Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Professors Constantinos Daskalakis, Piotr Indyk, and Steve Smith

March 16, 2017
Problem Set 3

Problem Set 3

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

Name: Faaya Abate Fulas

Collaborators: Ebenezer Nkwate, Yingni Hatty Wang

Part A

Problem 3-1. Submit this to gradescope.

(a)
- Modify BFS such that unvisited nodes make it into the queue iff their color is different from their parent node's color. This guarantees that when we trace back using parent pointers from t to s to find the shortest path, no two consecutive nodes will have the same color.
- Since this solution is still using the original BFS algorithm, the running time is $O(V + E)$

(b)
- Modify BFS such that each node that is visited is augmented with the color of the edge traversed to get to it. A node, x, on $level_i$ is assigned as a parent of a node, y, at $level_{i+1}$ iff the edge color of x is different from edge color of y. This guarantees that the modified BFS returns the shortest path where no two consecutive edges have the same color.
- Since, in the worst case, a single node will be visited k times, the running time of the modified BFS is $O(k(V + E))$

Problem 3-2. Submit this to gradescope.

(a)
- Run DFS such that every node in the graph is covered. Since DFS goes depth-first, it will cover all the connected nodes starting from the source node used.
- During each run of DFS, keep track of the visited nodes from that run in a list.
- Sort the lists returned from running DFS by size. This can be done in linear time ($O(V)$) using counting sort
- For $k = 1$, constructing an edge between a node in the list with the largest size and a node in the list with the second largest size will form a sub-graph with the largest number of connected nodes in the entire graph.

(b)
- For $k = 2$ edges, if an edge is drawn between a node in the list with the largest size and a node in the list with the second largest size, then a second edge is drawn between a node in the list with the second largest size and a node in the list with the third largest size, then the resulting sub-graph will contain the largest number of connected nodes in the entire graph.
- For any $0 < k \leq V$ edges, if an edge is drawn such that it forms a connection between nodes from the $k + 1$ lists (sorted in descending order by size), it will result in a sub-graph containing the maximum number of connected nodes in the entire graph.

Problem 3-3. Submit this to gradescope.

- Run DFS on the graph and remove the back-edge, $E(u, x)$ with weight w. Since the graph has only one cycle, it has one more edge than a tree would. Therefore, $E = V$ and DFS will take $O(V)$ time. The resulting graph is a tree, so $E = V - 1$ and there is exactly one path from s to t.
- Run BFS on the tree to find the path from s to t in $O(V + (V - 1)) = O(V)$ time and record the total cost of the path.
- Run BFS again on the tree to find the path from s to u and another BFS to find the path from x to t in $O(V)$ time. The total cost of the path from s to t via the back-edge, $E(u, x)$: $Cost(s, u) + Cost(x, t) + w$
- Compare the cost of the two paths, and choose the path with the lowest cost to find the shortest path between s and t.

Problem 3-4. Submit this to gradescope.

(a) A python dictionary entry has a hash value, a key pointer and a value pointer. In a 32 bit system, 4 bytes per pointer and 4 bytes for py_hash_t results in 12 bytes for a single python dictionary entry.The python documentation states that "Hash values are now values of a new type, py_hash_t , which is defined to be the same size as a pointer.", so I used 4 bytes instead of 8

(b) For sufficiently large number of insertions (to reduce the effect of the memory overhead) without deletions, a python dictionary doubles in size each time the load factor is reached.

(c) The load factor is $\approx 2/3$

(d) Yes. Since the table doubles when the load factor is reached, a new table of size $2m$ and a new hash function $h^{'}(k)$ has to be made, and every key in the previous table has to be rehashed into the new table before the key is inserted.

(e) As the value of n increases, the insertion time decreases. This happens because python uses open addressing and the rehashing frequency to handle collisions is lower for a larger n since the probability of collisions($\frac{1}{n}$) decreases as n increases.

(f)
- Since the hash function is not deterministic, it's not possible to find an object after insertion.
- Only integers can be hashed by the hash function. So, any other object would result in an error during insertion.

(g)
- Data Structure: A dictionary of all the nodes in the graph, with a set of all the node's neighbors as a value.
- To detect whether there is an edge between two nodes s and t, find s in the dictionary in $O(1)$ time and find t in the set of s's neighbors in $O(1)$ time.
- To find all of the neighbors of a node, find the node in the dictionary in $O(1)$ time. Its value pair is a set containing all of its neighbors.

(h) Tuples are hashable in python but arrays are not. This is because arrays are mutable, so if a change is made to the array once it has been inserted in a hash table, it can not be found.

(i) As d increases, it gets exponentially harder to find collisions. In a 32-bit system, for d$\geq$ 25, it takes longer to find a collision. The maximum time for finding collisions usually occurs at d=38, where the code is finding the collision when all the 32 bits of the hash values of the two strings match.

