Homework 11

Tushar Jain, N12753339
CSCI-GA 1170-003 - Fundamental Algorithms

November 30, 2017

Problem 1.a. Assume that all edge weights of an undirected graph G are equal to the same number w. Design the fastest algorithm you can to compute the MST of G. Argue the correctness of the algorithm and state its run-time. Is it faster than the standard $O(E + V \log V)$ run-time of Prim?

Solution.

If our graph is unweighted, or equivalently, all edges have the same weight, then any spanning tree is a minimum spanning tree. Therefore, we can use a BFS (or even DFS) to find such a tree in time linear in the number of edges.

```
def mst_via_bfs(graph, start):
    queue = [start]
    mst = {start:[]}

    while queue:
        node = queue.pop(0)
        neighbours = graph[node]
        for neighbour in neighbours:
            if neighbour not in mst.keys():
                queue.append(neighbour)

                insert_edge(mst, (node, neighbour))
                insert_edge(mst, (neighbour, node))

    return mst

def insert_edge(graph, edge):
    u, v = edge

    if graph.get(u):
        graph[u].append(v)
    else:
        graph[u] = [v]
```

Running Time:
Total running time is equivalent to that of BFS as we're just doing a BFS here, hence the running time is $O(V + E)$ which is less than that of Prim's Algorithm.

□

Problem 1.b. Now assume the all the edge weights are equal to w, except for a single edge $e\prime = (u\prime, v\prime)$ whose weight is $w\prime$ (note, $w\prime$ might be either larger or smaller than w). Show how to modify your solution in part (a) to compute the MST of G. What is the running time of your algorithm and how does it compare to the run-time you obtained in part (a) (or standard Prim)?

Solution.

If the $w\prime$ is less than all other weights, we chose this edge as the first edge when finding MST, else we do a normal BST & include this edge only if we're not able to form an MST.

Pseudo-Code:

```
def mst_via_bfs2(graph, start):
    e = find_unique_edge(graph)
    queue = [start]
    mst = {start:[]}
    count = 1

    while queue:
        node = queue.pop(0)
        neighbours = graph[node]
        for neighbour in neighbours:
            if neighbour not in mst.keys() and neighbour not in e:
                queue.append(neighbour)

                insert_edge(mst, (node, neighbour))
                insert_edge(mst, (neighbour, node))
                count += 1

    # can it also be implemented (w/o count) as len(mst) != len(G)?
    if count != len(G):
        return mst_via_bfs(graph, start) # from Q1a

    return mst

def insert_edge(graph, edge):
    u, v = edge

    if graph.get(u):
        graph[u].append(v)
    else:
        graph[u] = [v]

def find_unique_edge(graph):
    # weights is a dictionary on edges (tuple)
    from collections import Counter
    c = Counter(weights.values())

    for k,v in c.items():
        if v == 2: # since undirected => 2 edges
            unique = k
            break

    for k,v in weights.items():
        if unique == v:
            return k
```

Running Time:
Worst case would happen when there would be 2 BFS calls, therefore running time complexity is $O(V + E)$. □

Problem 2. Assume all edge weights of a connected undirected graph G are integers from 1 to W. Show how to modify Prims algorithm to achieve running time $O(E + VW)$. Hence, if $W = O(1)$, you get the optimal time $O(E + V)$.

Solution:

Since the weights are integers in a given range ($1 to W$), we can use a hashmap (dictionary in python) or buckets for each possible weight as key and the corresponding edges in an array/list. **Pseudo-Code:**

```
def mst_prim_modified(graph, start, weights ):
    n = len(graph)
    key = [sys.maxsize]*(n)
    p = [None]*(n)

    key[start] = 0

    import heapq

    q = heapq.heapify(G.v)

    while q:
        q_prime = lowest_bucket_with_root(q)
        u = heapq.heappop(q_prime)
        for v in G[u]:
            if v in q and weights[(u, v)] < key[v]:
                p[v] = u
                key[v] = weights[(u, v)]
```

Running Time:
heappop/extract $-$ *min* runs in $O(W)$ as we can find the first non-empty weight value & return the first element for that weight value. Also, $W = 1$ and decrease key would be in $O(1)$ as well since decreasing key would simply mean removing from one list and adding it another list corresponding with lower weight value. Therefore, total running time complexity is $O(E + VW)$.

