In [2]:
```python
import random
import timeit
import matplotlib.pyplot as plt
import numpy as np
import math
```

Part 1.1

Dijkstra's algorithm

In [3]:
```python
def dijkstra(graph, source, k):
    distances = dict()
    for node in graph:
        distances[node] = float('inf')
    distances[source] = 0
    intermediate_nodes = dict()
    for node in graph:
        intermediate_nodes[node] = []
    #Now we create a dictionary in which we keep a track of the number of
    relaxations = dict()
    for node in graph:
        relaxations[node] = 0


    pq = set(graph.keys()) #Adding all the initial nodes of the graph

    if k > len(graph)-1 or k < 0:
        raise ValueError("The value of K is always between 0 and N-1")

    while pq:
        min_node = min(pq)  #We first get the node with the minimum cost
        pq.remove(min_node)

        for adj_node in graph[min_node]:
            current_cost = distances[min_node] + graph[min_node][adj_node]
            if relaxations[min_node] >= k:
                break
            if current_cost < distances[adj_node]:
                distances[adj_node] = current_cost
                intermediate_nodes[adj_node] = intermediate_nodes[min_node]
                relaxations[adj_node] = relaxations[min_node] + 1
                pq.add(adj_node)

    ans = (distances, intermediate_nodes)
    return ans
```

Part 1.2

Bellmann Ford Algorithm

In [15]:

```
def bellman_ford(graph, source, k):
    distances = dict()
    for node in graph:
        distances[node] = float('inf')
    distances[source] = 0
    intermediate_nodes = dict()
    for node in graph:
        intermediate_nodes[node] = []
    #Now we create a dictionary in which we keep a track of the number of
    relaxations = dict()
    for node in graph:
        relaxations[node] = 0
    # Iterate over the vertices in the graph
    for _ in range(len(graph)):
        for node in graph:
            for adj_node, weight in graph[node].items():
                if relaxations[adj_node] > k:    #To make sure that the nu
                    intermediate_nodes[adj_node] = []  # Set predecessors 
                    distances[adj_node] = float('inf')  # Set distance to 
                else:
                    current_cost = distances[node] + weight
                    if current_cost < distances[adj_node]:
                        distances[adj_node] = current_cost
                        intermediate_nodes[adj_node] = intermediate_nodes[
                        relaxations[adj_node] = relaxations[node] + 1


    ans = (distances, intermediate_nodes)
    return ans
```

Test cases

In [16]:

```
graph = {
    'A': {'B': 2, 'C': 10},
    'B': {'D': 7, 'E': 1},
    'C': {'B': 1, 'D': 3},
    'D': {'E' : 2},
    'E': {}
}
source_node = 'A'
k = 1
ans = dijkstra(graph, source_node, k)
ans1 = bellman_ford(graph, source_node, k)
print(ans1)
print(ans==ans1)
```

```
({'A': 0, 'B': 2, 'C': 10, 'D': inf, 'E': inf}, {'A': [], 'B': ['A'],
'C': ['A'], 'D': [], 'E': []})
True
```

In [18]:
```python
graph = {
    'A': {'C': 44},
    'B': {'D': 7, 'E': 1},
    'C': {'B': 1},
    'D': {'E' : 2},
    'E': {'A': 5}
}
source_node = 'A'
k = 4
ans = dijkstra(graph, source_node, k)
ans1 = bellman_ford(graph, source_node, k)
print(ans1)
print(ans==ans1)
```

```
({'A': 0, 'B': 45, 'C': 44, 'D': 52, 'E': 46}, {'A': [], 'B': ['A', 'C'],
'C': ['A'], 'D': ['A', 'C', 'B'], 'E': ['A', 'C', 'B']})
True
```

Part 1.3

Experiments for comparing the Dijkstra's and Bellman Ford Algorithm

In [19]:
```python
def draw_plot(run_arr):
    x = np.arange(0, len(run_arr),1)
    fig=plt.figure(figsize=(20,8))
    plt.bar(x,run_arr)
    plt.axhline(np.mean(run_arr),color="red",linestyle="--",label="Avg")
    plt.xlabel("Number of runs")
    plt.ylabel("Time taken to find the shortest path")
    plt.title("Run time for retrieval")
    plt.show()
```

