**Network Routing and Graph Paths**

**Authors:**

Harshal Pandurang Bhoir and Dheeraj Mohanbabu.

**Abstract:**

A optimal routing algorithm called Dijkstra's Algorithm generates every possible path across the graph and then chooses the one with the lowest total cost. Iteratively computing a distance for each node in the graph, beginning at the start node and continuing until we reach the end node, is how this method operates.

**Introduction**

Digital Mapping Services in Google Maps, we have frequently attempted to calculate the distance in G-Maps between cities or between your location and the closest desired destination. Dijkstra's Algorithm is used to determine the shortest distance between two points along the path because there are numerous routes and paths linking them, but it must display the shortest distance. You may also have noticed that many social networking applications offer a list of friends that a specific user might know. Given that the system has more than a billion members, how effectively do you believe many social media businesses will integrate this feature? The shortest path between users, as determined via handshakes or connections between them, can be used to apply the conventional Dijkstra algorithm. In this project we have implemented Dijkstra algorithm on dataset from Amazon0302.

**Related Works**

There have been many applications and related works regarding the calculation of shortest path. Many of them seem to use the same kind of data but tend to lean more towards the use of deep learning. With the use of such powerful and deep algorithms, these researchers were able to get accuracy results near perfect (>95%). In our limited research there were very few credible sources using different, more standard, methods like Dijkstra algorithm to get the shortest path from source node to target node.

**Dataset Description**

The dataset we used was retrieved from the Amazon0302 which is very dense dataset for this implementation that is often used for testing and   
comparison of various algorithms.

**Graphical user interface, application, table, Excel

Description automatically generated**

This is the visualization of the dataset Amazon0302 ( node connections )

**Methodology**

A pathfinding algorithm called Dijkstra's Algorithm generates every possible path across the graph and then chooses the one with the lowest total cost.

Iteratively calculating a distance for each node in the graph, beginning at the start node and moving along until we reach the end node, is how this method operates. We have a "current node" in each iteration, and we calculate a best score for each node that can be reached from it.

**CODE:**

https://drive.google.com/drive/folders/1bCOFgjNXZiSsD4\_R7mRtXc09gvVX0Tih?usp=share\_link

import csv

import pandas as pd

import numpy as np

import random

with open ('Amazon0302.txt', 'r') as f:

head=['source','target']

first\_row = csv.reader(f,delimiter='\t')

df=pd.DataFrame(first\_row)

df.columns = head

# Set the random seed so that the values remain the same for each run

random.seed(0)

# Create a list of random values in the range 1 to 100

random\_values = [random.randint(1, 100) for \_ in range(len(df))]

# Add a third column to the dataframe with the random values

df["weight"] = random\_values

print (df.head(21))

df2=df[:20]

df2.to\_csv(r'Edge\_weight.txt', header=None, index=None, sep='\t', mode='a')

# Import necessary modules

import pandas as pd

import matplotlib.pyplot as plt

from collections import defaultdict

import random

# Define a function to implement the Dijkstra algorithm

def dijkstra(edges, source, target):

# Create a dictionary to store the distances from the source node to each node

distances = {node: float("inf") for node in edges}

distances[source] = 0

# Create a dictionary to store the previous node in the path from the source to each node

previous\_nodes = {node: None for node in edges}

# Create a set to store the nodes that have been visited

visited\_nodes = set()

# Create a set to store the nodes that are still to be visited

unvisited\_nodes = set(edges.keys())

# Loop through the unvisited nodes until we find the target node

while unvisited\_nodes:

current\_node = min(unvisited\_nodes, key=lambda x: distances[x])

if current\_node == target:

break

# Remove the current node from the set of unvisited nodes

unvisited\_nodes.remove(current\_node)

try:

# Update the distances to the neighboring nodes

for neighbor, weight in edges[current\_node]:

if distances[neighbor] > distances[current\_node] + weight:

distances[neighbor] = distances[current\_node] + weight

previous\_nodes[neighbor] = current\_node

except KeyError:

# If the key does not exist, set the value to None

value = None

# Create a list to store the shortest path

path = []

# Use the previous nodes dictionary to construct the shortest path

current\_node = target

try:

while current\_node:

path.append(current\_node)

current\_node = previous\_nodes[current\_node]

except KeyError:

# If the key does not exist, set the value to None

value = None

# Return the shortest path in reverse order

return path[::-1],distances

# In[20]:

df=pd.DataFrame()

# Read the input data from a Pandas dataframe

head=['source','target','weight']

df1 = pd.read\_csv("Edge\_weight.txt",sep='\t',names=head)

#df.names=head

# Create a new column "to" which is the same as the "from" column, but reversed

df['source'] = pd.concat([df1['source'], df1['target']])

# Add the third column to the end of the first column

df['target'] = pd.concat([df1['target'], df1['source']])

df['weight'] = pd.concat([df1['weight'], df1['weight']])

# Create a dictionary to store the edges and their weights

edges = defaultdict(list)

for \_, row in df.iterrows():

source, target, weight = row["source"], row["target"], row["weight"]

edges[source].append((target, weight))

a=[] # Create a list to store all the shortest path

# Use Dijkstra's algorithm to find the shortest path

# start at node 1

source = 2

for i in df['target'].unique():

print('Running Loop for node ', i)

target =i # end at the last node

shortest\_path = dijkstra(edges, source, target)

