Solution:

Since we only want to look up adjacent roads for roads, it only takes O(1) time (the same as for adjacency matrices). As for space complexity, adjacency lists have better space complexity than adjacency matrices because adjacency matrices only take O(e2), whereas adjacency lists only take O(e+v) (where e = edges and v = vertex). Additionally, since real-world graphs are sparse, an adjacency matrix requires a significant amount of storage. So  Adjacency list is a decent option in our instance.

**Code for RoadGraph class:**

**(u == initial location, v == final location, w == weight)**

class RoadGraph:

*# graph Constructor*

    def \_\_init\_\_(self, roads):

        n = len(roads)

*# implementing the adjacency list*

        self.adjList = [None] \* n

*# code for each above adjacency list creating adjacency list*

        for i in range(n):

            self.adjList[i] = []

*# add edges to the directed graph*

        for (u, v, w) in roads:

*# allocate node in adjacency list from src to dest*

            self.adjList[u].append((v, w))

*# PrintGraph is the Function for printing adjacency list representation of a graph*

def printGraph(graph):

    for u in range(len(graph.adjList)):

*# print all its neighboring vertices and current vertex*

        for (v, w) in graph.adjList[u]:

            print(f'({u} —> {v}, {w}) ', end='')

        print()

if \_\_name\_\_ == '\_\_main\_\_':

    roads = [(0, 1, 4), (0, 3, 2), (0, 2, 3), (2, 3, 2), (3, 0, 3)]

*# construct the structure of the graph*

    mygraph\_constructed = RoadGraph(roads)

*# displaing the adjacency list representation of the graph*

    printGraph(mygraph\_constructed)  *# (u->v, weight) representation*

Section 2 :

import heapq

class RoadGraph:

    vertices = []

    def \_\_init\_\_(self, roads):

        """

        takes roads as input and finds the highest vertex which is used as the size of the graph

        this highest vertex is found by looking at the end vertex in each tuple as some vertices dont

        connect to others but are connected to therefore looking at the end values finds the biggest vertex

        then each edge is added to its starting vertex's list in an adjacency matrix

        :param roads a list of tuples representing start, end vertices and distance:

        complexity:

        O(n)

        """

*# ToDo: Initialize the graph data structure here*

        vertexmax = roads[0][1]

*# loops through roads and finds the highest vertex used as the graph size*

        for i in range(len(roads)):

            if roads[i][1] > vertexmax:

                vertexmax = roads[i][1]

        vertexmax += 1

*# initialize an adjacency matrix of E\*V*

        self.vertices = [[0]\*vertexmax for i in range(vertexmax)]

*# loops through roads and gets the start end and length of each edge then adds it to the adjacency matrix*

        for i in range(len(roads)):

            road = roads[i]

            start = road[0]

            end = road[1]

            length = road[2]

            self.vertices[start][end] = length

    def routing(self, start, end, chores\_location):

*# ToDo: Performs the operation needed to find the optimal route.*

        """

        :param start int representing the starting vertex:

        :param end int representing the end vertex:

        :param chores\_location list of ints representing locations that at least one must be in the path:

        :return:

        """

*#visited = [False] \* (len(self.vertices))*

*#*

*#*

*#*

*# queue = []*

*# queue.append(start)*

*#*

*# vertIndex = start*

*# visited = [False] \* len(self.vertices)*

*# visited[start] = True*

*#*

*# path = []*

*#*

*# while len(queue) > 0:*

*#     vertIndex = queue.pop(0)*

*#     vertex = self.vertices[vertIndex]*

*#     #need to change min\_heap so that index is stored*

*#     min\_heap = self.vertices[vertIndex]*

*#     heapq.heapify(min\_heap)*

*#*

*#     print(min\_heap)*

*#*

*#     for i in range(len(vertex)):*

*#         #need to stop once end is in visited and a chore is in visited*

*#         next\_vertex = min\_heap.pop(0)*

*#         if next\_vertex != 0 and visited[i] == False:*

*#             print(i, next\_vertex)*

*#             visited[i] = True*

*#             queue.append(i)*

*#     #need to only add to path if it is visited*

*#     if len(path) == 0 or path[len(path)-1] != vertIndex and visited[vertIndex] == True:*

*#         path.append(vertIndex)*

*# print("path", path)*

*# 2 lists visited and unvisited*

*# distance to source is 0*

*# distance to other vertices is infinite*

*# visit smallest distance vertex which is source at start*

*# examine neighbours*

*# compute distance to neigbours current total distance + their distance*

*# if the distance is shorter than the known distance update smallest distance in table*

*# update prev vertex  for neighbours ie how we got there*

*# add current vertex to visisted*

*# create a list for storing whether a vertex has been looked at yet*

        visited = [False] \* (len(self.vertices))

        unvisisted = []

        path = []

*# tuples of (index of vertex, distance, prev vertex)*

        vertices = []

        for i in range(len(self.vertices)):

            tup = (i, float('inf'), 0)

            tuple(tup)

            vertices.append(tup)

        vertices[start] = start, 0, 0

        prev = start

        visited[start] = True

        unvisisted.append(start)

*# use a modified Dijikstras algo*

*# loops while the queue has elements in it*

        while len(unvisisted) > 0:

*# loops infinely*

*#print("distances", distances)*

*# gets the index and distance of the vertex with the smallest distance*

            minIndex, distance = Smallest\_get(

                self.vertices, visited, unvisisted, prev)

            unvisisted.append(minIndex)

            print("minIndex", minIndex, "distance", distance)

            vertex = vertices[minIndex][0]

*#print("vertex:", vertex)*

*#distance = vertices[minIndex][1]*

            prevVertex = vertices[minIndex][2]

            neighbourDistance = []

*# run loop over the vertex's neighbours*

            for i in range(len(self.vertices[minIndex])):

*# calculate the distance from the start to neighbour*

                ndistance = distance + vertices[minIndex][1]

*# if the stored value is more than neighbours distance*

                if ndistance < distance:

*# updating the  distance*

                    distance = ndistance

*# update the prev vertex*

                    prevVertex = prev

*#print("distance:", distance)*

*#print("prev:", prevVertex)*

                    vertices[vertex][1] = distance

                    vertices[vertex][2] = prevVertex

                    print("vertex", minIndex, distance, prevVertex)

*# adding the current vertex(not its neighbour) to visited*

            visited[minIndex] = True

*# seting prev to the current vertex*

            prev = minIndex

*# poping the current vertex from unvisited*

            unvisisted.pop(0)

*# print(visited)*

*# print(vertices)*

def Smallest\_get(vertices, visited, unvisited, prev):

    distance = 99

    index = 0

    print(vertices)

    for i in range(len(vertices[prev])):

*#print("vertex", vertices[i])*

*#print("distance < distance", vertices[i][1], distance)*

        if vertices[prev][i] < distance and i != prev:

            print("Index", index, "distance", distance)

            distance = vertices[i][1]

            index = i

            unvisited.insert(i, 0)

    print("exit")

    return index, distance

roads = [(0, 1, 4), (0, 3, 2), (0, 2, 3), (2, 3, 2), (3, 0, 3)]

roadgraph = RoadGraph(roads)

vertices = roadgraph.vertices

print(vertices)

roadgraph.routing(0, 1, [2, 3])