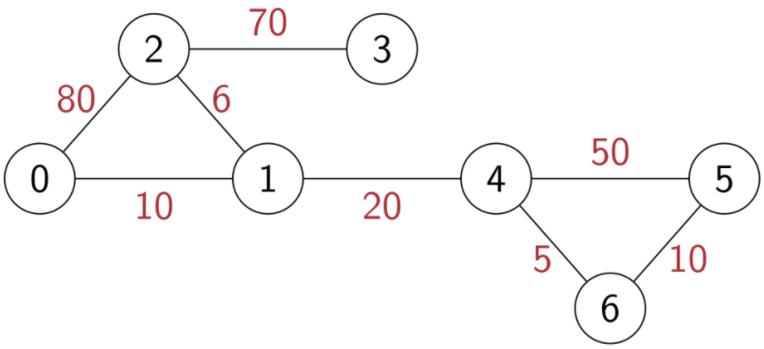
Shortest Paths in Weighted Graphs

Weighted Graphs

- Recall that BFS explores a graph level-by-level
- BFS computes the shortest path, in terms on number of edges, to every reachable vertex
- May assign values to edges
  - Cost, time, distance, ...
  - Weighted graph
- G = (V, E), W: E→R, where R represents the set of real numbers



- **Adjacency matrix**
  - Record weights along with edge information -- weight is always 0 if there is no edge

	0	1	2	3	4	5	6
0	(0,0)	(1,10)	(1,80)	(0,0)	(0,0)	(0,0)	(0,0)
1	(1,10)	(0,0)	(1,6)	(0,0)	(1,20)	(0,0)	(0,0)
2	(1,80)	(1,6)	(0,0)	(1,70)	(0,0)	(0,0)	(0,0)
3	(0,0)	(0,0)	(1,70)	(0,0)	(0,0)	(0,0)	(0,0)
4	(0,0)	(1,20)	(0,0)	(0,0)	(0,0)	(1,50)	(1,5)
5	(0,0)	(0,0)	(0,0)	(0,0)	(1,50)	(0,0)	(1,10)
6	(0,0)	(0,0)	(0,0)	(0,0)	(1,5)	(1,10)	(0,0)

- **Adjacency list**
  - Record weights along with edge information

0	[(1,10),(2,80)]
1	[(0,10),(2,6),(4,20)]
2	[(0,80),(1,6),(3,70)]
3	[(2,70)]
4	[(1,20),(5,50),(6,5)]
5	[(4,50),(6,10)]
6	[(4,5),(5,10)]

```python
#Weighted directed graph
#Adjacency matrix representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
size = 7
import numpy as np
W = np.zeros(shape=(size,size,2))
for (i,j,w) in dedges:
    W[i,j,0] = 1
    W[i,j,1] = w
print(W)

#Adjacency list representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
size = 7
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in dedges:
    WL[i].append((j,d))
print(WL)
```

```python
#Weighted undirected graph
#Adjacency matrix representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
edges = dedges + [(j,i,w) for (i,j,w) in dedges]
size = 7
import numpy as np
W = np.zeros(shape=(size,size,2))
for (i,j,w) in edges:
    W[i,j,0] = 1
    W[i,j,1] = w
print(W)

#Adjacency list representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
edges = dedges + [(j,i,w) for (i,j,w) in dedges]
size = 7
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in edges:
    WL[i].append((j,d))
print(WL)
```

Shortest paths in weighted graphs

- BFS computes shortest path, in terms of number of edges, to every reachable vertex
- In a weighted graph, add up the weights along the path
- Weighted shortest path need not have minimum number of edges
    - Shortest path from 0 to 2 is via 1

Single source shortest paths

- Find shortest paths from a fixed vertex to every other vertex
- Transport finished product from factory (single source) to all retail outlets
- Courier company delivers items from distribution centre (single source) to addresses

All pairs shortest paths

- Find shortest paths between every pair of vertices i and j
- Optimal airline, railway, road routes between cities

Negative edge weights

- Can negative edge weights be meaningful?
- Taxi driver trying to head home at the end of the day
    - Roads with few customers, drive empty (positive weight)
    - Roads with many customers, make profit (negative weight)
    - Find a route toward home that minimizes the cost

