Homework Assignment Week 5: Greedy Algorithms

Hyuntaek Oh

ohhyun@oregonstate.edu

CS 514_400 Algorithm

Nov. 4, 2024

1. (1) Implement Kruskal's and Prim's algorithms for minimum spanning tree.

```python
def MST_Kruskal(graph):
    # Get the number of vertices
    n = 0
    for u, v, w in graph:
        n = max(n, u+1, v+1)

    # Sort edges into non-decreasing order by weight w
    graph.sort(key=lambda edge: edge[2])

    # Initialize disjoint set
    parent = [i for i in range(n)]
    rank = [0] * n

    # Find minimum spanning tree and minimum cost
    mst = []
    min_cost = 0
    for u, v, weight in graph:
        if find(parent, u) != find(parent, v):
            mst.append((u, v))
            min_cost += weight
            union(parent, rank, u, v)

    return (min_cost, mst)
```

```python
def union(parent, rank, u, v):
    rootU = find(parent, u)
    rootV = find(parent, v)

    # Compare the rank of two parents
    if rootU != rootV:
        if rank[rootU] > rank[rootV]:
            parent[rootV] = rootU
        elif rank[rootU] < rank[rootV]:
            parent[rootU] = rootV
        else:
            parent[rootV] = rootU
            rank[rootU] += 1

5 usages
def find(parent, u):
    # Find parent of a node
    if parent[u] != u:
        parent[u] = find(parent, parent[u])
    return parent[u]
```

Fig 1. Kruskal's algorithm

```python
def MST_Prim(edges):
    # Create adjacent graph
    graph = {}
    for u, v, weight in edges:
        if u not in graph:
            graph[u] = {}
        if v not in graph:
            graph[v] = {}
        graph[u][v], graph[v][u] = weight, weight

    # Initialization
    key = {node: float('inf') for node in graph}
    parent = {node: -1 for node in graph}
    start_node = edges[0][0]
    key[start_node] = 0
    min_queue = [(0, start_node)]
```

```python
while min_queue:
    _, u = heapq.heappop(min_queue)

    # Update to smaller weight based on adjacent nodes
    for v, weight in graph[u].items():
        if weight < key[v]:
            key[v] = weight
            parent[v] = u
            heapq.heappush(min_queue, __item: (key[v], v))

# Find MST and the sum of minimum weights
mst = []
min_cost = 0
for i in parent:
    if parent[i] != -1:
        mst.append((parent[i], i))
        min_cost += key[i]

return (min_cost, mst)
```

Fig 2. Prim's algorithm

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

In the Prim's algorithm, the construction of a graph takes $O(E)$ time. The algorithm is implemented with a priority queue (min-heap) to keep track of the minimum edge weights. The time complexity of min-heap in the Prim's algorithm is $O(ElogV)$. To check and update weights, there is for-loop that takes $O(E)$ time. Thus, the total time complexity of Prim's algorithm is:

$$O(E) + O(ElogV) + O(E) = O(ElogV)$$

Kruskal's algorithm sorts all edges first, then processes them in order of non-decreasing weight, meaning that the time complexity of sorting all edges is $O(ElogE)$. Furthermore, Union and Find operations work for aggregation of two spanning graphs and finding parent of a specific node. The time complexity of Union and Find operations can be $O(E)$ since path compression and union by rank. So, the total time complexity of Kruskal's algorithm is:

$$O(ElogE) + O(E) = O(ElogE)$$

Real time taken in each case

Number of edges:	Kruskal_sparse (s):	Prim_sparse (s):	Kruskal_dense (s):	Prim_dense (s):
10^3	0.001000	0.001000	0.001000	0.000000
10^4	0.011000	0.019026	0.005975	0.004999
10^5	0.209929	0.308615	0.086769	0.062985
10^6	2.676248	5.281449	1.511027	1.692153
10^7	33.747358	76.651901	20.626441	29.987371

