Problem Set 3

All parts are due April 6, 2017 at 11:59PM.

Name: Caroline Pech

Collaborators: Carly Staub, Sonya Das

Part A

Problem 3-1.

(a) When G is a connected, unweighted, undirected graph with V vertices and E edges, if each vertex is colored with 1 of k colors, then the shortest path between vertices s and t such that no two consecutive vertices are the same color would involve running BFS (breadth first search) starting at vertex s. At each iteration of BFS, if any neighbor is the same color as the node, stop exploring that path and return None. Once the target node is reached, terminate. The run time of this algorithm is $O(V + E)$ because, in the worst case, one needs to explore all nodes and all edges of graph G to find vertex t, and, as shown in class, the run time of BFS is $O(V + E)$.

(b) Use BFS to find the shortest path, but one has to be aware of the vertex and previous edge that was traversed. Augment each vertex, once explored, with a set of colors that was used to traverse into the vertex. No vertex is queued or exhausted because each vertex has a possibility of being revisited each time that BFS is run on the graph (k times). If a vertex has already been entered via a path with a given color, BFS will not enter this vertex (i.e. if already visited by red path, vertex can't be visited via another red path). This is because a vertex V could have k colors for edges, and every edge could be able to exit V successfully, but only one of the k colors might be able to successfully traverse V as well as reach t. Once t is reached, append to a dictionary where the key is the length of the path and the value is a list of the path, which can be found by using parent pointers provided from BFS. After BFS has run k times, return the value for the key in the dictionary with the shortest path. That way if there are multiple paths from s to t discovered when running BFS k times, the shortest path is returned.

The run time of this algorithm is $O(k(V + E))$ time because each vertex is visited a maximum of k times. In the worst case, if there is no path from s to t, then BFS

will run k times exhaustively and return None. Inserting in a dictionary is $O(1)$, finding the minimum of the keys is $O(k)$, and returning the corresponding value is $O(1)$. Therefore the total remains $O(k(V + E))$.

Problem 3-2.

(a) If the Graphrican government would only like to build one road, an algorithm to run in $O(V + E)$ time is: choose a city from the V possible cities and run DFS to explore every city it is already connected to. Keep track of the vertices visited in a dictionary where the key is the current node and the value is the "parent", or the node that was visited previously. When DFS hits a node that has already been explored (but there are still unvisited nodes in the graph), begin a new dictionary for the next component. To minimize storage, if there are more than 2 connected components, after storing the first 2 dictionaries, any new dictionary that is created will only be stored if it ends up being larger than one of the 2 current stored dictionaries. This will override the smaller of the two, so that at any point, only 2 dictionaries are stored. Once DFS has terminated, find the lengths of all the dictionaries for each connected components and connect an edge (create a road) using one node from the longest dictionary and another node from the second largest dictionary. This maximizes the number of reachable cities because the one road connects all cities from the two largest components, and this returns the correct optimization because connecting the two largest components creates $x * y$ new pairs of reachable cities where x and y are the number of nodes in the largest two connected components. Therefore the product is the largest when x and y are the largest hence using the largest two connected components. Because DFS runs in $O(V + E)$ time then this algorithm also runs in $O(V + E)$ because, in the worst case, all vertices are visited and all edges are traversed.

Example: If a graph contains 3 connected components such that:

1. component 1 contains 3 vertices connected with 3 edges (in a triangle)
2. component 2 contains 2 vertices connected with 1 edge
3. component 3 contains one vertex with no edges

After DFS terminates, the two resultant dictionaries will be $\{v_0 : None, v_1 : v_0, v_2 : v_1\}$ and $\{v_3 : None, v_4 : v_3\}$ since the dictionary $\{v_5 : None\}$ will be overridden or never even stored depending on the ordering of vertices explored in DFS. An edge would be added to connect any two nodes from both components. Now 5 cities are all connected to 4 cities thereby maximizing the total reachable number of cities by adding one road.