Negative cycles

- A negative cycle is one whose weight is negative
    - Sum of the weights of edges that make up the cycle

- By repeatedly traversing a negative cycle, total cost keeps on decreasing
- If a graph has a negative cycle, shortest paths are not defined
- Without negative cycles, we can compute shortest paths even if some weights are negative

Summary

- In a weighted graph, each edge has a cost
  - Entries in adjacency matrix capture edge weights
- Length of a path is the sum of the weights
  - Shortest path in a weighted graph need not be the minimum in terms of number of edges
- Different shortest path problems
  - Single source: four one designated vertex to all others
  - All-pairs: between every pair of vertices
- Negative edge weights
  - Should not have negative cycles
  - Without negative cycles, shortest paths still well defined

Dijkstra's Algorithm : Single Source Shortest Path

- Dijkstra’s algorithm works for both directed and undirected graphs.
- Dijkstra’s algorithm doesn’t work for graphs with negative weights or negative weight cycles.
- This algorithm returns the shortest distance from the source to all other nodes, but after some modification like maintaining parent information of each node we can find out the shortest path.

Implementation

- Maintain two dictionaries with vertices as they keys
  - visited, initially False for all v (burnt vertices)
  - distance, initially infinity for all v (expected burn time)
- Set distance[s] to 0
- Repeat, until all reachable vertices are visited
  - Find unvisited vertex nextv with minimum distance
  - Set visited[nextv] to True
  - Recompute distance[v] for every neighbour v of nextv

Summary

- Dijkstra's algorithm computes single source shortest paths
- Use a greedy strategy to identify vertices to visit
  - Next vertex to visit is based on shortest distance computed so far
  - Need to prove that such a strategy is correct
  - Correctness requires edge weights to be non-negative

Complexity is $O(n^2)$

- Even with adjacency lists
- Bottleneck is identifying unvisited vertex with minimum distance
- Need a better data structure to identify and remove minimum (or max) from a collection.

```python
# Adjacency matrix implementation
def dijkstra(WMat, s):
  (rows, cols, x) = WMat.shape
  infinity = np.max(WMat) * rows + 1
  (visited, distance) = ({}, {})

  for v in range(rows):
    (visited[v], distance[v]) = (False, infinity)

  distance[s] = 0

  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                     if not visited[v]])
    nextvlist = [v for v in range(rows)
                   if (not visited[v]) and
                       distance[v] == nextd]

    if nextvlist == []:
      break

    nextv = min(nextvlist)
    visited[nextv] = True

    for v in range(cols):
      if WMat[nextv, v, 0] == 1 and (not visited[v]):
        distance[v] = min(distance[v], distance[nextv]
                          + WMat[nextv, v, 1])

  return distance
```

```python
# Adjacency list implementation
def dijkstra(WList, s):
  infinity = 1 + len(WList.keys()) * max([d for u in WList.keys()
                                          for (v, d) in WList[u]])
  (visited, distance) = ({}, {})

  for v in WList.keys():
    (visited[v], distance[v]) = (False, infinity)

  distance[s] = 0

  for u in WList.keys():
    nextd = min([distance[v] for v in WList.keys()
                     if not visited[v]])
    nextvlist = [v for v in WList.keys()
                   if (not visited[v]) and
                       distance[v] == nextd]
    if nextvlist == []:
      break

    nextv = min(nextvlist)
    visited[nextv] = True

    for (v, d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v], distance[nextv] + d)

  return distance
```

Bellman Ford algorithm : Single Source Shortest with Negative Weights

- Bellman-Ford works for both directed and undirected graphs with non-negative edges weights.
- Bellman-Ford does not work with an undirected graph with negative edges weight, as it will be declared as a negative weight cycle.
- Bellman-Ford works for a directed graph with negative edge weight, but not with negative weight cycle.

Complexity

- **O(n^3)** for adjacency matrix.
- **O(mn)** for adjacency list : where m is number of edges and n is number of vertices.