#print('shortest\_path',shortest\_path)

list\_values = shortest\_path[0]

dict\_values = sorted(dict(shortest\_path[1]).items(), key=lambda x: x[0], reverse=False)

#print('dict\_values',dict\_values)

a.append(list\_values)

# Print the shortest path

# Visualize the graph

#plt.plot()

#for source, targets in edges.items():

# for target, weight in targets:

# plt.plot([source, target], [1, 10], linestyle="dotted")

#plt.show()

print(df1)

# sort the list in ascending order of the last element of each sublist

a.sort(key=lambda x: x[-1] if len(x) > 0 else -1)

#print('list\_values',(list\_values))

#print('dict\_values',(dict\_values))

#print('a',a)

# Create a dataframe from the list and dictionary

df2 = pd.DataFrame(dict\_values, columns=["key", "value"])

b=[]

df2 = df2.rename(columns={'key': 'Target', 'value': 'Distance from {}'.format(source)})

#print(df2)

#for sublist in a:

# if len(sublist) > 0 and sublist[0] == source:

# b.append(sublist)

#print(b)

df2['Path']=a

df\_table=pd.DataFrame()

# iterate over the rows of the dataframe

for index, row in df2.iterrows():

# check if the list in the 'Path' column contains the number 3

if not source in row['Path']:

# if the list does not contain the number 3, set the value of the 'Path' column to 'inf'

df2.at[index, 'Path'] = 'inf'

df2.at[index, 'Distance from {}'.format(source)] = 'inf'

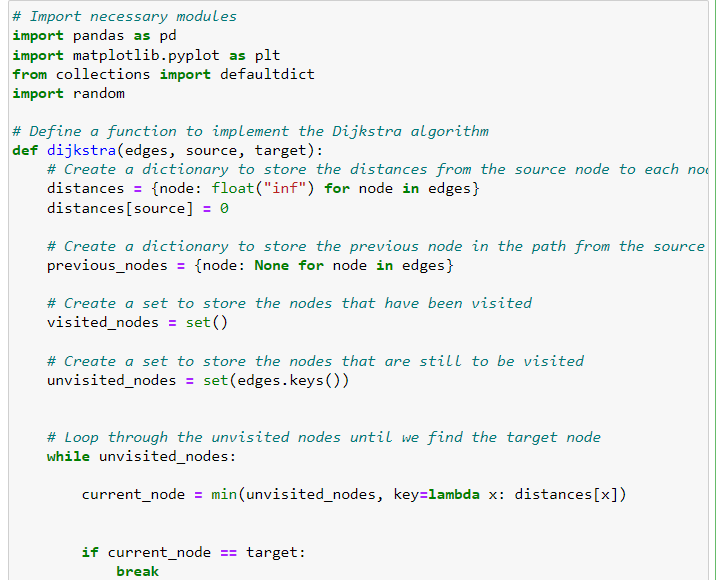
**Experimental Results**

Text

Description automatically generated

The time complexity of the code snippet linear in the size of the input data, since it reads the data from the input file and processes it row by row.

The space complexity may also be linear.



The time complexity of the Dijkstra algorithm is O(|E| + |V|log|V|), where |E| is the number of edges in the graph and |V| is the number of vertices. Also the algorithmis recursive and hence is of order O(n2)

The space complexity of the Dijkstra algorithm is O(|E| + |V|), Hence it is O(n)

Graphical user interface, text, application

Description automatically generated

The time complexity of the code snippet is O(n2)

The space complexity of the O(1)

Text

Description automatically generated

The time complexity of the code snippet is O(n2)

The space complexity of the O(1)

Diagram

Description automatically generated

Here we have implemented this algorithm using few user defined functions in pyhton3.

**Contribution**

- Dheeraj Mohan Babu: Algorithm design and code.

- Harshal Pandurang Bhoir: Code and documentation.

**Conclusion**

The goal of our experimentation was to see if we are able to construct a python function which returns shortest path form one node to targeted node. We began with a dataset form Amazon0302. We were successful in getting the shortest path using Dijkstra's Algorithm.

**Graphical user interface, text, application, table

Description automatically generated**

**Reference:**

• Broumi, Said, et al. “Applying Dijkstra Algorithm for Solving Neutrosophic Shortest Path Problem.” 2016 International Conference on Advanced Mechatronic Systems (ICAMechS), IEEE, Nov. 2016. Crossref, doi:10.1109/icamechs.2016.7813483.

• Qing, Guo, et al. “Path-Planning of Automated Guided Vehicle Based on Improved Dijkstra Algorithm.” 2017 29th Chinese Control And Decision Conference (CCDC), IEEE, May 2017. Crossref, doi:10.1109/ccdc.2017.7978471.

• Ji-Xian Xiao, and Fang-Ling Lu. “An Improvement of the Shortest Path Algorithm Based on Dijkstra Algorithm.” 2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE), IEEE, Feb. 2010. Crossref, doi:10.1109/iccae.2010.5451564

* Dataset Source:

https://snap.stanford.edu/data/