Mini Programming Assignment: Implement Dijkstra's Algorithm

Name: Ben Prescott

Date: 08/06/2021

Course: MSDS432 Summer 2021

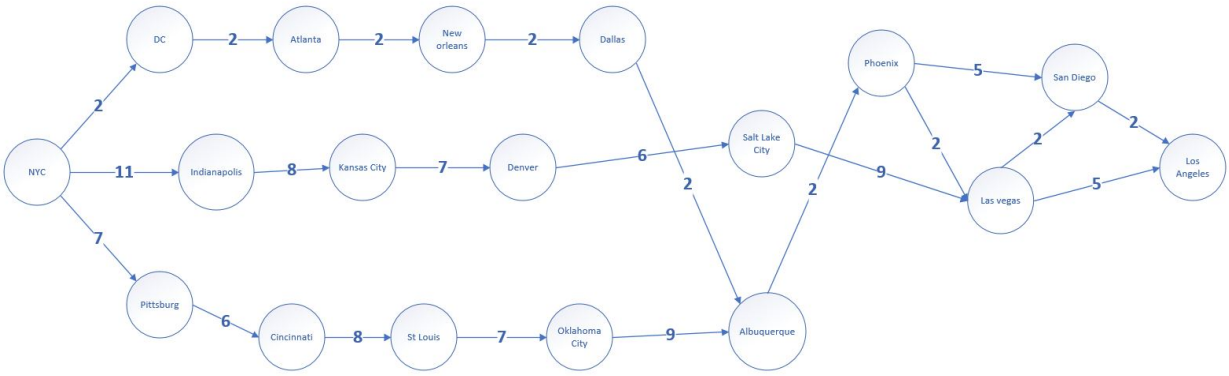
Executive Summary

Description:

This assignment includes the creation of a graph with possible routes between 18 major cities, then using breadth first search and Dijkstra's algorithm to determine the shortest paths between two cities. The graph is a weighted graph, with drive time (in hours) describes on the edges of each city to the next.

Once the graph is created, both algorithms are used to find the shortest path from New York City to Los Angeles. The execution time of each algorithm is logged, as well as the number of cities (nodes) each algorithm determines passing through, as well as the total drive time it will take to get from NYC to LA using the graph's edge weights.

The graph used for this assignment is seen below:



Algorithm Explanations

Breadth first search (BFS) and Dijkstra's algorithm both approach this problem differently. BFS does not use edge weights to determine the shortest route, but rather looks for the path with the lead amount of nodes to traverse to reach the end point. In this context, the shortest path would include visiting the least amount of cities, not necessarily looking for the shortest drive time. The time complexity of BFS is $O(V+E)$.

On the other hand, Dijkstra's algorithm does use edge weights to determine the recommended path. In this case, Dijkstra's algorithm would be searching for the path that includes the shortest overall drive time, using the edge weights to determine the output. Dijkstra's algorithm works on

finding the 'lowest cost' to neighboring nodes at each step in a graph. Once it has moved to the next node, it again checks for the shortest path available to the new node's neighbors, updating the neighbor's costs if it finds a shorter path. This process is repeated until the destination is reached. The worst case time complexity of Dijkstra's algorithm in this example is $O((V+E)logV)$, which indicates this algorithm is likely to be slower than BFS.

Approach

To start this assignment I created a graph using a Python dictionary type. The graph was modeled against the image above, creating the nodes (cities) and their connections to other nodes, and the travel times to each. This creates keys in the dictionary that represent the nodes, and values for each key that describes the connected cities and the travel time (in hours) to get to them.

Two additional dictionaries were required for the costs and the parents of the starting point node's connections, named 'costs' and 'parents'. These two dictionaries are used by Dijkstra's algorithm to update neighboring node's costs as it progresses through the graph, and the new node's parent node (where we traveled from). With NYC being the starting point, the three neighboring nodes (DC, Indianapolis, Pittsburg) have identified weights that were provided in the costs table, and DC, Indianapolis, and Pittsburg all had their parent set as NYC. Every other weight was set as infinity and parent set as 'None', allowing the algorithm to update costs and parent nodes as it progresses.

Results

The results showed what was to be expected of algorithm execution times - dijkstra's algorithm is noticably slower than breadth first search. However, as mentioned earlier, BFS is looking for the shortest path as it relates to nodes/edges to traverse. This resulted in BFS taking a different path from NYC to LA, going through Indianapolis and with a total of 6 stops. Considering BFS does not use weighted edges to determine the path, I calculated what the travel time would be using this path, resulting in a total drive time of 46 hours.

Dijkstra's algorithm found a different path from NYC to LA, traveling through the DC node and using a total of 9 stops. While this is 3 additional cities to hit in order to travel to Las Angeles, the total drive time is only 18 hours. This is significantly shorter in travel time, showing the value of a weighted graph and using an algorithm meant for weighted graphs.