Fig. 3 A table of running times of two different algorithms vs. number of edges

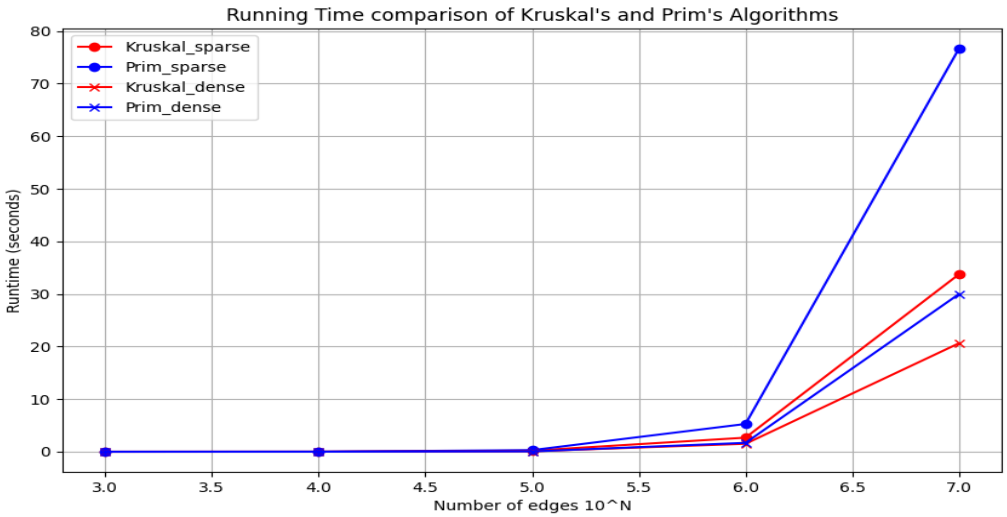


Fig. 4 A plot of two algorithms vs. number of edges

As shown in Fig. 3, the table shows the running times of two different algorithms, Kruskal and Prim, based on the number of edges. In the case of a sparse graph, the number of node is similar to the number of edges, Kruskal's algorithm is superior than Prim's algorithm. On the other hand, in the dense graph, the number of edges is equal to the squared number of nodes, the first three cases are Prim's algorithm is superior than Kruskal. The rest of the cases are different from my expectation since, in general, Prim's algorithm performs better than Kruskal.

Based on the theoretical time complexity, the times of Kruskal's algorithm from 10^3 to 10^7 are 9965.78, 132877.1, 1660640, 19931560, and 232434966 respectively, compared to the table above. Otherwise, the real times of Prim's algorithm from 10^3 to 10^7 have differences between sparse and dense graph since the number of node is based on the types of graph in the theoretical way. In the sparse case, the number of nodes is similar to the number of edges, and then the times of Prim's algorithm are similar to Kruskal's, meaning that 9965.78, 132877.1, 1660640, 19931560, and 232434966 respectively. On the other hand, in the dense case, the number of edges is equivalent to the squared number of nodes, meaning that 4954.19, 66438.56, 829920.80, 9965784.28, and 116261653.19.

As can be seen in Fig 4., the performances of two different algorithms in two different conditions show that Kruskal's algorithm is faster than Prim's algorithm. The increase in the number of edges affects the real time taken.

2. **You are driving on a long highway with gas stations at distances $d_1 < \ldots < d_n$ 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 d_n 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.**

To minimize the number of refueling stops, we need to choose the farthest gas stations within the restriction M miles (greedy selection). Let's assume that the number of stops is 6 ($n = 6$), including the final location G, and a car can run 100 miles with a full gas tank ($M = 100$). The distances of each stop $(d_1, d_2, \ldots, d_6)$ are 50, 90, 120, 160, 200, and 250 miles respectively. The figure below shows the results of how greedy choice can result in minimizing refueling stops.