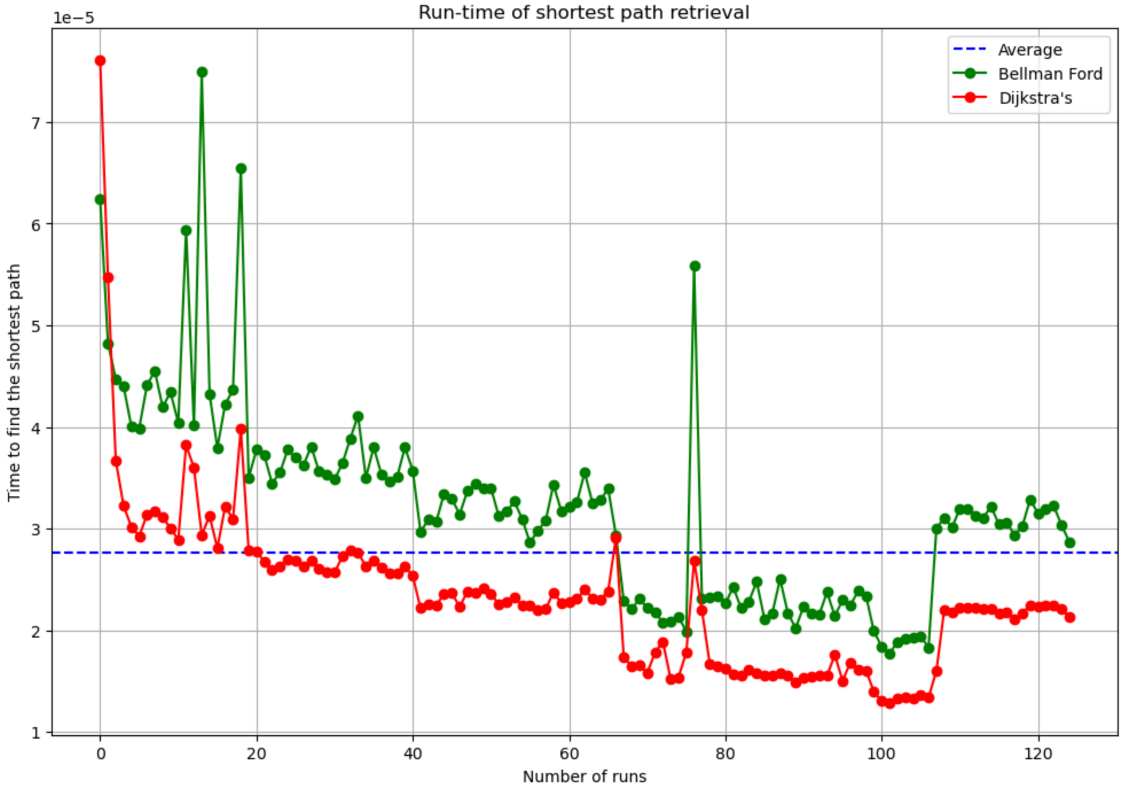
In [21]:
```python
def create_random_graph(nodes):
    graph = dict()

    for i in range(1, nodes + 1):
        graph[i] = dict()

    for i in range(1, nodes + 1):
        num_edges = random.randint(1, min(3, nodes - 1))  # Random number 
        edges = random.sample(range(1, nodes + 1), num_edges)  # Randomly 

        for edge in edges:
            if edge != i:
                weight = random.randint(1, 30)  # Random edge weight
                graph[i][edge] = weight

    return graph
```

In [20]:
```python
def limited_density(allowed,num_nodes):
    graph = {}

    for i in range(1, num_nodes + 1):
        graph[i] = {}

    for i in range(1, num_nodes + 1):
        max_edges = (num_nodes*(num_nodes - 1)) // 2
        num_edges = int((allowed/num_nodes)*max_edges)
        edges = random.choices(range(1, num_nodes + 1), k=num_edges)  # Ra

        for edge in edges:
            if edge != i:
                weight = random.randint(1, 30)  # Random edge weight
                graph[i][edge] = weight

    return graph
```

On the basis of the number of runs

In [667]:

```
runs = 125
source_node = 0
k = 3
run_timeDij = []
run_timeBel = []
x_list = []
for i in range(runs):
    x_list.append(i)
    graph = create_random_graph(20)
    dijkstra_graph = graph.copy()
    start_dij = timeit.default_timer()
    distances = dijkstra(dijkstra_graph, source_node, k)
    stop_dij = timeit.default_timer()
    run_timeDij.append(stop_dij-start_dij)

    bellman_graph = graph.copy()
    start_bel = timeit.default_timer()
    distances1 = bellman_ford(bellman_graph, source_node, k)
    stop_bel = timeit.default_timer()
    run_timeBel.append(stop_bel-start_bel)

plt.figure(figsize=(12,8))

average_y = (sum(run_timeBel) + sum(run_timeDij)) / (len(run_timeDij)*2)
plt.axhline(y=average_y, color='blue', linestyle='--', label='Average')

plt.xlabel("Number of runs")
plt.ylabel("Time to find the shortest path")
plt.title("Run-time of shortest path retrieval")
plt.grid(True, linestyle='-', alpha=1)

plt.plot(x_list, run_timeBel,marker="o",color="green",label="Bellman Ford"
plt.plot(x_list, run_timeDij,marker="o",color="red",label="Dijkstra's")

plt.legend()
plt.show()
```

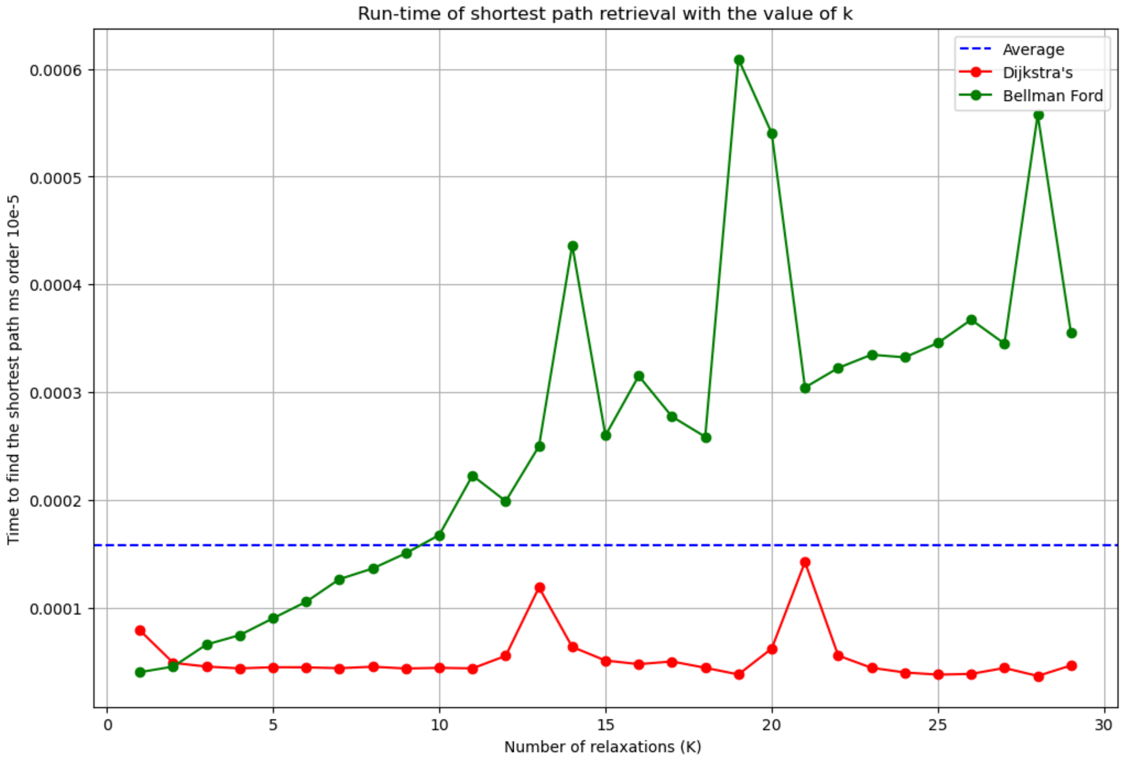


On the basis of the number of relaxations

In this we are creating a graph and for every iteration of the graph we are increasing the value of K (i.e the number of relaxations) and for every value of K we are finding the time for a fixed number of runs and the plotting the values on the basis of the average values.

In [670]:

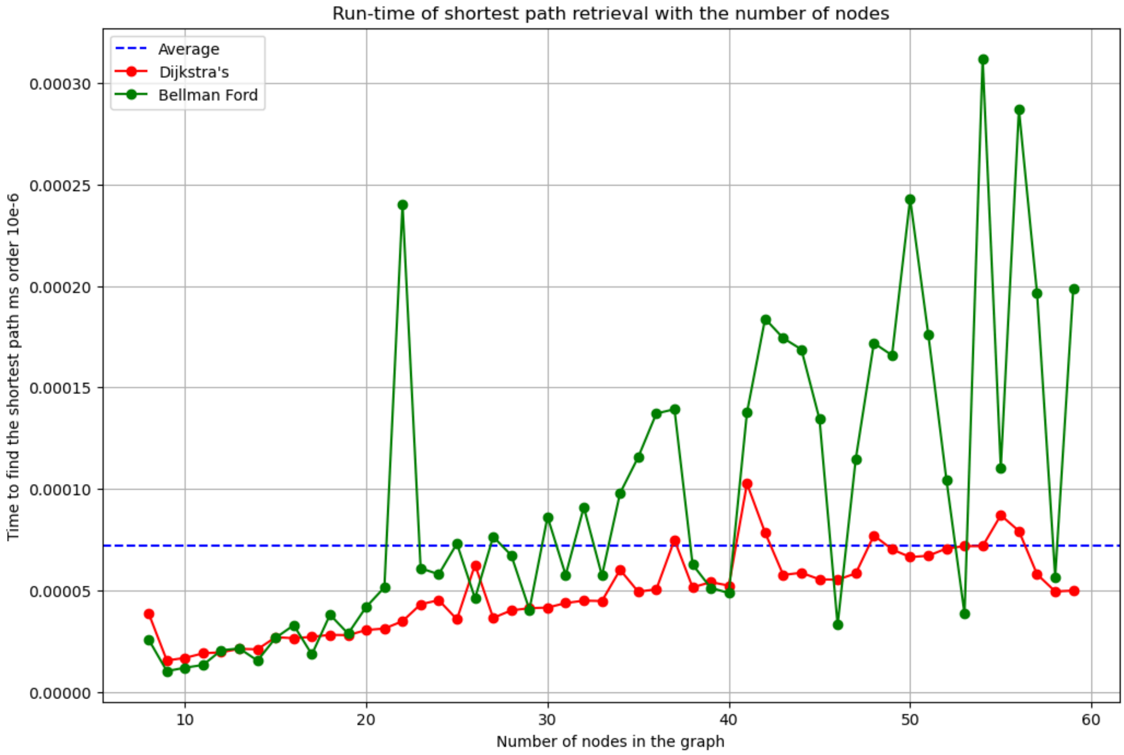
```
#With the value of k

runs = 10
graph = create_random_graph(30)

source_node = 2
k_values= []
run_timeDij = []
run_timeBel = []
for k in range(1,len(graph)):   #Calculates the average of 100 runs for eac
    k_values.append(k)

    dijkstra_graph = graph.copy()
    start_dij = timeit.default_timer()
    distances = dijkstra(dijkstra_graph, source_node, k)
    stop_dij = timeit.default_timer()
    run_timeDij.append(stop_dij-start_dij)


    bellman_graph = graph.copy()
    start_bel = timeit.default_timer()
    distances1 = bellman_ford(bellman_graph, source_node, k)
    stop_bel = timeit.default_timer()
    run_timeBel.append(stop_bel-start_bel)


plt.figure(figsize=(12,8))

average_y = (sum(run_timeBel) + sum(run_timeDij)) / (len(run_timeDij)*2)
plt.axhline(y=average_y, color='blue', linestyle='--', label='Average')

plt.xlabel("Number of relaxations (K) ")
plt.ylabel("Time to find the shortest path ms order 10e-5")
plt.title("Run-time of shortest path retrieval with the value of k")
plt.grid(True, linestyle='-', alpha=1)

plt.plot(k_values, run_timeDij,marker="o",color="red",label="Dijkstra's")

plt.plot(k_values, run_timeBel,marker="o",color="green",label="Bellman Ford
plt.legend()
plt.show()
```



