CS 514

Assignment 5 – Greedy Algorithms

report5.txt

Q1. Implement Kruskal's and Prim's algorithms for minimum spanning tree. Your algorithms take a weighted undirected graph as input and output a minimum spanning tree in the form of a set of edges and their total weight. Call these two functions MST_Kruskal and MST_Prim.

Answer-

Please find the implementation of Prim's and Kruskal's algorithms for minimum spanning tree

```
import queue

def MST_Prim(graph):
    min_spanning_tree = {}
    visited = set()
    priority_queue = queue.PriorityQueue()
    result_weight = 0
    result_list = []

    for x in graph:
        start, end, weight = x
        if start not in min_spanning_tree:
            min_spanning_tree[start] = list()
        if end not in min_spanning_tree:
            min_spanning_tree[end] = list()
        min_spanning_tree[start].append((start, end, weight))
        min_spanning_tree[end].append((end, start, weight))
    visited.add(0)
    for x in min_spanning_tree[0]:
        start, end, weight = x
        priority_queue.put((weight, start, end))
    while not priority_queue.empty():
        weight, start, end = priority_queue.get()
        if end not in visited:
            result_weight += weight
            result_list.append((min(start, end), max(start, end)))
            visited.add(end)
            for x in min_spanning_tree[end]:
                priority_queue.put((x[2], x[0], x[1]))

    return (result_weight, result_list)

def MST_Kruskal(graph):
    def find(parent, node):
        if parent[node] == node:
            return node
        return find(parent, parent[node])
```

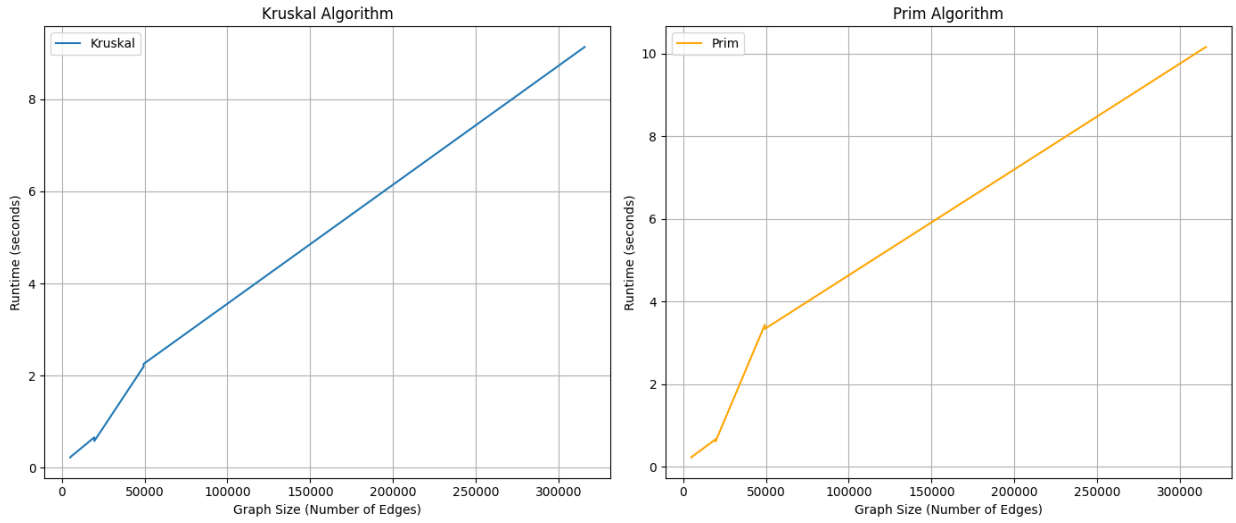
```
    def union(parent, rank, u, v):
        root_u = find(parent, u)
        root_v = find(parent, v)
        if rank[root_u] < rank[root_v]:
            parent[root_u] = root_v
        elif rank[root_u] > rank[root_v]:
            parent[root_v] = root_u
        else:
            parent[root_v] = root_u
            rank[root_u] += 1

    graph.sort(key=lambda x: x[2])

    min_spanning_tree = set()
    parent = {}
    rank = {}

    for u, v, weight in graph:
        if u not in parent:
            parent[u] = u
            rank[u] = 0
        if v not in parent:
            parent[v] = v
            rank[v] = 0

        if find(parent, u) != find(parent, v):
            min_spanning_tree.add((u, v, weight))
            union(parent, rank, u, v)

    total_weight = sum(weight for _, _, weight in min_spanning_tree)
    min_spanning_tree.discard((0, 0, 0))
    result_list = [(t[0], t[1]) for t in min_spanning_tree]

    return (total_weight, result_list)
```

Compare the running times of the two algorithms as a function of the graph sizes (number of edges). Show the running times a table and a plot in the report.