Summary

- Dijkstra's algorithm assumes non-negative edge weights
  - Final distance is frozen each time a vertex "burns"
  - Should not encounter a shorter route discovered later
- Without negative cycles, every shortest route is a path
- Every prefix of a shortest path is also a shortest path
- Iteratively find the shortest paths of length 1,2,...,n−1
- Update distance of each vertex with every iteration -- **Bellman-Ford algorithm**
- $$ time with adjacency matrix, **O(mn)** time with adjacency list
- if Bellman-Ford algorithm does not converge after n−1 iterations, there is a negative cycle

```
def bellman_ford(WMat, s):
  (rows, cols, x) = WMat.shape
  infinity = np.max(Wmat) * rows + 1
  distance = {}

  for v in range(rows):
    distance[v] = infinity

  for i in range(rows):
    for u in range(rows):
      for v in range(cols):
        if WMat[u, v, 0] == 1:
          distance[v] = min(distance[v],
                distance[u] + WMat[u, v, 1])

  return distance
```

```
def bellman_ford_list(WList, s):
  infinity = 1 + len(WList.keys()) * max
  ([d for u in WList.keys()for (v, d) in WList[u]])
  distance = {}

  for v in WList.keys():
    distance[v] = infinity

  distance[s] = 0

  for i in WList.keys():
    for u in WList.keys():
      for (v, d) in WList[u]:
        distance[v] = min(distance[v],
                          distance[u] + d)

  return distance
```

All pair of shortest path

- Find the shortest paths between every pair of vertices i and j.
- It is equivalent to if run Dijkstra or Bellman-Ford from each vertex.

Floyd-Warshall algorithm

- Floyd-Warshall's works for both directed and undirected graphs with non-negative edges weights.
- Floyd-Warshall's does not work with an undirected graph with negative edges weight, as it will be declared as a negative weight cycle.
- Floyd-Warshall's algorithm is an alternative way to compute transitive closure **B k [i, j] = 1** if we can reach j from i using vertices in {0, 1, . . . , k−1}
- Floyd-Warshall works for a directed graph with negative edge weight, but not with a negative weight cycle.
- Formula for Floyd-Warshall algorithm is given below:-
- $$SP^k[i,j] = min[SP^{k-1}[i,j],\ SP^{k-1}[i,k] + SP^{k-1}[k,j]]$$

Summary

- Warshall's algorithm is an alternative way to compute transitive closure
  - **B^k[i,j]=1** if we can reach j from i using the vertices in {0,1,...,k−1}

- Adapt Warshall's algorithm to compute all pairs of shortest paths
  - **SP^k[i,j]** if the length of the shortest path from i to j using vertices in {0,1,...,k−1}
  - **SP^n[i,j]** is the length of the overall shortest path

- **Works with negative edge weights assuming no negative cycles**
- **Simple nested loop implementation, time $O(n^3)$**
- Space can be limited to **$O(n^2)$** by re-using 2 "slices" SP and SP'

```python
#For adjacency matrix
def floyd_warshall(WMat):
  (rows, cols, x) = WMat.shape
  infinity = np.max(WMat) * rows * rows + 1
  Sp = np.zeros(shape=(rows, cols, cols + 1))

  for i in range(rows):
    for j in range(cols):
      SP[i, j, 0] = infinity

  for i in range(rows):
    for j in range(cols):
      if WMat[i, j, 0] == 1:
        SP[i, j, 0] = WMat[i, j, 1]

  for k in range(1, cols + 1):
    for i in range(rows):
      for j in range(cols):
        SP[i, j, k] = min(SP[i, j, k - 1]),
        SP[i, k - 1, k - 1] + SP[k - 1, j, k - 1])

  return SP[:, :, cols]
```

Minimum Cost Spanning Tree(MCST)

Spanning Tree(ST)

- Retain a minimal set of edges so that graph remains connected
- Recall that a minimally connected graph is a tree
- Adding an edge to a tree creates a loop
- Removing an edge disconnects the graph
- Want a tree that connects all the vertices — **spanning tree**
- More than one spanning tree, in general

