CS 599: Algorithms **(Due: November 10th, 2020)**

Submission Homework 6

Instructor: Prof. Prasad Tadepalli *Name:* Rashmi Jadhav, *Student ID:* 934-069-574

1: Implementations of Kruskal's and Prim's algorithms for minimum spanning tree (MST)

Kruskal's algorithm and Prim's algorithm for minimum spanning trees of weighted undirected graphs were implemented. To compare the results, random weighted undirected graphs with nodes 1100, 1200, ..., 2000 were generated as input to the two algorithms. The code snippets are as follows:

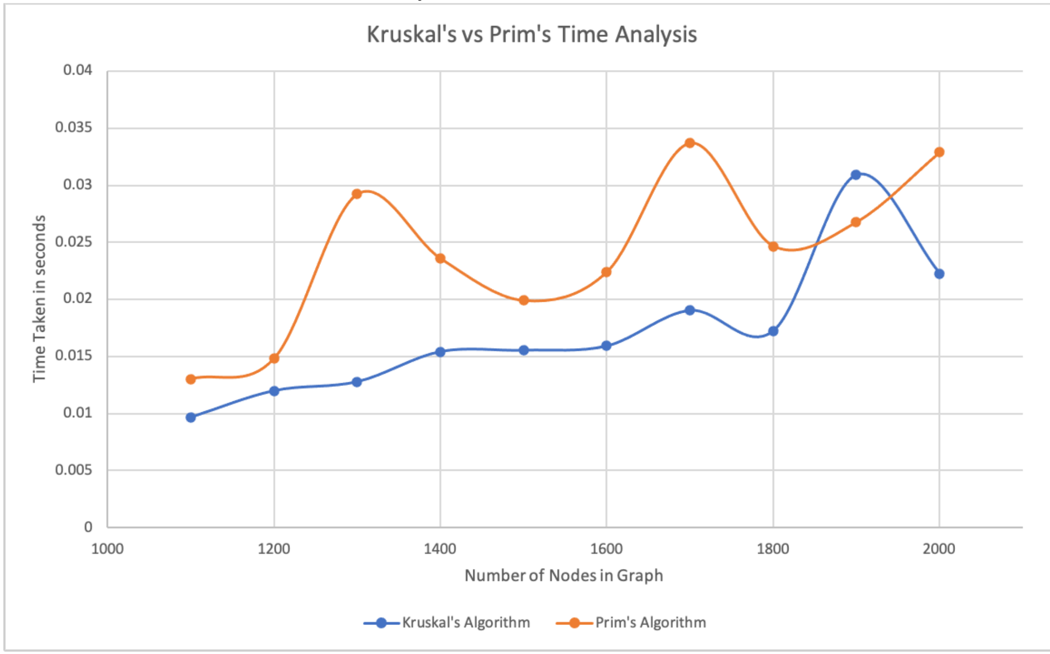
```
def kruskals_mst(G):
    start_time = time.time()
    min_spanning_tree = []
    mst_weight, index, mst_edges = 0, 0, 0

    G = sorted(G, key=lambda edge: edge[2])
    parent_list = []
    rank_list = []
    n_vertices_dict = find_total_vertices(G)

    for node in range(n_vertices_dict):
        rank_list.append(0)
        parent_list.append(node)

    while mst_edges < n_vertices_dict - 1:
        u, v, weight = G[index]
        index = index + 1
        u = find_root(parent_list, u)
        v = find_root(parent_list, v)
        if u != v:
            mst_edges = mst_edges + 1
            min_spanning_tree.append((u, v))
            mst_weight = mst_weight + weight
            union_by_rank(parent_list, rank_list, u, v)
    return mst_weight, min_spanning_tree, time.time() - start_time
```

```python
def prims_mst(G):
    start_time = time.time()
    dict_storage = defaultdict(list)

    init_values = G[0][0]
    source_values = G[0][0]
    for i, j, edge_weight in G:
        dict_storage[i].append((j, edge_weight))
        dict_storage[j].append((i, edge_weight))
    min_spanning_tree = []
    path_weight_from_vertex = {}
    heap_storage = [(0, init_values, source_values)]
    path_cost = 0
    is_edge_observed = False

    while heap_storage:
        weight, u, source_values = heapq.heappop(heap_storage)
        if u in path_weight_from_vertex:
            continue
        else:
            path_weight_from_vertex[u] = weight
            path_cost += weight
        if is_edge_observed:
            min_spanning_tree.append((source_values, u))
        is_edge_observed = True
        for pp, qq in dict_storage[u]:
            if pp not in path_weight_from_vertex:
                heapq.heappush(heap_storage, (qq, pp, u))
    return path_cost, min_spanning_tree, time.time() - start_time
```

The experimentation results for different inputs are as follows:

ALGORITHM	NODES IN GRAPH	EDGES IN MST	MST WEIGHT	TIME TAKEN (in s)
Kruskal's	1100	1099	93734	0.013353824615478516
Prim's	1100	1099	93734	0.016485214233398438
Kruskal's	1200	1199	105812	0.012279033660888672
Prim's	1200	1199	105812	0.01854085922241211
Kruskal's	1300	1299	110882	0.013502836227416992
Prim's	1300	1299	110882	0.027262210845947266
Kruskal's	1400	1399	117046	0.01696491241455078
Prim's	1400	1399	117046	0.024343013763427734
Kruskal's	1500	1499	121045	0.019934892654418945
Prim's	1500	1499	121045	0.03760528564453125
Kruskal's	1600	1599	130942	0.01913285255432129
Prim's	1600	1599	130942	0.024762868881225586
Kruskal's	1700	1699	136119	0.020281076431274414
Prim's	1700	1699	136119	0.03407478332519531
Kruskal's	1800	1799	148943	0.020554780960083008
Prim's	1800	1799	148943	0.02838611602783203
Kruskal's	1900	1899	153541	0.03091716766357422
Prim's	1900	1899	153541	0.029497146606445312
Kruskal's	2000	1999	164485	0.029284000396728516
Prim's	2000	1999	164485	0.03170275688171387

We can observe that both the algorithms give same weights of the minimum spanning trees on same input graphs. The algorithms were run for a 100 times to calculate the average time performances. Following is the chart that shows that Kruskal's is almost always better than Prim's.



2: Design for gas station problem

Start Point: S
Destination: T
Total gas stations between Start and Destination (including Destination): n
Gas Station distances from S: $d_1, d_2, d_3, ..., d_n$
Tank capacity: $M\ miles$
Problem: Minimize the number of refueling stops to reach the destination

Since the problem asks use to reach the destination from the start point, we assume that the gas station distances respect the vehicle's tank capacity. i.e., distance between two gas stations should not be greater than M miles, otherwise, we'd not reach the destination and get stuck at a station. Also, since we are not bothered about minimizing fuel costs, we fill the tank up to its capacity at each station.

This problem can be solved using a greedy approach such that we try to cover the maximum distance M as per the tanks capacity to eventually stop for a station. Let's say we start the journey and reach d_1, at this point, we update distance travelled to d_1 and we have used up $M - d_1$ miles from fuel. The algorithm will now get greedy and check next distance d_2. If $d_1 + d_2$ is still less than M miles, we update the distance travelled to $d_1 + d_2$ and greedily search for next d_i. However, if $d_1 + d_2 > M$, we'd travel only up to d_1 and fuel up to reach d_2 and again start a greedy hunt from the station where we filled gas.

```
dist_travelled = 0, n_stops_fueled = 1
for i in range 1 to n:
    if (d[i] + dist_travelled): ≤ M
        dist_travelled += d[i]
    else:
        dist_travelled = d[i]
        n_stops_fueled += 1
```

We observe that the time complexity of this greedy algorithm is $O(n)$. Breaking down the problem approach, we're essentially choosing a station to fuel only when the tank demands it. On the other hand, if we took any other approach by picking a station to fuel even if the tank can sustain going to the next station, we end up adding an extra fuel stop to our solution. Thus, choosing the stations by other policies would only lead to adding more stops to our overall solution. Hence, we can say that the greedy choice makes sure to defeat all other choices of fuel stops as they become suboptimal.

3: Maximum spanning tree

X: set of edges in a maximum spanning tree
S: set of vertices such that no edges in X cross from nodes in S to nodes in $V - S$
e: the heaviest edge not in X that crosses from S to $V - S$
To prove: X union e is a subset of a maximum spanning tree

Case 1: If e is already a part of maximum spanning tree, X union e is a subset of the maximum spanning tree T.
Case 2: Assume that there is another edge e' that is in maximum spanning tree T whereas e doesn't belong to T. Now, since MaxST should be connected, if we add e to T, there would be two paths between any two vertices in T. This makes T cyclic. If we remove e', then $T' = T \textit{ union } e - e'$. This removes the cycle and T' becomes connected and acyclic.
We need to show that T' is a MaxST. Say, weight of $T' = W(T')$ and weight of $T = W(T)$.
Therefore, $W(T') = W(T) + W(e) - W(e')$.
Since we know that $W(e) \geq W(e')$, $W(e) - W(e') \geq 0$. Hence, $W(T') \geq W(T)$
Thus, T' is a maximum spanning tree and X union e is a subset of maximum spanning tree.

4: Design for babrber's shop problem

If we want to minimize the total waiting time for all the customers, the average waiting time of all customers should be minimal. To arrive to a solution, let's analyze some cases. Let's say there are only two customers with customer A having a shorter service time than customer B's service time. If we serve customer A before customer B, A has a waiting time of 0 as no one was served before them. Customer B gets a wait time of customer A's service time. This makes total wait time for all customers to be equivalent to service time of customer A. Now consider another case wherein we serve customer B before customer A. B has a wait time of 0 and A get's a wait time of B's service time. This makes total wait time for all customers equivalent to service time of customer B. Clearly overall wait time in previous case was lesser than the later. Thus, we observe that serving smaller service time customers gives us an edge.
Extending this idea to 3 customers, if we sort these customers in ascending order of their service times, we'll eventually make sure that on an average, the later customers would have to wait less than if we had served higher service time customers first or if we had served the customers in any other random order. Thus, we choose ascending sorted order of service times to minimize the overall wait time.