(j) From reading the source files provided in the pset:
- Every built-in python object has its own hash function defined within its source file.
- The default hash function for user-created objects is defined in pyhash.c and it maintains the invariant that if x = y, then hash(x)= hash(y)

```
#####################3.4b,c,d
dic= {}
init= sys.getsizeof(dic)
time_elapsed= []
for i in range(10**7):
    if sys.getsizeof(dic) != init: #detect when the table size changes
        size = sys.getsizeof(dic)
        diff = size - init
        percentage = diff/init # percentage of change from its previous size
        loadfactor = (i-1)/(init/12.0)  #number of keys in the dictionary/number of entries in the dictionary
        print (i-1, size, init, percentage, loadfactor ,time_elapsed[i-1], time_elapsed[i-2])
        init= size

    #time elapsed during insertion
    start = time.time()
    dic[i] = 1
    end= time.time()
    time_elapsed.append(end - start)

# the table doubles for a large number of insertions without deletions
#load factor for larger insertions is approximately 0.666
print (time_elapsed[2730] == max(time_elapsed[1300:3000])) # True(i= 2730 was inserted when the load factor was reached)
```

Figure 1: Python code used used to answer parts b-d

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

d = dict()
n_values= [10**i for i in range(10)]
insert_times= []
for n in n_values:
    start_time = time.time()
    for i in range(10000):
        x = C(n)
        d[x]= random.random()
    end_time = time.time()
    time_taken = end_time - start_time # time taken for 10000 insertions for some value of n
    insert_times.append(time_taken)

##print (insert_times)
plt.plot(np.log10(n_values), insert_times, 'ro')
plt.show()
```

Figure 2: Python code used to answer part e

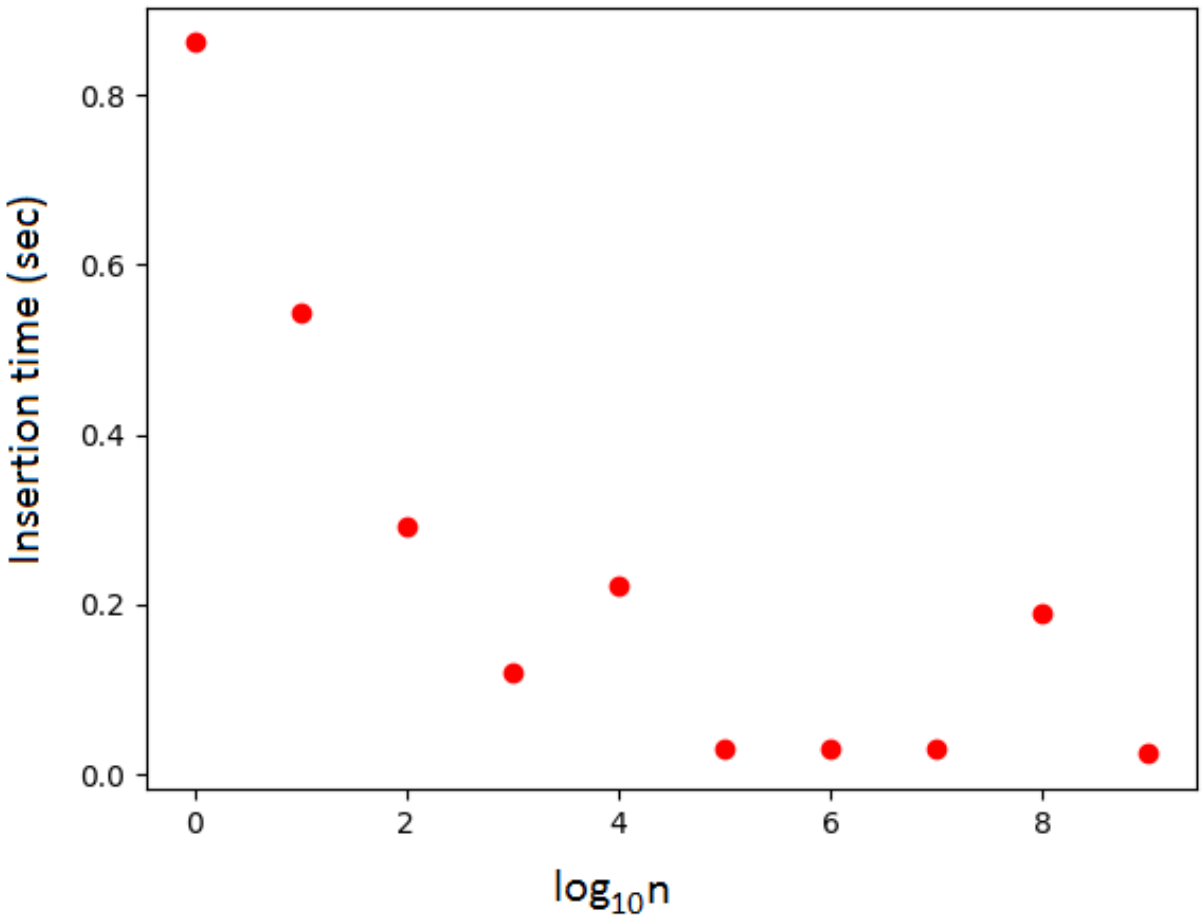

Figure 3: Graph to support answer for part e

