FINAL PROJECT:

FINDING THE CHEAPEST FLIGHT FOR THE MINERVANS

CS110: Solving Problems with Algorithms
Minerva Schools at KGI

Khanh Nguyen
December 2019

FINAL PROJECT

FINDING THE CHEAPEST FLIGHT FOR THE MINERVANS

Summary

In this project, we are interested in finding the cheapest flight for the Minervans. Because Minervans are all students and need to travel to a new city every 4 months, they always need to find the cheapest flight to save the cost. Therefore, the goal of this project is to examine which of the existing graph algorithms would be best to find the cheapest flight for the Minervans[1]. We examine 3 graph algorithms: Bellman-Ford, Dijkstra's, and Floyd-Warshall algorithm with a few test cases of the actual flight ticket from Seoul to Hyderabad, the cities that the Minervans need to travel this semester. To see whether the algorithm can be scaled up to a certain number of cities, we plot the running time of each algorithm on the same graph. Dijkstra's algorithm proves to be the most time-efficient, however, it can only examine the cheapest flight from one city to other cities in the graph. Floyd-Warshall algorithm is more versatile to explore the cheapest flight between any pairs of cities in the graph, but the running time scales up pretty fast. The extension of this project would be to extend the Dijkstra's algorithm to work with any pairs of cities in the graph and compare its running time with the Floyd-Warshall algorithm.

Problem statement

The goal of this project is to find the cheapest flight between two cities in the graph. The motivation for this comes from the fact that I am a student at Minerva, which enables us to

[1] #context: The problem can also be known as finding the shortest path between two nodes in a graph. However, I situate this problem in the Minerva context, in which we need to travel to come up with this problem statement.

study in 7 cities in the course of 4 years. Because of the excessive traveling, we always need to find the cheapest flight that would fit our student budget. We have the assumption that direct flight is more expensive than flights with one or multiple layovers. For example, this Spring the Minervans are traveling from Seoul, South Korea to Hyderabad, India. Even though some students are taking a direct route from Seoul to Hyderabad, others have found cheaper flight tickets with a layover in Singapore, Kuala Lumpur (Malaysia), or Hanoi (Vietnam)! Therefore, given a graph with the cities as the nodes and the cost of flying as the edges between the nodes, we want to find an algorithm that helps us find the cheapest flight.

To generalize the problem statement, we can also implement such an algorithm to find the cheapest traveling cost by other transportation, such as buses, to other parts in the cities, between cities or other countries.

Approach and implementation

[#PythonProgramming, #CodeReadability, #DynamicProgramming, #GreedyAlgorithm]

In this report, we define a graph as a collection of nodes and the paths (edges) between the nodes. Because of the nature of the problem above, we will only work with directed graphs, which are graphs with direction, such as a path from node A to node B is different from node B to node A. Each node can represent a city that we travel to, and the path between each node will bear a weight, which is the cost of flying from one city to another.

To solve the problem above, we will examine the performance of 3 graph algorithms: Bellman-Ford, Dijkstra's and Floyd-Warshall algorithm. Each algorithm is used to find the shortest path in a graph, which is also similar to finding the cheapest route, but the mechanism of each one is different.

- Bellman-Ford algorithm is used to find the shortest path between one source node, or start node[2], to the target node. The algorithm runs through each node in the graph, and for each node, it explores each edge that node can travel. From node u to node v, if there exists a less costly path from node u to node v, then we update the distance array that stores the traveling cost. From this feature, the Bellman-Ford algorithm is a dynamic programming algorithm because it exhibits the optimal substructure and overlapping subproblems. The optimal substructure is that from node 1 to node n we can have the shortest path as (1, 2, 3, ..., n), and we have node i, j such that $1 \leq i \leq j \leq n$\$. Then, we would have the problem of finding the shortest path from i to j. The overlapping subproblem is that we need to find the shortest path from node i to j in both cases just mentioned.

- Dijkstra's algorithm is also used to find the shortest path from the starting node to other nodes in the graph. However, Dijkstra's uses an additional data structure, priority queue, to prioritize the edge with the shortest distance first. Because Dijkstra's algorithm always goes to the next node which has the lowest cost to travel, yet it still gives the global optimal solution, we would say Dijkstra is a greedy algorithm.

- In the case of the Floyd-Warshall algorithm, this algorithm can find the shortest path between any pair of nodes in the graph by running 3 for loops and update the distance if a shorter path is found. Similarly to the description of the Bellman-Ford algorithm above, the Floyd-Warshall algorithm is also a dynamic programming algorithm because it exhibits the optimal substructure and overlapping subproblems. The optimal

[2] In this report, the term "source node" or "start node" is used interchangeably. Both terms refer to the node from which we start finding the shortest path from

substructure is that from node 1 to node n we can have the shortest path as (1, 2, 3, ..., n), and we have node i, j such that $1 \leq i \leq j \leq n$. Then, we would have the problem of finding the shortest path from i to j. The overlapping subproblem is that we need to find the shortest path from node i to j in both cases just mentioned.

