Graph Algorithms, Part I Traversal
Statistics 650/750
Week 4 Tuesday

Christopher Genovese and Alex Reinhart

19 Sep 2017

Announcements

- Windows users: See `documents/Info/windows-path.org` for information about setting your `PATH` to run R or Python from the shell (warning: instructions not yet tested)
- `cow-proximity` problem tweaked
- New `data-structures` and `graphs` problems in the problem bank
- Problem bank reorganized slightly

A Bit About Tools

Graphs

A **graph** is a structure that represents the relationships among a set of entities. It consists of

- A set of **nodes** (also called vertices) representing the entities.
- A set of **edges** each connecting two nodes.

Two nodes connected by an edge are called *neighbors*, or adjacent.

Graphs are enormously important in statistics, in computing, and more generally. Many things can be treated as graphs:

1. Social networks, where nodes are people and edges are relationships between them

2. Statistical models, where nodes are variables and edges represent dependence, causality, or correlation

3. Road maps, where nodes are intersections and buildings and edges represent connections between them via road

4. Electric circuits, where nodes are circuit nodes and edges the resistors, inductors, and capacitors connecting them

5. The Internet is a bunch of nodes (routers and individual computers) connected by Ethernet, WiFi, ISPs, undersea cables...

Treating something complicated, like a social network, as a graph lets us apply a range of powerful tools to work with it. By learning generally about graphs, we learn tools that can be used for many different kinds of problems.

Rather than remembering the details of various graph *algorithms*, it is useful to think about common graph *problems* and how they apply to your situation. Finding good algorithms then follows fairly easily.

Common graph problems:

- Finding connected components
- Finding shortest paths
- Matching and coloring
- Transitive closure
- Finding a minimal spanning tree
- Traversing the graph
- Topological sorting
- Finding separating edges and nodes
- Finding cycles and "tours"
- Maximizing flows through a network
- Calculating useful embeddings.

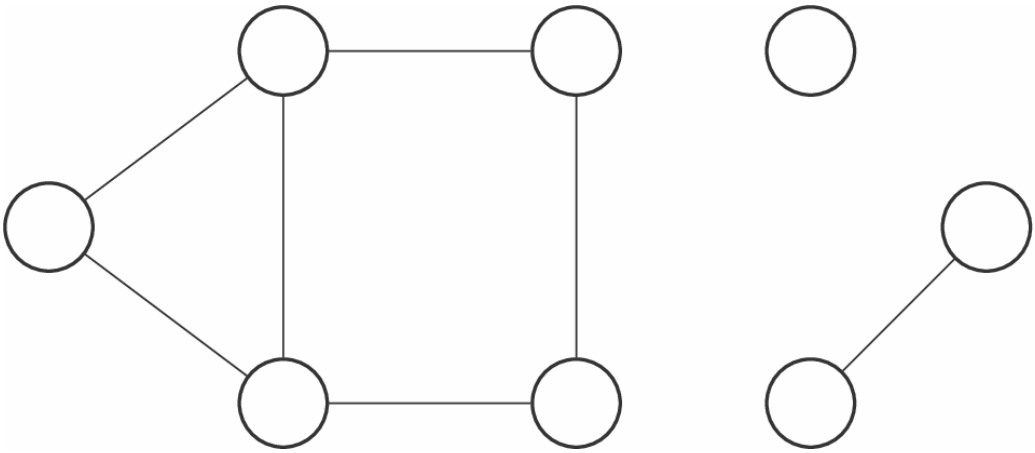
But first, let's learn about graphs.

