CSE 373 Section Handout #10
Graph Reference

SearchableGraph<V, E> methods:

```
public SearchableGraph<V, E>()    // undirected, unweighted
public SearchableGraph<V, E>(boolean directed, boolean weighted)
public void addEdge(V v1, V v2)
public void addEdge(V v1, V v2, E e)
public void addEdge(V v1, V v2, int weight)
public void addEdge(V v1, V v2, E e, int weight)
```

```
public void addVertex(V v)
public void clear()
public void clearEdges()
public boolean containsEdge(E e)
public boolean containsEdge(V v1, V v2)
public boolean containsVertex(V v)
public int cost(List<V> path)
public int degree(V v)
public E edge(V v1, V v2)
public int edgeCount()
public Collection<E> edges()
public int edgeWeight(V v1, V v2)
public int inDegree(V v)
public boolean isDirected()
```

```
public boolean isEmpty()
public boolean isReachable(V v1, V v2)
public boolean isWeighted()
public List<V> minimumWeightPath(V v1, V v2)
public Set<V> neighbors(V v)
public int outDegree(V v)
public void removeEdge(E e)
public void removeEdge(V v1, V v2)
public void removeVertex(V v)
public List<V> shortestPath(V v1, V v2)
public String toString()
public String toStringDetailed()
public int vertexCount()
public Set<V> vertices()
```

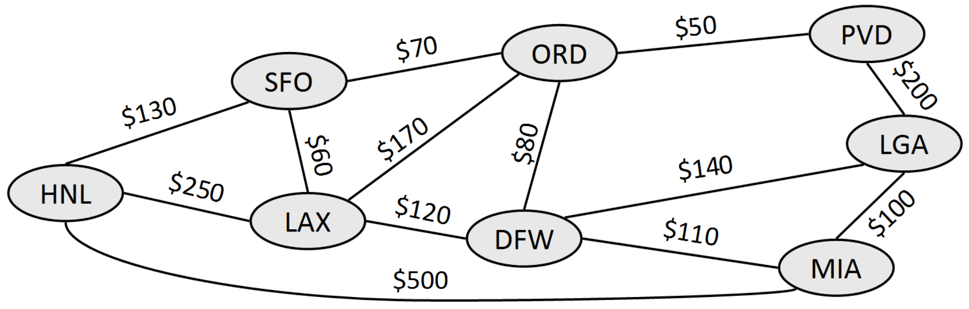
Dijkstra's algorithm pseudo-code:

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

  while pqueue is not empty:
    v := pqueue.removeMin().
    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.
  reconstruct path back from v2 to v1.
```

Topological sort pseudo-code:

```
// to be used on a DAG (directed, acyclic)
function topologicalSort():
  map := {each vertex -> its in-degree}.
  queue := {all vertices with in-degree 0}.
  ordering := { }.

  while queue is not empty:
    vertex v := queue.removeFirst().
    ordering += v.
    for each neighbor n of v:
      decrease n's in-degree in map by 1.
      if n's in-degree is now 0:
        queue += n.
```

Example graph of airports and flights:



CSE 373 Section Handout #10
Problems

1. **Graph implementation questions**
   What are 2 operations that can be performed quickly by each of the following graph implementations?
   What are 1-2 operations that perform poorly and run slowly on each implementation?
   Do the implementations have any other advantages or drawbacks besides runtime?

   a) adjacency matrix b) adjacency list

2. **Dijkstra's Algorithm 1**
   Perform Dijkstra's algorithm on the following graph to find the minimum-weight paths from **vertex A** to all other vertices. Reconstruct all paths and give the total cost of each path.

```
     ____________
    /    5       \
   /  4       2   ▼   7
  A ◄---► B ◄---► C ----► D
  |       ▲       |       ▲
 1|      2|       |3      |
  |       |       |      1/
  ▼   9   ▼   3   ▼      /
  E ◄---► F ◄---- G ----/
```

3. **Dijkstra's Algorithm 2**
   Perform Dijkstra's algorithm on the following graph to find the minimum-weight paths from **vertex B** to all other vertices. Reconstruct all paths and give the total cost of each path.

```
     4     2
  A --- B --- C
  |     |     |
 4|    6|     |1
  |     |     |
  D --- E --- F
  |  2     2  |
 1|           |1
  |     5     |
  G --------- H
```

4. **Topological Sort**
   Perform a topological sort on each of the two graphs below. Any valid topological sort ordering is correct as long as you follow the algorithm properly and show your work at each step.

```
Graph 4a:

    ____________
   /            \
  /              ▼
 A ◄---- B ----► C ◄---- D
 |       |       |       |
 |       |       |       |
 |       |       |      /
 ▼       ▼       ▼     /
 E ----► F ◄---- G ◄---/
```

```
Graph 4b:

P      ----------- A ◄--- U
▲     /            ▲      /
|     |            |     /
S     |     Y ◄--- M ◄--/
      |     ▲      |
      \     |      ▼
       ---► T ◄--- R
```

CSE 373 Section Handout #10
Problems, continued

The following problems use the `Graph` interface and `SearchableGraph` class shown in lecture.

5. `affordableFlights`
Write a method named `affordableFlights` that accepts three parameters: (1) a `Graph<String, String>` parameter representing airline flights (like in the picture on the syntax reference sheet, where each vertex represents a city airport, each edge represents a flight, and the weight of each edge represents the cost of the flight in dollars); (2) a string representing a starting city; and (3) and an integer representing a maximum amount of money to spend. Your method should return an alphabetized `Set` of all cities that can be visited from that starting city on a path with a total cost of the given max amount or less (not including the starting city itself). For example, if a variable named `graph` represents the graph given by the picture on the syntax reference, the call of `affordableFlights(graph, "PVD", 220)` should return `[DFW, LAX, LGA, ORD, SFO]`. *(You can just check vertex in the graph to see if it can be reached for the given cost limit, though this is not an ideal algorithm in terms of runtime.)*

6. `isConnected`
Write a method named `isConnected` that accepts a `Graph<String, String>` parameter and 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 just check every pair of vertices to see if they can reach each other, though this is not an ideal algorithm in terms of runtime.)*

7. `isCyclic`
Write a method named `isCyclic` that accepts a `Graph<String, String>` parameter and returns `true` if a path can be made from any vertex back to that same vertex (a cycle), or `false` if there are no cycles in the graph. To figure out whether a graph contains any cycles, use the following pseudo-code algorithm. *(You will have to maintain the mapping of vertex to unvisited/partial/visited yourself; the existing `vertexInfo` is not up to the task.)*

```
function isCyclic(graph):
    mark all vertices in the graph as UNVISITED.
    for each vertex v in the graph:
        if visit(graph, v) returns true, then the graph contains a cycle.
    if no vertex v returned true, the graph does not contain a cycle.

function visit(graph, v):
    mark v as being PARTIALLY VISITED.
    for each neighbor v2 of v:
        if v2 is PARTIALLY VISITED, then the graph contains a cycle.
        if v2 is UNVISITED:
            if visit(graph, v2) returns true, then the graph contains a cycle.
    mark v as FULLY VISITED.
    return false.
```

CSE 373 Section Handout #10
Solutions

1.

```
a) adjacency matrix
   - fast:  find out whether vertices A and B are neighbors;
            find edge weight between two vertices;
            add and remove edges
   - slow:  add/remove a vertex;
            (somewhat slow) find all neighbors of a given vertex
   - other: high memory usage (O(V^2))

b) adjacency list
   - fast:  find all neighbors of a given vertex;
            add/remove a vertex;
            add an edge
   - slow:  (somewhat fast) find out whether vertices A and B are neighbors;
            (somewhat slow) remove an edge
```

2.

