**CSE-5311-006**

**PROJECT**

**Sathwik Reddy Gowda Krishnareddy**

**ID:1002146597**

Selected Project 3 (Shortest Path) for my project work

Implement and compare the following shortest paths algorithm for weighted graph and unweighted graph:

● Breadth-first search (unweighted graph)

● The Bellman-Ford algorithm

● Dijkstra’s algorithm

**Observations Based on the testing and implementation of algorithms.**

Breadth-First Search (BFS):

Time Complexity: O(V+E) for an adjacency list representation, where V is the number of vertices and E is the number of edges. BFS is typically efficient in finding the shortest paths in unweighted graphs.

Dijkstra's Algorithm:

Time Complexity: O(V^2 ) with an array when using 2D array and queue logic. Can be improved to O((V+E)⋅logV) with a binary heap. Dijkstra's algorithm can only be used for finding shortest paths in graphs with non-negative weights. We will see an issue when there are negative weights on edges since we try to navigate towards the edges with less weight and negative edges give that impression.

Bellman-Ford Algorithm:

Time Complexity: O(V⋅E) and since O(E) comes up to V^2, total time complexity would be O(V^3).

Bellman-Ford can handle graphs with negative weights but is less efficient than Dijkstra's algorithm in the absence of negative weights. It is suitable for graphs with negative weights or negative cycles. Due to high time complexity suitable for sparse graphs and practically in real world generally graphs will be sparse. Fails if there is a negative cycle, but we will be able to identify if there is a negative cycle and report it.

Time Taken with Respect to Data Size:

BFS: Takes least time and is a very efficient algorithm, so easily scalable.

Dijkstra: Scales well for small and medium-sized graphs; however, its time complexity may become a concern for very large and dense graphs.

Bellman-Ford: Scales less efficiently than the others due to its higher time complexity. Even in test cases mentioned at the bottom of the document you can clearly see the time difference between Bellman Ford and other algorithms.

Which one is best in terms of what conditions?

Unweighted Graphs: BFS is often the best choice.

Non-negative Weighted Graphs: Dijkstra's algorithm is usually the most efficient.

Graphs with Negative Weights: Bellman-Ford is the only option, as Dijkstra's algorithm doesn't handle negative weights.

Possible Improvements:

Djikstra can be improved using Binary heap.

**Technical Details**

Programming Language used: **Python**

Built GUI using Python **ttkbootstrap**

Outputting the runtime analysis plots using **matplotlib.pyplot**

Generated random graph using **networkx**

**Data Structures used**:

In Breadth First Search using “Queue” from “queue” as **STACK** to keep stock of vertices to be further visited and pop the visited vertices.

In all the algorithms using “**Dictionary**” to keep track of the distance of all the nodes/vertices from source node/vertex.

**2D array** is used to represent Graph generated using networkx.

**Below is the GUI Developed and Options provided in the GUI**

**Number of Vertices –** User can input the number of vertices that the generated graph must have on which shortest path algorithms should run.

**Source Vertex –** Vertex from where the shortest path of rest of the vertices must be calculated

**Algorithm –** Drop down with the three algorithms, user can choose any one and see the plot of amount of time taken by that algorithm to find shortest path.

**Run –** Button to start the algorithm.

A screenshot of a computer

Description automatically generated

**Compare Checkbox-** When selected, as shown below. Users will be given an option to select algorithms to be compared. Users can enter up to three algorithms. When clicked on run, displays the plot showing time taken against each algorithm.

A screenshot of a computer

Description automatically generated

**Size Range Checkbox –** As shown in the screenshot below, this allows user to provide the range of number of vertices and the step size. Time taken by an algorithm to calculate the shortest path for graphs with different number of vertices, can be compared using this. After clicking run, the plot displays the time taken by an algorithm for different number of vertices provided in range, incremented based on the step size.

