Nick Troccoli
CS 106X

Section Handout #7: Graphs

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

Extra practice problems: CodeStepByStep – height, isBalanced

This week is all about **graphs**. This first page is a summary of different graph-related terms and search algorithms:

Graph: A data structure containing:
- a set of vertices V (sometimes called "nodes"),
- a set of edges E (sometimes called "arcs"), where each edge is a connection between two vertices

Degree: The 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.
- a path can be represented by the vertices visited or the edges followed.

Neighbors (or adjacent vertices): Two vertices connected directly by an edge.

Reachable: Vertex A is reachable from B is a path exists from B to A.

Directed graph: A graph where edges are one-way connections.

Undirected graph: A graph where edges don't have a direction (sometimes called "bidirectional" edges).

Connected graph: A graph is connected if every vertex is reachable from every other vertex. Directed graphs can be broken down more specifically into:
- **Strongly connected:** every vertex is reachable from every other following the direction of the edges.
- **Weakly connected:** every vertex is reachable from every other when disregarding the edges' directions.

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 (sometimes called a "self-loop").

Weight: The cost associated with a given edge.
- **Weighted graph:** one where the edges have weights.

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 before taking another step down any path.
- often implemented by maintaining a queue of vertices to visit.

Dijkstra's algorithm: Finds paths between one vertex and all other vertices by maintaining information about how to reach each vertex (i.e. total path cost and previous vertex) and continually improving that information until it reaches the best solution.
- often implemented by maintaining a *priority queue* of vertices to visit.

A* algorithm: A variation of Dijkstra's algorithm that incorporates a heuristic function to prioritize the order in which to visit the vertices.

Minimum spanning tree: The set of connected edges with the smallest total weight that touches every vertex in an undirected graph.

Kruskal's algorithm: An algorithm to find the minimum spanning tree of an undirected graph.

Depth-first search (DFS) pseudo-code:

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

Dijkstra's algorithm pseudo-code:

```
function dijkstra(v1, v2):
  •consider every vertex to have a cost of
   infinity, except v1 which has a cost of 0.
  •create a priority queue of vertexes, ordered
    by cost, storing only v1.

  while the pqueue is not empty:
    •dequeue a vertex v from the pqueue, and mark
     it as visited.
    •for each of the unvisited neighbors n of v,
     we now know that we can reach this neighbor
     with a total cost of (v's cost + the weight
     of the edge from v to n).
       •if the neighbor is not in the pqueue,
        or this is cheaper than n's current cost,
        we should enqueue the neighbor n to the
        pqueue with this new cost, and with v as
        its previous vertex.

when done, we can reconstruct the path from v2
back to v1 by following the previous pointers.
```

Breadth-first search (BFS) pseudo-code:

```
function bfs(v1, v2):
  create a queue of vertexes to visit,
    initially storing just v1.
  mark v1 as visited.

  while queue is 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 queue.
```

A* algorithm pseudo-code:

```
function astar(v1, v2):
  •consider every vertex to have a cost of
   infinity, except v1 which has a cost of 0.
  •create a priority queue of vertexes, ordered
    by heuristic cost, storing only v1 with a
    priority of H(v1, v2).

  while the pqueue is not empty:
    •dequeue a vertex v from the pqueue, and mark
     it as visited.
    •for each of the unvisited neighbors n of v,
     we now know that we can reach this neighbor
     with a total cost of (v's cost + the weight
     of the edge from v to n).
       •if the neighbor is not in the pqueue,
        or this is cheaper than n's current cost,
        we should enqueue the neighbor n to the
        pqueue with this new cost plus H(n, v2),
        and with v as its previous vertex.

when done, we can reconstruct the path from v2
back to v1 by following the previous pointers.
```

Important parts of the 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(v1, 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 *(graphs)*

For the graphs shown below, answer the following questions:

a) Which graphs are directed, and which are undirected?

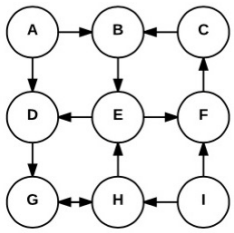
b) Which graphs are weighted, and which are unweighted?

c) Which graphs are connected, and which are not? Is any graph strongly connected?

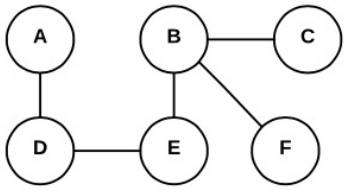
d) Which graphs are cyclic, and which are acyclic?

e) What is the degree of each vertex? (If it is directed, what is the in-degree and out-degree?)