```
graph : [A:0,/, B:inf,/, C:inf,/, D:inf,/, E:inf,/, F:inf,/, G:inf,/]
pqueue: [A:0, B:inf, C:inf, D:inf, E:inf, F:inf, G:inf]

remove A, process neighbors B/C/E, update B cost to 4, C to 5, E to 1
graph : [A:0,/, B:4,A, C:5,A, D:inf,/, E:1,A, F:inf,/, G:inf,/]
pqueue: [E:1, B:4, C:5, D:inf, F:inf, G:inf]

remove E, process neighbor F, update F cost to 10
graph : [A:0,/, B:4,A, C:5,A, D:inf,/, E:1,A, F:10,E, G:inf,/]
pqueue: [B:4, C:5, F:10, D:inf, G:inf]

remove B, process neighbors C/F, update F cost to 6
graph : [A:0,/, B:4,A, C:5,B, D:inf,/, E:1,A, F:6,B, G:inf,/]
pqueue: [C:5, F:6, D:inf, G:inf]

remove C, process neighbors D/G, update D cost to 12, G cost to 8
graph : [A:0,/, B:4,A, C:5,B, D:12,C, E:1,A, F:6,B, G:8,C]
pqueue: [F:6, G:8, D:12]

remove F ... no unprocessed neighbors, no changes.
remove G, process neighbor D, update D cost to 9
graph : [A:0,/, B:4,A, C:5,B, D:9,G, E:1,A, F:6,B, G:8,C]
pqueue: [D:9]

remove D... no unprocessed neighbors, no changes.
final paths:
A to B: [A, B], cost=4
A to C: [A, C], cost=5
A to D: [A, C, G, D], cost=9
A to E: [A, E], cost=1
A to F: [A, B, F], cost=6
A to G: [A, C, G], cost=8
```

CSE 373 Section Handout #10
Solutions, continued

3.

```
graph : [A:inf,/, B:0,/, C:inf,/, D:inf,/, E:inf,/, F:inf,/, G:inf,/, H:inf,/]
pqueue: [B:0, A:inf, C:inf, D:inf, E:inf, F:inf, G:inf, H:inf]

remove B, process neighbors A/C/E, update A cost to 4, C to 2, E to 6
graph : [A:4,B, B:0,/, C:2,B, D:inf,/, E:5,B, F:inf,/, G:inf,/, H:inf,/]
pqueue: [C:2, A:4, E:6, D:inf, F:inf, G:inf, H:inf]

remove C, process neighbor F, update F cost to 3
graph : [A:4,B, B:0,/, C:2,B, D:inf,/, E:5,B, F:3,C, G:inf,/, H:inf,/]
pqueue: [F:3, A:4, E:6, D:inf, G:inf, H:inf]

remove F, process neighbors E/H, update E cost to 5, H cost to 4
graph : [A:4,B, B:0,/, C:2,B, D:inf,/, E:5,F, F:3,C, G:inf,/, H:4,F]
pqueue: [A:4, H:4, E:5, D:inf, G:inf]

remove A, process neighbor D, update D cost to 8
graph : [A:4,B, B:0,/, C:2,B, D:8,A, E:5,F, F:3,C, G:inf,/, H:4,F]
pqueue: [H:4, E:5, D:8, G:inf]

remove H, process neighbor G, update G cost to 9
graph : [A:4,B, B:0,/, C:2,B, D:8,A, E:5,F, F:3,C, G:9,H, H:4,F]
pqueue: [E:5, D:8, G:9]

remove E, process neighbor D, update D cost to 7
graph : [A:4,B, B:0,/, C:2,B, D:7,E, E:5,F, F:3,C, G:9,H, H:4,F]
pqueue: [D:7, G:9]

remove D, process neighbor G, update G cost to 8
graph : [A:4,B, B:0,/, C:2,B, D:7,E, E:5,F, F:3,C, G:8,D, H:4,F]
pqueue: [G:8]


remove G... no unprocessed neighbors, no changes.
final paths:
B to A: [B, A], cost=4
B to C: [B, C], cost=2
B to D: [B, C, F, E, D], cost=7
B to E: [B, C, F, E], cost=5
B to F: [B, C, F], cost=3
B to G: [B, C, F, E, D, G], cost=8
B to H: [B, C, F, H], cost=4
```

CSE 373 Section Handout #10

4. a) one possible valid ordering:

