Chris Gregg
CS106B

Section #7
May 21, 2018

Section Handout #7: Graphs

Based on handouts by various current and past CS106B/X instructors and TAs.

This week has practice with graph structures, including graph properties and algorithms that act on graphs. The first page or so is a cheat sheet for terminology relating to graphs.

graph: A data structure containing a set of vertices V (sometimes called "nodes"), and a set of edges E ("arcs"), where each is a connection between 2 vertices.
degree: number of edges touching a given vertex.
path: A path from vertex A to B is a sequence of edges that can be followed starting from A to reach B. It can be represented as vertices visited, or edges taken.
neighbor or adjacent: Two vertices connected directly by an edge.
reachable: Vertex A is reachable from B if a path exists from B to A.
connected graph: A graph is connected if every vertex is reachable from every other.
cycle: A path that begins and ends at the same vertex.
acyclic graph: One that does not contain any cycles.
loop: An edge directly from a vertex to itself.
weight: Cost associated with a given edge.
weighted graph: One where edges have weights (see graph adjacent).
directed graph: A graph where edges are one-way connections.
undirected graph: A graph where edges don't have a direction.
depth-first search (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking. Often implemented recursively.
breadth-first search (BFS): Finds a path between two vertices by taking one step down all paths and then immediately backtracking. Often implemented by maintaining a queue of vertices to visit.

Depth-first search (DFS) pseudo-code:

```
dfs (v1, v2):
    mark v1 as visited, and add to path.
    perform a dfs from each of v1's unvisited neighbors n to v2:
        if dfs(n, v2) succeeds:
            a path is found! yay!
    if all neighbors fail:
        remove v1 from path.
```

Breadth-first search (BFS) pseudo-code:

```
bfs (v1, v2):
    create a queue of vertexes to visit, initially storing just v1.
    mark v1 as visited.
    while queue not empty and v2 is not seen:
        dequeue a vertex v from it,
        mark that vertex v as visited,
        add each unvisited neighbor n of v to the queue.
```

Important parts of Stanford BasicGraph library: (more online)

```
BasicGraph()
g.addEdge(v1, v2);
g.addVertex(vertex);
g.clear();
g.getEdge(v1, v2)
g.getEdgeSet()
g.getEdgeSet(vertex)
g.getNeighbors(vertex)
```

```
g.getVertex(name)
g.getVertexSet()
g.isConnected(v 1, v2)
g.isEmpty()
g.removeEdge(v1, v2) ;
g.removeVertex(vertex);
g.size()
g.toString()
```

```
struct Vertex {
    string name;
    Set<Edge *> edges;
};
```

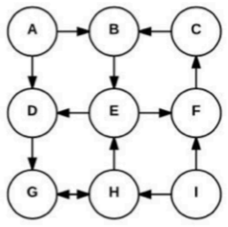
```
struct Edge {
    Vertex *start;
    Vertex *finish;
    double cost;
};
```

1. Graph Properties

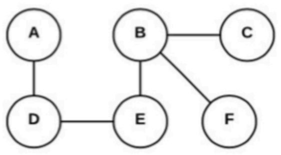
For each of the graphs shown below, answer the following questions.

a) Is the graph directed or undirected?
b) Is the graph weighted or unweighted?
c) Which graphs are connected, and which are not? Is any graph **strongly** connected?
d) Which graphs are cyclic, and which are acyclic?
e) What is the degree of each vertex? If directed, what is the in-degree and the out degree?

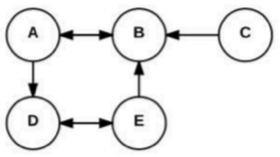
Graph 1:



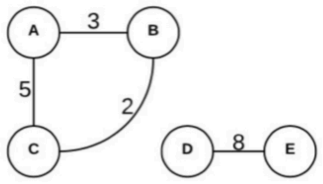
Graph 2:



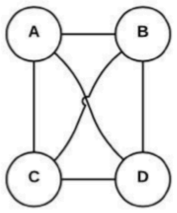
Graph 3:



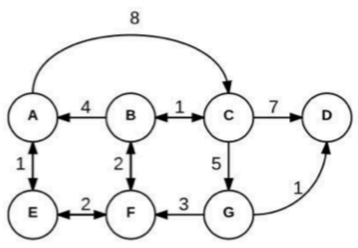
Graph 4:



Graph 5:



Graph 6:



2. Depth-First Search (DFS)
Write the paths that a depth-first search would find from vertex A to all other vertices in the following graphs. If a given vertex is not reachable from vertex A, write "no path" or "unreachable." Do this for graphs 1 and 6 above.

3. Breadth-First Search (BFS)
Write the paths that a breadth-first search would find from vertex A to all other vertices in the following graphs. If a given vertex is not reachable from vertex A, write "no path" or "unreachable." Do this for graphs 1 and 6 above.