Run-time of shortest path retrieval with the value of k
Time to find the shortest path ms order 10e-5
Number of relaxations (K)
Average
Dijkstra's
Bellman Ford
0.0001
0.0002
0.0003
0.0004
0.0005
0.0006
0
5
10
15
20
25
30

On the basis of the size of the graph

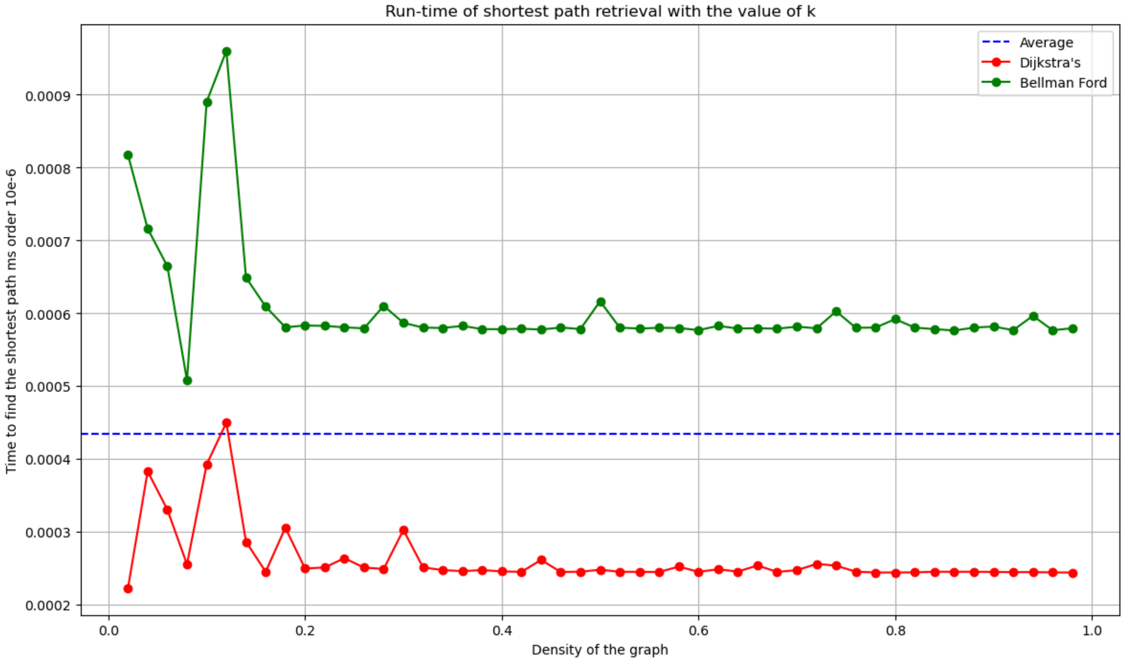
In [686]:
```
#With the value of k = 2

runs = 20
dijkstra_graph = graph.copy()
bellman_graph = graph.copy()
source_node = 0
k_values= []
no_of_nodes = 60
run_timeDij = []
run_timeBel = []
for n in range(8,no_of_nodes):
    k_values.append(n)  #This is the number of nodes
    k = random.randrange(1,n//4)
    graph_new = generate_random_graph(n)

    dijkstra_graph = graph_new.copy()
    start_dij = timeit.default_timer()
    distances = dijkstra(dijkstra_graph, source_node, k)
    stop_dij = timeit.default_timer()
    run_timeDij.append(stop_dij-start_dij)


    bellman_graph = graph_new.copy()
    start_bel = timeit.default_timer()
    distances1 = bellman_ford(bellman_graph, source_node, k)
    stop_bel = timeit.default_timer()
    run_timeBel.append(stop_bel-start_bel)


plt.figure(figsize=(12, 8))

average_y = (sum(run_timeBel) + sum(run_timeDij)) / (len(run_timeDij)*2)
plt.axhline(y=average_y, color='blue', linestyle='--', label='Average')

plt.xlabel("Number of nodes in the graph")
plt.ylabel("Time to find the shortest path ms order 10e-6")
plt.title("Run-time of shortest path retrieval with the number of nodes")
plt.grid(True, linestyle='-', alpha=1)

plt.plot(k_values, run_timeDij,marker = "o",color="red",label="Dijkstra's"

plt.plot(k_values, run_timeBel,marker = "o",color="green",label="Bellman F
plt.legend()
plt.show()
```