Spanning trees with cost

- Restoring a road or laying a fiber optic cable has a cost
- Minimum cost spanning tree
  - Add the cost of all the edges in the tree
  - Among the different spanning trees, choose one with the minimum cost
- Some facts about trees
  - **A tree on n vertices has exactly n − 1 edges**
  - **Adding an edge to a tree must create a cycle.**
  - **In a tree, every pair of vertices is connected by a unique path**

Building minimum cost spanning trees

- We will use these facts about trees to build minimum cost spanning trees
- Two natural strategies
- Start with the smallest edge and "grow" a tree
  - **Prim's Algorithm**
- Scan the edges in ascending order of weight to connect components without forming cycles
  - **Kruskal's Algorithm**

Summary

- **Prim's algorithm grows an MCST starting with any vertex**
- Implementation similar to Dijkstra's algorithms : Update rule for distance is different

- At each step, connect one more vertex to the tree using minimum cost edge from inside the tree to outside the tree

- Complexity is **O(n^2)**
  - Even with adjacency lists
  - Bottleneck is identifying unvisited vertex with minimum distance
  - Need a better data structure to identify and remove minimum (or max) from a collection

- **Kruskal's algorithm builds an MCST bottom up**
  - Start with n components, each an isolated vertex
  - Scan the edges in ascending order of cost
  - Whenever an edge merges disjoint components, add it to the MCST
- Correctness follows from Minimum Separator Lemma
- Complexity is **O(n^2)** due to naive handling of components
  - We will see how to improve to **O(m.logn)**
- If edge weights repeat, MCST is not unique
- "Choose minimum cost edge" will allow choices
  - Consinder a triangle on 3 vertices with all edges equal
- Different choices lead to different spanning trees
- In general, there may be a very large number of minimum cost spanning trees

```
#Prim's Algorithm using List
def prim_list2(WList):
  infinity = 1 + max([d for u in WList.keys()
                        for (v, d) in WList[u]])
  (visited, distance, nbr) = ({}, {}, {})

  for v in WList.keys():
    (visited[v], distance[v], nbr[v]) =
                    (False, infinity, -1)

  visited[0] = True

  for (v, d) in WList[0]:
    (distance[v], nbr[v]) = (d, 0)

  for i in range(1, len(WList.keys())):
    nextd = min([distance[v] for v in WList.keys()
                    if not visited[v]])
    nextvlist = [v for v in WList.keys()
                    if (not visited[v]) and
                          distance[v] == nextd]

    if nextvlist == []:
      break

    nextv = min(nextvlist)
    visited[nextv] = True

    for (v, d) in WList[nextv]:
      if not visited[v]:
        (distance[v], nbr[v]) = (min(distance[v], d),
                                 nextv)
  return nbr
```

```
#Kruskal's Algorithm using List
def kruskal(WList):
  (edges, components, TE) = ([], {}, [])

  for u in WList.keys():
    edges.extend([(d,u,v) for (v,d) in WList[u]])
    component[u] = u
  edges.sort()

  for (d, u, v) in edges:
    if component[u] != component[v]:
      TE.append((u, v))
      c = component[u]

      for w in WList.keys():
        if component[w] == c:
          component[w] = component[v]

  return TE
```

Union-Find data structure

- A set S partitioned into components {C1, C2, . . . , Ck }
  - Each s ∈ S belongs to exactly one Cj
- Support the following operations
  - MakeUnionFind(S) — set up initial singleton components {s}, for each s ∈ S
  - Find(s) — return the component containing s
  - Union(s,s') — merges components containing s, s 0

```
#Naïve Implementation of Union-Find
class MakeUnionFind:
    def __init__(self):
        self.components = {}
        self.size = 0
    def make_union_find(self,vertices):
        self.size = vertices
        for vertex in range(vertices):
            self.components[vertex]=vertex
    def find(self,vertex):
        return self.components[vertex]
    def union(self,u,v):
        c_old = self.components[u]
        c_new = self.components[v]
        for k in range(self.size):
            if Component[k] == c_old:
                Component[k] = c_new
```

```
#Improved Implementation of Union-Find
class MakeUnionFind:
    def __init__(self):
        self.components = {}
        self.members = {}
        self.size = {}
    def make_union_find(self,vertices):
        for vertex in range(vertices):
            self.components[vertex] = vertex
            self.members[vertex] = [vertex]
            self.size[vertex] = 1
    def find(self,vertex):
        return self.components[vertex]
    def union(self,u,v):
        c_old = self.components[u]
        c_new = self.components[v]
        '''Always add member in components which
           have greater size'''
        if self.size[c_new] >= self.size[c_old]:
            for x in self.members[c_old]:
                self.components[x] = c_new
                self.members[c_new].append(x)
                self.size[c_new] += 1
        else:
            for x in self.members[c_new]:
                self.components[x] = c_old
                self.members[c_old].append(x)
                self.size[c_old] += 1
```