Data preparation

Each algorithm has the same input and output. However, each one requires different types of data structures. More details about the similarities and differences are presented in the table below:

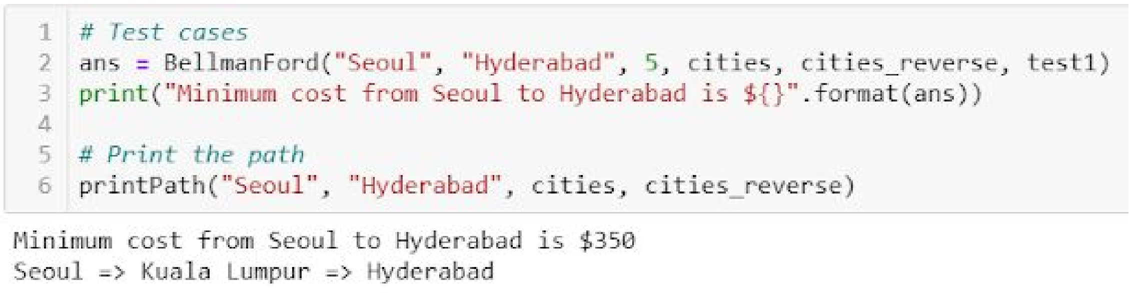
	Bellman-Ford algorithm	**Dijkstra's algorithm**	**Floyd-Warshall algorithm**
Data input	● Starting node ● Target node ● Number of nodes in the graph ● Edge list to represent the path and cost of each path from one node to another		
Data output	Shortest distance from the starting node to the target node	Shortest distance from the starting node to the target node	Can find the shortest distance between any pair of nodes
Type of graph	Directional Non-directional Positive/Negative weights	Directional Non-directional Positive weights	Directional Positive/Negative weights No negative cycle
Data structures	One-dimensional array for distance Edge list or adjacency matrix	One-dimensional array for distance Edge list or adjacency matrix Priority queue to save the pair of node and its weight	Two-dimensional array Edge list or adjacency matrix
Complexity	O(Edges*Vertices)	O(Edges*Log(Vertices)	O(Vertices^3)

Table 1: Comparison between Bellman-Ford algorithm, Dijkstra's algorithm and Floyd-Warshall algorithm

Back to the problem of finding the cheapest flight, we have a list of 9 cities in the graph: Seoul, Hyderabad, Ho Chi Minh, Kuala Lumpur, Singapore, Bangkok, Tokyo, Hongkong, and Mumbai. Because we need to store the information in the data structure of Python list for the graph and distance, we then translate these 9 cities into ID numbers as in Appendix 1.

We have 3 test cases for these 9 cities, which can be found in Appendix 2. The test case is presented as an edge list, with the format of ("start node, "end node", "cost of traveling in USD"). The data is the actual price from Google Flights and rounded to the nearest multiple of 5. For example, one edge in the edge list is ("Seoul", "Bangkok", 315), which means that the path is from Seoul to Bangkok, and the flight ticket is 315 USD. We use the same 3 test cases for 3 algorithms we're working with.

While we can use the test cases to determine the accuracy of the algortihm, Three test cases is not rigorous enough to check whether the algorithm is comprehensive to cover all of the cases. However, we are using the test case for another purpose, which is to demonstrate the path from one city to another. The photo below is the example of Bellman-Ford algorithm finding cheapest flight from Seoul to Hyderabad (350 USD), and the path is Seoul, Kuala Lumpur, Hyderabad (Appendix 3 - Bellman-Ford algorithm).



```
# Test cases
ans = BellmanFord("Seoul", "Hyderabad", 5, cities, cities_reverse, test1)
print("Minimum cost from Seoul to Hyderabad is ${}".format(ans))

# Print the path
printPath("Seoul", "Hyderabad", cities, cities_reverse)
```

```
Minimum cost from Seoul to Hyderabad is $350
Seoul => Kuala Lumpur => Hyderabad
```

Figure 1: A code snippet showing the result of the Bellman-Ford algorithm and its path from the start city (Seoul) to the target city (Hyderabad)

Running time analysis [#ComplexityAnalysis, #ComputationalCritique]

Because the 3 algorithms passes all the 3 test cases provided, we will assume that the performance of the algorithms is acceptable. The next step to is analyze the running time of each algorithms with respect to growth in the input size.

Theoretical analysis

We use E to represent the number of edges and V to represent the number of nodes/vertices.

- Bellman-Ford: O(E*V)

- Step 1: Prepare the data structures: We need to prepare a one-dimensional graph from the edge list and a one dimensional distance array. Creating the graph takes O(E) with E as the number of the edges, and creating the distance array takes O(V) with V as the number of vertices in the graph.

- Step 2: Loop through all the nodes and edges: This process take O(V*E) because this is a two-nested for loops.

After combining the running time of 2 steps above, we can generalize the running time of the algorithm as O(V*E) because this is the one with highest order of growth.

- Dijkstra's: O(E*Log(E))

- Step 1: Prepare the data structures: We need to prepare a two-dimensional graph from the edge list and a one dimensional distance array. Creating the graph takes O(E) and creating the distance array takes O(V).

- Step 2: Run the while loop with all the edges. Pushing all the edges to the priority queue takes O(E*Log(E)) time and extracting a node takes O(Log(E)) time.

After combining the running time of 2 steps above, we can generalize the running time of the algorithm as O(E*Log(E)) because this is the one with highest order of growth. We also need to note that this analysis is only pertain to the way we define the graph in the randomly generated graph in Appendix 4, because we define the number of edges to be around the square of number of nodes divided by two.

- Floyd-Warshall: O(V^3)

- Step 1: Prepare the data structures: We need to prepare a two-dimensional graph from the edge list and a two-dimensional distance array. Each of the process would take O(V^2)

- Step 2: Loop through all the nodes: This process takes O(V^3) because this is a three-nested for loops.

After combining the running time of 2 steps above, we can generalize the running time of the algorithm as O(V^3) because this is the one with highest order of growth.