In [22]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from timeit import default_timer as timer
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
np.set_printoptions(suppress=True)
```

Creating Connections Graph

In [12]:
```
graph = {}
```

```
# Defining Nodes
graph['NYC'] = {}
graph['DC'] = {}
graph['Pittsburg'] = {}
graph['Indianapolis'] = {}
graph['Atlanta'] = {}
graph['Kansas City'] = {}
graph['Cincinnati'] = {}
graph['New Orleans'] = {}
graph['Denver'] = {}
graph['St Louis'] = {}
graph['Oklahoma City'] = {}
graph['Dallas'] = {}
graph['Albuquerque'] = {}
graph['Salt Lake City'] = {}
graph['Phoenix'] = {}
graph['Las Vegas'] = {}
graph['San Diego'] = {}
graph['Los Angeles'] = {}

# Defining Connections And Weights

# Top Path
graph['NYC']['DC'] = 2
graph['DC']['Atlanta'] = 2
graph['Atlanta']['New Orleans'] = 2
graph['New Orleans']['Dallas'] = 2
graph['Dallas']['Albuquerque'] = 2
graph['Albuquerque']['Phoenix'] = 2
graph['Phoenix']['Las Vegas'] = 2
graph['Phoenix']['San Diego'] = 5
graph['Las Vegas']['San Diego'] = 2
graph['Las Vegas']['Los Angeles'] = 5
graph['San Diego']['Los Angeles'] = 2

# Middle Path
graph['NYC']['Indianapolis'] = 11
graph['Indianapolis']['Kansas City'] = 8
graph['Kansas City']['Denver'] = 7
graph['Denver']['Salt Lake City'] = 6
graph['Salt Lake City']['Las Vegas'] = 9
graph['NYC']['Indianapolis'] = 11

# Bottom Path
graph['NYC']['Pittsburg'] = 7
graph['Pittsburg']['Cincinnati'] = 6
graph['Cincinnati']['St Louis'] = 8
graph['St Louis']['Oklahoma City'] = 7
graph['Oklahoma City']['Albuquerque'] = 9

# Creating Costs Table

infinity = float('inf')
costs = {}
costs['DC'] = 2
costs['Indianapolis'] = 11
costs['Pittsburg'] = 7
costs['Atlanta'] = infinity
costs['Kansas City'] = infinity
costs['Cincinnati'] = infinity
costs['New Orleans'] = infinity
```

```
costs['Denver'] = infinity
costs['St Louis'] = infinity
costs['Oklahoma City'] = infinity
costs['Dallas'] = infinity
costs['Albuquerque'] = infinity
costs['Salt Lake City'] = infinity
costs['Phoenix'] = infinity
costs['Las Vegas'] = infinity
costs['San Diego'] = infinity
costs['Los Angeles'] = infinity

# Creating Parents Table

parents = {}
parents['DC'] = 'NYC'
parents['Indianapolis'] = 'NYC'
parents['Pittsburg'] = 'NYC'
parents['Atlanta'] = None
parents['Kansas City'] = None
parents['Cincinnati'] = None
parents['New Orleans'] = None
parents['Denver'] = None
parents['St Louis'] = None
parents['Oklahoma City'] = None
parents['Dallas'] = None
parents['Albuquerque'] = None
parents['Salt Lake City'] = None
parents['Phoenix'] = None
parents['Las Vegas'] = None
parents['San Diego'] = None
parents['Los Angeles'] = None

# Array to keep track of processed nodes
processed = []
```

Breadth First Search Route

In [9]:

```
# Defining a BFS algorithm

def bfs(graph, start, end):
    # maintain a queue of paths
    queue = []
    # push the first path into the queue
    queue.append([start])
    while queue:
        # get the first path from the queue
        path = queue.pop(0)
        # get the last node from the path
        node = path[-1]
        # path found
        if node == end:
            return path
        # enumerate all adjacent nodes, construct a new path and push it into the queue
        for adjacent in graph.get(node, []):
            new_path = list(path)
            new_path.append(adjacent)
            queue.append(new_path)
```

```
# Using breadth first search to find the path from NYC to LA, recording the execution t
start = timer()
bfs_path = bfs(graph, 'NYC', 'Los Angeles')
end = timer()

bfstime = end-start
```

In [10]:

```
prevnum = 0
nextnum = 1
bfs_total = 0

for n in range(len(bfs_path)-1):
    num = graph[bfs_path[prevnum]][bfs_path[nextnum]]
    prevnum += 1
    nextnum += 1
    bfs_total += num
print(' --> '.join(bfs_path),'\n')
print("The time the drive will take is",bfs_total,"hours")
```

