece of string

Not been there yet

Been there, haven’t explored all the paths out.

Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

start

Not been there yet
Been there, haven’t explored all the paths out.
Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet
Been there, haven’t explored all the paths out.
Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet

Been there, haven’t explored all the paths out.

Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.

start
Depth First Search
Exploring a labyrinth with chalk and a piece of string

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.

```
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet
Been there, haven’t explored all the paths out.
Been there, have explored all the paths out.
```
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet

Been there, haven’t explored all the paths out.

Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet

Been there, haven’t explored all the paths out.

Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string
Depth First Search

Exploring a labyrinth with chalk and a piece of string

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string.

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.

start
Depth First Search
Exploring a labyrinth with chalk and a piece of string

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.

start
Depth First Search
Exploring a labyrinth with chalk and a piece of string

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.

start
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Labyrinth: EXPLORED!
Depth First Search
Exploring a labyrinth with pseudocode

• Each vertex keeps track of whether it is:
  • Unvisited
  • In progress
  • All done

• Each vertex will also keep track of:
  • The time we first enter it.
  • The time we finish with it and mark it all done.

You might have seen other ways to implement DFS than what we are about to go through. This way has more bookkeeping, but more intuition – also, the bookkeeping will be useful later!
Depth First Search

currentTime = 0

• **DFS**(w, currentTime):
  • w.startTime = currentTime
  • currentTime += 1
  • Mark w as **in progress**.
  • for v in w.neighbors:
    • if v is **unvisited**:
      • currentTime
        = DFS(v, currentTime)
      • currentTime += 1
    • w.finishTime = currentTime
    • Mark w as **all done**
  • return currentTime
Depth First Search

DFS(w, currentTime):
• w.startTime = currentTime
• currentTime ++
• Mark w as in progress.
• for v in w.neighbors:
  • if v is unvisited:
    • currentTime = DFS(v, currentTime)
  • currentTime ++
• w.finishTime = currentTime
• Mark w as all done
• return currentTime

Start: 0
currentTime = 1
Depth First Search

currentTime = 1

• DFS(w, currentTime):
  • w.startTime = currentTime
  • currentTime ++
  • Mark w as in progress.
  • for v in w.neighbors:
    • if v is unvisited:
      • currentTime = DFS(v, currentTime)
      • currentTime ++
    • w.finishTime = currentTime
    • Mark w as all done
  • return currentTime

• DFS(A, 0):
  • w.startTime = 1
  • currentTime ++
  • Mark w as in progress.
  • for v in w.neighbors:
    • if v is unvisited:
      • currentTime = DFS(v, currentTime)
      • currentTime ++
    • w.finishTime = currentTime
    • Mark w as all done
  • return currentTime

• DFS(B, 1):
  • w.startTime = 2
  • currentTime ++
  • Mark w as in progress.
  • for v in w.neighbors:
    • if v is unvisited:
      • currentTime = DFS(v, currentTime)
      • currentTime ++
    • w.finishTime = currentTime
    • Mark w as all done
  • return currentTime

• DFS(C, 3):
  • w.startTime = 4
  • currentTime ++
  • Mark w as in progress.
  • for v in w.neighbors:
    • if v is unvisited:
      • currentTime = DFS(v, currentTime)
      • currentTime ++
    • w.finishTime = currentTime
    • Mark w as all done
  • return currentTime

• DFS(D, 6):
  • w.startTime = 7
  • currentTime ++
  • Mark w as in progress.
  • for v in w.neighbors:
    • if v is unvisited:
      • currentTime = DFS(v, currentTime)
      • currentTime ++
    • w.finishTime = currentTime
    • Mark w as all done
  • return currentTime

Start: 0

currentTime = 1

w

unvisited
in progress
all done
Depth First Search

currentTime = 2

• **DFS**(w, currentTime):
  • w.startTime = currentTime
  • currentTime ++
  • Mark w as **in progress**.
  • **for** v in w.neighbors:
    • **if** v is **unvisited**:
      • currentTime = **DFS**(v, currentTime)
      • currentTime ++
    • w.finishTime = currentTime
  • Mark w as **all done**
  • **return** currentTime