Run-time of shortest path retrieval with the number of nodes
Average
Dijkstra's
Bellman Ford
Time to find the shortest path ms order 10e-6
Number of nodes in the graph
0.00000
0.00005
0.00010
0.00015
0.00020
0.00025
0.00030
10
20
30
40
50
60

On the basis of the density of the graph

In [687]:
```
#With the value of k = 2

runs = 50
source_node = 0
k = 3
k_values= []
no_of_nodes = 50
run_timeDij = []
run_timeBel = []
for n in range(1,runs):
    k_values.append((n/runs))  #This is the number of nodes
    graph = limited_density(n,no_of_nodes)  #In this we are creating graphs
    #of increasing densities

    dijkstra_graph = graph.copy()
    start_dij = timeit.default_timer()
    distances = dijkstra(dijkstra_graph, source_node, k)
    stop_dij = timeit.default_timer()
    run_timeDij.append(stop_dij-start_dij)


    bellman_graph = graph.copy()
    start_bel = timeit.default_timer()
    distances1 = bellman_ford(bellman_graph, source_node, k)
    stop_bel = timeit.default_timer()
    run_timeBel.append(stop_bel-start_bel)

plt.figure(figsize=(14, 8))


average_y = (sum(run_timeDij) + sum(run_timeBel)) / (len(run_timeDij)*2)
plt.axhline(y=average_y, color='blue', linestyle='--', label='Average')

plt.xlabel("Density of the graph")
plt.ylabel("Time to find the shortest path ms order 10e-6")
plt.title("Run-time of shortest path retrieval with the value of k")
plt.grid(True, linestyle='-', alpha=1)

plt.plot(k_values, run_timeDij,marker="o",color="red",label="Dijkstra's")

plt.plot(k_values, run_timeBel,marker="o",color="green",label="Bellman Ford
plt.legend()
plt.show()
```

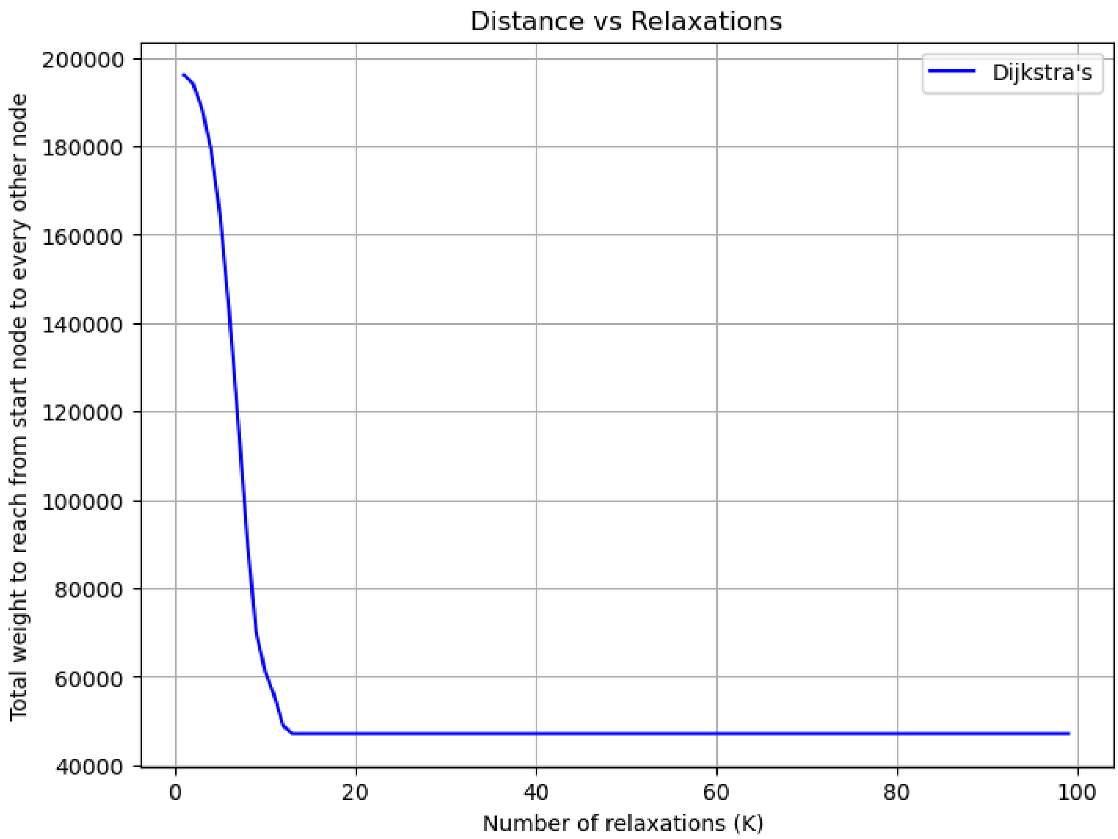


Run-time of shortest path retrieval with the value of k
Average
Dijkstra's
Bellman Ford
Time to find the shortest path ms order 10e-6
Density of the graph
0.0002
0.0003
0.0004
0.0005
0.0006
0.0007
0.0008
0.0009
0.0
0.2
0.4
0.6
0.8
1.0

