Graphs

A graph is

- a set of nodes
- a set of edges each connecting two nodes
Graphs

A *directed graph* is

- a set of **nodes** 🔄
- a set of **edges** ➔

  each connecting one node to another node

We’ll just use “graph” to mean “directed graph”
Graphs: Lists

At most one outgoing edge ⇒ list
Graphs: Trees

Reach each node in only one way ⇒ tree
Graphs: DAG

Can’t get to a node from itself ⇒

*directed acyclic graph (DAG)*
Graphs: Cycles

Can get to a node from itself $\Rightarrow$ graph
Roots

Some nodes might be considered roots — often nodes that reach all others

[Diagram of interconnected nodes]

[Diagram of interconnected nodes]

[Diagram of interconnected nodes]
Roots

Some nodes might be considered *roots* — often nodes that reach all others.

A graph containingly only trees is a *forest*.
Roots

Some nodes might be considered *roots* — often nodes that reach all others

Can reach all nodes from some root ⇒ a *connected* graph
Roots

Some nodes might be considered *roots* — often nodes that reach all others.

Multiple candidate roots:
Representing Graphs

Graphs can be represented in different ways:

- Nodes as structs/objects, edges as pointers/references
- Nodes as objects, edges in a dictionary
- Nodes a integers, edges as a list of pairs of numbers

Unless you’re solving abstract graph problems, typically you have an existing data definition that you might think of as a graph — probably matching the first case
Designing Programs: Lists

\[
\begin{align*}
&\text{(define } (F \ n) \\
&\quad (\text{cond} \\
&\quad\quad [(\text{empty? } n) \ldots] \\
&\quad\quad [\text{else } \ldots (F \ (\text{rest } n)) \ldots])
\end{align*}
\]

for \( (n = \text{root}; \) \\
\quad \text{n} \neq \text{NULL;} \\
\quad \text{n} = \text{n->next} \) { \\
\quad \ldots.
\}
Designing Programs: Trees

(define (F n)
  (cond
    [(empty? n) ...]
    [else ... (F (child1 n))
     ... (F (childN n)) ...]))

- Depth-first vs. breadth-first
- Might express recursion through a stack or queue
Designing Programs: DAGs

- Sometimes, treat a DAG as a tree
Designing Programs: DAGs

- Sometimes, treat a DAG as a graph...
Designing Programs: Graphs

Like a tree, but accumulate seen

```
(define (F n)
  (cond
    [(seen? n) ...]
    [else (seen! n)
      (cond
        [(empty? n) ...]
        [else ... (F (child1 n))
          ... (F (childN n)) ...]]]]))
```
Designing Programs: Graphs

Depth-first:

or
Designing Programs: Graphs

Breadth-first:

or
Classical Graph Algorithm

Find the shortest path to a node:

Solution: breadth-first search
Classical Graph Algorithm

Find the shortest weighted path to a node:

Neither breadth-first nor depth-first works
Classical Graph Algorithm

Find the shortest weighted path to a node:

Solution: use a **priority queue**
- Enqueue node with distance so far
- Dequeue node that has shortest distance so far

A priority queue gives us “closest-first”
- Instead of a queue (breadth-first)
- Instead of a stack (depth-first)
Shortest Weighted Path
Shortest Weighted Path

0

A

B

C

D

B 1
C 9
Shortest Weighted Path
Shortest Weighted Path

1

C 5
C 9
D 10
Shortest Weighted Path

A

B

C

D

5

1

9

4

3

9

1

9

3

C  9

D  10
Shortest Weighted Path

5

A

B

C

D

D 6
C 9
D 10
Shortest Weighted Path

6

A

B

C

D

C 9
D 10

1 9
4
3

9
1

9
Tracking Seen Nodes

Two common ways to track “already seen” nodes:

• Reserve space in the node for a mutable boolean
  + Easy to implement (in C)
  − Easy to pollute state

• Use a container
  − More work to implement (in C)
  + Avoids extra state