□

Problem 3. To implement Kruskals algorithm, we need a disjoint-set data structure that can perform the operations MAKE-SET, FIND-SET and UNION. To get Kruskals algorithm to run in time $O(E \log V)$, we need this disjoint-set implementation to have the following property: any sequence of m operations runs in $O(m \log n)$ time, where n is the number of MAKE-SET operations. Describe and implement the disjoint-set data structure with the required running time, and prove its correctness and running time. **Hint:** if you take two trees such that each tree has at least 2^h nodes (where h is the height of the tree) and connect the root of the shallower tree to that of the deeper tree, then the resulting tree also has at least 2^h nodes (where h is the height of the resulting tree).

Solution:

To implement a disjoint-set forest with the union-by-rank and path-compression heuristic, we will keep track of ranks. In each node, we also maintain an integer value called rank, which would be an upper bound on the height of the node ,i.e., the number of edges in the longest path between the node itself and a descendant leaf.

Union By Rank:

- Rank of singleton sets created by MAKE-SET is 0.
- When taking union of 2 tress of equal rank, we arbitrarily choose one ot be the parent and increment its rank by one.

- When taking union of 2 trees of unequal rank, the tree with lower rank becomes the child, and the ranks are unchanged.

Path Compression:

- When running Find-set(x), we make all nodes on the path from x to the root direct children of the root.

Pseudo-Code:

```
p = {}
rank = {}

def make_set(x):
    p[x] = x
    rank[x] = 0

def union(x,y):
    px, py = find_set(x), find_set(y)

    if rank[px] > rank[py]:
        p[yp] = px
    else:
        p[px] = py
        if rank[px] == rank[py]:
            rank[py] += 1

def find_set(x):
    if x != p[x]:
        p[x] = find_set(p[x]) # path-compression
    return p[x]
```

Running Time:
$A_k(j)$ is a variation of **Ackerman's function**

$$A_k(j) = \begin{cases} j+1 \text{ if } k=0, \\ A_{k-1}^{(j+1)}(j) \text{ if } k \geq 1 \end{cases}$$

$A_4(1) = 16^{512}$ is much larger than the number of atoms in the universe(10^{80})!
The result is in terms of $\alpha(n)$, a single parameter inverse of $A_k(j)$ defined as the lowest k for which $A_k(1)$ is at least n:

$$\alpha(n) = mink : A_k(1) \geq n$$

$\alpha(n)$ grows every slowly, for all practical purposes we can treat $\alpha(n)$ as a constant. The running time is $O(m\alpha(n))$ for a sequence of m MAKE-SET, FIND-SET and UNION operations. □

Problem 4. Run the Bellman-Ford algorithm on the directed graph shown below, using vertex z as the source. In each pass, relax edges in this order: $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$ and show the d and π values after each pass. Now, change the weight of edge (z, x) to 4 and run the algorithm again, using s as the source.

Solution:
Using z as source with given relax edges order:
The d values are as follows:

s	t	x	y	z
∞	∞	∞	∞	0
2	∞	7	∞	0
2	5	7	9	0
2	5	6	9	0
2	4	6	9	0

And the π values are as follows:

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
z	NIL	z	NIL	NIL
z	x	z	s	NIL
z	x	y	s	NIL
z	x	y	s	NIL

Now, changing the weight if edge (z, x) to 4, using s as the source and using the same relax edge ordering: The d values are as follows:

s	t	x	y	z
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2
0	2	4	7	2
0	2	4	7	-2

And the π values are as follows:

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
NIL	s	NIL	s	NIL
NIL	s	y	s	t
NIL	x	y	s	t
NIL	x	y	s	t

$\square$