The running times in a table –

	Prim's		Kruskal's	
No. of edges	Min Weight	Time taken	Min Weight	Time taken
4994	6372	0.23031210899353027	6372	0.22024798393249512
4994	6171	0.23633790016174316	6171	0.22255301475524902
19607	1194	0.6677289009094238	1194	0.6562180519104004
19607	1171	0.6174440383911133	1171	0.5748019218444824
49370	2229	3.4393179416656494	2229	3.1927926540374756
49370	2201	3.341308832168579	2201	3.2518630027770996
140359	1499	6.687819004058838	1499	6.8356029987335205
140359	1499	6.7576799392700195	1499	7.047821044921875
315745	999	10.1645188331604	999	10.137946844100952
315745	999	10.86064887046814	999	11.20964503288269

The plot is as shown below –



We know the time complexity of Prim's algorithm which implements a priority queue has a time complexity of O(|E| +|V|log|V|) and tends to perform better for denser graphs. But as |E| have much large value compared to |V|, this graph looks linear – O(|E|). The Kruskal's algorithm is faster with sparse graph but loses speed when the density of graph increases and it's time complexity is O(|E|log|V|).

Q2. You are driving on a long highway with gas stations at distances *d1*< ...< *dn* miles from the starting location *S*. Your car can run *M* miles with a full gas tank. You start with a full gas tank and want to reach the final location which is at *dn* miles from *S*. How would you choose the gas stations to minimize the number of refueling stops? Argue that no other choice can make fewer stops.

Answer –

Solution - The problem of minimizing the number of refueling stops while driving on a long highway with gas stations at different distances can be solved using a greedy approach. The idea behind this approach is to always choose the gas station closest to the farthest point we can reach with our current fuel tank.

1. Starting with a full tank: As we are beginning our journey at point S with a full gas tank, ensures that we can cover the initial (first) part of the trip without any stops.

2. Selecting the farthest reachable gas station: At each step, we check to identify the gas station that is the farthest away from our current location (while being within our car's range, M miles). This choice ensures that we are utilizing our current tank of gas to the maximum, as it allows us to travel the longest distance before needing a refill.

3. Refueling at the selected station: Once we identify the farthest reachable gas station, we stop at that station to refill our tank. This ensures that we have a full tank for the next segment of your journey.

4. Iterative process: We continue this process of identifying the farthest reachable gas station and refueling there until we have covered the entire distance to our final destination, dn miles from the starting point.

In all, we compute the distance between the current station and the next station, and if it is larger than the distance we can travel with the current amount of fuel, we refuel at the current state. Otherwise, we do not stop at the current station. Finally, if the distance between two consecutive stations is larger than M, then it's invalid.

```
def minimizeRefuelingStops(distances, M):
    n = len(distances)  # Number of gas stations
    stops = []  # A list to store the selected gas stations
    current_station = 0  # (considering S as starting point 0-Start from the first station)
    last_refuel = 0  # last refueling station

    while current_station < n:
        # Find the farthest reachable gas station within the car's range
        while current_station < n and distances[current_station] - distances[last_refuel] <= M:
            current_station += 1

        # If we couldn't find a farther station within range, we must backtrack
        if current_station == last_refuel:
            return "Impossible to reach the destination
        # Add the last refueling station to the list of selected stations
        stops.append(distances[last_refuel])
        last_refuel = current_station - 1

    return stops
```

The runtime of this algorithm is Θ(n).

