Last name _____

First name _____

LARSON—MATH 356—LAB WORKSHEET 06
Bipartite Test & Dijkstra's Algorithm!

Reminders

1. Homework *h*04, the Test 1 Review, is due 11:59 p.m. on Monday, Mar. 22.
2. Test 1 is Tuesday, Mar. 23.
3. There is a special Bonus Talk on Wednesday 1:00 (the Discrete Math Seminar).
4. Read ahead in our textbook. We're into Chp. 2 and trees!

1. Log in to your Sage/CoCalc account.
   (a) Start the Chrome browser.
   (b) Go to `http://cocalc.com` and sign in.
   (c) You should see an existing Project for our class. Click on that.
   (d) Click "New", call it **s06**, then click "Sage Worksheet".
   (e) For each problem number, label it in the Sage cell where the work is. So for Problem 1, the first line of the cell should be `#Problem 1`.
   (f) When you are finished with the worksheet, click "make pdf", email me the pdf (at `clarson@vcu.edu`, with a header that says **Math 356 s06 worksheet attached**).

Saving and Re-using Code

2. Any code you want to re-use can be saved in a `.sage` file (a text file with a name that just helps you remember what's in the file) and can be *loaded* into memory at any time. There is a "graphs.sage" file in your Cocalc project Handouts—copy that into your "home" directory (to make it yours—then you can change it—the copy in the Handouts folder will change as I change it).

3. Go to your files list, click on "graphs.sage" to see what that file looks like. Add your own comment to this file (like "#I can add my own graphs here!").

4. Now *load* that file. Run: `load('graphs.sage')`.

5. Run: `my_graphs` to see what we have so far.

Testing if a (Connected) Graph is Bipartite

Our goal is to write a function `is_bipartite(G)` that takes a connected graph G as input and returns TRUE is G is bipartite and FALSE if it is not bipartite.

6. One step in the algorithm described in class is to check if a set of vertices in a graph has an edges between them. Copy and evaluate the following function. Does it work (run tests)? Why?

```
def has_edge(g,S):
    E = g.edges(labels=False)
    for v in S:
        for w in S:
            if (v,w) in E or (w,v) in E:
                return True
    return False
```

7. Here is a simplified version of the algorithm described in class. Instead of keeping track of the vertices at each distance from an input vertex v_1, we just keep finding the next set of neighbors (which must be the vertices at the next distance) and testing if that set has any edges. If it doesn't we keep going. If we ever do find an edge our graph can't be bipartite (as, we argued, there must be an odd cycle).

   Copy and evaluate the following function. Does it work (run tests)? Why?

```
def is_bipartite(g):

    V = g.vertices()
    v=V[0]
    so_far = [v]
    current = [v]
    N = neighbors_of_set(g,current)

    while len(so_far) < len(V):
        current = []
        for w in N:
            if w not in so_far:
                current.append(w)
                so_far.append(w)
        if has_edge(g,current):
            return False
        N = neighbors_of_set(g,current)
    return True
```

Implement Dijkstra's algorithm

Now we want a function `dijkstra(G,v)` that takes a weighted graph G vertices v_1 and v_2 as inputs and outputs the length (total weight) of a shortest weighted path from v_1 to v_2.

8. First we'll need some weighted graphs we can use to test our algorithm. We can easily generate these with random edges and random weights. Copy and evaluate the following function. Does it work (run tests)? Why?

```
def random_weighted_graph(n,m):
    g = Graph(n)
    for v in srange(n):
        for w in srange(n):
            if v < w:
                g.add_edge(v,w,randint(0,m))
    return g
```

9. In the Dijkstra's algorithm we discussed in class, we repeatedly tested edges from a "solved" set S to a not-yet-solved set $\bar{S}$. To do this we needed to get the weights from those edges. Copy and evaluate the following function. Does it work (run tests)? Why?

```
def get_weight(u,w,E):
    for e in E:
        if e[0]==u and e[1]==w:
            return e[2]
        if e[1]==u and e[0]==w:
            return e[2]
    return None
```

10. Some of the pairs of vertices from S to $\bar{S}$ aren't actually adjacent. If we try to test these our code will break. Here's a function to test if there is actually an edge between a pair of vertices. S to a not-yet-solved set $\bar{S}$. Copy and evaluate the following function. Does it work (run tests)? Why?

```
def is_edge(u,w,E):
    for e in E:
        if e[0]==u and e[1]==w:
            return True
        if e[1]==u and e[0]==w:
            return True
    return False
```

11. The following implementation uses Python *dictionaries*. What are they? How do they work?

12. We have all the ingredients now to code the Dijkstra's algorithm from class. Copy and evaluate the following function. Does it work (run tests)? Why?

```
def dijkstra(g,v1,v2):

    E = g.edges()
    max_label = max(e[2] for e in E)+1
    V = g.vertices()
    S = [v1]
    V.remove(v1)
    S_bar = V
    labels = {v1:0}

    while len(S_bar) > 0:
        smallest_label = max_label
        for u in S:
            for w in S_bar:
                if is_edge(u,w,E):
                    if labels[u] + get_weight(u,w,E) < smallest_label:
                        smallest_label = labels[u] + get_weight(u,w,E)
                        best_vertex = w

        S.append(best_vertex)
        S_bar.remove(best_vertex)
        labels[best_vertex]=smallest_label

    return labels[v2]
```