(b) If the Graphrican government has the capability to build k roads where $0 < k \leq V$, then to generalize the algorithm given in part a, choose a node from a list of unvisited nodes (initially all nodes in the graph) and run DFS until hitting a node that has already been visited. Back track and continue until visited all possible nodes are visited within that connected component and add to a dictionary at each iteration where the keys are the current node and the value is the parent (the node from the iteration prior). When DFS jumps to the next connected component in order to explore all unvisited nodes, begin a new dictionary so that there is a dictionary for each connected component. To minimize storage, if there are more than $k + 1$ connected components, after storing the first $k + 1$ dictionaries, any new dictionary that is created will only be stored if it ends up being larger than one of the $k + 1$ current stored dictionaries. This will override the smallest of the k, so that at any point, only $k + 1$ dictionaries are stored. After DFS terminates, connect an edge between a node in the largest dictionary and a node in the second largest dictionary. Pop the largest dictionary from the list of stored dictionaries and recurse until the k edges are added to the graph (connecting the $k+1$ largest components.

Example: If a graph contains 5 connected components such that:

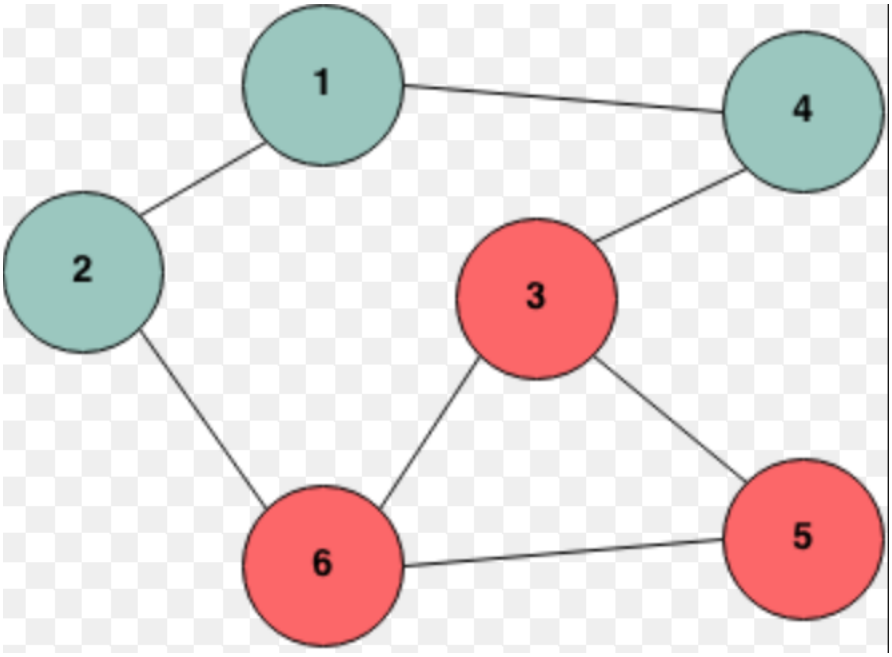
1. component 1 contains 3 vertices connected with 3 edges (in a triangle)
2. component 2 contains 3 vertices connected with 3 edges (in a triangle)
3. component 3 contains 2 vertices with an edge connecting them
4. component 4 contains one vertex with no edges
5. component 5 contains one vertex with no edges

In this case, select k = 2. After DFS terminates, the three resultant dictionaries will be $\{v_0 : None, v_1 : v_0, v_2 : v_1\}$, $\{v_3 : None, v_4 : v_3, v_5 : v_4\}$, and $\{v_6 : None, v_7 : v_6\}$. The largest two dictionaries are dictionary 1 and dictionary 2. Therefore the first edge would be connected between two nodes in each of the first two dictionaries. Now, the new dictionary representing that connected component would be the largest, and the second edge would connect that subgraph with the next largest connected component.

Proof of Correctness: Similar to the analysis in part A, when connecting two connected components, there are $x * y$ new reachable pairs of cities where x and y are the number of nodes in each respective subgraph. Multiplying the largest 2 numbers has the largest product, therefore the largest two subgraphs should be connected by an edge. When k edges are used, there are $x*y+y*z+x*z+...$ new pairs of reachable cities where the number of variables = k and there are $k + (k - 1) + k(-2) + ...$ multiplications that are then summed.