Flavors of Graphs

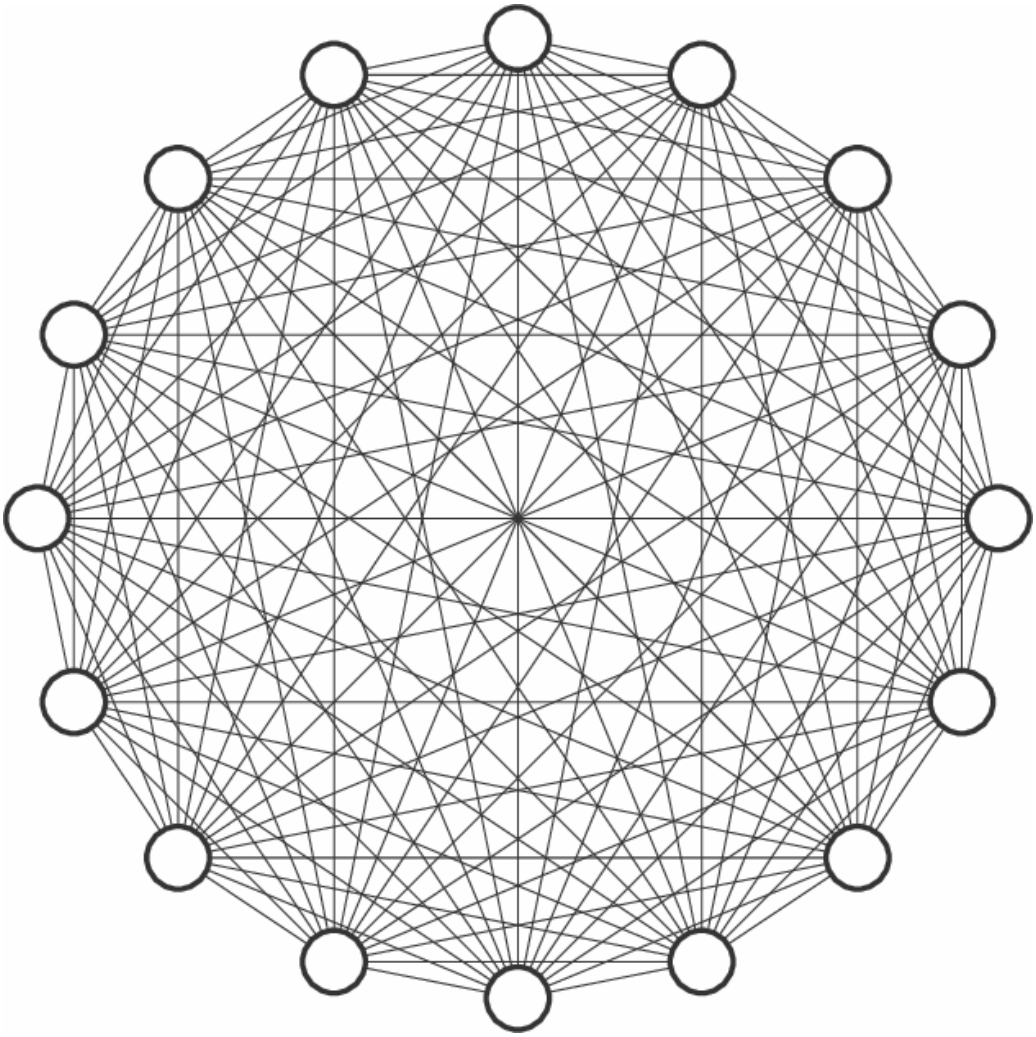
Mathematically, a graph $G = (N, E)$, where N is a *set* of nodes and E is a set containing 2-sets of the form $\{n_1, n_2\}$, that represents an edge between nodes n_1 and n_2. (They are 2-sets because they are not ordered: $\{n_2, n_1\}$ represents the same edge as $\{n_1, n_2\}$.)

A graph is **connected** if one can move between any two nodes by a series of steps between adjacent nodes. That is, any node is a neighbor of a neighbor of a neighbor of a... of any other node.

Graphs are often shown, er, graphically with the nodes as circles (or other shapes) and the edges as lines or curves linking the nodes.



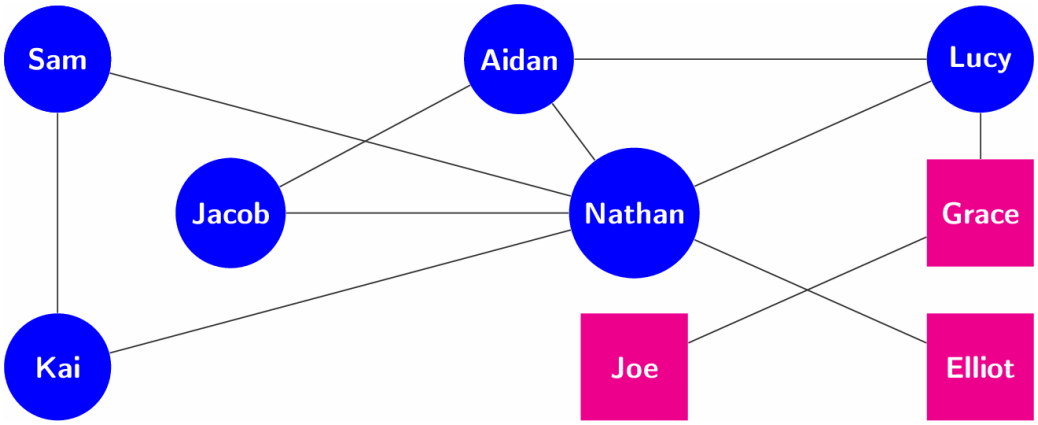
A graph can be basic, like a single node with no edges, or complex, like:



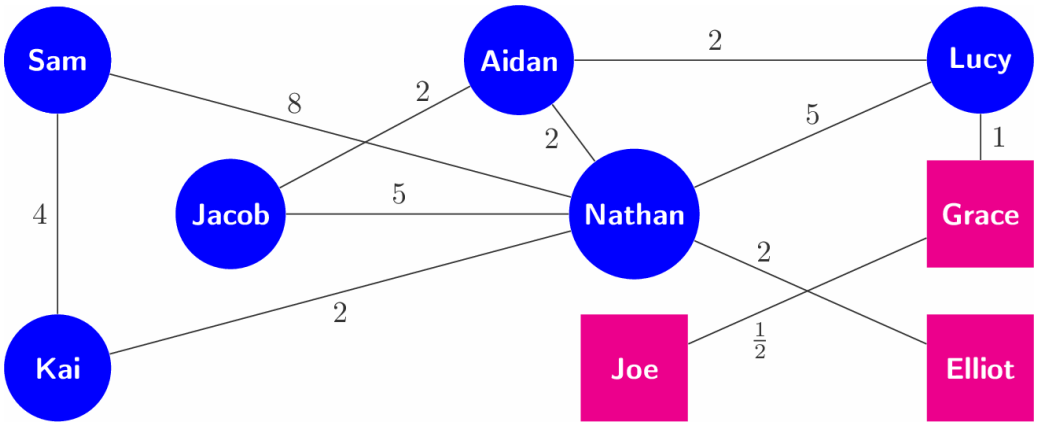
Properties (especially Labels and Weights)

While the nodes and edges define the relationships, we often want to encode more information. To that end, we can associate properties with the nodes and the edges. These properties can be arbitrary data (or meta-data), but most common are **labels** and **weights**, which are strings and numbers, respectively, associated with nodes and/or edges.

Here is a graph with labeled nodes:



Here is a graph with labeled nodes and weighted edges:



Weights might be used to represent many things: the strength of correlation between variables, the distance between two points on the road network, or the number of emails sent between two people, among many other potential applications.

Directed versus Undirected Graphs

The graphs displayed above are **undirected**: the edges $\{n_1, n_2\}$ represent *symmetric* relationships between the nodes. For example, Kai is friends with Nathan if and only if Nathan is friends with Kai.

In contrast, a **directed** graph, or *digraph*, has edges with direction that represent asymmetric relationships. For example, Joe might "like" Grace without Grace liking Joe.

Mathematically, the edges in a digraph are not sets but tuples, specifically ordered pairs (n_1, n_2) representing an edge *from* node n_1 *to* node n_2.

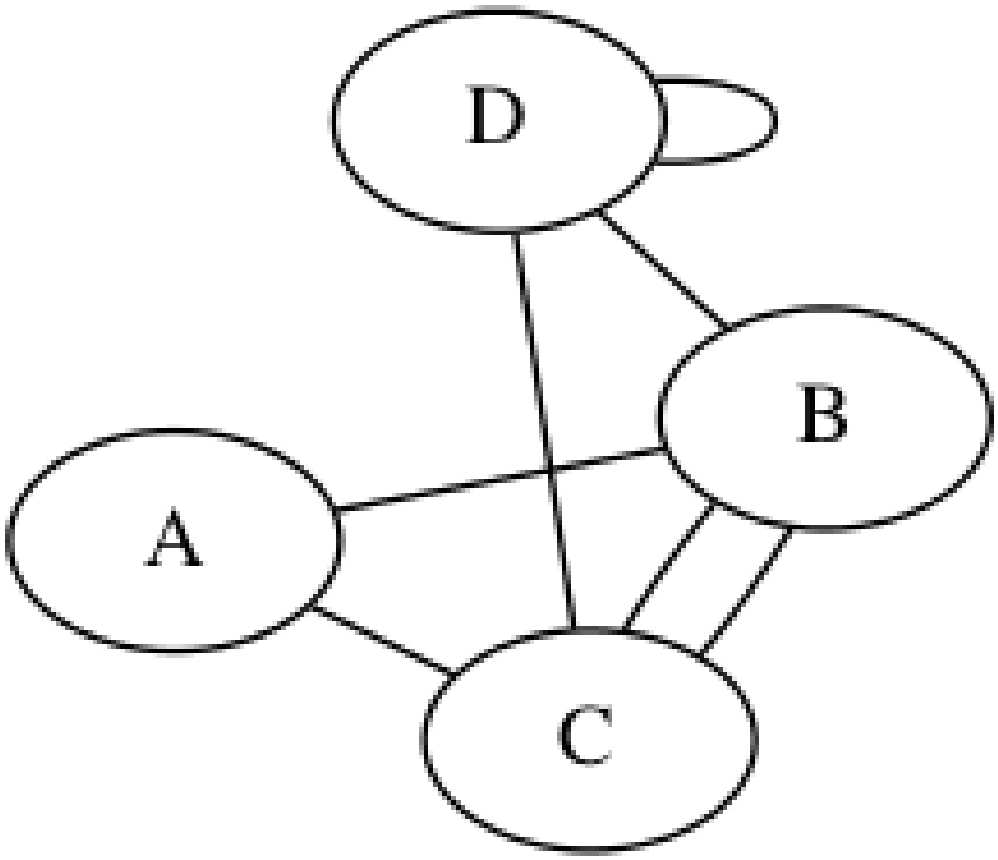
Graphically, we display directed edges as *arrows*:

