CS106X
Autumn 2015 Cynthia Lee

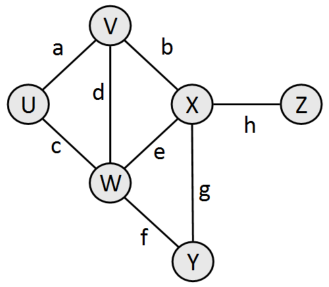
Section 7 (Week 8) Handout

Problem and solution authors include Marty Stepp.

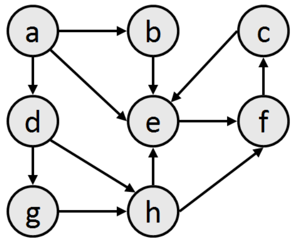
Reference Sheet:

This week is about graphs with vertexes and edges. The first couple pages are cheat sheets for graph terminology and common search algorithms. Recommended problems: do 1-4 to get comfortable. 5-7 are good problems to start with, and give 8 a shot if you are feeling adventurous.

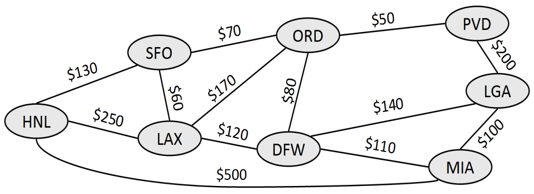
graph: A data structure containing:
  a set of **vertexes** V *(sometimes called "nodes")*,
  a set of **edges** E *("arcs")*, where each is a connection between 2 vertexes.



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.
  can be represented as vertexes visited, or edges taken
neighbor or **adjacent**: Two vertexes connected directly by an edge.
reachable: Vertex A is reachable from B if a path exists from A to B.
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 below)*.
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 vertexes by exploring each possible path as far as possible before backtracking.
  Often implemented recursively.



breadth-first search (BFS): Finds a path between two vertexes by taking one step down all paths and then immediately backtracking.
  Often implemented by maintaining a queue of vertexes to visit.
Dijkstra's algorithm: Finds paths between one vertex and all other vertexes by maintaining information about how to reach each vertex (cost and previous vertex) and continually improving that information until it reaches the best solution.
  Often implemented by maintaining a *priority queue* of vertexes to visit.
A* algorithm: A variation of Dijkstra's algorithm that incorporates a heuristic function to prioritize the order in which to visit the vertexes.
minimum spanning tree: the set of connected edges with the smallest total weight that covers every vertex in the graph
Kruskal's algorithm: An algorithm to find the minimum spanning tree of a graph

Depth-first search (DFS) pseudo-code:

```
function dfs(v1, v2):
  dfs(v1, v2, { }).

function dfs(v1, v2, path):
  path += v1.
  mark v1 as visited.
  if v1 is v2:
    a path is found!

  for each unvisited neighbor n of v1:
    if dfs(n, v2, path) finds a path:
      a path is found!

  path -= v1.  // path is not found.
```

Breadth-first search (BFS) pseudo-code:

```
function bfs(v1, v2):
  queue := {v1}.
  mark v1 as visited.

  while queue is not empty:
    v := queue.dequeue().
    if v is v2:
      a path is found!

    for each unvisited neighbor n of v:
      mark n as visited.
      queue.enqueue(n).

  // path is not found.
```

Dijkstra's algorithm pseudo-code:

```
function dijkstra(v1, v2):
  for each vertex v:
    v's cost := infinity.
    v's previous := none.
  v1's cost := 0.
  pqueue := {v1, at priority 0}.

  while pqueue is not empty:
    v := pqueue.dequeue().
    mark v as visited.
    for each unvisited neighbor n of v:
      cost := v's cost +
              weight of edge (v, n).
      if cost < n's cost:
        n's cost := cost.
        n's previous := v.
        enqueue/update n in pqueue.
  reconstruct path back from v2 to v1.
```