Of the three running time above, Dijkstra's algorithm seems to outperform the other two. If we compare Dijkstra's with Bellman-Ford, log(E) is way smaller than V. And in the case of Floyd-Warshall algorithm, O(V^3) is way higher than O(E*Log(E)).

Experimental analysis

To examine the running time experimentally, we test the algorithm with randomly generated graphs with number of nodes ranging from 3 to 100 (Appendix 4). For each number of node, we run the algorithms for 50 times then average the results. As we can see from figure 2

[3], Dijkstra is the fastest algorithm with the running time rises only little as the size of input grows. The running time for Floyd-Warshall and Bellman-Ford is relatively the same and scales up pretty pretty fast after the size of input reaches 40. This can be explained because Floyd-Warshall algorithm has the running time of O(V^3), which scales exponentially. The running time of Bellman-Ford algorithm is O(E*V), but E = $\frac{V*(V-1)}{2}$, so its running time is only slightly shorter than Floyd-Warshall.

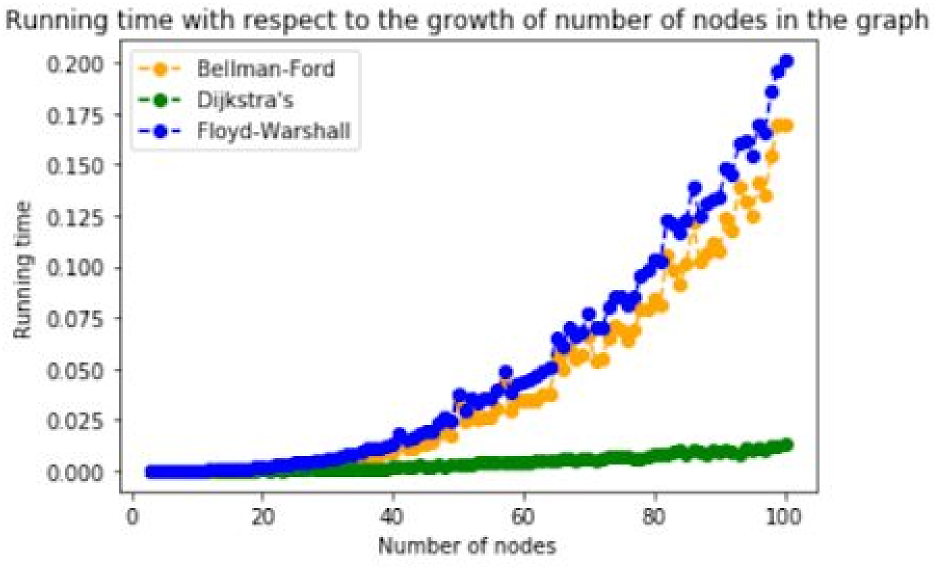


Figure 2: The running time of 3 algorithms with respect to the growth of number of nodes in the graph

Limitation

This is not a comprehensive approach to solve the problem of the cheapest flight, but within the time constraint I managed to implement the three graphs with 15 hours of coding. The algorithm would be more robust if I have more tests case to work with. I generated the graphs randomly by the number of nodes (Appendix 4) but I cannot check whether the graphs

[3] #dataviz: I visualize the running time of each algorithm and plot all of them in the same plot. Each running time is color-coded with a legend. The plot is clearly defined with the title and labeled-axes.

are valid. Also for this particular context, I do not include the check negative cycle in the Floyd-Warshall algorithm because I want to return the shortest distance.

Suggestion

Because Dijkstra's algorithm is the most time-efficient, we can experiment to extend it to work with any pairs of nodes in the graph. After that, we can compare its running time with the Floyd-Warshall algorithm to see which one performs better.

Conclusion[4]

To find the cheapest flight between the two cities, we implemented three graph algorithms: Bellman-Ford, Dijkstra's and Floyd-Warshall. Even though each algorithm manages to find the shortest distance from one node to the target node, the running time of each algorithm is different. Dijkstra's algorithm is the most time-efficient, but it can only examine the cheapest flight from one city to other cities in the graph. In contrast, Floyd-Warshall algorithm can find the cheapest flight between any pairs of cities in the graph, but the running time scales up pretty fast. The extension of this project would be to extend the Dijkstra's algorithm, which is the most time-efficient one, to work with any pairs of cities in the graph and compare its running time with the Floyd-Warshall algorithm.

List of applied LOs

#PythonProgramming: The Python code in this final project works well to supplement the written report. I did include test cases to both testing the performance of the algorithm and to demonstrate the effect of implementing such algorithm, which is to know the

[4] #organization: This report is well-organized into 8 parts: summary, problem statement, approach and implementation, data preparation, limitation, suggestion, conclusion. The report is explained in prose whereas the supporting Python code is in the appendix. Thanks to this structure, a busy reader can focus on the part of his or her interest and examine the code if needed.

cheapest way to fly and which route to take. I also plotted the running time to compare the performances of three algorithms.

#CodeReadability: I carefully commented all code and explained the reason why I include a particular command. The names of the variables are either self-explanatory or explained through the comment. Thanks to this, the readers would be able to follow and replicate the results.

#DynamicProgramming: Both Bellman-Ford and Floyd-Warshall algorithms are forms of dynamic programming. In this report, I explain them in terms of optimal substructure and overlapping subproblems, which are the two aspects of dynamic programming. I also implemented the working code for Bellman-Ford and Floyd-Warshall in the Python appendix.

