Shortest Path Algorithm

- Dijkstra's Algo
  - Greedy
  - Find shortest path to each edge 1 by 1
- Bellman Ford
  - Max path length will be V-1
  - Compute shortest path for each path length

In [ ]:

```
```

In [2]:

```
class AdjDirectedGraph:
    def __init__(self):
        self.adj_list = dict() # hashmap<str, pair<str, int> > = map< src, <dest,

    # add edge from a->b
    def add(self, a, b, w): # a: src, b: dest, w: weight

        if a not in self.adj_list:
            self.adj_list[a] = []

        if b not in self.adj_list:
            self.adj_list[b] = []

        self.adj_list[a].append( (b, w) )

    def print(self):
        for k,v in self.adj_list.items():
            print(k, v)

    def get_adj(self, key):
        return self.adj_list[key]

    def vertices(self):
        return self.adj_list.keys() # return list of keys only/vertices
```

In [8]:

```
# Djikstra's algo

import heapq
# Adj List repr

class DummyHeap:
    def __init__(self):
        self.distances = {}

    def print(self):
        print(self.distances)

    def add_key(self, key): #key = (dest, weight)
        self.distances[key[0]] = key[1]

    def empty(self):
        return len(self.distances) == 0

    def remove_min(self):  # pop
        if self.empty():
            return None

        res = None
        # iterate over all vertices
        for v,d in self.distances.items(): # list of <vertex, distance> key,v
            if res is None:
                res = v
            elif self.distances[v] < self.distances[res]:
                res = v

        value = self.distances.pop(res)
        return (res, value)

    def decrease_key(self, key, value):
        self.distances[key] = value

    def get_key(self, key):
        return self.distances[key]


def find_shortest_path(graph, src, dest):
    # contains distance from source to all vertices, such that we are able to
    hq = DummyHeap()      # SC: O(V)
    hq.add_key( [src, 0]) # SC: O(V)

    distances = {}
    for v in graph.vertices():
        if v == src:
            distances[v] = 0
        else:
            distances[v] = float('inf') # set to big value

    hq.print()

    while not hq.empty():

        v, curr_dist = hq.remove_min()
        print("removed", v)

```

```
        # get all neighbours
        neighbors = graph.get_adj(v)
        for neighbor in neighbors:
            d = neighbor[0]
            wt = neighbor[1]
            if curr_dist + wt < distances[d]:
                distances[d] = curr_dist + wt
                hq.decrease_key(d, curr_dist + wt)
        print('distances', distances)

    return distances[dest]

# O(E log V)
# SC: O(V)

"""
  (A)--4->(B)---8-->(C)
   | \     |       *
   |  \    |      /
   2   8   2     1
   |     \ |   /
   *      **  /
  (D)--5->(E)
"""

g = AdjDirectedGraph()
g.add("A", "B", 4)
g.add("B", "C", 8)
g.add("A", "E", 8)
g.add("A", "D", 2)
g.add("B", "E", 2)
g.add("D", "E", 5)
g.add("E", "C", 1)
g.print()
print(g.vertices())

print()
print("shortest path distance", find_shortest_path(g, "A", "B"))
print()
print("shortest path distance", find_shortest_path(g, "A", "C"))
print()
print("shortest path distance", find_shortest_path(g, "A", "E"))

```

```
A [('B', 4), ('E', 8), ('D', 2)]
B [('C', 8), ('E', 2)]
C []
E [('C', 1)]
D [('E', 5)]
dict_keys(['A', 'B', 'C', 'E', 'D'])

{'A': 0}
removed A
distances {'A': 0, 'B': 4, 'C': inf, 'E': 8, 'D': 2}
removed D
distances {'A': 0, 'B': 4, 'C': inf, 'E': 7, 'D': 2}
removed B
distances {'A': 0, 'B': 4, 'C': 12, 'E': 6, 'D': 2}
removed E
distances {'A': 0, 'B': 4, 'C': 7, 'E': 6, 'D': 2}
removed C
distances {'A': 0, 'B': 4, 'C': 7, 'E': 6, 'D': 2}
shortest path distance 4

{'A': 0}
removed A
distances {'A': 0, 'B': 4, 'C': inf, 'E': 8, 'D': 2}
removed D
distances {'A': 0, 'B': 4, 'C': inf, 'E': 7, 'D': 2}
removed B
distances {'A': 0, 'B': 4, 'C': 12, 'E': 6, 'D': 2}
removed E
distances {'A': 0, 'B': 4, 'C': 7, 'E': 6, 'D': 2}
removed C
distances {'A': 0, 'B': 4, 'C': 7, 'E': 6, 'D': 2}
shortest path distance 7

{'A': 0}
removed A
distances {'A': 0, 'B': 4, 'C': inf, 'E': 8, 'D': 2}
removed D
distances {'A': 0, 'B': 4, 'C': inf, 'E': 7, 'D': 2}
removed B
distances {'A': 0, 'B': 4, 'C': 12, 'E': 6, 'D': 2}
removed E
distances {'A': 0, 'B': 4, 'C': 7, 'E': 6, 'D': 2}
removed C
distances {'A': 0, 'B': 4, 'C': 7, 'E': 6, 'D': 2}
shortest path distance 6
```