Simple versus Non-Simple

A *simple* graph has

- no edges from a node to itself,
- at most one edge between any pair of nodes;

otherwise, the graph is *non-simple*. The graphs above were all simple; here's a non-simple example.



Most graphs we work with are simple graphs.

Sparse versus Dense

A graph is *sparse* if only a small fraction of the possible edges are present; otherwise it is *dense*. Which of the examples above are sparse? Which are dense?

Later we'll see that the difference between a sparse and a dense graph has implications for how we choose to represent a graph in our programs.

Consider a large social network, like Facebook, where edges represent "friendship" between two members. Is such a graph sparse or dense?

Cyclic versus Acyclic

A **cycle** in a digraph is a sequence of adjacent nodes (a path, respecting direction) that begin and end with the same node. (There are several different flavors of cycle.)

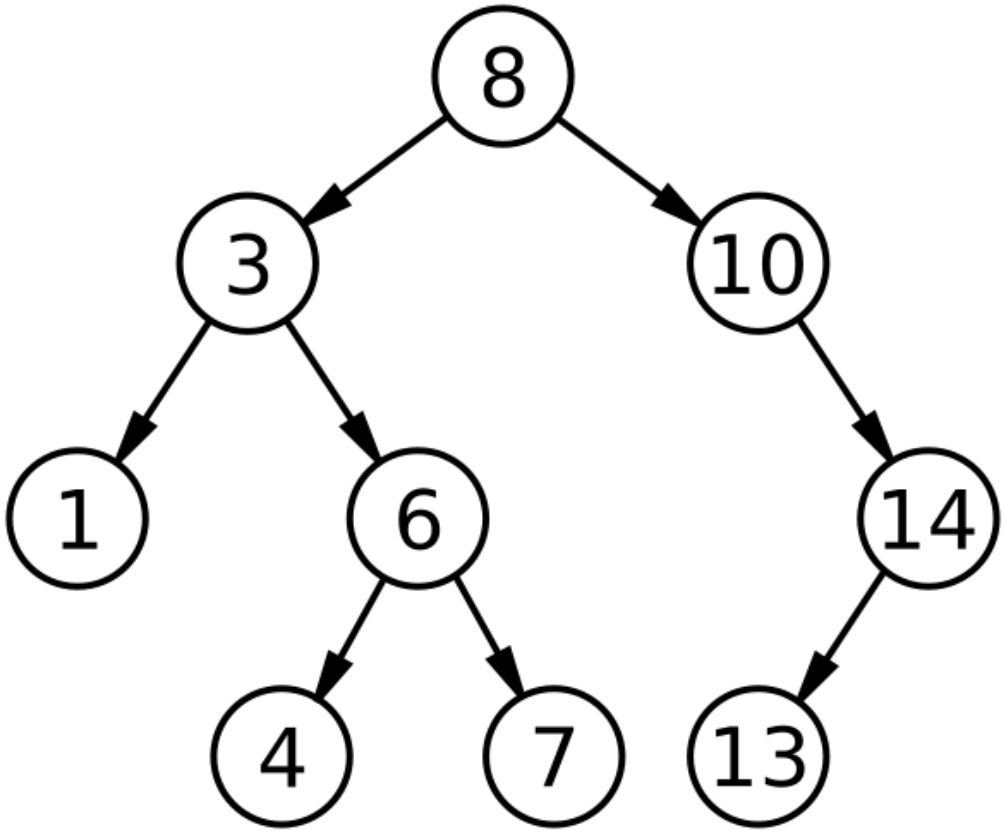
An **acyclic graph** has no cycles (else it is *cyclic*). A directed, acyclic graph is called a **DAG**.

One of the most important types of acyclic graph is a **tree**.

A graph G is a tree if any of the following (equivalent) properties hold:

- G is connected, acyclic graph.
- G is connected but deleting any edge makes it disconnected.
- Any two distinct nodes in G are connected by exactly one *simple path* (a path containing distinct nodes only).
- A finite G with n nodes is a tree if it is acyclic and has $n - 1$ edges.
- A finite G with n nodes is a tree if it is connected and has $n - 1$ edges.

There are many varieties of tree. Here's a binary tree, for example, where each node has only two children:



Representing Graphs

There are several common data structures used to represent graphs. For a graph G, let $n = \#N$ and $e = \#E$ denote the total number of nodes and edges.

When we talked about data structures, we made a distinction between the *abstract operations* a data structure supports and the actual implementation, in code and in memory, of the data structure. The same applies here: a graph is an abstract idea, and there are a few operations we want it to support:

- Check if there is an edge between two nodes
- Add and remove nodes
- Add and remove edges between nodes
- Traverse the graph

Let's discuss a few common implementations, which have different tradeoffs in computational complexity and storage costs.

Adjacency Matrix

An $n \times n$ matrix where an entry at row i and column j indicates whether there is an edge connecting i and j. In digraphs, rows represent source nodes and columns represent destination nodes. In undirected graphs, adjacency matrices are symmetric.

Adjaceny matrices make it easy to look up whether an edge exists. They are useful for dense, simple graphs. Properties must be stored elsewhere.

	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	0	1	1	0	0
C	1	0	0	0	0	1	1
D	0	1	0	0	0	0	0
E	0	1	0	0	0	0	0
F	0	0	1	0	0	0	0
G	0	0	1	0	0	0	0

In graphs with weighted edges, the adjacency matrix often stores the weights on each edge instead of 1. Instead of 0 for no-edge, it is sometimes convenient to use ∞.

Adjacency List

For each node, we store a list of the node's neighbors:

A B
B A C U
C B U E F G
U B C
E C F
F C E G
G C F

We can allow self links and multi-links (via repeats in the list). We can also store data (or a pointer to it) in the list.

Incidence Matrix

An $n \times e$ boolean matrix, where rows represent nodes and columns represent edges, and where the i,jth element being true means that node i is "incident on" edge j.

Operations and Performance

Operation	Adjacency Matrix	Adjacency List	Incidence Matrix	Winner
u, v adjacent?	O(1)	O(n)	O(e)	Adjacency Matrix
degree(u)	O(n)	O(neighbors)	O(e)	Adjacency List
insert node	$O(n^2)$	O(1)	O(n e)	Adjacency List
insert edge	O(1)	O(neighbors)	O(n e)	Adjacency List
delete node	$O(n^2)$	O(e)	O(n e)	depends
delete edge	O(1)	O(e)	O(n e)	Adjacency Matrix
memory (small G)	$O(n^2)$	O(n + e)	O(n e)	Adjacency List
memory (big G)	$O(n^2)$	O(n + e)	O(n e)	Adjacency Matrix
traversal	Theta(n + e)	Theta(n^2)		Adjacency List

Adjacency lists are a good default, but pay attention to the problem at hand.

Group Exercise

Design two data structures/classes that represent an undirected graph in the Adjacency Matrix and List representation, respectively.

You can assume the nodes are identified by numbers from 1..n or 0..(n-1), whichever is more convenient.

Write functions to translate between the two representations.

(For later) Choose one of the operations above in the table and implement it.

Ideas?

```r
adj_matrix_to_adj_list <- function(A) {
    n <- nrow(A)
    adj_list <- vector("list", n)

    for ( i in 1:n ) {
        adj_list[[i]] <- which(A[i,] == 1)
    }

### OR

    adj_list <- lapply(1:n, function(i) { which(A[i,] == 1) })

    return( adj_list )
}
```

Traversal

To traverse a graph is to visit every node and/or edge systematically. This seems boring, but it's actually an important part of many things we want to do with graphs: finding connected components, finding paths between nodes, calculating graph statistics, and much more. Even "finding paths between nodes" is useful for an incredible number of problems, from Google Maps to internet routing, and even tasks as plainly statistical as kernel density estimation can be phrased in terms of traversals of graphs (or trees).

There are several strategies for traversing a graph, each with different useful properties.

Start with an undirected, simple graph.

Q: If I showed you a large, undirected, simple graph and asked you to count the nodes exactly, how would you do it?

```
Keep careful track of which nodes and edges remain to be visited,
which of those visited have incident nodes/edges left to consider,
and which have been fully processed.
```

Strategies

A traversal strategy produces a sequence of nodes and edges, with each node and edge listed exactly once. There are two, very commonly used strategies:

Breadth-First Search (BFS) visit all neighbors of the current node before visiting any of *their* neighbors.