Problem 3-3. First run DFS in order to assign tree edges and 1 back edge in the graph. Because there is one cycle then there must be one back edge. When DFS tries to explore a node that has already been visited, then it has completed the cycle and the edge previous can simply be set as

the back edge. Then, run BFS twice. The first time BFS should be run on the normal graph. Store the cumulative weight of the shortest path that BFS returns as well as the path. Then run BFS on the graph, but first remove the assigned back edge. Compare the cumulative weight of the shortest path that this BFS iteration produces and return the path with lesser weight.
For BFS on the graph without the back edge: In a weighted undirected tree $G = (V, E, w)$, BFS from a vertex s finds single-source shortest paths from s (via parents pointers) in $O(V + E)$ time because in a tree, there is only one path between two vertices, and BFS finds it. BFS on the graph with the back edge may produce the correct shortest path depending on where s and t are relative to the cycle, which is why the edge needs to be included in at least one run of BFS. As seen below, if vertex 6 is s and vertex 3 is t, then if the edge between 3 and 6 is designated as the back edge, then BFS without the back edge would result in a length of 2 (this doesn't factor in weights but for a general idea) vs. BFS including the edge results in a path of length 1.



If there is exactly one cycle, then there are at most V edges because a tree of V vertices has $V - 1$ edges, and you only need to add one edge to make one cycle in the graph. Therefore the run times of DFS and BFS become $O(V + E) = O(V + V) = O(2V) = O(V)$.

Problem 3-4. Submit this to gradescope.

(a)

```
typedef struct {
    /* Cached hash code of me_key. */
    Py_hash_t me_hash;
    PyObject *me_key;
    PyObject *me_value; /* This field is only meaningful for
        combined tables */
```

```
} PyDictKeyEntry;
```

As seen above, the dictionary entry has two pointers, represented by the two '*', which are each 64-bits in a 64-bit system. Therefore, 64/8 = 8 so each pointer is 8 bytes. The Py_hash_t type is usually 8 bytes as well, so in total $8 + 8 + 8 = 24$ bytes.

(b) A Python dictionary size increases when $n/m = \alpha$. The table size increases by approximately double when this happens as shown below.

```
import sys

dict = {}
for i in range(50):
  dict[i] = i
  print(sys.getsizeof(dict))
```

The corresponding output is:

288 288 288 288 288 480 480 480 480 480 480 864 864 864 864 864 864 864 864 864 1632 1632 1632 1632 1632 1632 1632 1632 1632 1632 1632 1632 1632 1632 1632 1632 1632 1632 1632 1632 3168 3168 3168 3168 3168 3168 3168

(c) The load factor of a python dictionary is $2/3$.
Using a smaller example of the code from part b:

```
import sys

dict = {}
for i in range(10):
    dict[i] = i
    print(i, sys.getsizeof(dict))
```

The corresponding output is:

```
0 288
1 288
2 288
3 288
4 288
5 480
6 480
7 480
8 480
9 480
```

We see that when the number of entries approaches 2/3, the table roughly doubles. In

this example, it doubled for the sixth entry and $6/9 = 2/3$.

(d) It takes longer to insert keys when the load factor is reached because the probability of a collision increases when there are more elements hashed to the same table, so the table size has to increase to lower the probability of collisions.

```
import time

d1 = dict()
d2 = dict()

for i in range(1, 2796205):
  d1[i] = 0
  d2[i] = 0


t = time.time()
for i in range(2796205, 5592406):
  d1[i] = 0
t1 = time.time() - t


t = time.time()
for i in range(2796205, 5592407):
  d2[i] = 0
t2 = time.time() - t

print(t2 - t1)
```

The corresponding output is below.