Comparing the Accuracy

Dijkstra's algorithm

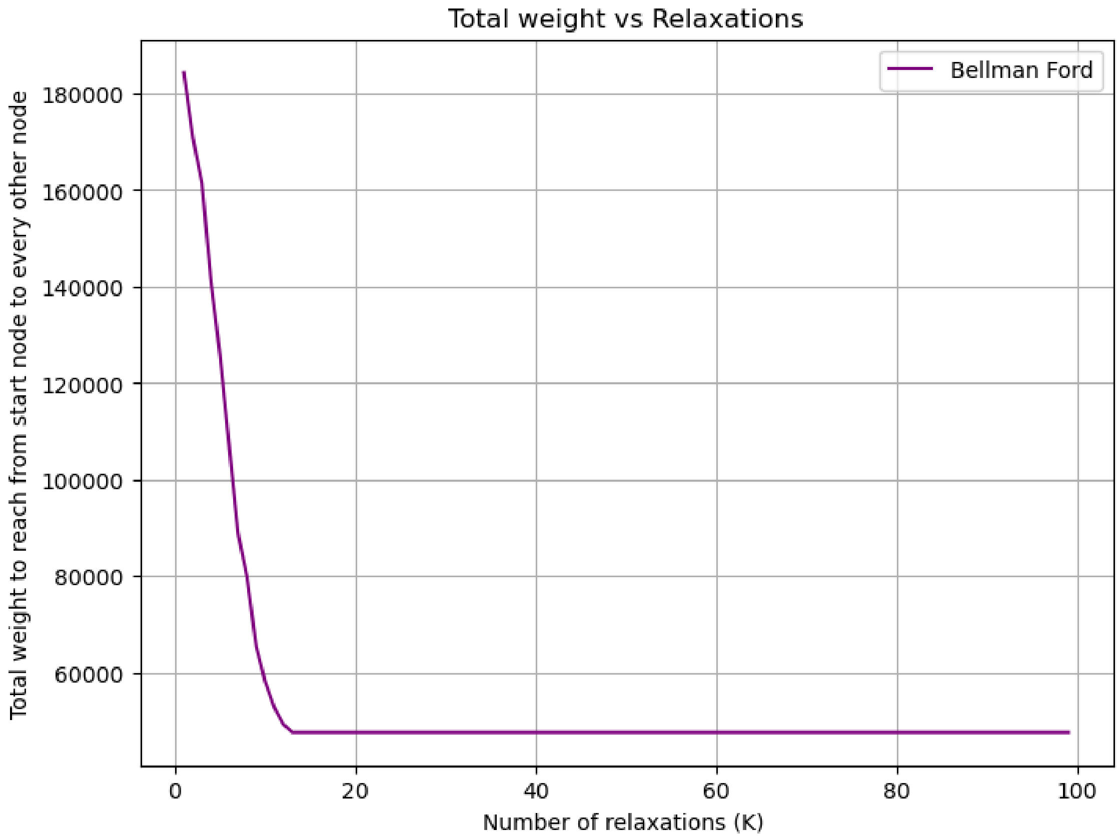
In [388]:
```
graph = create_random_graph(100)

source_node = 1

k_values= []
run_timeDij = []
for k in range(1,len(graph)):
    k_values.append(k)
    # distances = dijkstra(graph, source_node, k)
    distances1 = dijkstra(graph, source_node, k)
    val = list(distances1[0].values())
    for i in range (len(val)):
        if (val)[i]==float('inf'):
            (val)[i]=2000
    run_timeDij.append(sum(val))

plt.figure(figsize=(8,6))


plt.xlabel("Number of relaxations (K) ")
plt.ylabel("Total weight to reach from start node to every other node")
plt.title("Total weight vs Relaxations")
plt.grid(True, linestyle='-', alpha=1)

plt.plot(k_values, run_timeDij,color="blue",label="Dijkstra's")

plt.legend()
plt.show()
```



Distance vs Relaxations
Dijkstra's
Total weight to reach from start node to every other node
Number of relaxations (K)
200000
180000
160000
140000
120000
100000
80000
60000
40000
0
20
40
60
80
100

Bellman Ford Algorithm

In [392]:
```
graph = create_random_graph(100)

source_node = 1

k_values= []
run_timeDij = []
for k in range(1,len(graph)):
    k_values.append(k)
    # distances = dijkstra(graph, source_node, k)
    distances1 = bellman_ford(graph, source_node, k)
    val = list(distances1[0].values())
    for i in range (len(val)):
        if (val)[i]==float('inf'):
            (val)[i]=2000
    run_timeDij.append(sum(val))

plt.figure(figsize=(8,6))



plt.xlabel("Number of relaxations (K) ")
plt.ylabel("Total weight to reach from start node to every other node")
plt.title("Total weight vs Relaxations")
plt.grid(True, linestyle='-', alpha=1)

plt.plot(k_values, run_timeDij,color="purple",label="Bellman Ford")

plt.legend()
plt.show()
```



Part 2

Dijkstra's and Bellman Ford's are single source shortest path algorithms.However, many times we are faced with problems that require us to solve shortest path between all pairs. This means that the algorithm needs to find the shortest path from every possible source to every possible destination. For every pair of vertices u and v, we want to compute shortest path d$istance(u, v)$ and the second-to-last vertex on the shortest path $previous\ (u, v)$.

How would you design an all-pair shortest path algorithm for both positive edge weights and negative edge weights? Implement a function that can address this. Dijkstra has complexity $\Theta(E + VlogV)$, or $\Theta\ (V2)$ if the graph is dense and Bellman-Ford has complexity $\Theta\ (VE)$, or $\Theta(V3)$ if the graph is dense.

Knowing this, **what would you conclude the complexity of your two algorithms to be for dense graphs?** Explain your conclusion in your report. You do not need to verify this empirically.

In [27]:
```
import random

def generate_random_graph(num_nodes):
    graph = {}

    for i in range(1, num_nodes + 1):
        graph[i] = {}

    for i in range(1, num_nodes + 1):
        num_edges = random.randint(1, min(3, num_nodes - 1))  # Random numb
        edges = random.sample(range(1, num_nodes + 1), num_edges)  # Rando

        for edge in edges:
            if edge != i:
                weight = random.randint(0, 20)  # Random edge weight
                graph[i][edge] = weight

    return graph

def generate_random_graph_2(num_nodes):
    graph = {}

    for i in range(1, num_nodes + 1):
        graph[i] = {}

    for i in range(1, num_nodes + 1):
        num_edges = random.randint(1, min(3, num_nodes - 1))  # Random numb
        edges = random.sample(range(1, num_nodes + 1), num_edges)  # Rando

        for edge in edges:
            if edge != i:
                weight = random.randint(-20, 30)  # Random edge weight, neg
                graph[i][edge] = weight

    return graph
```

(2.1) FOR POSITIVE EDGE-WEIGHTS

In [28]:
```
def all_pair_shortest_paths_positive(graph, k):
    all_shortest_paths = {}
    all_second_to_last = {}

    # Helper function to perform Dijkstra's algorithm
    def dijkstra_2(graph, start):
        distances = {node: float('inf') for node in graph}
        distances[start] = 0
        visited = set()

        while len(visited) < len(graph):
            min_distance = float('inf')
            min_node = None
            for node in graph:
                if node not in visited and distances[node] < min_distance:
                    min_distance = distances[node]
                    min_node = node

            if min_node is None:
                break

            visited.add(min_node)

            for neighbor, weight in graph[min_node].items():
                new_distance = distances[min_node] + weight
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance

        return distances

    # Iterate through each node in the graph
    for source in graph:
        distances, previous_vertices = dijkstra(graph, source, k)
        all_shortest_paths[source] = distances
        # Extractin second-to-last vertices from previous_vertices
        second_to_last = {}
        for vertex in graph:
            second_to_last[vertex] = None
        for destination, prev in previous_vertices.items():
            if prev:
                second_to_last[destination] = prev[-1]
        all_second_to_last[source] = second_to_last


    return all_shortest_paths, all_second_to_last
```

Test case 2.1

In [29]:
```
# Example usage:

k = 2
x = generate_random_graph(4)
print("Graph:\n", x)
all_shortest_paths, all_second_to_last = all_pair_shortest_paths_positive(x

print("Shortest Paths:")
for source, distances in all_shortest_paths.items():
    print(f"From node {source}: {distances}")

print("\nSecond-to-Last Nodes:")
for source, second_to_last in all_second_to_last.items():
    print(f"For node {source}: {second_to_last}")
```

```
Graph:
 {1: {3: 19, 4: 6}, 2: {3: 7, 1: 9}, 3: {4: 12, 2: 2}, 4: {1: 13, 3: 5, 
2: 14}}
Shortest Paths:
From node 1: {1: 0, 2: 20, 3: 11, 4: 6}
From node 2: {1: 9, 2: 0, 3: 7, 4: 15}
From node 3: {1: 11, 2: 2, 3: 0, 4: 12}
From node 4: {1: 13, 2: 7, 3: 5, 4: 0}

Second-to-Last Nodes:
For node 1: {1: None, 2: 4, 3: 4, 4: 1}
For node 2: {1: 2, 2: None, 3: 2, 4: 1}
For node 3: {1: 2, 2: 3, 3: None, 4: 3}
For node 4: {1: 4, 2: 3, 3: 4, 4: None}
```

(2.2) FOR NEGATIVE EDGE-WEIGHTS

In [30]:
```python
def all_pair_shortest_paths_negative(graph, k):
    vertices = list(graph.keys())
    edges = [(u, v, w) for u in graph for v, w in graph[u].items()]
    n = len(vertices)

    all_shortest_paths = {}
    all_second_to_last = {}

    # Bellman-Ford algorithm
    def bellman_ford2(graph, start, k):
        distances = {node: float('inf') for node in graph}
        distances[start] = 0
        previous_vertices = {node: [] for node in graph}

        for _ in range(k):
            for u, v, w in edges:
                if distances[u] + w < distances[v]:
                    distances[v] = distances[u] + w
                    previous_vertices[v] = [u]

                elif distances[u] + w == distances[v]:
                    previous_vertices[v].append(u)

        # Check for negative cycles
        for u, v, w in edges:
            if distances[u] + w < distances[v]:
                raise ValueError("Graph contains a negative cycle")

        return distances, previous_vertices

    # Iterate through each node in the graph
    for source in vertices:
        distances, previous_vertices = bellman_ford2(graph, source, k)
        all_shortest_paths[source] = distances

        # Extract second-to-last nodes
        second_to_last = {}
        for destination in vertices:
            second_to_last[destination] = None
            if previous_vertices[destination]:
                second_to_last[destination] = previous_vertices[destinatio
        all_second_to_last[source] = second_to_last

    return all_shortest_paths, all_second_to_last
```