```
NYC --> Indianapolis --> Kansas City --> Denver --> Salt Lake City --> Las Vegas --> Los
Angeles

The time the drive will take is 46 hours
```

Dijkstra's Algorithm

In [13]:

```
# Defining dijkstra's algorithm
def find_lowest_cost_node(costs):
    lowest_cost = float("inf")
    lowest_cost_node = None
    # Go through each node.
    for node in costs:
        cost = costs[node]
        # If it's the lowest cost so far and hasn't been processed yet...
        if cost < lowest_cost and node not in processed:
            # ... set it as the new lowest-cost node.
            lowest_cost = cost
            lowest_cost_node = node
    return lowest_cost_node

# Find the lowest-cost node that you haven't processed yet.
node = find_lowest_cost_node(costs)
# If you've processed all the nodes, this while loop is done.

start = timer()

while node is not None:
    cost = costs[node]
    # Go through all the neighbors of this node.
    neighbors = graph[node]
    for n in neighbors.keys():
        new_cost = cost + neighbors[n]
        # If it's cheaper to get to this neighbor by going through this node...
        if costs[n] > new_cost:
            # ... update the cost for this node.
            costs[n] = new_cost
            # This node becomes the new parent for this neighbor.
            parents[n] = node
```

```
    # Mark the node as processed.
    processed.append(node)
    # Find the next node to process, and loop.
    node = find_lowest_cost_node(costs)

end = timer()

dijkstratime = end-start

print("Cost from the start to each node:")
print(costs)
```

```
Cost from the start to each node:
{'DC': 2, 'Indianapolis': 11, 'Pittsburg': 7, 'Atlanta': 4, 'Kansas City': 19, 'Cincinna
ti': 13, 'New Orleans': 6, 'Denver': 26, 'St Louis': 21, 'Oklahoma City': 28, 'Dallas':
8, 'Albuquerque': 10, 'Salt Lake City': 32, 'Phoenix': 12, 'Las Vegas': 14, 'San Diego':
16, 'Los Angeles': 18}
```

In [14]:

```
# Find the path from NYC to LA
current_node , start_node = 'Los Angeles', 'NYC'

dijkstra_path = [current_node]

while current_node != start_node:
    last_loc = parents[current_node]
    current_node = last_loc
    dijkstra_path.append(last_loc)
```

In [15]:

```
# Find the total drive time hours for the path

prevnum = 0
nextnum = 1
dijkstra_total = 0

for n in range(len(dijkstra_path[::-1])-1):
    num = graph[dijkstra_path[::-1][prevnum]][dijkstra_path[::-1][nextnum]]
    prevnum += 1
    nextnum += 1
    dijkstra_total += num
print(' --> '.join(dijkstra_path[::-1]),'\n')
print("The time the drive will take is",dijkstra_total,"hours")
```

```
NYC --> DC --> Atlanta --> New Orleans --> Dallas --> Albuquerque --> Phoenix --> Las Ve
gas --> San Diego --> Los Angeles

The time the drive will take is 18 hours
```

Comparison Table & Graphs

In [20]:

```
# Creating a Pandas DataFrame for table reference
pd.set_option("display.precision", 10)
results = pd.DataFrame()
results['Algorithm'] = ['Dijkstras Algorithm','Breadth First Search']
results['Trip Time (hours)'] = [dijkstra_total,bfs_total]
results['Execution Times'] = [dijkstratime,bfstime]
results['Path'] = [' --> '.join(dijkstra_path[::-1]), ' --> '.join(bfs_path)]
```

```
results['Number of Stops'] = [len(dijkstra_path)-1, len(bfs_path)-1]
results
```

Out[20]:

	Algorithm	Trip Time (hours)	Execution Times	Path	Number of Stops
0	Dijkstras Algorithm	18	0.0002532	NYC --> DC --> Atlanta --> New Orleans --> Dal...	9
1	Breadth First Search	46	0.0000721	NYC --> Indianapolis --> Kansas City --> Denve...	6

Visualize Results

In [21]:

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

fig, axes = plt.subplots(1, 3, sharex=True, figsize=(16,8))
sns.set_style('darkgrid')
fig.suptitle('Algorithm Performance - Dijkstra vs Breadth First')

# Execution Times
sns.barplot(ax=axes[0], x=results['Algorithm'], y=results['Execution Times'])
axes[0].set_title('Execution Times')

# Trip Time
sns.barplot(ax=axes[1], x=results['Algorithm'], y=results['Trip Time (hours)'])
axes[1].set_title('Trip Time (hours)')

# Number Stops
sns.barplot(ax=axes[2], x=results['Algorithm'], y=results['Number of Stops'])
axes[2].set_title('Number of Stops')

plt.plot();
```

```
<Figure size 864x576 with 0 Axes>
```