#GreedyAlgorithm: Dijkstra's algorithm is a form of greedy algorithm. I explain the algorithm in terms of the greedy approach: it always goes to the next node which has the lowest cost to travel thanks to the priority queue. The algorithm still gives global optimal solution and not stuck in the local maxima, so Dijkstra is a greedy algorithm. I also implemented the working code for Dijkstra's algorithm in the Python appendix.

#ComplexityAnalysis: In section "Running time analysis", I analyzed the running time of each of the three algorithms with justification. The theoretical result is also confirmed by the experimental approach.

#ComputationalCritique: In section "Running time analysis", I plot the running time experimentally with randomly generated graphs to compare with the theoretical performance of the algorithms. I also include the section "Limitation" and "Suggestion" to critique what my final project has done well and what could be improved.

Appendix 1: Dictionary of cities

In [1]:

```
# Here is the list of cities
cities_list = ["Seoul", "Hyderabad", "Ho Chi Minh", "Kuala Lumpur",
              "Singapore", "Bangkok", "Tokyo", "Hongkong", "Mumbai"]

# Transform the list to dictionary so we can access id number of each city later
cities = {key: value for value, key in enumerate(cities_list)}
cities
```

Out[1]:

```
{'Seoul': 0,
 'Hyderabad': 1,
 'Ho Chi Minh': 2,
 'Kuala Lumpur': 3,
 'Singapore': 4,
 'Bangkok': 5,
 'Tokyo': 6,
 'Hongkong': 7,
 'Mumbai': 8}
```

In [2]:

```
# Create an inversed dictionary to translate the id number to city name later
cities_reverse = {value: key for key, value in cities.items()}
cities_reverse
```

Out[2]:

```
{0: 'Seoul',
 1: 'Hyderabad',
 2: 'Ho Chi Minh',
 3: 'Kuala Lumpur',
 4: 'Singapore',
 5: 'Bangkok',
 6: 'Tokyo',
 7: 'Hongkong',
 8: 'Mumbai'}
```

Appendix 2: Test cases

In [3]:

```
# 3 test cases to find the cheapest flight for Minervans
# The data is the actual price from Google Flights and rounded to the nearest multiple of 5
test1 = [("Seoul", "Bangkok", 315), ("Seoul", "Singapore", 360), ("Seoul", "Kuala Lumpur", 245),
         ("Bangkok", "Hyderabad", 280), ("Kuala Lumpur", "Hyderabad", 105), ("Singapore",
"Hyderabad", 250)]
test2 = [("Seoul", "Bangkok", 315), ("Seoul", "Singapore", 360), ("Seoul", "Kuala Lumpur", 245),
         ("Bangkok", "Hyderabad", 280), ("Kuala Lumpur", "Hyderabad", 105), ("Singapore",
"Hyderabad", 250),
         ("Seoul", "Ho Chi Minh", 200), ("Ho Chi Minh", "Singapore", 45), ("Bangkok", "Singapore",
70)]
test3 = [("Tokyo", "Bangkok", 180), ("Tokyo", "Hongkong", 550), ("Hongkong", "Hyderabad", 470),
         ("Bangkok", "Hyderabad", 280), ("Bangkok", "Singapore", 70), ("Singapore", "Hyderabad", 25
0),
         ("Bangkok", "Mumbai", 180), ("Mumbai", "Hyderabad", 40)]
```

Appendix 3: 3 graph algorithms

Bellman-Ford algorithm

In [4]:

```
# Declare global variables
INF = float("inf")
MAX = 9 # MAX is 9 because we have 9 cities in the list of cities

# Class Edge to store the information of each edge in the edge list
class Edge:
    def __init__(self, source, target, weight):
        self.source = source
        self.target = target
        self.weight = weight

# Function to create one-dimensional graph from the edge list
def graph_1D(dictionary, edge_list):
    graph = []
    # Loop through each edge in the edge_list
    for i in range(len(edge_list)):
        # u, v are the 2 cities
        # w is the price flying from city u to city v
        u, v, w = edge_list[i]
        # Get the id number of each city through the dictionary
        u = dictionary[u]
        v = dictionary[v]
        # Append it to the one-dimensional graph
        graph.append(Edge(u, v, w))
    return graph

# Function Bellman-Ford
def BellmanFord(start, target, nodes, dictionary, dictionary_reverse, edge_list):
    """

    Inputs:
    start: Starting node
    target: Target node
    nodes: Number of nodes in the graph
    dictionary: The dictionary to translate the names of the node to their ID numbers
    dictionary_reverse: The dictionary to translate the ID numbers of the nodes to their names
    edge_list: The edge list representing the flying routes between cities

    Output:
    dist[dictionary[target]]: The shortest distance from the start node to the target node

    """
    # Prepare the data structures
    graph = graph_1D(dictionary, edge_list) # To store the nodes in the graph
    dist = [INF for i in range(MAX)] # To store the distance from one node to another
    global pathBF # To store the path from one node to another
    pathBF = [-1 for i in range(MAX)]
    edges = len(edge_list) # Number of edges in the graph

    dist[dictionary[start]] = 0 # Initialize the function by setting the distance from the
starting node to itself as 0
    # Loop through all the nodes (n-1) times
    for i in range(1, nodes):
        # Loop through all the edges
        for j in range(edges):
            u = graph[j].source
            v = graph[j].target
            w = graph[j].weight
            # If there exists a route from source u to target v with lower cost than the current on
e
            # then we update the distance of target v
            if (dist[u] != INF) and (dist[u] + w < dist[v]):
                dist[v] = dist[u] + w
                # We also update the path to node v through u
                pathBF[v] = u

    # Return the shortest distance from start node to target node
    return dist[dictionary[target]]

def printPath(start, target, dictionary, dictionary_reverse):
    """

    Inputs:
    start: Starting node
    target: Target node
    dictionary: The dictionary to translate the names of the node to their ID numbers
    dictionary_reverse: The dictionary to translate the ID numbers of the nodes to their names


    Output:
    The function directly prints out the path from the starting node to the target node

    """
    # Take the ID number of the start and target node
    s, t = dictionary[start], dictionary[target]
    # The list to store the sequence of the nodes
    lst = []

    # Base case: If the target node is the same as the start node, then we just simply print it ou
t
    if t == s:
        print(dictionary_reverse[s])
        return
    # Base case: There is no path from the start node to the target node, noted by "-1"
    # So we print "No path"
    elif pathBF[t] == -1:
        print("No path")
        return

    while True:
        # Tracking the path from the start node to the target node in the reversed order
        # We are back tracking from the target node back to the start node
        lst.append(t)
        t = pathBF[t]
        # When we reach the starting node, break the while loop
        if t == s:
            lst.append(s)
            break

    # Reverse the sequence to print it out
    lst.reverse()
    # Print out the order from start node to target node
    for i in range(len(lst)-1):
        print(dictionary_reverse[lst[i]], end = " => ")
    print(dictionary_reverse[lst[-1]])
```