In [ ]:

In [14]:

```
# Djikstra's algo

import heapq

def find_shortest_path(graph, src, dest):
    hq = [(0,src)] # SC: O(V)
    distances = {} # SC: O(V)

    for v in graph.vertices():
        if v == src:
            distances[v] = 0
        else:
            distances[v] = float('inf') # set to big value
    print(hq)

    while len(hq) != 0:

        dist, v  = heapq.heappop(hq)
        print("removed", v, dist)

        # get all neighbours
        neighbors = graph.get_adj(v)

        for neighbor in neighbors:
            d = neighbor[0]
            wt = neighbor[1]

            if (dist + wt) < distances[d]:
                distances[d] = dist + wt
                heapq.heappush(hq, (dist + wt, d) ) # push a duplicate key with s
            print('heap', hq)

    print(distances)
    return distances[dest]


"""
  (A)--4->(B)---8-->(C)
   | \     |       *
   |  \    |      /
   2   8   2     1
   |     \ |   /
   *      **  /
  (D)--5->(E)
"""

g = AdjDirectedGraph()
g.add("A", "B", 4)
g.add("B", "C", 8)
g.add("A", "E", 8)
g.add("A", "D", 2)
g.add("B", "E", 2)
g.add("D", "E", 5)
g.add("E", "C", 1)
g.print()
print(g.vertices())

print()
print("shortest path distance", find_shortest_path(g, "A", "B"))
```

```
print()
print("shortest path distance", find_shortest_path(g, "A", "C"))
print()
print("shortest path distance", find_shortest_path(g, "A", "E"))
```

```
A [('B', 4), ('E', 8), ('D', 2)]
B [('C', 8), ('E', 2)]
C []
E [('C', 1)]
D [('E', 5)]
dict_keys(['A', 'B', 'C', 'E', 'D'])

[(0, 'A')]
removed A 0
heap [(4, 'B')]
heap [(4, 'B'), (8, 'E')]
heap [(2, 'D'), (8, 'E'), (4, 'B')]
removed D 2
heap [(4, 'B'), (8, 'E'), (7, 'E')]
removed B 4
heap [(7, 'E'), (8, 'E'), (12, 'C')]
heap [(6, 'E'), (7, 'E'), (12, 'C'), (8, 'E')]
removed E 6
heap [(7, 'C'), (7, 'E'), (12, 'C'), (8, 'E')]
removed C 7
removed E 7
heap [(8, 'E'), (12, 'C')]
removed E 8
heap [(12, 'C')]
removed C 12
{'A': 0, 'B': 4, 'C': 7, 'E': 6, 'D': 2}
shortest path distance 4

[(0, 'A')]
removed A 0
heap [(4, 'B')]
heap [(4, 'B'), (8, 'E')]
heap [(2, 'D'), (8, 'E'), (4, 'B')]
removed D 2
heap [(4, 'B'), (8, 'E'), (7, 'E')]
removed B 4
heap [(7, 'E'), (8, 'E'), (12, 'C')]
heap [(6, 'E'), (7, 'E'), (12, 'C'), (8, 'E')]
removed E 6
heap [(7, 'C'), (7, 'E'), (12, 'C'), (8, 'E')]
removed C 7
removed E 7
heap [(8, 'E'), (12, 'C')]
removed E 8
heap [(12, 'C')]
removed C 12
{'A': 0, 'B': 4, 'C': 7, 'E': 6, 'D': 2}
shortest path distance 7

[(0, 'A')]
removed A 0
heap [(4, 'B')]
heap [(4, 'B'), (8, 'E')]
heap [(2, 'D'), (8, 'E'), (4, 'B')]
removed D 2
heap [(4, 'B'), (8, 'E'), (7, 'E')]
removed B 4
heap [(7, 'E'), (8, 'E'), (12, 'C')]
heap [(6, 'E'), (7, 'E'), (12, 'C'), (8, 'E')]
removed E 6
heap [(7, 'C'), (7, 'E'), (12, 'C'), (8, 'E')]
```

```
removed C 7
removed E 7
heap [(8, 'E'), (12, 'C')]
removed E 8
heap [(12, 'C')]
removed C 12
{'A': 0, 'B': 4, 'C': 7, 'E': 6, 'D': 2}
shortest path distance 6
```

In [ ]:

In [22]:

```
# Bellman Ford

def find_shortest_path(edges, v, src, dest):
    distances = {src: 0} # SC: O(V)

    for i in range(1, v): # traverse V-1 times
        print()
        dist_bkp = distances.copy() # SC: O(V)
        for edge in edges:
            s, d, wt = edge
            # use only for looking up weather to consider src for current path le
            src_dist = dist_bkp.get(s, float('inf'))

            if src_dist != float('inf') and distances.get(d, float('inf')) > src_
                distances[d] = src_dist + wt
            print(dist_bkp, distances)

    print("distances", distances, dest)
    return distances[dest]

# O(V*E)
# SC: O(V)

"""
  (A)--4--(B)---8---(C)
   | \     |       /
   |  \    |      /
   2   8   2     1
   |     \ |   /
   |      \|  /
  (D)--5--(E)
"""
# distance {"A": 0, "B":4, "E":8, "D":2}   {"A": 0, "B":4, "E":6, "D":2, "C":12}
# dist_bkp {"A": 0}                        {"A": 0, "B":4, "E":8, "D":2}

g = [] # list of edges
g.append( ("A", "B", 4) )
g.append( ("B", "C", 8) )
g.append( ("A", "E", 8) )
g.append( ("A", "D", 2) )
g.append( ("B", "E", 2) )
g.append( ("D", "E", 5) )
g.append( ("E", "C", 1) )
print(g)

print()
print("shortest path distance", find_shortest_path(g, 5, "A", "B"))
print()
print("shortest path distance", find_shortest_path(g, 5, "A", "C"))
print()
print("shortest path distance", find_shortest_path(g, 5, "A", "E"))
```

```
[('A', 'B', 4), ('B', 'C', 8), ('A', 'E', 8), ('A', 'D', 2), ('B',
'E', 2), ('D', 'E', 5), ('E', 'C', 1)]


{'A': 0} {'A': 0, 'B': 4}
{'A': 0} {'A': 0, 'B': 4}
{'A': 0} {'A': 0, 'B': 4, 'E': 8}
{'A': 0} {'A': 0, 'B': 4, 'E': 8, 'D': 2}
{'A': 0} {'A': 0, 'B': 4, 'E': 8, 'D': 2}
{'A': 0} {'A': 0, 'B': 4, 'E': 8, 'D': 2}
{'A': 0} {'A': 0, 'B': 4, 'E': 8, 'D': 2}

{'A': 0, 'B': 4, 'E': 8, 'D': 2} {'A': 0, 'B': 4, 'E': 8, 'D': 2}
{'A': 0, 'B': 4, 'E': 8, 'D': 2} {'A': 0, 'B': 4, 'E': 8, 'D': 2,
'C': 12}
{'A': 0, 'B': 4, 'E': 8, 'D': 2} {'A': 0, 'B': 4, 'E': 8, 'D': 2,
'C': 12}
{'A': 0, 'B': 4, 'E': 8, 'D': 2} {'A': 0, 'B': 4, 'E': 8, 'D': 2,
'C': 12}
```

In [15]:

```
d = {}
d2 = d.copy()
d[1] = 2
print(d, d2)
```

```
{1: 2} {}
```

In [ ]:

```

```

Question

https://leetcode.com/problems/cheapest-flights-within-k-stops/description/ (https://leetcode.com/problems/cheapest-flights-within-k-stops/description/)

Python

```python
class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: in
t, dst: int, k: int) -> int:
        distances = {src: 0}
        edges = flights

        for i in range(k+1): # traverse V-1 times
            dist_bkp = distances.copy()
            for edge in edges:
                s, d, wt = edge
                src_dist = dist_bkp.get(s, float('inf'))

                if src_dist != float('inf') and distances.get(d, float('i
nf')) > src_dist + wt:
                    distances[d] = src_dist + wt

        return distances.get(dst, -1)
```

C++

```
class Solution {
public:
    int findCheapestPrice(int n, vector<vector<int>>& flights, int src, i
nt dst, int k) {
```

Java

```
class Solution {
    public int findCheapestPrice(int n, int[][] flights, int src, int ds
t, int k) {
        Map<Integer, Integer> distances = new HashMap<>();
        distances.put(src, 0);

        for (int i = 0; i <= k; i++) {
            Map<Integer, Integer> dist_bkp = new HashMap<>(distances);

            for (int[] edge : flights) {
                int s = edge[0];
                int d = edge[1];
                int wt = edge[2];

                int src_dist = dist_bkp.getOrDefault(s, Integer.MAX_VALU
E);

                if (src_dist != Integer.MAX_VALUE && distances.getOrDefau
lt(d, Integer.MAX_VALUE) > src_dist + wt) {
                distances.put(d, src_dist + wt);
                }
            }
        }
        return distances.get(dst)== null ? -1:distances.get(dst);
    }
}
```