Start:0

• unvisited
• in progress
• all done
Depth First Search

\textbf{DFS}(w, \text{currentTime}):

\begin{itemize}
  \item w.startTime = \text{currentTime}
  \item currentTime ++
  \item Mark w as \textbf{in progress}.
  \item \textbf{for} v in w.neighbors:
    \begin{itemize}
      \item if v is \textbf{unvisited}:
        \begin{itemize}
          \item currentTime = DFS(v, currentTime)
          \item currentTime ++
        \end{itemize}
      \end{itemize}
  \item w.finishTime = \text{currentTime}
  \item Mark w as \textbf{all done}
  \item \textbf{return} \text{currentTime}
\end{itemize}
Depth First Search

currentTime = 21

• **DFS**(w, currentTime):
  • w.startTime = currentTime
  • currentTime ++
  • Mark w as *in progress*.
  • **for** v in w.neighbors:
    • **if** v is *unvisited*:
      • currentTime = **DFS**(v, currentTime)
      • currentTime ++
    • w.finishTime = currentTime
  • Mark w as *all done*
  • **return** currentTime

Start: 0

End: 21

Takes until currentTime = 20

unvisited

in progress

all done
Depth First Search

\[
\text{DFS}(w, \text{currentTime}): \\
\begin{align*}
&\text{w.startTime} = \text{currentTime} \\
&\text{currentTime}++ \\
&\text{Mark } w \text{ as \textit{in progress}.} \\
&\text{for } v \text{ in } w.\text{neighbors}: \\
&\quad \text{if } v \text{ is \textit{unvisited}:} \\
&\quad\quad \text{currentTime} = \text{DFS}(v, \text{currentTime}) \\
&\quad\quad \text{currentTime}++ \\
&\quad \text{w.finishTime} = \text{currentTime} \\
&\quad \text{Mark } w \text{ as \textit{all done}} \\
&\quad \text{return } \text{currentTime}
\end{align*}
\]
DFS finds all the nodes reachable from the starting point.

In an undirected graph, this is called a connected component.

One application: finding connected components.
To explore the whole graph

• Do it repeatedly!
Why is it called depth-first?

• We are implicitly building a tree:

• And *first* we go as *deep* as we can.
Running time
To explore just the connected component we started in

• We look at each edge only once.
• And basically don’t do anything else.
• So...

\[ O(m) \]

• (Assuming we are using the linked-list representation)
• (Details on board)
Running time
To explore the whole thing

• Explore the connected components one-by-one.
• This takes time \([on \, board]\)

\[O(n + m)\]

Here the running time is \(O(m)\) like before

Here \(m=0\) but it still takes time \(O(n)\) to explore the graph.
You check:

DFS works fine on directed graphs too!

Only walk to C, not to B.

Siggi the studious stork
Pre-lecture exercise

• How can you sign up for classes so that you never violate the pre-req requirements?

• More practically, given a package dependency graph, how do you install packages in the correct order?
Application: topological sorting

- Question: in what order should I install packages?

Suppose the dependency graph has no cycles: it is a Directed Acyclic Graph (DAG)
Can’t always eyeball it.
Let’s do DFS

Discussion and observations on board.
Finish times seem useful

Claim: In general, we’ll always have:

Suppose the underlying graph has no cycles

To understand why, let’s go back to that DFS tree.
A more general statement
(this holds even if there are cycles)
This is called the “parentheses theorem” in CLRS

• If \( v \) is a descendant of \( w \) in this tree:

\[
\text{timeline}
\begin{array}{cccc}
\text{w.start} & \text{v.start} & \text{v.finish} & \text{w.finish} \\
\end{array}
\]

• If \( w \) is a descendant of \( v \) in this tree:

\[
\begin{array}{cccc}
\text{v.start} & \text{w.start} & \text{w.finish} & \text{v.finish} \\
\end{array}
\]

• If neither are descendants of each other:

\[
\begin{array}{cccc}
\text{v.start} & \text{v.finish} & \text{w.start} & \text{w.finish} \\
\end{array}
\]

(or the other way around)
So to prove this ->

If A → B

Then B.finishTime < A.finishTime

Suppose the underlying graph has no cycles

• **Case 1**: B is a descendant of A in the DFS tree.

• Then

  A.startTime < B.startTime < B.finishTime < A.finishTime

• aka, B.finishTime < A.finishTime.
So to prove this ->

NOTE: In class this case was missing!!!
I messed up 😞
But it’s here now.