Complexity

- MakeUnionFind(S) — $O(n)$
- Find(i) — $O(1)$
- Union(i,j) — $O(n)$
- Sequence of m Union() operations takes time $O(mn)$

Complexity

- MakeUnionFind(S) — $O(n)$
- Find(i) — $O(1)$
- Union(i,j) — $O(logn)$

Improved Kruskal's using algorithm using Union-find:

Complexity

- Tree has $n-1$ edges, so $O(n)$ Union() operations
- $O(nlogn)$ amortized cost, overall
- Sorting E takes $O(mlogm)$
  - Equivalently $O(mlogn)$, since $m \leq n^2$
- Overall time, $O((m+n)logn)$

```python
#Improved Kruskal's algorithm using Union-find
def kruskal(WList):
    (edges,TE) = ([],[])
    for u in WList.keys():
        edges.extend([(d,u,v) for (v,d) in WList[u]])
    edges.sort()
    mf = MakeUnionFind() #Given on Page1
    mf.make_union_find(len(WList))
    for (d,u,v) in edges:
        if mf.components[u] != mf.components[v]:
            mf.union(u,v)
            TE.append((u,v,d))
        '''We can stop the process if the size becomes
           equal to the total number of vertices'''
        # Which represent that a spanning tree is completed
        if mf.size[mf.components[u]]>= mf.size[mf.components[u]]:
            if mf.size[mf.components[u]] == len(WList):
                break
        else:
            if mf.size[mf.components[v]] == len(WList):
                break
    return(TE)
```

Priority Queue

Need to maintain a collection of items with priorities to optimize the following operations

- **delete max( )**
  - Identify and remove item with highest priority
  - Need not be unique
- **insert( )**
  - Add a new item to the list

Implementing Priority Queues

One dimensional :

- Unsorted list
  - insert( ) is ***O(1)***
  - delete_max( ) is ***O(n)***
- Sorted list
  - delete_max( ) is ***O(1)***
  - insert( ) is ***O(n)***
- Processing n items requires $O(n^2)$

Two dimensional :

- $\sqrt{N} \times \sqrt{N}$ array with sorted rows
  - insert( ) is $O(\sqrt{N})$
  - delete_max is $O(\sqrt{N})$
  - Processing N items is $O(N\sqrt{N})$

Binary tree

A binary tree is a tree data structure in which each node can contain at most 2 children, which are referred to as the left child and the right child.

Heap

Heap is a binary tree, filled level by level, left to right. There are two types of the heap:

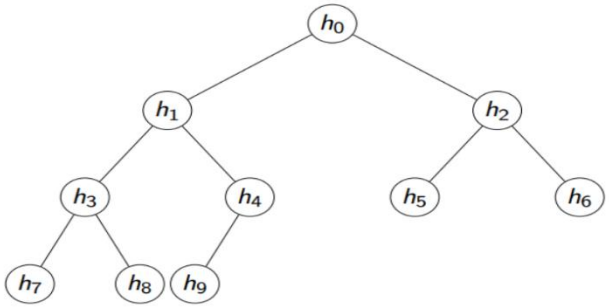
- Max heap - For each node V in heap except for leaf nodes, the value of V should be greater or equal to its child's node value.
- Min heap - For each node V in heap except for leaf nodes, the value of V should be less or equal to its child's node value.
- **We can represent heap using array(list in python)**