```
map:   {A=1, B=0, C=3, D=0, E=1, F=3, G=2}
queue: [B, D], order: []

map:   {A=0, C=2, D=0, E=1, F=2, G=2}       // remove/process B; update A,C,F
queue: [D, A], order: [B]

map:   {A=0, C=1, E=1, F=2, G=1}            // remove/process D; update C,G
queue: [A], order: [B, D]

map:   {C=0, E=0, F=2, G=1}                 // remove/process A; update C,E
queue: [C,E], order: [B, D, A]

map:   {E=0, F=2, G=0}                      // remove/process C; update G
queue: [E, G], order: [B, D, A, C]

map:   {F=1, G=0}                           // remove/process E; update F
queue: [G], order: [B, D, A, C, E]

map:   {F=0}                                // remove/process G; update F
queue: [G], order: [B, D, A, C, E, G]

map:   {}                                   // remove/process F; complete
queue: [], order: [B, D, A, C, E, G, F]
```

b) one possible valid ordering:

```
map:   {A=2, M=1, P=1, R=1, S=0, T=2, U=0, Y=2}
queue: [S, U], order: []

map:   {A=2, M=1, P=0, R=1, T=2, U=0, Y=2}
queue: [U, P], order: [S]                   // remove/process S; update P

map:   {A=1, M=0, P=0, R=1, T=2, Y=2}
queue: [P, M], order: [S, U]                // remove/process U; update M,A

map:   {A=1, M=0, R=1, T=2, Y=2}
queue: [M], order: [S, U, P]                // remove/process P

map:   {A=0, R=0, T=2, Y=1}
queue: [A, R], order: [S, U, P, M]          // remove/process M; update A,R,Y

map:   {R=0, T=1, Y=1}
queue: [R], order: [S, U, P, M, A]          // remove/process A; update T

map:   {T=0, Y=1}
queue: [T], order: [S, U, P, M, A, R]       // remove/process R; update T

map:   {Y=0}
queue: [Y], order: [S, U, P, M, A, R, T]    // remove/process T; update Y

map:   {}
queue: [], order: [S, U, P, M, A, R, T, Y]  // remove/process Y; complete

                    "SUP MARTY?"   "NOT MUCH, SUP WITH YOU DOGG?"
```

CSE 373 Section Handout #10
Solutions, continued

5.

```
public static Set<String> affordableFlights(Graph<String, String> graph,
        String startCity, int maxCost) {
    Set<String> affordable = new TreeSet<String>();
    for (String city : graph.vertices()) {
        if (!city.equals(startCity)) {
            List<String> cheapest = graph.minimumWeightPath(startCity, city);
            if (graph.cost(cheapest) <= maxCost) {
                affordable.add(city);
            }
        }
    }
    return affordable;
}
```

6.

```
public static boolean isConnected(Graph<String, String> graph) {
    for (String v1 : graph.vertices()) {
        for (String v2 : graph.vertices()) {
            if (!graph.isReachable(v1, v2)) {
                return false;
            }
        }
    }
    return true;
}
```

7.

```
public static boolean isCyclic(Graph<String, String> graph) {
    Map<String, String> marking = new HashMap<String, String>();
    for (String v : graph.vertices()) {
        marking.put(v, "UNVISITED");
    }
    for (String v : graph.vertices()) {
        if (visit(graph, marking, v)) {
            return true;
        }
    }
    return false;
}

private static boolean visit(Graph<String, String> graph,
        Map<String, String> marking, String v) {
    marking.put(v, "PARTIAL");
    for (String v2 : graph.neighbors(v)) {
        String v2mark = marking.get(v2);
        if (v2mark.equals("PARTIAL")) {
            return true;
        } else if (v2mark.equals("UNVISITED")) {
            if (visit(graph, marking, v2)) {
                return true;
            }
        }
    }
    marking.put(v, "VISITED");
    return false;
}
```