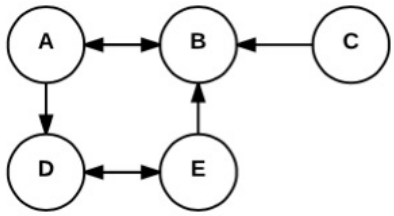
Graph 1:



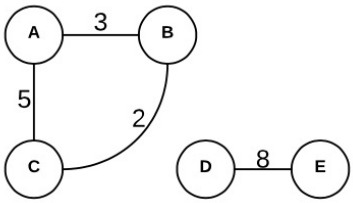
Graph 2:



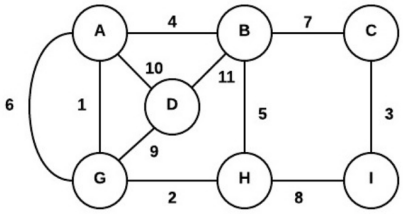
Graph 3:



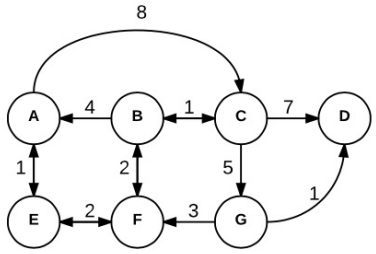
Graph 4:



Graph 5:



Graph 6:



2. Depth-first search *(graphs)*

Write the paths that a depth-first search would find from vertex A to all the other vertices in **graph 1** and **graph 6** above, assuming that the order of exploration is north, west, east, south. If a given vertex is not reachable from vertex A, write "no path" or "unreachable".

3. Breadth-first search *(graphs)*

Write the paths that a breadth-first search would find from vertex A to all other vertices in **graph 1** and **graph 6**. Which paths are shorter than the ones found by DFS in the previous problem?

4. Minimum weight paths *(graphs)*

Which paths found by DFS and BFS on **graph 6** in the previous problems are not minimal weight? What are the minimal weight paths from vertex A to all other vertices? *(Just inspect the graph manually.)*

5. Dijkstra's and A* *(graphs)*

Trace through Dijkstra's algorithm to find the shortest paths from vertex A to all other vertices in **graph 6**. Qualitatively, how would A* perform differently relative to Dijkstra's if it were to use the cartesian distance between two vertices as its heuristic? Do Dijkstra's and A* always find the same path?

6. kthLevelFriends *(graphs)*

Imagine a graph of Facebook friends, where users are vertices and friendships are edges. Write a function called **`kthLevelFriends`** that returns the set of people who are exactly *k* hops away from the vertex *v* (and not fewer). For example, if *k* = 1, those are *v*'s direct friends; if *k* = 2, they are *v*'s friends-of-friends. If *k* = 0, return a set containing only *v* itself. You may assume all the input arguments are valid.

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

7. hasCycle *(graphs)*

Write a function named **`hasCycle`** that returns true if a graph contains any cycles, or false if not.

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

8. findMinimumVertexCover *(graphs)*

Write function named **`findMinimumVertexCover`** that returns a set of vertex pointers identifying a minimum vertex cover. A *vertex cover* is a subset of an undirected graph's vertices such that 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. Consider the following graph on the left:

Each of the four illustrations after it on the right shows some vertex cover (shaded nodes are included in the vertex cover, and hollow 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 there is no smaller vertex cover.

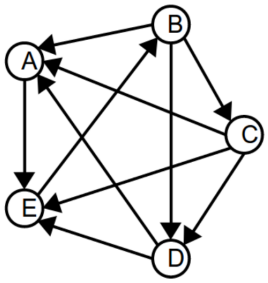
Understand that because the graph is undirected, that means for every edge that leads from some vertex v1 to v2, there will be an edge that leads from v2 to v1. If there are two or more minimum vertex covers, then you can return any one of them. The implementation of this function should consider every possible vertex subset, keeping track of the smallest one that covers the entire graph.

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

9. tournamentWinners *(graphs)*

Write a function **`tournamentWinners`** that accepts a graph representing a tournament and returns a set of all the winners in that tournament. A *tournament* is a graph of a contest among *n* players. Each player plays a game against each other player, and either wins or loses it (no ties). A player is represented by a vertex, and a directed edge shows the winner pointing to the loser. In the tournament shown, player A won against player E, but lost against players B, C, and D.

A tournament *winner* is a player who, for each other player, either won a game against that player, or won a game against a player who won their game against that player (or both). For example, in the graph on the right, players B, C, and E are tournament winners. However, player D is not a tournament winner, because they neither beat player C, nor beat anyone who in turn beat player C. Although player D won against player E, who in turn won against player B, who then won against player C, under our definition player D is not a winner.




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