- **Case 2:** B is a **NOT** descendant of A in the DFS tree.
- Then we must have explored B before A.
  - Otherwise we would have gotten to B from A, and B would have been a descendant of A in the DFS tree.
- Then
  - aka, \( B.\text{finishTime} < A.\text{finishTime} \).

Suppose the underlying graph has no cycles
Back to this problem

• Question: in what order should I install packages?

Suppose the dependency graph has no cycles: it is a Directed Acyclic Graph (DAG)
In reverse order of finishing time

• Do DFS
• Maintain a list of packages, in the order you want to install them.
• When you mark a vertex as all done, put it at the beginning of the list.

start:3
finish:4

dpkg

start:9
finish:10

coreutils

start:0
finish:11
dpkg

start:2
finish:5

libselinux1

start:3
finish:4

multiarch_support

start:7
finish:8
tar

start:1
finish:6
libbz2

start:9
finish:10

coreutils

start:2
finish:5

libselinux1

start:3
finish:4

multiarch_support

start:7
finish:8
tar

start:1
finish:6
libbz2
For implementation, see IPython notebook

```
In [69]: print(G)

CS161Graph with:
   Vertices:
     dkpg,coreutils,multiarch_support,libselinux1,libbz2.tar, 
   Edges:
     (dkpg,multiarch_support) (dkpg,coreutils) (dkpg,tar) (dkpg,libbz2 
     ) (coreutils,libbz2) (coreutils,libselinux1) (libselinux1,multiarch_support) (libbz2,libselinux1)

In [71]: V = topoSort(G)
   for v in V:
       print(v)

dkpg
tar
coreutils
libbz2
libselinux1
multiarch_support
```
What did we just learn?

- **DFS** can help you solve the **TOPLOGICAL SORTING PROBLEM**
  - That’s the fancy name for the problem of finding an ordering that respects all the dependencies

- Thinking about the DFS tree is helpful.
Example:

This example skipped in class – here for reference.

- Unvisited
- In progress
- All done

Start: 0
This example skipped in class – here for reference.
Example

This example skipped in class – here for reference.
Example

This example skipped in class – here for reference.
Example

This example skipped in class – here for reference.
This example skipped in class – here for reference.
Example

This example skipped in class – here for reference.
Example

Start: 0
Leave: 7

Start: 1
Leave: 6

Start: 2
Leave: 5

Start: 3
Leave: 4

Do them in this order:

A
C
D
B

This example skipped in class – here for reference.
Another use of DFS

- In-order enumeration of binary search trees

Given a binary search tree, output all the nodes in order.

Instead of outputting a node when you are done with it, output it when you are done with the left child and before you begin the right child.
Part 2: breadth-first search
How do we explore a graph?

If we can fly
Breadth-First Search
Exploring the world with a bird’s-eye view

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps
Breadth-First Search
Exploring the world with a bird’s-eye view

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

start
Breadth-First Search
Exploring the world with a bird’s-eye view

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps

start
Breadth-First Search

Exploring the world with a bird’s-eye view

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

start
Breadth-First Search
Exploring the world with a bird’s-eye view

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

World: EXPLORED!
Breadth-First Search
Exploring the world with pseudocode

- Set $L_i = []$ for $i=1,\ldots,n$
- $L_0 = \{w\}$, where $w$ is the start node
- For $i = 0, \ldots, n-1$:
  - For $u$ in $L_i$:
    - For each $v$ which is a neighbor of $u$:
      - If $v$ isn’t yet visited:
        - mark $v$ as visited, and put it in $L_{i+1}$

$L_i$ is the set of nodes we can reach in $i$ steps from $w$

Go through all the nodes in $L_i$ and add their unvisited neighbors to $L_{i+1}$
BFS also finds all the nodes reachable from the starting point.

It is also a good way to find all the connected components.
Running time
To explore the whole thing

• Explore the connected components one-by-one.
• Same argument as DFS: running time is $O(n + m)$

Verify these!

• Like DFS, BFS also works fine on directed graphs.
Why is it called breadth-first?

• We are implicitly building a tree:

  • And first we go as broadly as we can.

YOINK!

Call this the “BFS tree”
Pre-lecture exercise

• What Samuel L. Jackson’s Bacon number?

(Answer: 2)
I wrote the pre-lecture exercise before I realized that I really wanted an example with distance 3

Kevin Bacon

Oliver Sacks

Tilda Swinton

It is really hard to find people with Bacon number 3!
Application: shortest path

• How long is the shortest path between w and v?
Application: shortest path

• How long is the shortest path between w and v?