Depth-First Search (DFS) visit all neighbors of the next visited node before visiting the other neighbors of the *current* node.

These strategies differ in how the sequence of nodes is handled.

During traversal, we will assign nodes three possible states:

`fresh` the node has not yet been considered or processed

`visited` the node has been found, but some of its neighbors' edges remain fresh.

`processed` the node has been visited and all its incident nodes have as well.

Because the state of an edge can be inferred from its surrounding nodes, it is usually sufficient to keep track of the node states.

We will maintain a *traversal state* object that contains the current state of all nodes and a record of the history of traversal, including from what node was each node visited, the "times" at which each node is visited and processed, and an indicator (which we can set) of whether the search should stop.

As part of this state, we will also track an arbitrary *accumulator* object, which we will use to store any data or results that we accumulate over the course of the traversal.

To make the traversal flexible, we will pass three *processing functions* to the algorithms:

`before_node` called on fresh nodes when they are visited

`after_node` called on visited nodes when they are processed

`on_edge` called on edge when first traversed

These will be able to access the current node or edge as well as the current traversal state. By specifying difference choices of these functions, we can make our graph traversal algorithm perform many different tasks.

Aside: Stacks and Queues

A brief refresher from our discussion of data structures.

- Stack

  A **stack** is a data structure for processing objects in a *Last In-First Out* manner. The most recent object added is said to be at the "top" of the stack.

  Stacks support two primary (and one optional) operations:

  `push(obj)` add object to the top of the stack

  `pop()` remove the object from the top of the stack and return it

`peek()` return the object from the top of the stack without removing it

It is an error to pop an empty stack.

- Queue

  A **queue** is a data structure for processing objects in a *First In-First Out* manner. The most recent object added is said to be at the "back" of the queue; the next object to be processed is said to be at the "front" of the queue.

  Queues support two primary operations:

  enqueue(obj) add object to the back of the queue

  dequeue() remove the object at the front of the queue and return it

  It is an error to dequeue and empty queue. (Don't confuse dequeue with deque, a different but related data structure.) Sometimes the `front()` operation is provided, which like `peek()`, looks at the front object without removing it

Breadth-First Search (BFS)

We will maintain a **queue** of nodes, initialized with the start node. We successively dequeue a node and process it and enqueue all of that nodes fresh neighbors (processing all the edges along the way).

Inputs:

`graph` an undirected graph

`start` a node at which to start the search

`acc` an accumulator object of arbitrary type

`before_node` a function(node,ts) called when a node is `VISITED`

`after_node` a function(node,ts) called when a node is `PROCESSED`

`on_edge` a function(from,to,ts) called when an edge is traversed

`ts` a traversal state object (newly initialized if `None`)

Output:

- An updated traversal state `ts'`

The algorithm:

```
function bfs(graph, start, acc, before_node=None, after_node=None,
             on_edge=None, ts=None):

  Initialize traversal state ts if not supplied
  Create a new empty queue

  enqueue(start)
  mark start visited
  process start with before_node [if supplied]

  while queue is not empty:
     current_node = dequeue queue

     for neighbors of current_node:
         if neighbor is not processed:
             process edge (current_node, neighbor) with on_edge [if supplied]

         if neighbor is fresh:
             enqueue neighbor
             mark neighbor visited
             set parent[neighbor] = current_node in ts
             process neighbor with before_node [if supplied]

     mark current_node processed
     process current_node with after_node [if supplied]
```

Assume we also have a functions `bfs_all` that calls `bfs` on successive fresh nodes, updating the same traversal state and returning it.

```python
def bfs_all(graph, acc, before_node=None, after_node=None, on_edge=None):
    ts = None

    for node in graph.nodes():
        if ts is None or ts.fresh(node):
            ts = self.bfs(node, acc, before_node, after_node, on_edge, ts)

    return ts
```

Questions? Discussion...
What do the parent pointers do here?

`The node from which node i was visited is assigned to parent[i].`
`These pointers traces the visitation paths of the algorithm.`

- Examples of using BFS

  1. You have a function

     ```
     def inc(graph, node, ts):
         ts.accumulator += 1
     ```

     and call `tstate = bfs(start, 0, inc)`. What is `tstate.accumulator` after the call?

     When the traversal starts the accumulator is 0. Each time a node is visited, this accumulator is incremented. Hence, this counts the nodes in the graph.

  2. You have a function

     ```
     def inc_if_blue(graph, node, ts):
         props = graph.get_node_properties(node)
         if  props['color'] == 'blue':
             ts.accumulator += 1
     ```

     and call `tstate = bfs(start, 0, inc_if_blue)`. What is `tstate.accumulator` after the call?