`H = [h0, h1, h2, h3, h4, h5, h6, h7, h8, h9]`

left child of `H[i] = H[2 * i + 1]`

Right child of `H[i] = H[2 * i + 2]`

Parent of `H[i] = H[(i-1) // 2], for i > 0`




```python
#Max Heap
class maxheap:
    def __init__(self):
        self.A = []
    def max_heapify(self,k):
        l = 2 * k + 1
        r = 2 * k + 2
        largest = k
        if l < len(self.A) and self.A[l]>self.A[largest]:
            largest = l
        if r < len(self.A) and self.A[r]>self.A[largest]:
            largest = r
        if largest != k:
            self.A[k],self.A[largest]=self.A[largest],self.A[k]
            self.max_heapify(largest)

    def build_max_heap(self,L):
        self.A = []
        for i in L:
            self.A.append(i)
        n = int((len(self.A)//2)-1)
        for k in range(n, -1, -1):
            self.max_heapify(k)

    def delete_max(self):
        item = None
        if self.A != []:
            self.A[0],self.A[-1] = self.A[-1],self.A[0]
            item = self.A.pop()
            self.max_heapify(0)
        return item

    def insert_in_maxheap(self,d):
        self.A.append(d)
        index = len(self.A)-1
        while index > 0:
            parent = (index-1)//2
            if self.A[index] > self.A[parent]:
                self.A[index],self.A[parent]=self.A[parent],
                self.A[index]
                index = parent
            else:
                break
```

```python
#Min Heap
class minheap:
    def __init__(self):
        self.A = []
    def min_heapify(self,k):
        l = 2 * k + 1
        r = 2 * k + 2
        smallest = k
        if l < len(self.A) and self.A[l]<self.A[smallest]:
            smallest = l
        if r < len(self.A) and self.A[r]<self.A[smallest]:
            smallest = r
        if smallest != k:
            self.A[k], self.A[smallest]=self.A[smallest],self.A[k]
            self.min_heapify(smallest)

    def build_min_heap(self,L):
        self.A = []
        for i in L:
            self.A.append(i)
        n = int((len(self.A)//2)-1)
        for k in range(n, -1, -1):
            self.min_heapify(k)

    def delete_min(self):
        item = None
        if self.A != []:
            self.A[0],self.A[-1] = self.A[-1],self.A[0]
            item = self.A.pop()
            self.min_heapify(0)
        return item

    def insert_in_minheap(self,d):
        self.A.append(d)
        index = len(self.A)-1
        while index > 0:
            parent = (index-1)//2
            if self.A[index] < self.A[parent]:
                self.A[index],self.A[parent] = self.A[parent],
                self.A[index]
                index = parent
            else:
                break
```

Complexity

Heaps are a tree implementation of priority queues

- insert( ) is ***O(logN)***
- delete max( ) is ***O(logN)***
- heapify( ) builds a heap in ***O(N)***

Complexity : Heap Sort

- Start with an unordered list
- Build a heap — ***O(n)***
- Call delete max() n times to extract elements in descending order — ***O(nlogn)***
- After each delete max(), heap shrinks by 1
- Store maximum value at the end of current heap
- **In place *O(nlogn)* sort**

Binary Search Tree (BST)

A **binary search tree** is a binary tree that is either empty or satisfies the following conditions:

For each node V in the Tree

- The value of the left child or left subtree is always less than the value of V.
- The value of the right child or right subtree is always greater than the value of V

```
# considering dictionary as a heap for given code
def min_heapify(i,size):
    lchild = 2*i + 1
    rchild = 2*i + 2
    small = i
    if lchild < size-1 and HtoV[lchild][1] < HtoV[small][1]:
        small = lchild
    if rchild < size-1 and HtoV[rchild][1] < HtoV[small][1]:
        small = rchild
    if small != i:
        VtoH[HtoV[small][0]] = i
        VtoH[HtoV[i][0]] = small
        (HtoV[small],HtoV[i]) = (HtoV[i], HtoV[small])
        min_heapify(small,size)

def create_minheap(size):
    for x in range((size//2)-1,-1,-1):
        min_heapify(x,size)

def minheap_update(i,size):
    if i!= 0:
        while i > 0:
            parent = (i-1)//2
            if HtoV[parent][1] >  HtoV[i][1]:
                VtoH[HtoV[parent][0]] = i
                VtoH[HtoV[i][0]] = parent
                (HtoV[parent],HtoV[i]) = (HtoV[i], HtoV[parent])
            else:
                break
            i = parent

def delete_min(hsize):
    VtoH[HtoV[0][0]] = hsize-1
    VtoH[HtoV[hsize-1][0]] = 0
    HtoV[hsize-1],HtoV[0] = HtoV[0],HtoV[hsize-1]
    node,dist = HtoV[hsize-1]
    hsize = hsize - 1
    min_heapify(0,hsize)
    return node,dist,hsize
```

```
#Heap sort Implementation:
def max_heapify(A,size,k):
    l = 2 * k + 1
    r = 2 * k + 2
    largest = k
    if l < size and A[l] > A[largest]:
        largest = l
    if r < size and A[r] > A[largest]:
        largest = r
    if largest != k:
        (A[k], A[largest]) = (A[largest], A[k])
        max_heapify(A,size,largest)

def build_max_heap(A):
    n = (len(A)//2)-1
    for i in range(n, -1, -1):
        max_heapify(A,len(A),i)

def heapsort(A):
    build_max_heap(A)
    n = len(A)
    for i in range(n-1,-1,-1):
        A[0],A[i] = A[i],A[0]
        max_heapify(A,i,0)
```

```
#Updated Implementation for adjacency matrix using min heap:
#global HtoV map heap index to (vertex,distance from source)
#global VtoH map vertex to heap index
HtoV, VtoH = {},{}
def dijkstra(WMat,s):
    (rows,cols,x) = WMat.shape
    infinity = float('inf')
    visited = {}
    heapsize = rows
    for v in range(rows):
        VtoH[v]=v
        HtoV[v]=[v,infinity]
        visited[v] = False
    HtoV[s]= [s,0]
    create_minheap(heapsize)

    for u in range(rows):
        nextd,ds,heapsize = delete_min(heapsize)
        visited[nextd] = True
        for v in range(cols):
            if WMat[nextd,v,0] == 1 and (not visited[v]):
                '''update distance of adjacent of v if it is
                   less than to previous one'''
                HtoV[VtoH[v]][1] = min(HtoV[VtoH[v]][1],ds+WMat[nextd,v,1])
                minheap_update(VtoH[v],heapsize)
```

```
#Updated Implementation for adjacency list using min heap:
HtoV, VtoH = {},{}
#global HtoV map heap index to (vertex,distance from source)
#global VtoH map vertex to heap index
def dijkstralist(WList,s):
    infinity = float('inf')
    visited = {}
    heapsize = len(WList)
    for v in WList.keys():
        VtoH[v]=v
        HtoV[v]=[v,infinity]
        visited[v] = False
    HtoV[s]= [s,0]
    create_minheap(heapsize)

    for u in WList.keys():
        nextd,ds,heapsize = delete_min(heapsize)
        visited[nextd] = True
        for v,d in WList[nextd]:
            if not visited[v]:
                HtoV[VtoH[v]][1] = min(HtoV[VtoH[v]][1],ds+d)
                minheap_update(VtoH[v],heapsize)
```

Complexity : BST

- find( ), insert( ) and delete( ) all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with n nodes may have height ***O(n)***
- Balanced trees have height ***O(logn)***
- Will see how to keep a tree balanced to ensure all operations remain ***O(logn)***

BY – ADITYA DHAR DWIVEDI

Balanced search tree (AVL Tree) - : - Greedy Algorithm

Binary search tree

- find(), insert() and delete() all walk down a single path
- Worst-case: height of the tree An unbalanced tree with n nodes may have height O(n)

AVL Tree

- Balanced trees have height ***O(logn)***
- Using rotations, we can maintain height balance
- Height balanced trees have height ***O(logn)***
- find(), insert() and delete() all walk down a single path, take time ***O(logn)***
- Minimum number of node $S(h)=S(h-2)+S(h-1)+1$
- Maximum number of nodes 2^h-1

Greedy Algorithm

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion
- Never go back and revise an earlier decision
- Drastically reduces space to search for solutions
- Greedy strategy needs a proof of optimality
- Example :
    - Dijkstra's
    - Prim's
    - Kruskal's
    - Interval scheduling
    - Minimize lateness
    - Huffman coding

Algorithm

1. Sort all jobs which based on end time in increasing order.
2. Take the interval which has earliest finish time.
3. Repeat next two steps till you process all jobs.
4. Eliminate all intervals which have start time less than selected interval's end time.
5. If interval has start time greater than current interval's end time, at it to set. Set current interval to new interval.