It’s three!
To find the **distance** between w and all other vertices v

- Do a BFS starting at w
- For all v in $L_i$
  - The shortest path between w and v has length $i$
  - A shortest path between w and v is given by the path in the BFS tree.
- If we never found v, the distance is infinite.
Proof idea (on board)
Proof idea

• Suppose by induction it’s true for vertices in $L_0, L_1, L_2$
  • For all $i < 3$, the vertices in $L_i$ have distance $i$ from $v$.

• Want to show: it’s true for vertices of distance 3 also.
  • aka, the shortest path between $w$ and $v$ has length 3.

• Well, it has distance at most 3
  • Since we just found a path of length 3

• And it has distance at least 3
  • Since if it had distance $i < 3$, it would have been in $L_i$. 

THIS SLIDE SKIPPED IN CLASS

Just the idea... see CLRS for details!
What did we just learn?

- The BFS tree is useful for computing distances between pairs of vertices.
- We can find the shortest path between $u$ and $v$ in time $O(m)$.

The BFS tree is also helpful for:

- Testing if a graph is bipartite or not.
Pre-lecture exercise: fish

• Some pairs of species will fight if put in the same tank.
• You only have two tanks.
• Connected fish will fight.
Application: testing if a graph is bipartite

• Bipartite means it looks like this:

Can color the vertices red and orange so that there are no edges between any same-colored vertices

Example:
- are in tank A
- are in tank B
- if the fish fight

Example:
- are students
- are classes
- if the student is enrolled in the class
Is this graph bipartite?
How about this one?
How about this one?
This one?
Solution using BFS

• Color the levels of the BFS tree in alternating colors.
• If you never color two connected nodes the same color, then it is bipartite.
• Otherwise, it’s not.
Breadth-First Search
For testing bipartite-ness

start

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps
Breadth-First Search
For testing bipartite-ness

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps
Breadth-First Search
For testing bipartite-ness

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

start
Breadth-First Search
For testing bipartite-ness

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps

start
Breadth-First Search
For testing bipartite-ness

CLEARLY BIPARTITE!
Breadth-First Search
For testing bipartite-ness

start

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps
Breadth-First Search
For testing bipartite-ness
Breadth-First Search
For testing bipartite-ness

start

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps
Breadth-First Search

For testing bipartite-ness
Breadth-First Search
For testing bipartite-ness

WHOA NOT BIPARTITE!
Hang on now.

• Just because **this** coloring doesn’t work, why does that mean that there is **no** coloring that works?

I can come up with plenty of bad colorings on this legitimately bipartite graph...

Plucky the pedantic penguin
Some proof required

- If BFS colors two neighbors the same color, then it’s found an **cycle of odd length** in the graph.

Ollie the over-achieving ostrich

Make this proof sketch formal!

There must be an even number of these edges

This one extra makes it odd
Some proof required

• If BFS colors two neighbors the same color, then it’s found an cycle of odd length in the graph.
• So the graph has an odd cycle as a subgraph.
• But you can never color an odd cycle with two colors so that no two neighbors have the same color.
  • [Fun exercise!]

• So you can’t legitimately color the whole graph either.
• Thus it’s not bipartite.
What did we just learn?

BFS can be used to detect bipartite-ness in time $O(n + m)$. 
Outline

• Part 0: Graphs and terminology

• Part 1: Depth-first search
  • Application: topological sorting
  • Application: in-order traversal of BSTs

• Part 2: Breadth-first search
  • Application: shortest paths
  • Application (if time): is a graph bipartite?
Recap

- **Depth-first search**
  - Useful for topological sorting
  - Also in-order traversals of BSTs

- **Breadth-first search**
  - Useful for finding shortest paths
  - Also for testing bipartiteness

- **Both DFS, BFS:**
  - Useful for exploring graphs, finding connected components, etc
Still open (next few classes)

• We can now find components in undirected graphs...
  • What if we want to find strongly connected components in directed graphs?

• How can we find shortest paths in weighted graphs?

• What is Samuel L. Jackson’s Erdos number?
  • (Or, what if I want everyone’s everyone-else number?)
Next Time

• Strongly Connected Components

Before Next Time

• Pre-lecture exercise: Strongly Connected What-Now?