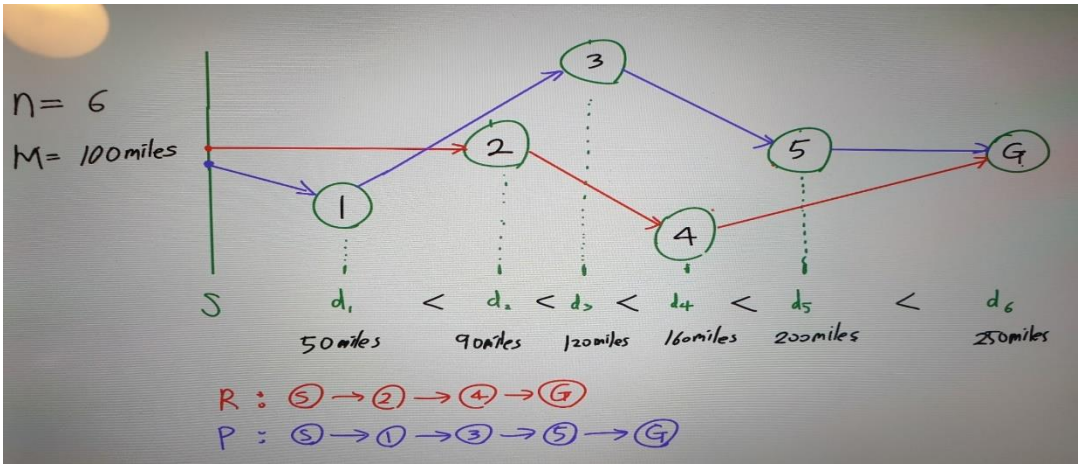


Fig. 5 An example of driving 100 miles to reach the final location G

As can be seen in Fig. 5, Red line chooses the farthest gas station from starting point S and chosen gas station to drive as far as possible in a greedy way. On the other hand, Purple line selects the non-maximum distance at the first step, and then chooses the farthest gas station in each step. In other words, at the first step, Red and Purple lines are different selection (always maximum and non-maximum), after that, both Red and Purple lines runs maximum distances.

From the Red line, the first selection to a gas station is $node\ 2$ since the maximum distance the Red line can run is 100 miles:

$$S + node\ 2 = 0 + 90\ (\leq 100\ miles) = 90$$

Then, the next selection would be $node\ 4$, $160\ miles$, since the maximum distance is 190 after refueling at $node\ 2$. So, this means:

$$(S + node2) + M = 90 + 100 = 190\ (\geq 160\ miles)$$

The last selection is to choose the final location G, which is $250\ miles$ since the car can run $290\ miles$. Thus, only two stops at the selected gas station $(S \rightarrow 2 \rightarrow 4 \rightarrow G)$ are needed to reach the destination. On the other hand, Purple line shows three stops at the chosen gas station $(S \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow G)$ since it selected one non-maximum distance at the first step, leading to one more stop. Both lines are similar strategy to find paths, while, at the first step, one non-greedy selection leads to non-greedy result, meaning that the more non-greedy selection occurs, the more stops a car needs to refuel the gas.