Test Case 2.2

In [31]:
```
k = 2
x = generate_random_graph_2(3)
print("Graph:\n", x)

all_shortest_paths, all_second_to_last = all_pair_shortest_paths_negative(x

# Print the results
print("Shortest Paths:")
for source, distances in all_shortest_paths.items():
    print(f"From node {source}: {distances}")

print("\nSecond-to-Last Nodes:")
for source, second_to_last in all_second_to_last.items():
    print(f"For node {source}: {second_to_last}")
```

```
Graph:
 {1: {2: 27}, 2: {3: 11}, 3: {1: 3}}
Shortest Paths:
From node 1: {1: 0, 2: 27, 3: 38}
From node 2: {1: 14, 2: 0, 3: 11}
From node 3: {1: 3, 2: 30, 3: 0}

Second-to-Last Nodes:
For node 1: {1: None, 2: 1, 3: 2}
For node 2: {1: 3, 2: None, 3: 2}
For node 3: {1: 3, 2: 1, 3: None}
```

The **all_pair_shortest_paths_negative** function computes the shortest paths between all pairs of nodes in a graph where edge weights may be negative. It employs the Bellman-Ford algorithm to handle negative weights and identify negative cycles. Here's a concise summary of its functionality:

It initializes variables to represent the graph's vertices and edges, and creates empty dictionaries to store shortest paths and second-to-last vertices.

The code defines a helper function **bellman_ford2** which implements Bellman Ford's algorithm and computes the shortest paths from a given source node to all other nodes in the graph. This algorithm runs for a specified number of iterations (k), updating distances and previous vertices along the way.

After the iterations, the algorithm checks for negative cycles by iterating over all edges again. If it finds a shorter path, it indicates the presence of a negative cycle.

The function returns nested dictionaries containing the shortest paths and their second-to-last vertices for each source node in the graph.

Complexity of algorithm:

The algorithm, "all_pair_shortest_paths_negative(graph, k)", computes all-pair shortest paths in a dense graph with positive and negative edge weights. It utilizes Bellman-Ford's algorithm as a medium to compute shortest paths from one node to all other nodes and then returns the second-to-last vertices from the paths obtained.

The complexity of the algorithm can be analyzed as follows:

For each node in the graph, the algorithm calls Bellman-Ford's algorithm. In a dense graph, Bellman-Ford's algorithm complexity is approximately **Θ(VE)**, which simplifies to **Θ(V^3)** since E can be at most **V^2**. Hence, calling Bellman-Ford's algorithm for every single node to calculate the shortest path gives a total complexity of **Θ(V^4)**.

After obtaining the shortest paths for each node, the algorithm extracts the second-to-last vertices from the paths obtained. This involves iterating through the previous vertices for each destination node, which takes **Θ(V)** time.

Thus, the algorithm's overall complexity remains **Θ(V^4)** for computing all-pair shortest paths in a dense graph with negative edge weights.

Part 3

A_Star algorithm

In [44]:
```python
from math import *

class PriorityQueue:
    def __init__(self):
        self.elements = []

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

    def insert(self, item, priority):
        self.elements.append((item, priority))


    def get(self):
        index = 0
        V = len(self.elements)
        for i in range(V):
            if self.elements[i][1] < self.elements[index][1]:
                index = i
        return self.elements.pop(index)[0]


#This function calculates the heuristic cost
def sup2h(graph, destination):    #This function finds the heuristic cost b
    h = dict()
    for node in graph:
        a1, b1 = node
        a2,b2 = destination
        h[node] = sqrt((a2-a1)**2 + (b2-b1)**2)
    return h


def A_Star(graph, start, destination, heuristic):
    pq = PriorityQueue()
    pq.insert(start, 0)
    predecessor_dictionary = dict()
    total_weight = dict()    #This is the dictionary which stores the total
    predecessor_dictionary[start] = None #The predecessor of the start_node
    total_weight[start] = 0


    while not pq.empty():
        current = pq.get()      #This will be the start node as initially w
        #therefore the first element to be popped is the start_node
        if current == destination:      #Incase the start and the destinati
            break

        for neighbor, weight in graph[current].items():
            #print(next_node)
            new_cost = total_weight[current] + weight
            if neighbor not in total_weight or new_cost < total_weight[neig
                total_weight[neighbor] = new_cost
                priority = new_cost + heuristic[neighbor]    #This is the c
                pq.insert(neighbor, priority)
                predecessor_dictionary[neighbor] = current
    #print("The predecessor dictionary:", came_from)
    # Reconstruct path if goal is reached
    if destination not in predecessor_dictionary:       #This is when there
        return None   # No path found

    current, path = destination, [current]
```

```python
    new_dict=dict()
    while current != start:
        current = predecessor_dictionary[current]
        path.append(current)
    path.reverse()
    answer = (predecessor_dictionary, path)
    return answer
```

Test Case

In [48]:
```python
graph = {
    (0, 0): {(1, 1): 3, (3,3): 1},
    (1, 1): {(0, 0): 4, (2, 2): 6},
    (2, 2): {(1, 1): 1, (3, 3): 2},
    (3, 3): {(2, 2): 2, (4, 4): 3},
    (4, 4): {}
}

start_node = (0, 0)
dest_node = (3, 3)
heuristic = sup2h(random_graph,dest_node)
ans = A_Star(graph, start_node, dest_node, heuristic)
print(ans)
```

```
({(0, 0): None, (1, 1): (0, 0), (3, 3): (0, 0)}, [(0, 0), (3, 3)])
```