4. *k*th Level Friends
Imagine a graph of social network friends, where users are vertices and friendships are edges. Write a function that takes in a social network graph, a Vertex in that graph, and a value k and returns the set of people who are exactly k hops away from the Vertex (and not fewer). For example, if k = 1, those are v's direct friends; if k = 2, they are your friends-of-friends. If k = 0, return a set containing only the user. Assume input arguments are valid.

```
Set<Vertex *> kthLevelFriends(BasicGraph& graph, Vertex *v, int k) {…
```

5. Cycling
Write a function that returns true if a graph contains any cycles, or false if not.

```
bool hasCycle(BasicGraph& graph) { ...
```

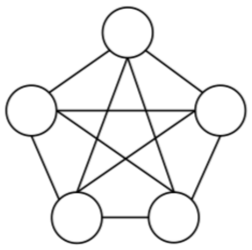
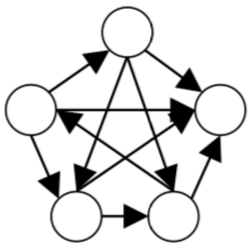
6. Is it Reachable?
Write a function named **isReachable** that returns true if a path can be made from the vertex v1 to the vertex v2, or false if not. If the two vertices are the same, return true. Use either BFS or DFS, described in the reference above. Bonus: do this problem twice with both BFS and DFS.

```
bool isReachable(BasicGraph& graph, Vertex* v1, Vertex* v2) { ...
```

7. Is it Connected?
Write a function that takes in a graph and returns true if a path can be made from every vertex to any other vertex, or false if there is any vertex that cannot be reached by a path from some other vertex. An empty graph is defined as being connected.

```
bool isConnected(BasicGraph& graph) { ...
```

8. Crowning the Champion
Recall that a complete graph is an undirected graph where every single graph node is connected to every other node. A tournament graph is a directed graph that comes from a complete graph where you impose a direction on each and every arc. Informally, a tournament graph is a summary of who prevailed over whom in an exhaustive competition of one-on-one matches, where every single person eventually competes – exactly once – against everyone else. Below, on the left, is a complete graph on five nodes, and on the right is one possible tournament graph that can be derived from the complete graph.

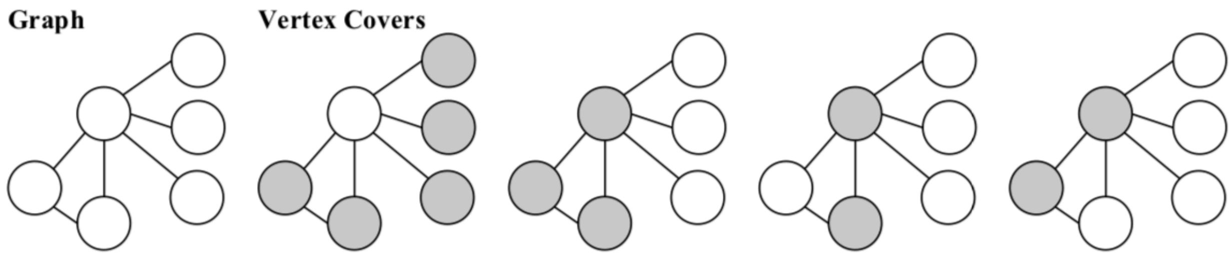
The tournament graph on the above right (with numbering starting at the top and increasing in a clockwise order) states that player 1 beat players 2, 3, and 4 (but not 5), that player 2 lost to everybody, and so on.

A *tournament champion* is a node in a tournament representing someone who, for every other player, either directly prevailed over that player, or prevailed over someone who prevailed over that player. In other words, a node is a champion if one can travel from it to every other node via a path of at most 2 arcs. In the above example, players 1, 3, and 5 are all champions, but players 2 and 4 are not. Take some time to figure out why that is. Then, write a function that, given a tournament graph, returns the tournament champions for that graph.

```
Set<Vertex *> crownTournamentChampions(BasicGraph& graph) { ...
```

9. [CHALLENGE PROBLEM] Minimum Vertex Cover

A vertex cover is a subset of an undirected graph's vertices such that each and every edge in the graph is incident to at least one vertex in the subset. A minimum vertex cover is a vertex cover of the smallest possible size (where the size is determined by the number of nodes in the cover). Suppose you have the following graph on the left.



Each of the four illustrations on the right shows some vertex cover where shaded nodes are included in the vertex cover and unshaded ones are excluded. Each one is a vertex cover because each edge touches at least one vertex in the cover. The two vertex covers on the right are minimum vertex covers because you can't have a vertex cover with fewer than two nodes.

Write a function that, given a graph, returns a minimum vertex cover for the graph. If there are multiple minimum vertex covers, you can return an arbitrary one.

```
Set<Vertex *> findMinimumVertexCover(BasicGraph& graph) { ...
```