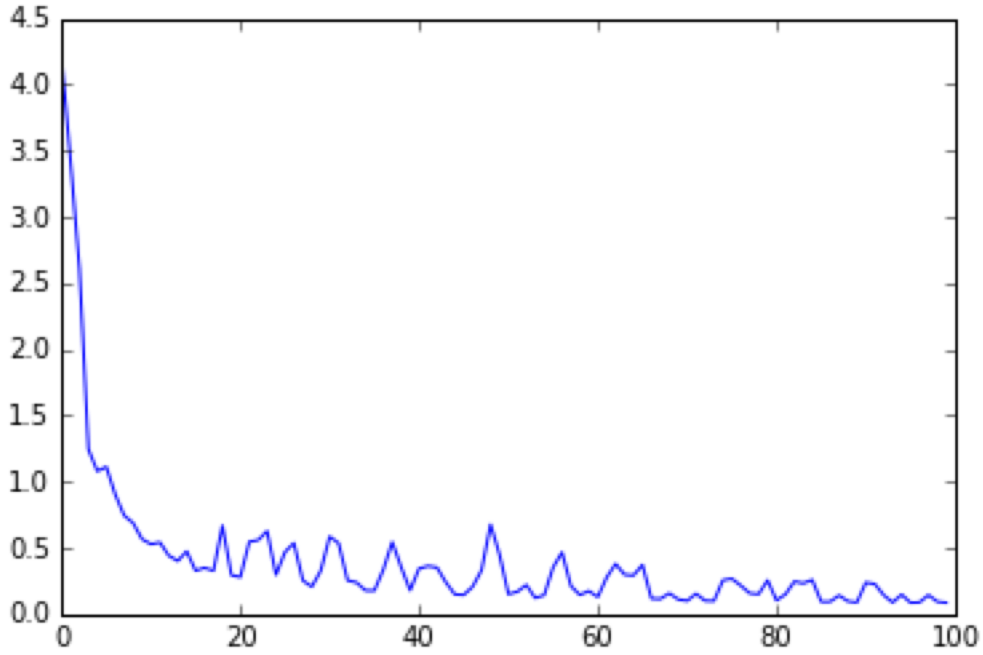
```
0.5477070808410645
```

As seen above, the difference in hashing one additional value is a full half second (I used large numbers here to accentuate the time difference).

(e) When there are hash collisions, Python uses open addressing to hash the object to a different bin. With the given hash function, a greater n means less tiem to insert because there is a lower probability for collisions. Open addressing is easier with a larger dictionary because there are more potential bins for the object to be placed in. In the code provide for this part, everything tries to hash to the same bin.

```
import matplotlib.pyplot as plt
import random
import time
```

```
class C(object):
   def __init__(self, n):
      self.hash = random.randint(0,n)
   def __hash__(self):
      return self.hash

def insert(n, num):
   d = dict()
   start = time.time()
   for i in range(num):
      x = C(n)
      if x in d:
         d[x].append(random.random())
      else:
         d[x] = [random.random()]
   end = time.time()
   return end-start

def plot(n_array, num):
   times = []
   for n in n_array:
      times.append(insert(n, num))
   plt.plot(n_array, times)

plot(list(range(0,100)), 10000)
```



This shows that as n increases, the time to insert declines, which, again, is because collisions are less likely.

(f) The hash function included is problematic because the hash function uses the random module meaning that it violates the principle that when you hash the same value, it should return the same output. Therefore you won't be able to search for a value once you have hashed it.

```
import random
class D(object):
   def __init__(self, n):
      self.n = n
   def __hash__(self):
      return random.randint(0, self.n)
d = {}
x = D(100)
d[x] = 0
if x in d:
   print(True)
else:
   print(False)
```

Output:

```
False
```

Another example:

```
import random
class D(object):
   def __init__(self, n):
      self.n = n
   def __hash__(self):
      return random.randint(0, self.n)
d = {}
x = D(100)
d[x] = 0
del d[x]
```

Output:

```
Traceback (most recent call last):
  File "/Users/carolinepech/Documents/6.006 q4.py", line 44,
     in <module>
    del d[x]
KeyError: <__main__.D object at 0x10558bc50>
```

These two examples show why this hash function is improper because you can't find something that you have hashed and, similarly, if you try to delete something that you have inserted, it throws an error.

(g) The best way to use sets to get the best of both worlds of adjacency lists and adjacency matrices is to use a dictionary where the keys are the nodes and the values are sets of neighbors to that node. For this structure, insert, find, and delete all take $O(1)$. Checking if 2 nodes are neighbors takes $O(1)$ and returning a list of neighbors is also $O(1)$. This data structure can do all of these operations in $O(1)$ because dictionaries and sets have insert, find, and delete in $O(1)$ amortized. These operations are faster than for adjacency lists and adjacency matrices because they have to search throug $O(V)$ or $O(\text{neighbors})$ when searching and comparing. Adjacency matrices have $O(1)$ look up time, but this structure also searches in constant time.