Proof of correctness of the algorithm -
Suppose there is a possible solution.
Let S be solution from your greedy algorithm and x be the station that you visit first in S.
Let S' be any list of stations that does not contain x and z be the first station in S'.
The greedy algorithm chooses the first station where
distances[next_station] − distances[current_station] as the maximum value and less than M. Therefore, we know that x > z.
Hence, len(S') = 1 + minimizeRefuelingStops (distances[z + 1...n]) and len(S) = 1 + minimizeRefuelingStops (distances[x + 1...n]), which shows that S is not worse than S'.
Therefore, the solution from the greedy algorithm is optimal.

We note that any sequence of stops that took less stops than the greedy algorithm would have to 'pass' the greedy algorithm at some point along the route.

Using induction, we can see that if the greedy algorithm is the farthest it can be after the first stop, and after the nth stop it is the farthest it could be given stop n - 1, then the greedy algorithm must be the farthest it can be for all stops along the route.

Q3. Let 'maximum spanning tree' be defined as a spanning tree with the maximum total weight. Define the *cut property* for maximum spanning tree as follows. Suppose *X* is a set of edges in a maximum spanning tree. Choose a set of vertices *S* such that no edges in *X* cross from nodes in *S* to

nodes in *V-S*. Let *e* be the heaviest edge not in *X* that crosses from *S* to *V-S*. Show that *X union {e}* is a subset of a maximum spanning tree.

The cut property of a maximum spanning tree states that a specific subset of edges, denoted as Y, within a maximum spanning tree always remains a part of any other maximum spanning tree.

Proof:
Let H be a graph, and Y represent a set of edges in a maximum spanning tree M of H. To show that Y ∪ {k} is a subset of a maximum spanning tree when selecting a set of nodes T such that no edges in Y cross from T to V-T and k is the heaviest edge not in Y that crosses from T to V-T, we proceed as follows:

1. Begin with the given maximum spanning tree M, with Y as a subset of its edges.
2. Select a set of nodes T such that Y respects the cut created by T.
3. Identify the heaviest edge k that crosses from T to V-T but is not part of Y. This implies that there is another edge l in Y that also crosses the same cut.
4. Remove edge l from M to form a spanning tree M - {l}.
5. Add edge k to M - {l} to create a new tree N = (M - {l}) union {k}.

To prove that N is also a maximum spanning tree:
- The weight of M, denoted as W(M), can be expressed as W(M) = W(M - {l}) + W(l), where W(l) ≤ the weight of the heaviest edge in Y.
- The weight of N is W(N) = W(M - {l}) + W(k), with W(k) ≥ the weight of the heaviest edge in Y.

Comparing the weights of M and N, we find:

W(N) = W(M - {l}) + W(k) = W(M) - W(l) + W(k) ≥ W(M)

This indicates that the weight of N is either greater or equal to the weight of M, confirming that N is also a maximum spanning tree. In conclusion, adding the heaviest edge k, which crosses the cut formed by T, to Y does not reduce the total weight of the spanning tree, verifying the cut property for maximum spanning trees.

Q4. A barber shop serves *n* customers in a queue. They have service times *t1, ..., tn*. Only one customer can be served at any time. The waiting time for any customer is the sum of the service times of all previous customers. How would you order the customers so that the total waiting time for all customers will be minimized? Carefully justify your answer.

There exists a greedy solution to solve this problem. To minimize the total waiting time for all the customers, we can serve the customers in an increasing order of their service time.

This minimizes the total waiting time for all customers in the queue because it serves the customers with the shortest service times first. By doing so, it allows customers with longer service times to start their service earlier and reduces the time they spend waiting.

Justification –

There are n customers with service times as t1,t2,…tn. Given that the waiting time is the sum of service times of all previous customers.

Consider customer i, where 1<=i <= n. The waiting time of this customer will be –

$W[i] = \Sigma_1^{i-1} (ti)$ #sum of service time of all the previous customers.

We carry out the task in the following steps -

1. Sort the n requests in non-decreasing order of t[i] values
2. Service the requests in this order.

Intuitive proof of correctness of algorithm:

Consider what happens when we decide which request to service, and there are more than one requests.

Consider any 2 such requests.

If you schedule the request with the larger t[i], it would mean that the other request would have to wait for that larger amount of time before being serviced, thus adding that large quantity to the total waiting time.

On the other hand, in case you service the smaller request first, the addition to the total waiting time would only be equal to the t[i] value of the smaller request. Hence, whenever you have a choice, go for the request with the smaller t[i] value.

Hence, this solution is the most optimal solution.