Week 3 Greedy algorithms

> A greedy algorithms is an algorithm that follows the problem solving approach of making a locally optimal choice at each stage with the hope of finding a global optimum

Common Probelms can be solved using agreedy approach:

- Interval sheduling/partitioning
- Scheduling to minimize lateness
- **Shortest path**
- **Minimun spanning trees**

How to design greedy algorithm:

Step1: Understand problem

Step2: Start with simple algorithm

Step3: Test it correctness:

- Try some simple examples to get feel for algorithm
- If none of them break algorithm, see if there's underlying structural property we can use to prove correctness

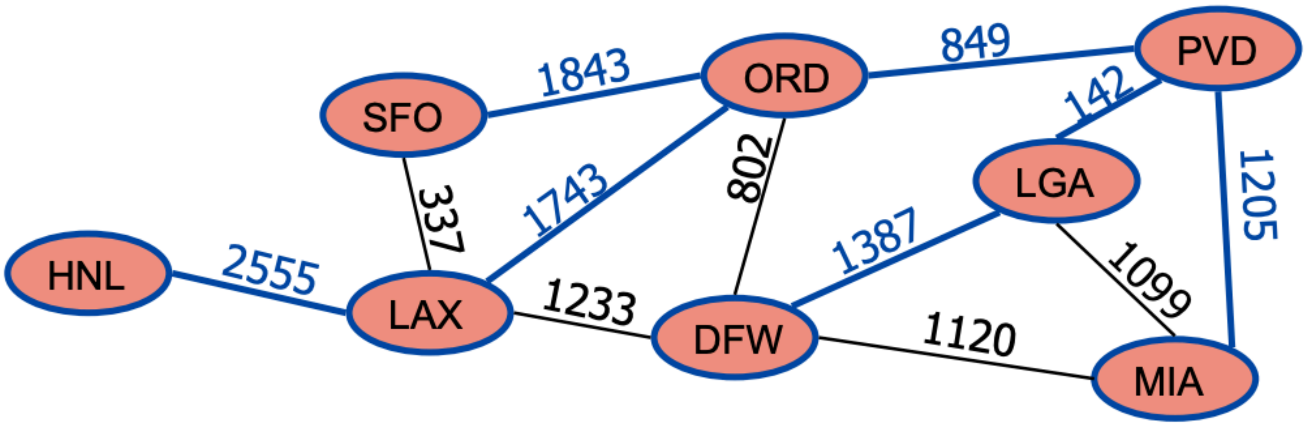
Step4: Better understanding algorithm, and improve

Shortest path

> Problems: Given an edge weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v, where the weight of a path is the sum of the weights of its edges.

> **Applications:** Internet packet routing, flight reservations and driving directions

```
Property:
1. A subpath of a shortest path is itself a shortest path
2. There is a tree of shortest paths from a start vertex to all the
other vertices (shortest path tree)
```



Dijkstra's Algorithm 🧐

Maintain a set of explored nodes S for which we have determined the shortest path distance $D(u)$from s to u. (s is the starting vertex)

- Input:
  - Graph $G = (V, E)$
  - Edges with non-negative weights
  - Start vertex s
- Output:
  - Distance from s to all $v \in V$
  - Shortest path tree rooted at s
- Initialize S = $\{s\}$, D(s) = 0, D[v] = infinite for all v (vertices) in $V - s$
- Repeatedly choose unexplored node v which minimizes:

  > $D[v]$ = min{ $D[u]$ + $w(l_e)$ }, for each $u \in S$,

  - l_e is a single edge (neighbor edge) connected between v and u
  - $w(l_e)$ — the weight of edge l_e
  - v can be any vertices not explored before (not in S)

Pseudocode

```
def Dijkstra(G = (V, E), w, s):
  # initialize algorithm
  D = [] # shortest Distance list from vertex s to other vertices
  parent = {} # Represent the rooted tree generated by Dijkstra
  for v in V do
    D[v] = sys.maxint #set D[v] to infinity
  D[s] = 0
```