A* algorithm pseudo-code:

```
function astar(v1, v2):
  for each vertex v:
    v's cost := infinity.
    v's previous := none.
  v1's cost := 0.
  pqueue := {v1, at priority H(v1, v2)}.

  while pqueue is not empty:
    v := pqueue.dequeue().
    mark v as visited.
    for each unvisited neighbor n of v:
      cost := v's cost +
              weight of edge (v, n).
      if cost < n's cost:
        n's cost := cost.
        n's previous := v.
        enqueue n at priority (cost + H(n, v2)).
  reconstruct path back from v2 to v1.
```

Important parts of Stanford Graph 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;

    double cost;      // initially 0.0
    bool visited;     // initially false
    Vertex* previous; // initially NULL
};
```

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

    bool visited;   // initially false
};
```

Problem 1: Graph Searching

a. Graph properties

For the graphs shown below, answer the following questions:

i) Which graphs are directed, and which are undirected?
ii) Which graphs are weighted, and which are unweighted?
iii) Which graphs are connected, and which are not? Is any graph strongly connected?
iv) Which graphs are cyclic, and which are acyclic?
v) What is the degree of each vertex? (If it is directed, what is the in-degree and out-degree?)

```
Graph 1:

A --> B <-- C
|     |     ^
|     |     |
V     V     |
D <-- E --> F
|     ^     ^
|     |     |
V     |     |
G <-> H <-- I
```

```
Graph 2:

A     B----C
|     |\_
|     |  \_
D-----E    F
```

```
Graph 3:

A <--> B <-- C
|      ^
V      |
D <--> E
```

```
Graph 4:

    3
  A---B
  |  /
 5| /2
  |/
  C   D---E
        8
```

```
Graph 5:

A-----B
|\   /|
| \ / |
|  +  |
| / \ |
|/   \|
C-----D
```

```
Graph 6:
    ____________
   /    8       \
  /  4       1   V   7
 A <---- B <---> C ----> D
 ^       ^       |       ^
1|      2|       |5      |
 |       |       |      1/
 V   2   V   3   V      /
 E <---> F <---- G ----/
```

b. depth-first search (DFS)

Write the paths that a depth-first search would find from vertex A to all other vertexes in graphs 1 and 6. If a given vertex is not reachable from vertex A, write "no path" or "unreachable".

c. breadth-first search (BFS)

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

d. minimum weight paths

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 nodes? *(Just inspect the graph manually.)*

e. **`kthLevelFriends`**. Imagine a graph of Facebook friends, where users are vertexes and friendships are edges. Write a function

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

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 your friends-of-friends. If *k* = 0, return a set containing only the user. (Assume input arguments are valid.)

f. **`isReachable`**. Write a function

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

that returns `true` if a path can be made from the vertex `v1` to the vertex `v2` , or false if not. If the two vertexes 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.

g. **`isConnected`**. Write a function

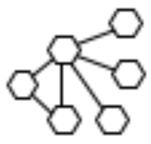
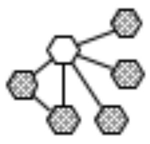
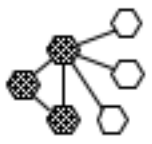
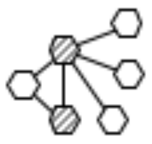
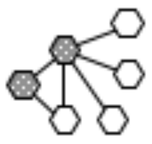
```
bool isConnected(BasicGraph& graph)
```

that returns `true` if a path can be made from every vertex to any other vertex, or false if there is any vertex cannot be reached by a path from some other vertex. An empty graph is defined as being connected. You can use the `isReachable` function from the previous problem to help solve this one.

8. **`findMinimumVertexCover`**. Write a function

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

that returns a set of vertex pointers identifying a minimum vertex cover. A *vertex cover* is a subset of an undirected graph's vertexes 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. Consider the following graph on the left:

Graph	Vertex Covers			
				

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. Think of this as a backtracking problem. The implementation of this function should consider every possible vertex subset, keeping track of the smallest one that covers the entire graph. Try all possible vertex combinations using a "choose-explore-unchoose" pattern and keep track of state along the way.