     As before, we are incrementing the accumulator when we visit nodes, but this time only for blue nodes. Hence, we are counting the number of blue nodes.

  3. You have a function

     ```
     def parents(graph, node, ts):
         parent = ts.parent[node]
         my_name = graph.get_node_properties(node, 'label')
         if parent:
             p_name = graph.get_node_properties(parent, 'label')
         else:
             p_name = None
         ts.accumulator[my_name] = p_name
     ```

     and call `tstate = bfs(start, {}, inc_if_blue)`. What is `tstate.accumulator` after the call?

     Here, we pass in an empty dictionary and record for each node, its label and the label of its "parent" – the node we came from during BFS. The accumulator thus contains a dictionary mapping node labels to parent labels.

4. Find a path between nodes in the graph:

```
def find_path(from_node, to_node, parents):
    path = []
    end = to_node
    while from_node != end and end is not None:
        path.append(end)
        end = parents[end]
    if end is not None:
        path.append(from_node)
        path.reverse()
        return path
    else:
        return None

# ts = bfs(...)
# find_path(0, 3, ts.parent)
```

What kind of path does BFS find?

- Exercises

1. You have a function

```
def blue_labels(graph, node, ts):
    props = graph.get_node_properties(node)
    if  props['color'] == 'blue':
        ts.accumulator.append(props['label'])
```

and call `tstate = bfs(start, [], before_node=blue_labels)`. What is `tstate.accumulator` after the call?

2. Write a function `find_path(parents, start, end)` that takes the BFS tree (through the parent pointers) and returns a list of node IDs giving a path from `start` to `end`, or `None` if there is no such path.

3. Configure BFS to find the **connected components** of a graph, these are the sets of nodes such that within each set there is a path between any two nodes.

```
def collect_visited(graph, node, state):
    """Accumulates list of nodes as they are visited."""
    state.accumulator.append(node)
```

```

def grab_component(graph, components, start, state=None):
    """Collect one connected component and reset state accumulator."""

    state = graph.bfs(start, [], before_node=collect_visited, ts=state)
    components.append(state.accumulator)
    state.accumulator = []

    return state

def connected_components(g):
    """Returns a list of connected components for a graph g"""

    components = []
    ts = None

    for node in g.nodes():
        if ts.fresh(node):
            ts = grab_component(g, components, node, state=ts)

    return components
```

---

4. Configure BFS to determine if the graph can be *two-colored*, meaning that we can assign one of two colors to every node without two nodes of the same color sharing an edge between them. A two-colorable graph is said to be **bipartite**. Find the two coloring or return None/null/NA if the graph is not bipartite.

```
def complementary_color(color):
    return 1 - color

def check_edge(graph, node, neighbor, state):
    node_color = graph.get_node_properties(node, "color")
    nghb_color = graph.get_node_properties(neighbor, "color")

    if node_color == nghb_color:
        ts.accumulator = False  # Bipartite indicator
        ts.finished = True

    graph.update_node_properties(neighbor,
                                 color=complementary_color(node_color))

```

```
15 def two_coloring(g):
16     """Returns a two-coloring of a graph g if bipartite, else False."""
17
18     ts = None
19
20     for node in self.nodes():
21         if ts is None or ts.fresh(node):
22             g.update_node_properties(node, color=0)
23             ts = self.bfs(node, True, on_edge=check_edge, ts=ts)
24
25         if ts.finished:
26             break
27
28     if ts.accumulator:
29         return [(node, g.get_node_properties(node, "color")) for node in g.nodes()]
30     else:
31         return False
```

Depth First Search (DFS)

In contrast to BFS, in DFS we will maintain a **stack** of nodes, initialized with the start node.

We successively pop a node and process it and push all of that node's fresh neighbors (processing all the edges along the way). There is a recursive logic to DFS: for each fresh neighbor, call DFS on it (maintaining state).

```
DFS(start):
  for neighbor in neighbors(start):
     if neighbor is FRESH:
        DFS(neighbor)
```

Wait, where's the stack?