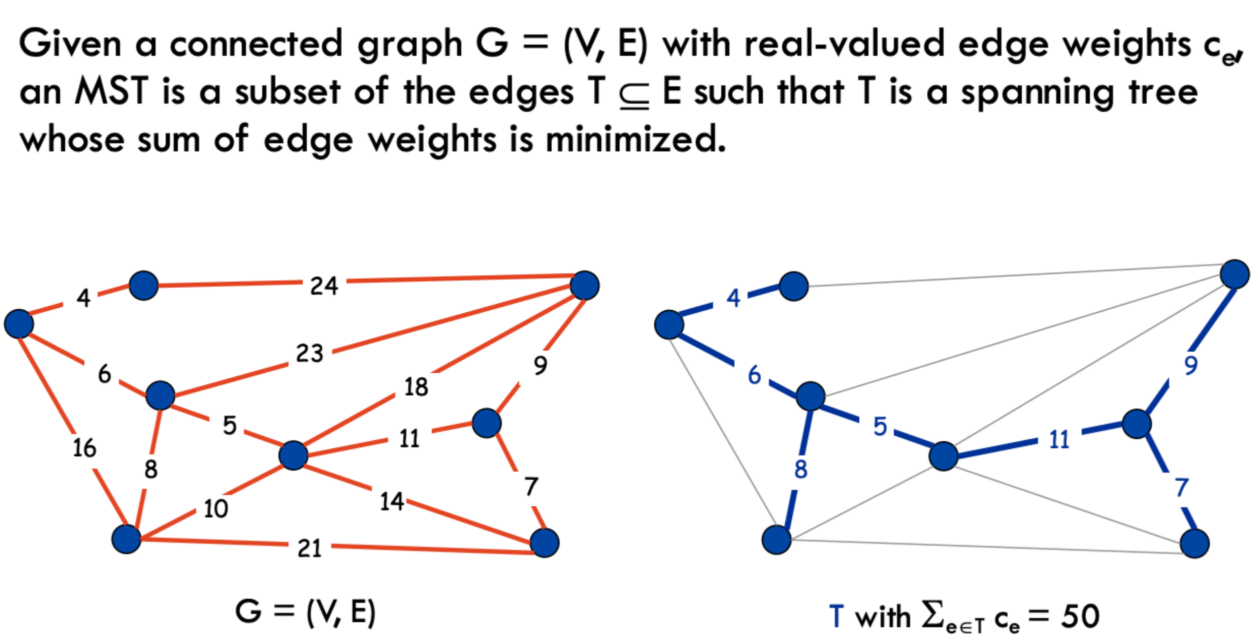
```
  Q <- new priority queue for { (v, D[v]) : v in V}

  #iteratively add vertices to S
  while Q is not empty do
    u <- Q.remove_min()
      for z in G.neighbors(u) do
        if (D[u] + w[u,z]) < D[z] then
          D[z] <- D[u] + w[u, z]
          Q.update_priority(z, D[z])
          parent[z] <- u # set the z.parent as vertex u
  return D, parent
```

Time Complexity

Assume the graph is connected: $m >= n-1$, the algorithm spends $O(m)$ time on everything except PQ operations

- But using normal **Heap** PQ, Disjkstar runs in $O(m * logn)$ time
  - Remove_min() in each operation cost $O(logn)$, and iterate m times
  - Using Fibonacci heaps, instead we get $O(m + nlogn)$

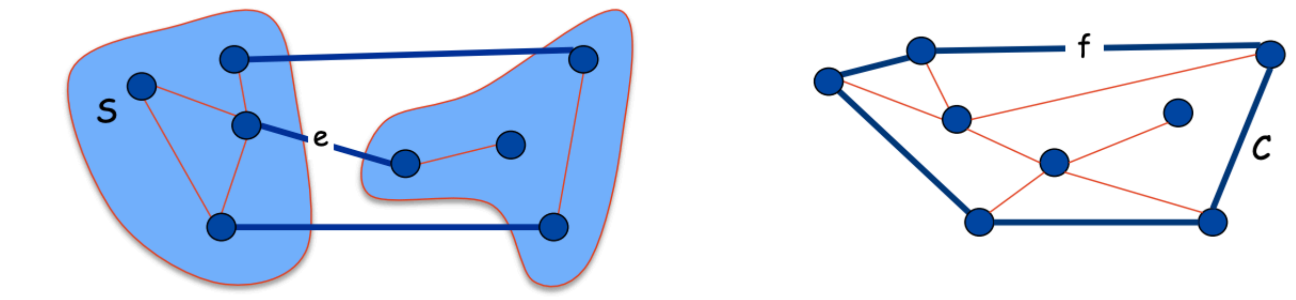
Minimum Spanning Tree



MST Properties

Assumption: All edge costs c_e are distinct

- ***Cut properties***: Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e
- **Cycle Property**: Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



e is in the MST

f is not in the MST

Terminology

Cutset — **A Cut is a subset of nodes S**. The corresponding **cutset D** is the subset of edges with exatly one endpoint in **S**.