In [5]:

```
# Test cases
ans = BellmanFord("Seoul", "Hyderabad", 5, cities, cities_reverse, test1)
print("Minimum cost from Seoul to Hyderabad is ${}".format(ans))

# Print the path
printPath("Seoul", "Hyderabad", cities, cities_reverse)
```

```
Minimum cost from Seoul to Hyderabad is $350
Seoul => Kuala Lumpur => Hyderabad
```

In [6]:

```
ans = BellmanFord("Seoul", "Hyderabad", 6, cities, cities_reverse, test2)
print("Minimum cost from Seoul to Hyderabad is ${}".format(ans))

# Print the path
printPath("Seoul", "Hyderabad", cities, cities_reverse)
```

```
Minimum cost from Seoul to Hyderabad is $350
Seoul => Kuala Lumpur => Hyderabad
```

In [7]:

```
ans = BellmanFord("Tokyo", "Hyderabad", 6, cities, cities_reverse, test3)
print("Minimum cost from Tokyo to Hyderabad is ${}".format(ans))

# Print the path
printPath("Tokyo", "Hyderabad", cities, cities_reverse)
```

```
Minimum cost from Tokyo to Hyderabad is $400
Tokyo => Bangkok => Mumbai => Hyderabad
```

Dijkstra's algorithm

In [8]:

```
# Import library
import queue

# Declare global variables
INF = float("inf")
MAX = 9

# Class Node to store the information of each node in the graph: its ID number and distance
class Node:
    def __init__(self, ID, distance):
        self.ID = ID
        self.distance = distance

    # To organize the node from the shortest distance to longest
    def __lt__(self, other):
        return self.distance <= other.distance

# Function to create two-dimensional graph from the edge list
def graph_2D(dictionary, edge_list):
    graph = [[] for i in range(MAX)]
    for i in range(len(edge_list)):
        city1 = dictionary[edge_list[i][0]]
        city2 = dictionary[edge_list[i][1]]
        distance = edge_list[i][2]
        graph[city1].append(Node(city2, distance))
    return graph

# Function Dijkstra
def Dijkstra(start, target, nodes, dictionary, dictionary_reverse, edge_list):
    """
    Inputs:
    start: Starting node
    target: Target node
    nodes: Number of nodes in the graph
    dictionary: The dictionary to translate the names of the node to their ID numbers
    dictionary_reverse: The dictionary to translate the ID numbers of the nodes to their names
    edge_list: The edge list representing the flying routes between cities

    Output:
    dist[dictionary[target]]: The shortest distance from the start node to the target node

    """

    # Prepare the data structures
    graph = graph_2D(dictionary, edge_list) # To store the nodes in the graph
    dist = [INF for i in range(MAX)] # To store the distance from one node to another
    global pathD # To store the path from one node to another
    pathD = [-1 for i in range(MAX)]

    # Priority queue to get the nodes with shortest distance
    pq = queue.PriorityQueue()
    # Initialize the function by putting the start node to the queue, distance 0
    pq.put(Node(dictionary[start], 0))
    dist[dictionary[start]] = 0

    # Run the while loop when we still have nodes in the priority queue
    while not pq.empty():
        # Get the top node in the queue
        top = pq.get()
        u = top.ID
        w = top.distance
        # For all neighbor nodes in graph of u
        for neighbor in graph[u]:
            # If there exists a route from node u to neighbor with lower cost than the current one
            # then we update the distance of neighbor node
            if neighbor.distance + w < dist[neighbor.ID]:
                dist[neighbor.ID] = w + neighbor.distance
                # We put this node and its updated distance to the priority queue
                pq.put(Node(neighbor.ID, dist[neighbor.ID]))
                # We also update the path to node neighbor through u
                pathD[neighbor.ID] = u

    # Return the shortest distance from start node to target node
    return dist[dictionary[target]]

def printPath(start, target, dictionary, dictionary_reverse):
    """

    Inputs:
    start: Starting node
    target: Target node
    dictionary: The dictionary to translate the names of the node to their ID numbers
    dictionary_reverse: The dictionary to translate the ID numbers of the nodes to their names


    Output:
    The function directly prints out the path from the starting node to the target node

    """
    # Take the ID number of the start and target node
    s, t = dictionary[start], dictionary[target]
    # The list to store the sequence of the nodes
    lst = []

    # Base case: If the target node is the same as the start node, then we just simply print it ou
t
    if t == s:
        print(dictionary_reverse[s])
        return
    # Base case: There is no path from the start node to the target node, noted by "-1"
    # So we print "No path"
    elif pathD[t] == -1:
        print("No path")
        return
    while True:
        # Tracking the path from the start node to the target node in the reversed order
        # We are back tracking from the target node back to the start node
        lst.append(t)
        t = pathD[t]
        # When we reach the starting node, break the while loop
        if t == s:
            lst.append(s)
            break

    # Reverse the sequence to print it out
    lst.reverse()
    # Print out the order from start node to target node
    for i in range(len(lst)-1):
        print(dictionary_reverse[lst[i]], end = " => ")
    print(dictionary_reverse[lst[-1]])
```