Analysis

- Initially, sort n bookings by finish time - ***O(nlogn)***
- Single scan***, O(n)***
- overall ***O(nlogn)***

```
#Interval Scheduling:
def tuplesort(L, index):
    L_ = []
    for t in L:
        L_.append(t[index:index+1] +
                  t[:index]+t[index+1:])
    L_.sort()

    L__ = []
    for t in L_:
        L__.append(t[1:index+1] +
                   t[0:1]+t[index+1:])
    return L__

def intervalschedule(L):
    sortedL = tuplesort(L, 2)
    accepted = [sortedL[0][0]]
    for i, s, f in sortedL[1:]:
        if s > L[accepted[-1]][2]:
            accepted.append(i)
    return accepted
```

Minimize Lateness

Algorithm

1. Sort all job in ascending order of deadlines
2. Start with time t = 0
3. For each job in the list
    1. Schedule the job at time t
    2. Finish time = t + processing time of job
    3. t = finish time
4. Return (start time, finish time) for each job

```
from operator import itemgetter

def minimize_lateness(jobs):
    schedule =[]
    max_lateness = 0
    t = 0

    sorted_jobs = sorted(jobs,key=itemgetter(2))

    for job in sorted_jobs:
        job_start_time = t
        job_finish_time = t + job[1]

        t = job_finish_time
        if(job_finish_time > job[2]):
            max_lateness =  max (max_lateness,
                                 (job_finish_time-job[2]))
        schedule.append((job[0],job_start_time,
                         job_finish_time))

    return max_lateness, schedule
```

Analysis

- Sort the requests by D(i) — ***O(nlogn)***
- Read all schedule in sorted order — ***O(n)***
- overall ***O(nlogn)***

Huffman Coding

Algorithm

1. Calculate the frequency of each character in the string.
2. Sort the characters in increasing order of the frequency.
3. Make each unique character as a leaf node.
4. Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.
5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies.
6. Insert node z into the tree.
7. Repeat steps 3 to 5 for all the characters.
8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.

Analysis

- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency
- Store frequencies in an array
- Linear scan to find minimum values
- $|A|=k$, number of recursive calls is k–1
- Complexity is $O(k^2)$
- Instead, maintain frequencies in an heap
- Extracting two minimum frequency letters and adding back compound letter are both $O(\log k)$
- Complexity drops to ***O(klogK)***

```
1
2  class Node:
3      def __init__(self,frequency,symbol = None,left = None,right = None):
4          self.frequency = frequency
5          self.symbol = symbol
6          self.left = left
7          self.right = right
8
9  # Solution
10
11 def Huffman(s):
12     huffcode = {}
13     char = list(s)
14     freqlist = []
15     unique_char = set(char)
16     for c in unique_char:
17         freqlist.append((char.count(c),c))
18     nodes = []
19     for nd in sorted(freqlist):
20         nodes.append((nd,Node(nd[0],nd[1])))
21     while len(nodes) > 1:
22         nodes.sort()
23         L = nodes[0][1]
24         R = nodes[1][1]
25         newnode = Node(L.frequency + R.frequency, L.symbol + R.symbol,L,R)
26         nodes.pop(0)
27         nodes.pop(0)
28         nodes.append(((L.frequency + R.frequency, L.symbol + R.symbol),newnode))
29
30     for ch in unique_char:
31         temp = newnode
32         code = ''
33         while ch != temp.symbol:
34             if ch in temp.left.symbol:
35                 code += '0'
36                 temp = temp.left
37             else:
38                 code += '1'
39                 temp = temp.right
40         huffcode[ch] = code
41     return huffcode
```