Find the Minimum Spanning Tree (MST):

> Prim's Algorithm

> Cut property — Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e

So in Prim's Algorithm, everytime we add an edge, we follow cut property.

```
def prim(G = (V, E), c):
  u <- arbitrary vertex in V
  S <- { u } #let vertex u to be our strating vertex
  T <- NULL # That's our initlization of MST

  while len(S) < len(V) do
    # u in S and v not in S
    (u, v) <- min(S.cutset()) # min cost edge with exactly one endpoint in S
    add (u, v) to T # v never discover before, add it to T
    add v to S
  return T
```

Implementation

```
# Sample Graph Strature
class Vertex:
    def __init__(self, value):
        self.value = value;
        self.adjacent_edges = [];

    def getValue(self):
        return self.value;

    def __repr__(self):
        return self.value;

    def __str__(self):
        return self.value;

    def addEdge(self, edge):
        self.adjacent_edges.append(edge);

    def getAdjacentEdges(self):
        return self.adjacent_edges;

class Edge:
    def __init__(self, name, weight):
        self.name = name;
        self.weight = weight;
        self.neighborVertices = []

    def __str__(self):
        return self.name + " " + str(self.weight);

    def __repr__(self):
        return self.name + " " + str(self.weight);

    #  For compare x < y
    def __lt__(self, other):
        return self.weight < other.weight

    def addVertex(self, vertex: Vertex):
        if len(self.neighborVertices) == 2:
            print("Already connected with two vertices");
            return;
```

```
        self.neighborVertices.append(vertex);

    def get_Neighbor_Vertices(self):
        return self.neighborVertices;

    def get_weight(self):
        return self.weight;


class Adjacency_list_graph:
    def __init__(self):
        self.vertices = [];
        self.edges = []

    def addVertex(self, vertex: Vertex):
        self.vertices.append(vertex)

    def addVertices(self, list):
        self.vertices.extend(list);

    def addEdge(self, edge: Edge):
        self.edges.append(edge);
        for v in self.vertices:
            if v in edge.get_Neighbor_Vertices():
                v.addEdge(edge);

    def addEdges(self, edgeList):
        self.edges.extend(edgeList);
        for edge in edgeList:
            for v in self.vertices:
                if v in edge.get_Neighbor_Vertices():
                    v.addEdge(edge);

    def getVertices(self):
        return self.vertices;

    def getEdges(self):
        return self.edges;
```

```
from queue import PriorityQueue
```

```python
def prim(G: Adjacency_list_graph):
    V = G.getVertices()
    discovered_Edge = PriorityQueue()
    spanningTree = []
    S = [V[0]]
    for edge in V[0].getAdjacentEdges():
        discovered_Edge.put((edge.get_weight(), edge))

    while len(S) < len(V):
        # discovered_Edge.sort(key = sortWeight);
        print([edge for edge in discovered_Edge.queue]);

        weight, local_min_edge = discovered_Edge.get()
        end_point1 = local_min_edge.get_Neighbor_Vertices()[0]
        end_point2 = local_min_edge.get_Neighbor_Vertices()[1]
        print("Add light edge: {} -- {}".format(end_point1, end_point2));
        print("");
        spanningTree.append((end_point1, end_point2))

        if end_point1 not in S:
            nexVertex = end_point1;
        else:
            nexVertex = end_point2;

        S.append(nexVertex)
        for e in nexVertex.getAdjacentEdges():
            if (e.get_weight(), e) not in discovered_Edge.queue and e !=
local_min_edge:
                discovered_Edge.put((e.get_weight(), e))
    return spanningTree
```

Time Complexity — Similar analysis to Dijkstra's Algorithm

Using heap implementation: $O(mlogn)$

Using Fibonacci heap: O(m + nlogn)

Kruskal's Algorithm🌲🌲

Consider edges in ascending order of weight.

Case1: If adding e to T creates a cycle, discard a according to cycle property.

Case2: Otherwise, insert $e = (u, v)$ into T according to cut property where S = set of nodes in $u's$ connected component.

Kruskal's Algorithm: Time complexity

Sorting edges takes $O(mlogm)$ time

We need to be able to test if adding a new edge creates a cycle, in which case we skip the edge

- Option1: Run DFS in each iteration to see of the number of connect components stays the same. This leads to $O(m * n)$ time for the main loop

Union Find ADT

Union find — data structure defined on a ground set of elements A

Used to keep track of an evolving partition of A

Supported operations:

- $make\_sets(A)$ — make |A| singleton sets with elements in A
- $find(a)$ — returns an id for the set element a belongs to
- $union(a, b)$ — union the sets elements a and b belong to

Implmentation

```
def Kruskal(V, E, c):
  sort E in increasing c-value
  answear = []
  comp = make_sets(V)
  for (u, v) in E do
    # check if u and v are already connected in comp
    if comp.find(u) != comp.find(v) then
      answer.append((u,v))
      comp.union(u,v)
  return answer
```

Sets are represented with a lists. An array points to the set each element belongs to:

- $make\_sets(A)$ Create and initialzie the array, take $O(n)$
- $find(a)$ is a simple lookup in the array, take $O(1)$
- $union(a, b)$ Add elements in u's set to v's set, take $O(n)$

Kruskal's Algorithm would run in $O(n^2)$ time

However, in **Better union-find implementation**:

> Sets are represented with a lists. An array points to the set each element belongs to. Keep track of cardinality of each set. When taking the union of two sets change the smallest.

Kruskal’s algorithm would be **optimized in $O(mlogn)$ time**