In [9]:

```
# Test cases
ans = Dijkstra("Seoul", "Hyderabad", 5, cities, cities_reverse, test1)
print("Minimum cost from Seoul to Hyderabad is ${}".format(ans))

# Print the path
printPath("Seoul", "Hyderabad", cities, cities_reverse)
```

```
Minimum cost from Seoul to Hyderabad is $350
Seoul => Kuala Lumpur => Hyderabad
```

In [10]:

```
ans = Dijkstra("Seoul", "Hyderabad", 6, cities, cities_reverse, test2)
print("Minimum cost from Seoul to Hyderabad is ${}".format(ans))

# Print the path
printPath("Seoul", "Hyderabad", cities, cities_reverse)
```

```
Minimum cost from Seoul to Hyderabad is $350
Seoul => Kuala Lumpur => Hyderabad
```

In [11]:

```
ans = Dijkstra("Tokyo", "Hyderabad", 6, cities, cities_reverse, test3)
print("Minimum cost from Tokyo to Hyderabad is ${}".format(ans))

# Print the path
printPath("Tokyo", "Hyderabad", cities, cities_reverse)
```

```
Minimum cost from Tokyo to Hyderabad is $400
Tokyo => Bangkok => Mumbai => Hyderabad
```

Floyd-Warshall algorithm

In [12]:

```
# Declare global variables
INF = float("inf")
MAX = 9

# Function Floyd-Warshall algorithm
def FloydWarshall(start, target, nodes, dictionary, dictionary_reverse, edge_list):
    """

    Inputs:
    start: Starting node
    target: Target node
    nodes: Number of nodes in the graph
    dictionary: The dictionary to translate the names of the node to their ID numbers
    dictionary_reverse: The dictionary to translate the ID numbers of the nodes to their names
    edge_list: The edge list representing the flying routes between cities

    Output:
    dist[dictionary[start]][dictionary[target]]: The shortest distance from the start node to the
target node

    """

    # Initialize the data structures
    graph = [[INF for i in range(MAX)] for j in range(MAX)] # To store the nodes in the graph
    dist = [[INF for i in range(MAX)] for j in range(MAX)] # To store the distance from one node to
another
    global pathFW
    pathFW = [[-1 for i in range(MAX)] for j in range(MAX)] # To store the path from one node to an
other

    # If the starting and ending nodes are the same, then there is no route between them
    for i in range(MAX):
        for j in range(MAX):
            if i == j:
                graph[i][j] = 0

    # Update the information in graph, dist, path according to the edge list
    for i in range(len(edge_list)):
        city1 = dictionary[edge_list[i][0]]
        city2 = dictionary[edge_list[i][1]]
        distance = edge_list[i][2]
        graph[city1][city2] = distance
        dist[city1][city2] = distance
        pathFW[city1][city2] = city1

    # Run 3 for loops to update the distance
    for k in range(MAX):
        for i in range(MAX):
            for j in range(MAX):
            # If there exists a node k such that going from i to j through k is shorter than going
directly from i to j
            # then we update the distance from i to j
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    # We also update the path from node i to node j accordingly
                    # We also update the path from node i to node j accordingly
                    pathFW[i][j] = pathFW[k][j]

    # Return the shortest distance from start node to target node
    return dist[dictionary[start]][dictionary[target]]

def printPath(start, target, dictionary, dictionary_reverse):
    """

    Inputs:
    start: Starting node
    target: Target node
    dictionary: The dictionary to translate the names of the node to their ID numbers
    dictionary_reverse: The dictionary to translate the ID numbers of the nodes to their names


    Output:
    The function directly prints out the path from the starting node to the target node

    """
    # Take the ID number of the start and target node
    s, t = dictionary[start], dictionary[target]
    # The list to store the sequence of the nodes
    lst = []

    # Run the while loop until we reach the start node
    while s != t:
        # Tracking the path from the start node to the target node in the reversed order
        # We are back tracking from the target node back to the start node
        lst.append(t)
        t = pathFW[s][t]

    # Finally, add the start node to the list
    lst.append(s)
    # Reverse the sequence to print it out
    lst.reverse()

    # Print out the order from start node to target node
    for i in range(len(lst)-1):
        print(dictionary_reverse[lst[i]], end = " => ")
    print(dictionary_reverse[lst[-1]])
```

In [13]:

```
# Test cases
ans = FloydWarshall("Seoul", "Hyderabad", 5, cities, cities_reverse, test1)
print("Minimum cost from Seoul to Hyderabad is ${}".format(ans))

# Print the path
printPath("Seoul", "Hyderabad", cities, cities_reverse)
```

```
Minimum cost from Seoul to Hyderabad is $350
Seoul => Kuala Lumpur => Hyderabad
```