3. **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.**

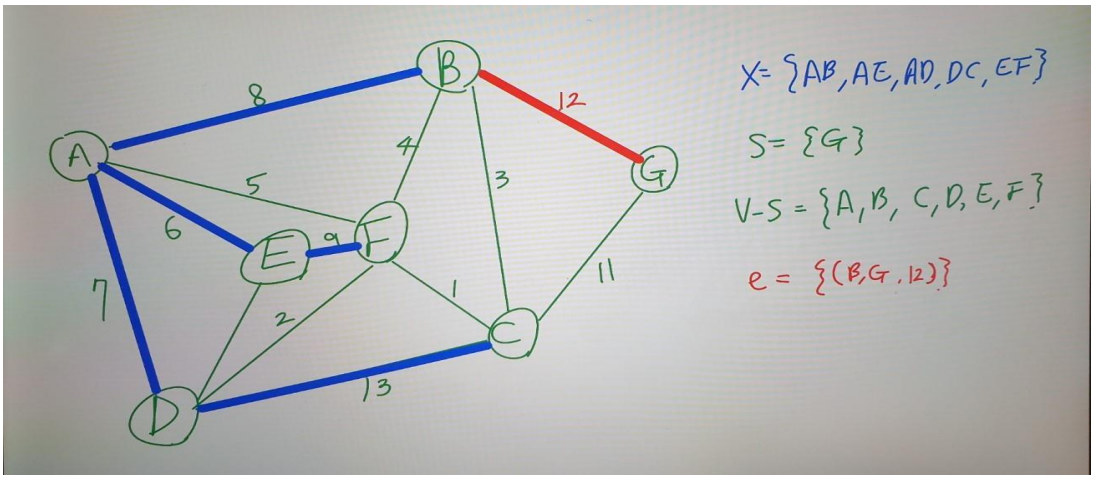


Fig. 6 Maximum Spanning Tree with the edge e

As shown in Fig. 6, a maximum spanning tree is based on a set of the edges in X, $\{(A, B, 8), (A, E, 6), (A, D, 7), (D, C, 13), (E, F, 9)\}$. The cut property for the maximum spanning tree is that if the vertices are partitioned into two sets S and $V - S$, and there is an edge e connecting these sets with the heaviest weight among all edges crossing the cut, then e should be a part of any maximum spanning tree, as it maximizes the overall weights of the spanning tree.

Let's assume that T is a maximum spanning tree that contains X but not e. Since e is the heaviest edge that crosses the cut between S and $V - S$, including e in T would increase the total weight of T without forming a cycle (no edges in X cross this cut). Therefore, by adding e to X, we can maximize the total weight while still maintaining the spanning tree properties, meaning that $X \cup \{e\}$ does not violate any properties of a maximum spanning tree and increases the total weight compared to any spanning tree not containing e.

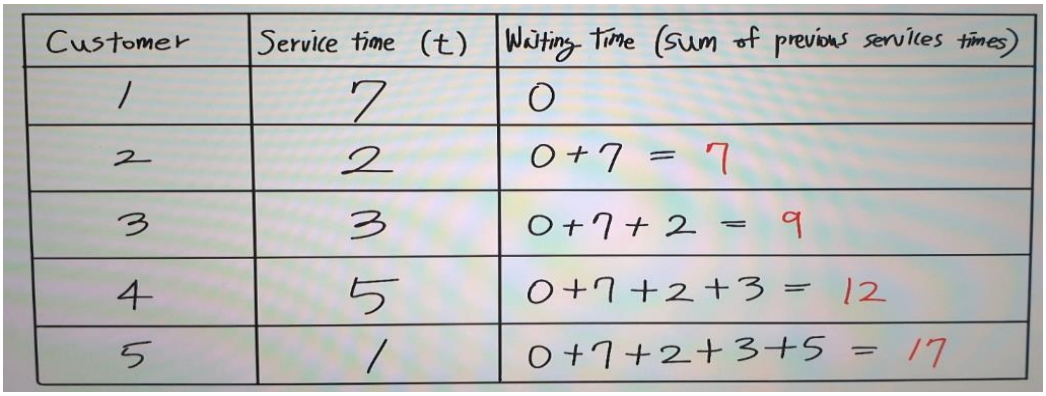
Thus, $X \cup \{e\}$ must be a subset of a maximum spanning tree, as including e satisfies the cut property by ensuring that the highest weight edge across the cut is part of the tree, preserving the requirement of maximum spanning tree.