Table 1: Analysis of Run Time

Audio Name	Insert	Find	Delete	Check if Neighbors	List Neighbors
$\{node : setofneighbors\}$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

(h) Tuples are not always hashable in Python, because tuples may hold references to unhashable objects such as lists or dicts. A tuple is only hashable if all of its items are hashable as well. Arrays are never hashable. This is because an object must be immutable in order to hash it so that you can search for it. For example, if you were able

to hash an array, but then you changed that array after hashing it, you would no longer be able to find that array.

```
d = {}
immutable_tuple = ('a', 'b')
d[immutable_tuple] = 1
print(d)

mutable_tuple = ('a', [1,2])
d[mutable_tuple] = 2
print(d)

l = [1,2,3]
d[l] = 3
print(d)
```

Output:

```
{('a', 'b'): 1}
Traceback (most recent call last):
  File "/Users/carolinepech/Documents/6.006 q4.py", line 38,
     in <module>
    d[mutable_tuple] = 2
TypeError: unhashable type: 'list'

Traceback (most recent call last):
  File "/Users/carolinepech/Documents/6.006 q4.py", line 39,
     in <module>
    d[l] = 3
TypeError: unhashable type: 'list'
```

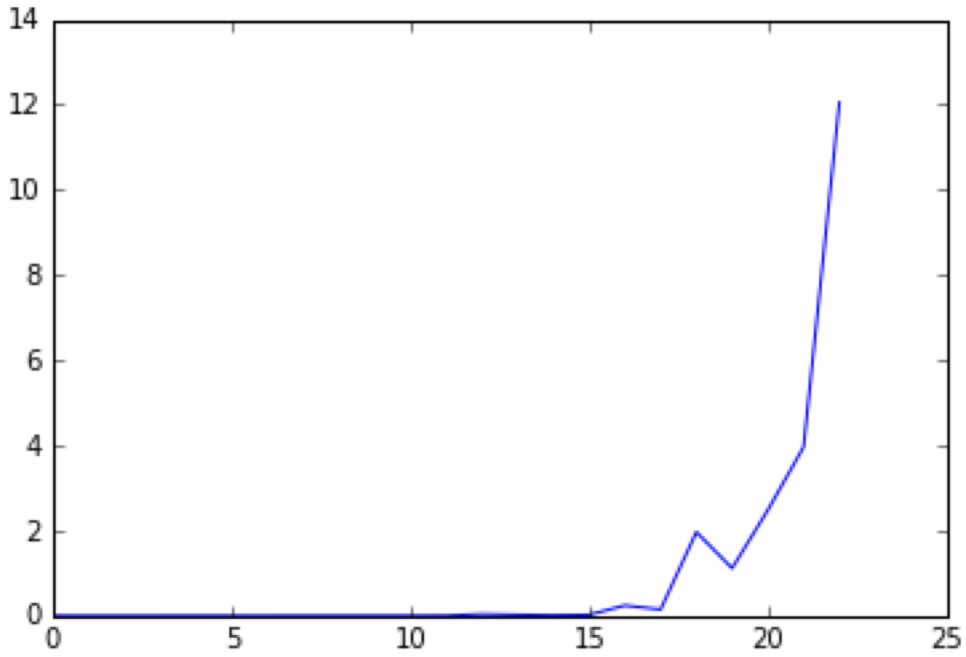
(i) As the value of d increases, it becomes harder to find a collision, which is why in the graph below, it takes a longer period of time to find 2 randomly generated strings with the same remainder. The value of d where it takes too long to find a collision is 24, but each time it produces a slightly different result.

Input:

```
import matplotlib.pyplot as plt
import random
import time

def find_collision_time(d):
    collision = False
    rand_str = random.randint(1000000000, 9999999999)
```

```
    start = time.time()
    while not collision:
        temp = random.randint(1000000000, 9999999999)
        if temp%2**d == rand_str%2**d:
            collision = True
    end = time.time()
    return end-start

def plot(d_array):
    collision = []
    for d in d_array:
        collision.append(find_collision_time(d))
    plt.plot(d_array, collision)

plot([range(23)])
```

Output (y-axis is time in seconds and x-axis is d)



(j) Part j

Part B

Problem 3-5. Submit your implementation on `alg.csail.mit.edu`.