In [14]:

```
ans = FloydWarshall("Seoul", "Hyderabad", 6, cities, cities_reverse, test2)
print("Minimum cost from Seoul to Hyderabad is ${}".format(ans))

# Print the path
printPath("Seoul", "Hyderabad", cities, cities_reverse)
```

```
Minimum cost from Seoul to Hyderabad is $350
Seoul => Kuala Lumpur => Hyderabad
```

In [15]:

```
ans = FloydWarshall("Tokyo", "Hyderabad", 6, cities, cities_reverse, test3)
print("Minimum cost from Tokyo to Hyderabad is ${}".format(ans))

# Print the path
printPath("Tokyo", "Hyderabad", cities, cities_reverse)
```

```
Minimum cost from Tokyo to Hyderabad is $400
Tokyo => Bangkok => Mumbai => Hyderabad
```

Appendix 4: Running time examination

I'm writing modified code for each algorithm because the input from the randomly generated graph function does not require a dictionary to translate from string to integer.

Bellman-Ford algorithm to test running time

In [16]:

```python
# Declare global variables
INF = float("inf")

# Class Edge to store the information of each edge in the edge list
class Edge:
    def __init__(self, source, target, weight):
        self.source = source
        self.target = target
        self.weight = weight

# Function to create one-dimensional graph from the edge list
def graph_1D(edge_list):
    graph = []
    # Loop through each edge in the edge_list
    for i in range(len(edge_list)):
        # u, v are the 2 cities
        # w is the price flying from city u to city v
        u, v, w = edge_list[i]
        # Append it to the one-dimensional graph
        graph.append(Edge(u, v, w))
    return graph

# Function Bellman-Ford
def BellmanFord_General(start, target, nodes, edge_list):
    """
    Inputs: s, n, m
    start: Starting node
    target: Target node
    nodes: Number of nodes in the graph
    edge_list: The edge list representing the flying routes between cities

    Output:
    True/False: whether we can find the shortest distance
    """
    # Prepare the data structures
    MAX = nodes + 1
    graph = graph_1D(edge_list) # To store the nodes in the graph
    dist = [INF for i in range(MAX)] # To store the distance from one node to another

    # Initialize the function by setting the distance from the starting node to itself as 0
    dist[start] = 0
    # Loop through all the nodes (n-1) times
    for i in range(1, nodes):
        # Loop through all the edges
        for j in range(len(edge_list)):
            u = graph[j].source
            v = graph[j].target
            w = graph[j].weight
            # If there exists a route from source u to target v with lower cost than the current on
e
            # then we update the distance of target v
            if (dist[u] != INF) and (dist[u] + w < dist[v]):
                dist[v] = dist[u] + w

    # Return the shortest distance from start node to target node
    return dist[target]
```

Dijkstra's algorithm to test running time

In [17]:

```python
# Import library
import queue

# Declare global variables
INF = float("inf")

# Class Node to store the information of each node in the graph: its ID number and distance
class Node:
    def __init__(self, ID, distance):
        self.ID = ID
        self.distance = distance

    # To organize the node from the shortest distance to longest
    def __lt__(self, other):
        return self.distance <= other.distance

# Function to create two-dimensional graph from the edge list
def graph_2D(edge_list):
    graph = [[] for i in range(len(edge_list)+1)]
    for i in range(len(edge_list)):
        city1 = edge_list[i][0]
        city2 = edge_list[i][1]
        distance = edge_list[i][2]
        graph[city1].append(Node(city2, distance))
    return graph


def Dijkstra_General(start, target, nodes, edge_list):
    """
    Inputs:
    start: Starting node
    target: Target node
    nodes: Number of nodes in the graph
    dictionary: The dictionary to translate the names of the node to their ID numbers
    dictionary_reverse: The dictionary to translate the ID numbers of the nodes to their names
    edge_list: The edge list representing the flying routes between cities

    Output:
    dist[dictionary[target]]: The shortest distance from the start node to the target node

    """

    # Prepare the data structures
    MAX = nodes + 1
    graph = graph_2D(edge_list)
    dist = [INF for i in range(MAX)]

    # Priority queue to get the nodes with shortest distance
    pq = queue.PriorityQueue()
    # Initialize the function by putting the start node to the queue, distance 0
    pq.put(Node(start, 0))
    dist[start] = 0

    # Run the while loop when we still have nodes in the priority queue
    while not pq.empty():
        # Get the top node in the queue
        top = pq.get()
        u = top.ID
        w = top.distance
        # For all neighbor nodes in graph of u
        for neighbor in graph[u]:
            # If there exists a route from node u to neighbor with lower cost than the current one
            # then we update the distance of neighbor node
            if neighbor.distance + w < dist[neighbor.ID]:
                dist[neighbor.ID] = w + neighbor.distance
                # We put this node and its updated distance to the priority queue
                pq.put(Node(neighbor.ID, dist[neighbor.ID]))

    # Return the shortest distance from start node to target node
    return dist[target]
```

