Diameter of Graph HW-6

October 29, 2024

1 Problem Statement

Given a WEIGHTED graph, return its diameter (the length of a largest shortest path) as well as all instances of shortest path attaining that diameter. So the input is a graph, from the structure used in class (nothing else), and the output is a pair: one integer indicating the diameter and a list of list of nodes, each representing a shortest path of diameter length - Test not only on small graphs, but on large random graphs

1.1 General Idea

- Could run Dijkstra or Bellman-Ford $|V|$ times.
- Could run Floyd-Warshall.
- Could run multi-commodity flow

1.2 Code

[1]:
```
def new_queue():
    return []

def empty_queue(q):
    return len(q)==0

def enqueue(q,e):
    return q.append(e)

def dequeue(q):
    return q.pop(0)
```

[8]:
```
# functions for min-heap

def newheap(n):
    return [0]+[0]*n

# Inserting element e into min-heap a at the end
def insert(a, e):
    a[0] = a[0] + 1
    a[a[0]] = e
    min_heap_fix_up(a,a[0])
```

```
# Fix up from position i to restore min-heap property of heap a
def min_heap_fix_up(a, i):
    while i > 1:
        p = i // 2
        # determines if heap is max-heap or min-heap
        #       \/
        if a[p] > a[i]:
            a[p],a[i] = a[i],a[p]
            i = p
        else:
            return

# remove the top element and fix the rest of the heap from that point
def extractsmallest(a):
    e,a[1],a[0] = a[1],a[a[0]],a[0]-1
    min_heap_fix_down(a,1)
    a[a[0]+1]=0
    return e

# starting from i, go down and fix the heap by swapping parent and child
def min_heap_fix_down(a, i):
    while 2*i <= a[0]:
        c = 2*i
        if c+1 <= a[0]:
            # switch to the smaller of the two children
            if a[c+1] < a[c]:
                c = c+1
        # if the child is smaller then swap with parent
        if a[i] > a[c]:
            a[i],a[c] = a[c],a[i]
            i = c
        else:
            return
```

[3]:
```
# for all graphs

def newgraph(v=None):
    return {v:{}} if v is not None else {}

def nodes(G):
    return list(G)

def nodecount(G):
    return len(G)

def add_nodes(G,nodes):
    for node in nodes:
        G[node]=set()
    return G


# for un-weighted graphs

# def add_neighbours(G,node,neighbours):
#     for v in neighbours:
#         G[node].add(v)
#         G[v].add(node)


# for weighted graphs

def addedge(G,e):
    (u,v,w) = e
    addarc(G,(u,v,w))
    addarc(G,(v,u,w))

def addarc(G,e):
    (u,v,w) = e
    H = G.get(u,None)
    if H is None:
        G[u] = {v:w}
    else:
        G[u][v] = w
    H = G.get(v,None)
    if H is None:
        G[v]={}

def neighbors(G,u):
    a=[]
    for v,w in G[u].items():
        a.append((v,w))
    return a

def edges(G):
    all=[]
    for u in G.keys():
        for v,w in neighbors(G,u):
            if u<v:
                all.append((u,v,w))
    return all

def arcs(G):
    all=[]
    for u in G.keys():
        for v,w in neighbors(G,u):
            all.append((u,v,w))
    return all

def edge_p(G,u,v):
    return v in neighbours(G,u)

def nodeingraph_p(v,G): # True iff node is in the graph
    H = G.get(v,False)
    return H != False

def boundaryedge_p(u,v,G): # True iff u is in G and v is not
    H = nodeingraph_p(u,G)
    return H!=False and not nodeingraph_p(v,G)

def components(G):
    count = 0
    visited = [False]*len(nodes(G))
    for v in nodes(G):
        if not visited[v]:
            count = count + 1
            bfs(G,v,visited)
    return count
```

