# Greedy Technique

Chapter 9

# Greedy Technique

- It is considered as a general technique by the computer scientists though it is applicable only to optimization problems.
- **Optimization problem:**

- Objective function (either maximize or minimize)
- Constraints
- Feasible solutions and Optimal solution

- Ex:  Change making problem – greed on highest value coin
- It is a logical strategy of making a sequence of best choices among the currently available alternatives

# Greedy Approach

- It suggests constructing a solution **through a sequence of steps,** each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.
- On each step the **choice made** must be:

  \- **Feasible** (satisfy all the constraints)

  \- **Locally Optimal** (Best local choice available)

  \- **Irrevocable** (Once decision is made, it cannot be changed in subsequent steps)
- At each step we try to grab the best alternative based on **Greedy criteria.**

# Definition

- Given n points connect them in the cheapest possible way so that there **will be a path between every pair of points.**
- **Spanning Tree** is defined as a connected acyclic sub-graph of a connected graph that contains all the vertices of the graph.
- **Minimum Spanning Tree** of a weighted connected graph is its **spanning tree of the smallest weight** where the weight of a tree is defined as the sum the weights on all its edges.

# Prim’s Algorithm

- Construct a Minimum Cost Spanning Tree through a sequence of expanding subtrees
- Initial subtree contains a single vertex. (select arbitrarily)
- In each iteration include an edge with smallest weight to the subtree
- Stop the algorithm when all the graph vertices have been included to the subtree .

Note: There will be n-1 iterations

# Prim's Algorithm

Algorithm  Prim(G)

//Prim's algorithm for constructing a minimum spanning tree.

//Input: A weighted connected graph G =( V,E )

//Output: $E_T$ , Set of edges composing a minimum spanning tree of G.

$V_T \leftarrow \{V_0\}$

$E_T \leftarrow Ø$

for i ← 1 to |V | – 1 do

find minimum weight edge e*= (v*, u*) among all the edges (v,u) such that v is in $V_T$ and u is in $V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

return $E_T$

# Analysis

- It depends on the data structure used to represent the graph and the priority queue to represent $V - V_T$
- If the graph is represented as an adjacency matrix and the priority queue is represented as an unordered array the the time complexity is $\Theta(|V|^2)$. (on each of $|V| - 1$ iterations the array is traversed to find the minimum and update the necessary priorities for the remaining vertices)
- Priority Queue can be implemented as a **min heap.**
- If the graph is represented as an adjacency list, and the priority queue is implemented as min heap the the time complexity is $O(|E| \log |V|)$ (i.e., $|V| - 1$ deletions and $|E|$ verifications to change the element priorities in a minheap of size $|V|$

$$|V| - 1 + |E|\, O(\log |V|) = O(|E| \log |V|)).$$

# Kruskal’s Algorithm

- It constructs the Minimum spanning tree as an expanding sequence of sub-graphs which are always acyclic, but not necessarily connected.
- Sorts the edges in increasing order of their weights.
- Start with an empty sub-graph
- Scan the list and add an edge to the current sub-graph if it does not form a cycle , otherwise skip the edge

# Kruskal's Algorithm

Algorithm kruskal (G)

//Kruskal's Algorithm for constructing a minimum spanning tree.

//Input: weighted connected graph G =( V,E )

//Output: $E_t$ , set of edges in minimum spanning tree of G.

Sort E in non-decreasing order by weights $w(e_{i1}) <= \ldots\ldots\ldots <= w(e_{ik})$

$E_T \leftarrow Ø$

ecounter $\leftarrow 0$

$k \leftarrow 0$

while ecounter < |V | – 1 do

    k =k+1

    if $E_T \cup \{ e_{ik} \}$ is acyclic

        $E_T \leftarrow E_T \cup \{e_{ik}\}$

        ecounter = ecounter +1

return $E_T$

# Kruskal’s Algorithm

Different view of Kruskal’s algorithm

- Algorithm progresses through a series of forests containing all the vertices of the graph and few edges.
- Initial forest contains |V| trivial trees
- On each iteration algorithm takes the next edge (u,v) from the sorted list and finds the trees containing them.
- If they are not same unite them, otherwise skip the edge
- Final forest contains a single tree which is the minimum cost spanning tree.

# Disjoint subsets and Union-Find Algorithms

- Makeset(x) – creates a one element set {x}
- Find(x) – returns a subset containing x
- Union(x, y) – construsts the union of the disjoint subsets Sx and Sy, containing x and y.
- **Alternate procedures**
- quick-find and quick-union

# Dijkstra’s Algorithm

- Single Source Shortest Path Problem
- Input is a weighted Connected Graph with a Source vertex
- Find the Shortest path from this vertex to all the other vertices
- First find the shortest path to nearest vertex, next nearest vertex and so on

# Dijkstra's Algorithm

- Procedure:
- Label all vertices (nearest vertex, length)
- Select the nearest vertex u*
- Move u* from the set of fringes to the set of tree vertices
- For the remaining fringe vertices u that is connected to u*, if $\mathbf{d_{u*} + w(u*, u) < d_u}$ , update the label of u

# Dijkstra’s Algorithm

Algorithm   Dijkstra (G,S)

//Dijkstra’s Algorithm for single source shortest paths.

//Input: Weighted graph $G = (V, E)$ and  source vertex s.

//Output: The length of $d_v$ of a shortest path from s to v and its penultimate vertex $P_v$ for every vertex v in V

initialize (Q)

For every vertex v in V do

    $d_v \leftarrow$ œ; $P_v \leftarrow$ null

    Insert (Q, v, $d_v$)

$d_s \leftarrow 0$; decrease (Q, s, $d_s$); $V_T \leftarrow Ø$

for i ← 0 to |V| -1 do

    u* ← DeleteMin(Q)

    $V_T \leftarrow V_T$ U {u*}

    for every u in V - $V_T$ adjacent to u* do

        if $d_{u*} + w(u^*, u) < d_u$

            $d_u \leftarrow d_{u*} + w(u^*, u)$

            $P_u \leftarrow u^*$

            Decerase (Q, u, $d_u$)

# Analysis ( similar to Prim's Algorithm)

- It depends on the data structure used to represent the graph and the priority queue to represent $V - V_T$
- If the graph is represented as an adjacency matrix and the priority queue is represented as an unordered array the the time complexity is $\Theta(|V|^2)$. (on each of $|V| - 1$ iterations the array is traversed to find the minimum and update the necessary priorities for the remaining vertices)
- Priority Queue can be implemented as a **min heap.**
- If the graph is represented as an adjacency list, and the priority queue is implemented as min heap the the time complexity is $O(|E| \log |V|)$ (i.e., $|V| - 1$ deletions and $|E|$ verifications to change the element priorities in a minheap of size $|V|$

$$|V| - 1 + |E|\, O(\log |V|) = O(|E| \log |V|)).$$

# Huffman Trees

- Fixed length encoding (eg. ASCII Codes)
- Variable length encoding (processing is difficult)
- Prefix-free codes or prefix codes –
- In prefix codes no codeword is a prefix of a codeword of another character.

# Huffmann's Algorithm

- Initialize n one-node trees and label them with the character of the alphabet. Record the frequency of each character in its tree's root to indicate tree's **weight.** (sum of the frequencies in the tree's leaves)
- Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight, make them the left and right sub-tree of a new tree and record the sum of their weights in the root of the new tree as its weight
- The tree constructed by the above algorithm is called a **Huffmann Tree** and the code generated is called the **Huffmann Code**

- Thank You