Floyd-Warshall algorithm to test running time

Floyd-Warshall algorithm to test running time

In [18]:

```
# Declare global variables
INF = float("inf")

# Function Floyd-Warshall
def FloydWarshall_General(start, target, nodes, edge_list):
    """

    Inputs:
    start: Starting node
    target: Target node
    nodes: Number of nodes in the graph
    dictionary: The dictionary to translate the names of the node to their ID numbers
    dictionary_reverse: The dictionary to translate the ID numbers of the nodes to their names
    edge_list: The edge list representing the flying routes between cities

    Output:
    dist[dictionary[start]][dictionary[target]]: The shortest distance from the start node to the
target node

    """

    # Initialize the data structures
    MAX = nodes + 1
    graph = [[INF for i in range(MAX)] for j in range(MAX)] # To store the nodes in the graph
    dist = [[INF for i in range(MAX)] for j in range(MAX)]  # To store the distance from one node t
o another

    # If the starting and ending nodes are the same, then there is no route between them
    for i in range(MAX):
        for j in range(MAX):
            if i == j:
                graph[i][j] = 0

    # Update the information in graph, dist, path according to the edge list
    for i in range(len(edge_list)):
        city1 = edge_list[i][0]
        city2 = edge_list[i][1]
        distance = edge_list[i][2]
        graph[city1][city2] = distance
        dist[city1][city2] = distance

    # Run 3 for loops to update the distance
    for k in range(MAX):
        for i in range(MAX):
            for j in range(MAX):
            # If there exists a node k such that going from i to j through k is shorter than going
directly from i to j
            # then we update the distance from i to j
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    # Return the shortest distance from start node to target node
    return dist[start][target]
```

Generate data to push into the algorithms

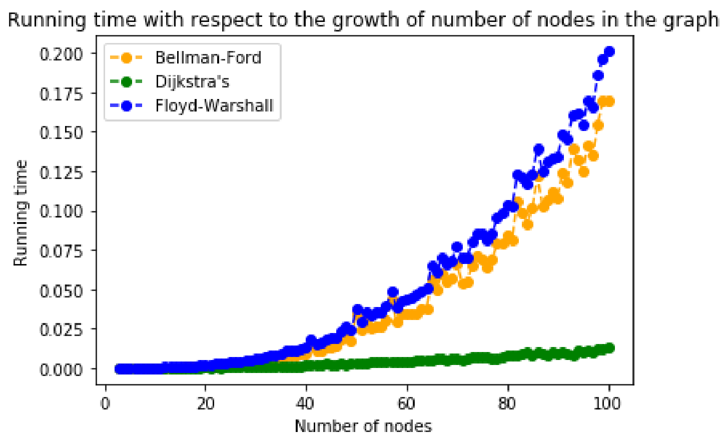
In [19]:

```
# Import library
import random

def random_graph_generator(n):
    """
    Function to generate random graphs to push to the 3 graph algorithms

    Input: n: Number of nodes in the graph

    Output: graph: An edge list with each edge comprised of a start node, end node, and cost from
start to end node
    """
    # Declare the number of edges in the graph
    # Actually the number of edges can be any as long as it is less than n*(n-1)/2
    # But for the sake of calculating the running time, we predefine the number of edges here
    num_edges = n*(n-1)//2

    # Initialize the edge list
    edge_list = []

    # Run through all edges in the edge list
    for i in range(num_edges):
        a = random.randint(0, n-1) # Start node
        b = random.randint(0, n-1) # End node
        w = random.randint(0, 1000) if a != b else 0 # Random traveling cost from node a to b
        edge_list.append((a, b, w)) # Append the edge to the edge list

    # Return the edge list
    return edge_list
```

Plot the results

In [21]:

```
# Import plotting and time libraries
import matplotlib.pyplot as plt
import time

# Initialize the storage array to store the running time of 3 algorithms
bf_time = [] # Bellman-Ford function
d_time = [] # Dijkstra function
fw_time = [] # Floyd-Warshall function
x = [i for i in range(3, 101)] # This is the x-axis: Input size from 3 to 100 (inclusize)

# Running time of each algorithm with respect to the growth in input size
# Input ranges from 3 to 100 (inclusive)
for n in range(3, 101, 1):
    # Create temporary arrays to store the time
    # For each length we would run the function 50 times, then average the results
    bf_time_temp = []
    d_time_temp = []
    fw_time_temp = []

    # Run each function 50 times for each input size
    for i in range(50):
        # Create the edge list with nodes
        data = random_graph_generator(n)

        # Timing the running time of Bellman-Ford algorithm
        start_BF = time.time()
        BellmanFord_General(0, n-1, n, data)
        end_BF = time.time()
        bf_time_temp.append(end_BF-start_BF)

        # Timing the running time of Dijkstra's algorithm
        start_D = time.time()
        Dijkstra_General(0, n-1, n, data)
        end_D = time.time()
        d_time_temp.append(end_D-start_D)

        # Timing the running time of Floyd-Warshall algorithm
        start_FW = time.time()
        FloydWarshall_General(0, n-1, n, data)
        end_FW = time.time()
        fw_time_temp.append(end_FW-start_FW)

    # Average the result of 50 running times. Then append it to the storage array
    bf_time.append(sum(bf_time_temp)/50)
    d_time.append(sum(d_time_temp)/50)
    fw_time.append(sum(fw_time_temp)/50)

# Plot the running time of each function versus its growth in input size
plt.plot(x, bf_time, color = "orange", linestyle = "dashed", marker='o', label = "Bellman-Ford")
plt.plot(x, d_time, color = "green", linestyle = "dashed", marker='o', label = "Dijkstra's")
plt.plot(x, fw_time, color = "blue", linestyle = "dashed", marker='o', label = "Floyd-Warshall")
plt.xlabel("Number of nodes")
plt.ylabel("Running time")
plt.title("""Running time with respect to the growth of number of nodes in the graph""")
plt.legend()
plt.show()
```


Running time with respect to the growth of number of nodes in the graph
Bellman-Ford
Dijkstra's
Floyd-Warshall
Running time
Number of nodes