```
###############3.4f
e={}
class D(object):
    def __init__(self, n):
        self.n = n
    def __hash__(self):
        return random.randint(0, self.n)

###inserting a string
e[D("3")]= 1

###finding an object after insertion
x= D(1000)
e[x] = 1

print (e[x])
```

Figure 4: Code to support answer for part f

```
def randomword(length):
   return ''.join(random.choice(string.ascii_lowercase) for i in range(length))

d_vals= [i for i in range(1,40)]
collisions= []
time_max, d_max =0,0
for d in d_vals:
    encountered= {}
    start_time = time.time()
    while True:
        str1= randomword(10)
##        remainder = hash(str1)% (2**d)
        remainder= bin(hash(str1))[-d:]
        if (remainder in encountered) and (str1 != encountered[remainder]):
            end_time= time.time()
            if time_max < end_time-start_time:
                d_max= d
                time_max= end_time-start_time
            collisions.append((str1, encountered[remainder], end_time-start_time))
            break
        encountered[remainder]=str1

print(d_max, time_max )

plt.plot(d_vals, [c[2] for c in collisions], 'ro')
plt.show()
```

Figure 5: Code used to answer part i

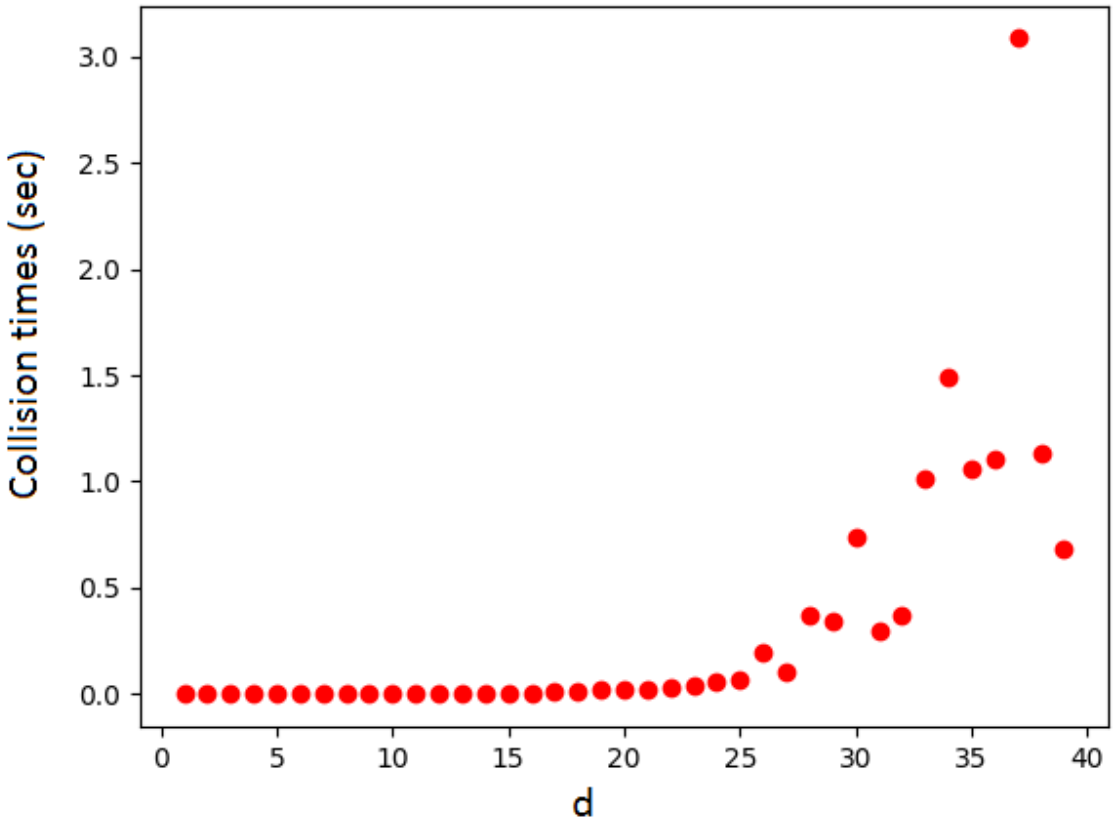

Figure 6: Graph to support answer for part i