4. **A barber shop serves n customers in a queue. They have service time $t_1, \ldots, t_n$. 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.**

To minimize the total waiting time for all customers, we should choose the customers with shorter service times. The waiting time for each customer depends on the sum of service times of previous customers. For example, the waiting time for the first

customer in line is 0, the second customer's waiting time is the service time of the first customer, $0 + t_1$. Then, the third customer's waiting time is the sum of the service times of the first and second customers, $0 + t_1 + t_2$, and so on. To achieve the minimum total waiting time, we should arrange customers in non-decrease order of their service times. When customers with shorter service times are served first, each subsequent customer's waiting time is minimized, as they wait only for the shorter service times that came before them.

Two Examples of calculating the total waiting time are below.

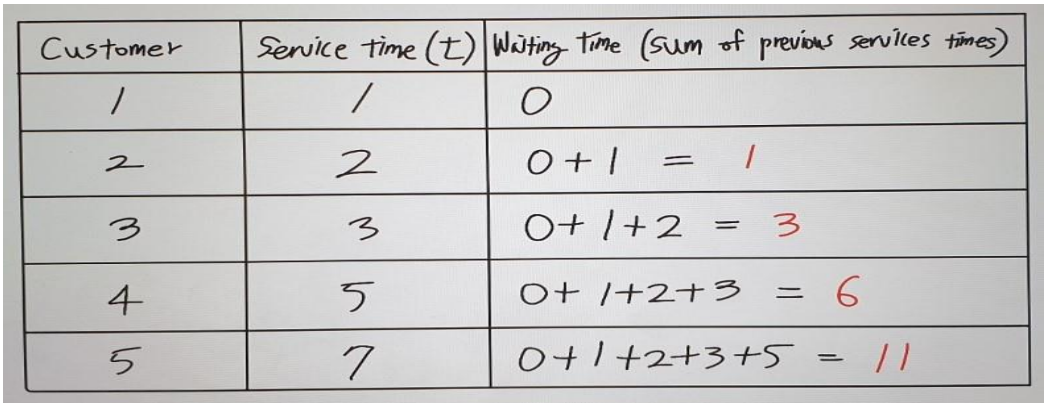


Customer	Service time (t)	Waiting time (sum of previous services times)
1	7	0
2	2	0+7 = 7
3	3	0+7+2 = 9
4	5	0+7+2+3 = 12
5	1	0+7+2+3+5 = 17

Fig. 7 A table of the total waiting time in random order

Fig. 7 shows the total waiting time of five customers in **random order**. The service time is $t_1 = 7,\ t_2 = 2,\ t_3 = 3,\ t_4 = 5,$ and $t_5 = 1$. The waiting time of the first customer is 0, and the second customer's waiting time is 7 since the second customer should wait until the first customer's service time. The third customer's waiting time is the sum of the first and second customers' service time, $0 + 7 + 2 = 9$, and so on. The total waiting time is 17:

$$Total\ waiting\ time_{random} = t_1 + t_2 + t_3 + t_4 = 7 + 2 + 3 + 5 = 17$$



Customer	Service time (t)	Waiting time (sum of previous services times)
1	1	0
2	2	0+1 = 1
3	3	0+1+2 = 3
4	5	0+1+2+3 = 6
5	7	0+1+2+3+5 = 11

Fig. 8 A table of the total waiting time in ascending order

On the other hand, Fig. 8 the total waiting time of five customers in **non-decrease**

order. The service time is $t_1 = 1,\ t_2 = 2,\ t_3 = 3,\ t_4 = 5,$ and $t_5 = 7$. The waiting time of the first customer is 0, and the second customer's waiting time is 1 since the second customer should wait until the first customer's service time. The third customer's waiting time is the sum of the first and second customers' service time, $3\ (0 + 1 + 2)$, and so on. The total waiting time is 11:

$$Total\ waiting\ time_{ascending} = t_1 + t_2 + t_3 + t_4 = 1 + 2 + 3 + 5 = 11$$

As a result, we know that the order of service time affects the total waiting time. If the largest number in the service times is the last order, the sum of the rest of service times would be the minimum. As can be seen in examples above, by arranging customers in ascending order of their service times, the total waiting time is minimized from 17 to 11 in this example.