[4]:
```
def dijkstra(G,r):
    n = nodecount(G)
    d,p = [float('inf')]*n,[None]*n
    d[r]=0
    heap,T = newheap(n*n),newgraph(r)
    for (v,w) in neighbors(G,r):
        d[v],p[v] = w,r
        insert(heap,(r,v,w))
    while nodecount(T) < n:
        (u,v,w) = extractsmallest(heap)
        if boundaryedge_p(u,v,T):
            addarc(T,(u,v,w))
            for (t,w) in neighbors(G,v):
                if d[t] > d[v]+w:
                    d[t],p[t] = d[v]+w,v
                    insert(heap,(v,t,d[t]))
    return d,p,T
```

[5]:
```
def diameter(G):
    """Where G is a graph"""
    return d,list_of_paths
```

1.3 Tests

[9]:
```
G1 = newgraph()
add_nodes(G1,[1,2,3,4,5,6])
add_neighbours(G1,1,[2, 4, 5])
add_neighbours(G1,2,[3, 4])
add_neighbours(G1,3,[5])
add_neighbours(G1,4,[5])

G2 = newgraph()
add_nodes(G2,[0,1,2,3,4,5])
add_neighbours(G2,0,[1,2,3])
add_neighbours(G2,1,[2,3])
add_neighbours(G2,2,[3,4,5])
add_neighbours(G2,3,[4,5])
add_neighbours(G2,4,[5])

dijkstra(G1, 1)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[9], line 16
     13 add_neighbours(G2,3,[4,5])
     14 add_neighbours(G2,4,[5])
---> 16 dijkstra(G1, 1)

Cell In[4], line 6, in dijkstra(G, r)
      4 d[r]=0
      5 heap,T = newheap(n*n),newgraph(r)
----> 6 for (v,w) in neighbors(G,r):
      7     d[v],p[v] = w,r
      8     insert(heap,(r,v,w))

Cell In[3], line 41, in neighbors(G, u)
     39 def neighbors(G,u):
     40     a=[]
---> 41     for v,w in G[u].items():
     42         a.append((v,w))
     43     return a

AttributeError: 'set' object has no attribute 'items'
```

[10]:
```
import networkx as nx
import matplotlib.pyplot as plt

def visualize_graph(G):
    # Initialize a networkx graph
    nx_graph = nx.Graph()

    # Add edges to the networkx graph
    for u in G.keys():
        for v, w in G[u].items():
            if u < v:  # Avoid adding duplicate edges in an undirected graph
                nx_graph.add_edge(u, v, weight=w)

    # Draw the graph
    pos = nx.spring_layout(nx_graph)  # positions for all nodes
    weights = nx.get_edge_attributes(nx_graph, 'weight')

    # Draw nodes, edges, and labels
    nx.draw(nx_graph, pos, with_labels=True, node_color="lightblue",␣
↪node_size=500, font_size=10, font_weight="bold")
    nx.draw_networkx_edge_labels(nx_graph, pos, edge_labels=weights)

    # Display the plot
    plt.show()

# Example usage
G = newgraph()
add_nodes(G, [0, 1, 2, 3])
addedge(G, (0, 1, 5))
addedge(G, (1, 2, 3))
addedge(G, (2, 3, 1))
addedge(G, (3, 0, 2))

visualize_graph(G)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[10], line 28
     26 G = newgraph()
     27 add_nodes(G, [0, 1, 2, 3])
---> 28 addedge(G, (0, 1, 5))
     29 addedge(G, (1, 2, 3))
     30 addedge(G, (2, 3, 1))

Cell In[3], line 25, in addedge(G, e)
     23 def addedge(G,e):
     24     (u,v,w) = e
---> 25     addarc(G,(u,v,w))
     26     addarc(G,(v,u,w))

Cell In[3], line 34, in addarc(G, e)
```

```
     32     G[u] = {v:w}
     33 else:
---> 34     G[u][v] = w
     35 H = G.get(v,None)
     36 if H is None:

TypeError: 'set' object does not support item assignment
```

1.4 Proof of Correctness

1.5 Runtime