For our algorithm, we take the inputs:

`graph` an undirected graph

`start` a node at which to start the search

`acc` an accumulator object of arbitrary type

`before_node` a function(node,ts) called when a node is VISITED

`after_node` a function(node,ts) called when a node is PROCESSED

`on_edge` a function(from,to,ts) called when an edge is traversed

`ts` a traversal state object (newly initialized if None)

We output an updated traversal state `ts'`.

```
function dfs(graph, start, acc, before_node=None, after_node=None,
             on_edge=None, ts=None):

  tick the clock
  state[node] = VISITED
  visited_time[node] = time

  do before_node processing of node [if supplied]

  for each neighbor of node:
      do on_edge processing of edge(node <-> neighbor) [if supplied]

      if state[neighbor] is FRESH:
          parent[neighbor] = node
          dfs(graph, neighbor, acc, before_node, after_node, on_edge, ts)

  state[node] = PROCESSED
  tick the clock
  processed_time[node] = time

  do after_node processing of node [if supplied]
```

Alternately, we can explicitly use a stack, looping until the stack is empty:

```
function dfs(graph, start, acc, before_node=None, after_node=None,
             on_edge=None, ts=None):
  time = 0
  stack is empty

  if ts is None initialize traversal state:
      state of all nodes = FRESH
      parent[start] = None
      accumulator = acc
      finished = False
  else:
      use ts as traversal state
```

```
push (start, True) onto stack

while stack is not empty and not finished:
    peek at (current, is_node?) on top of stack

    if is_node? is False:
        do on_edge processing of current edge (if specified)
        pop the stack
    else if state[current] is FRESH:
        tick the clock
        state[current] = VISITED

        do before_node processing of current(if specified)

        for each neighbor of current:
            if neighbor is FRESH:
                parent[neighbor] = current
                push (neighbor, True) on stack
                push (edge[current<->neighbor], False) on stack

    else if state[current] is VISITED:
        tick the clock
        state[current] = PROCESSED
        do after_node processing of current (if specified)
        pop the stack
    else:
        pop the stack

return traversal state
```

Again, suppose we have `dfs_all` which continues searching until no fresh nodes are found:

```
1 def dfs_all(self, acc, before_node, after_node, on_edge):
2     ts = None
3
4     for node in self.nodes():
5         if ts is None or ts.fresh(node):
6             ts = self.dfs(node, acc, before_node, after_node, on_edge, ts)
```

```
7
8     return ts
```

- Example Configuring DFS

  1. Task: Print traversal history as DFS runs
     Basic idea: Mark each node as it is being visited and processed, and mark each edge as it is being traversed. Here, we will use node labels to keep track.
     Solution: See print-history.py for the solution.
  2. Task: Detect cycles in a graph with DFS.
     Basic Idea: In an undirected graph, if there are no back edges, we have a tree – hence, no cycles. But any back edge creates a cycle. So, look for back edges.

     ```
     1  def detect_back_edge(g, from_node, to_node, ts):
     2      if ts.visited(to_node) and ts.parent[from_node] != to_node:
     3          # Found back edge
     4          from_label = gr.get_node_properties(from_node, 'label')
     5          to_label = gr.get_node_properties(to_node, 'label')
     6
     7          print("Found cycle with nodes {from_n}"
     8                "and {to_n}".format(from_n=from_label, to_n=to_label))
     9
     10         ts.finished = True
     ```

     Pass this as the `on_edge` argument.

- DAGs and Topological Sort

  A **topological sort** of a DAG is a linear ordering of the DAG's nodes such that if (u, v) is a directed edge in the graph, node u comes before node v in the ordering.

  Given a DAG, how do we use DFS to do a topological sort?

  Algorithm `topological-sort`:

  ```
  Input: A DAG G
  Output: A list of nodes representing a topological sort

  Steps: Run DFS on G, configured with after_node so that
  after each node is processed, we push it onto the front
  of a linked list (or equivalently onto a stack).

  Return the list of nodes.
  ```

- Other Applications and Exercises

  1. Configure `dfs` to count the number of "descendants" of a node.
  2. Configure `dfs` to compute a path between two nodes.

- Application: Directed Graphs and Strongly Connected Components

  A directed graph is strongly connected if there is a *directed* path between any two nodes.

  The strongly connected components of a directed graph are its maximal, strongly connected subgraphs.

  We can find the strongly connected components with two DFS's. For an arbitrary node v, the graph is strongly connected if we can find a directed path from v to any other node u **and** a directed path from any node w to v.

  Let G be a directed graph and let G' have the same nodes and edges with all the edges *reversed*. Pick an arbitrary node v.

  The algorithm for *detecting* strong connectivity is basically as follows:

  1. Do `DFS(G, v)`.
  2. If the traversal does not contain all nodes, then there are nodes we cannot reach from v. Hence, G cannot be strongly connected.
  3. Do `DFS(G', v)`.
  4. If this traversal does not contain all nodes, then there are nodes in G from which we cannot reach v. Hence, G cannot be strongly connected.

  To find the strongly connected components, we just do a little processing.

  In step 1, record the `processed_times`. In step 3, do `DFS_ALL(G')` with the nodes ordered as the reversal of the `processing_times`.