Algorithms&Complexity - In Class Assignments (Week 3)

Greedy Algorithms

Manuela Cleves

1. Let's use Dijkstra's algorithms. Assume that we want to travel the **longest** possible route.

Problem: Find the longest path from the node s to every other node in the weighted directed acyclic graph G=(V,E). Weights of all edges are ≥0.

Usual process for Dijkstra's Algorithm:

Problem: Find the shortest path from the node s to every other node in the weighted graph G=(V,E). Weights of all edges are >0.

- Mark the base node with a distance of zero and mark it as the current node.
- Find all nodes connected to the current node. Calculate the shortest distance between them and the base node. Don't record these distances if they are longer than a previously recorded distances.
- Mark the current node as visited.
- Mark the unvisited node with the shortest distance to the base node as current. Go to step 2.
- If all nodes are visited stop the algorithm.

Complexity of this algorithm is Θ(V^2) and Θ(E+V log$V$) if you use min heap or Fibonacci heap to store un-visited nodes.

Visualization for future reference: https://www.youtube.com/watch?v=wtdtkJgcYUM

In [10]:
```
def longest_route(graph, start):
    # Transform weights into their negative values (this way the shortest route
    negative_graph = {node: {neighbor: -weight for neighbor, weight in neighbo

    # Initialize distances with negative infinity for all nodes
    distances = {node: float('-inf') for node in graph}
    distances[start] = 0

    # Queue for Dijkstra's algorithm
    queue = [(0, start)]

    while queue:
        current_distance, current_node = min(queue)

        # Remove the current node from the ueue
```

```
        queue.remove((current_distance, current_node))

        # If the node has been visited before with a longer distance, skip
        if current_distance < distances[current_node]:
            continue

        # Update distances to neighbors
        for neighbor, weight in negative_graph[current_node].items():
            distance = current_distance + weight

            # If a shorter path is found, update the distance
            if distance > distances[neighbor]:
                distances[neighbor] = distance
                queue.append((distance, neighbor))

    return distances

# Example usage
# Create our graph with nodes that are represented by letters ('s', 'a', 'b',
graph = {
    's': {'a': 2, 'b': 3},
    'a': {'b': 1, 'c': 4},
    'b': {'c': 1},
    'c': {}
}

start_node='s'
longest = longest_route(graph, start_node)

#Print function that lets us know the longest possible path from 's' to each o
print("Longest Paths from {}: {}".format(start_node, longest))
```

```
Longest Paths from s: {'s': 0, 'a': -2, 'b': -3, 'c': -4}
```

2. Problem: Given a set of coins, find the minimum number of coins which would equate to the input value.

Input: C={c_1,c_2,...,c_n} – list of coins in the cash register, V – input value.

In [1]:

```
def coin_calculator(coins,value):
    #First, sort the coins
    coins.sort(reverse=True)
    #Initialize parameters
    remaining_value=value
    coins_in=0

    #Now, to count the coins
    for coin in coins:
        #Count the coins of the current value
        number_coins=remaining_value//coin
        #Recalculate value
        remaining_value%=coin
        #Update total coins in
        coins_in+=number_coins

        if remaining_value==0:
            break
    return coins_in
```

```
#Example

C=[25, 25, 5, 5, 1]
value=100

result = coin_calculator (C,value)

print ("Minimim number of coins:", result)
```

```
Minimim number of coins: 4
```

In [ ]: