1. (a) Start Chrome browser.
   (b) Go to http://cocalc.com and “Sign In”.
   (c) Click project Math 356.
   (d) Click “New”, call it c35, then click “Sage Worksheet”.

We need some examples of weighted graphs. We previously coded the following function that puts random integer weights on the edges of a given graph \( g \). Copy that here. Let \( pete \) be the Petersen graph. Evaluate \( pete=graphs.PetersenGraph() \). Then let \( pete1=random_integer_edge_weights(pete,10) \). “Show” it with:
\[ pete1.show(edge_labels=True) \]
and draw what you get.

Our current algorithmic problem is to find a shortest weighted path between a pair of vertices in a graph. We discussed the Dijkstra’s algorithm pseudo-code below (from our book). Let’s implement it. Instead of using lists—which depend on an ordering of the vertices—let’s use dictionaries. Different data structures have advantages and disadvantages—it’s less natural to iterate over dictionaries—but we can access their entries by their keys very efficiently.

2. We found we needed to access edge weights: given vertices \( v \) and \( w \) what is \( w(vw) \)?
We can use:

\[
def weight(g,v,w):
    W = [z for (x,y,z) in g.edges() if (x==v and y==w) or (x==w and y==v)]
    if len(W) > 0:
        return W.pop()
\]

Test this function with edges in \( pete1 \).

3. Evaluate \( D=\{"A":5, "B":\infty\} \). \( D \) is a Python dictionary. The strings “A” and “B” are keys, while 5 and \( \infty \) are values. We can access the values with the keys. Evaluate \( D["A"] \) and \( D["B"] \). You can change the value of a dictionary entry by assignment: to change the value associated to “A” to 4, use \( D["A"]=4 \). That’s it. To see that it changed evaluate \( D["A"] \). To see the whole dictionary just evaluate \( D \).

4. We’ll see that its natural to use dictionaries for \( D \) and \( P \) (even the pseudo-code suggests it). For \( Q \) lists are probably more natural (Python lists even have a .remove() method!). The ingredients we’ll need are \( \text{len}(Q) \) for “\( \text{length}(Q) \)” in the pseudo-code, \( g\text{.neighbors}(v) \) for the neighbors of \( v \) in graph \( g \) (that’s “\( \text{adj}(v) \)” in the pseudo-code). Now we’re ready to write an implementation of Dijkstra’s algorithm using the above pseudo-code and these ingredients. Remember those “←” symbols are assignment operators—that’s just “=” in Python.
Algorithm 3.5 A general template for Dijkstra’s algorithm.

Input: An undirected or directed graph \( G = (V, E) \) that is weighted and has no self-loops. The order of \( G \) is \( n > 0 \). A vertex \( s \in V \) from which to start the search. Vertices are numbered from 1 to \( n \), i.e. \( V = \{1, 2, \ldots, n\} \).

Output: A list \( D \) of distances such that \( D[u] \) is the distance of a shortest path from \( s \) to \( v \). A list \( P \) of vertex parents such that \( P[u] \) is the parent of \( v \), i.e. \( v \) is adjacent from \( P[u] \).

1. \( D \leftarrow [\infty, \infty, \ldots, \infty] \) \( \quad \) \( > n \) copies of \( \infty \)
2. \( D[s] \leftarrow 0 \)
3. \( P \leftarrow [\ ] \)
4. \( Q \leftarrow V \) \( \quad \) \( > \) list of nodes to visit
5. while length(Q) > 0 do
   6. find \( v \in Q \) such that \( D[v] \) is minimal
   7. \( Q \leftarrow \text{remove}(Q, v) \)
   8. for each \( u \in \text{adj}(v) \cap Q \) do
      9. if \( D[u] > D[v] + w(vu) \) then
         10. \( D[u] \leftarrow D[v] + w(vu) \)
         11. \( P[u] \leftarrow v \)
   12. return \((D, P)\)

5. Our level sets problem is, given a connected graph \( g \) and vertex \( v \) as inputs, to find the sets \( L_0(v), L_1(v), \ldots, L_i(v) \), the set of vertices at distance \( i \) from \( v \). There will be a last non-empty set. At this point we’ll have a partition of the vertices. Test it!

```python
def dijkstra(g,s):
    D={}
    V= g.vertices()
    for v in V:
        D[v]=infinity
    D[s]=0
    P={}
    P[s]=s
    Q=g.vertices()
    while len(Q) > 0:
        minD = min([D[v] for v in Q])
        v = [v for v in D.keys() if v in Q and D[v]==minD].pop()
        Q.remove(v)
        AdjQ = [w for w in g.neighbors(v) if w in Q]
        for u in AdjQ:
            if D[u] > D[v] + weight(g,v,u):
                D[u] = D[v] + weight(g,v,u)
                P[u] = v
    return D,P
```

def level_sets(g,v):
    L = {}
    D = dijkstra(g,v)[0]
    for d in D.values():
        L[d] = []
    for v in g.vertices():
        d = D[v]
        L[d].append(v)
    return L