A screenshot of a computer

Description automatically generated

**Below is the list of functions used in GUI, code and the purpose of each function.**

Class: shortestPathAlgorithmsApp

Method/Function

­­\_\_init\_\_ - Constructor to create an instance and initialize the GUI screen.

def \_\_init\_\_(self,root):

        self.root = root

        self.root.title("Shortest Path Algorithms")

        self.root.geometry('1080x650')

        self.initializeGUI()

initializeGUI – Called inside \_\_init\_\_, has the code related to all the fields of the screen.

 def initializeGUI(self):

        # Creating a Frame

        self.mainFrame = ttkb.Frame(self.root,width=500)

        self.mainFrame.grid(row=1, column=0, sticky="w", padx=10, pady=5)

        # Overall box label

        self.boxLabel = ttkb.Label(text="Shortest Path Algorithms",font=("Poppins",20))

        self.boxLabel.grid(row=0, column=0, sticky="w", padx=10, pady=5)

        # Input Size field label

        self.inputSizeLabel = ttkb.Label(self.mainFrame,text="NUMBER OF VERTICES",font=("Poppins",15))

        self.inputSizeLabel.grid(row=2, column=0, sticky="w", padx=10, pady=5)

        # Input Size field entry

        self.inputSizeEntry = ttkb.Entry(self.mainFrame)

        self.inputSizeEntry.grid(row=2, column=1, sticky="W",padx=10,pady=5)

        self.inputSizeEntry.insert(0,"100")

        # Input Size Range field label

        self.inputSizeRangeLabel = ttkb.Label(self.mainFrame,text="RANGE OF VERTICES",font=("Poppins",15))

        self.inputSizeRangeLabel.grid(row=2, column=0, sticky="w", padx=10, pady=5)

        self.inputSizeRangeLabel.grid\_remove()

        # Input Size Range Low field entry

        self.inputSizeRangeLowEntry = ttkb.Entry(self.mainFrame)

        self.inputSizeRangeLowEntry.grid(row=2, column=1, sticky="W",padx=10,pady=5)

        self.inputSizeRangeLowEntry.insert(0,"100")

        self.inputSizeRangeLowEntry.grid\_remove()

        # Input Size Range High field entry

        self.inputSizeRangeHighEntry = ttkb.Entry(self.mainFrame)

        self.inputSizeRangeHighEntry.grid(row=2, column=2, sticky="W",padx=10,pady=5)

        self.inputSizeRangeHighEntry.insert(0,"100")

        self.inputSizeRangeHighEntry.grid\_remove()

        # Input Size Range Interval Step Size Label

        self.inputSizeRangeStepSizeLabel = ttkb.Label(self.mainFrame,text="STEP SIZE",font=("Poppins",15))

        self.inputSizeRangeStepSizeLabel.grid(row=2, column=3, sticky="w", padx=10, pady=5)

        self.inputSizeRangeStepSizeLabel.grid\_remove()

        # Input Size Range Interval Step Size entry

        self.inputSizeRangeStepSizeEntry = ttkb.Entry(self.mainFrame)

        self.inputSizeRangeStepSizeEntry.grid(row=2, column=4, sticky="W",padx=10,pady=5)

        self.inputSizeRangeStepSizeEntry.insert(0,"10")

        self.inputSizeRangeStepSizeEntry.grid\_remove()

        # Source Vertex field label

        self.sourceVertexLabel = ttkb.Label(self.mainFrame,text="SOURCE VERTEX",font=("Poppins",15))

        self.sourceVertexLabel.grid(row=3, column=0, sticky="w", padx=10, pady=5)

        # Source Vertex field entry

        self.sourceVertexEntry = ttkb.Entry(self.mainFrame)

        self.sourceVertexEntry.grid(row=3, column=1, sticky="W",padx=10,pady=5)

        self.sourceVertexEntry.insert(0,"0")

        # Algorithm drop down label

        self.algorithmLabel = ttkb.Label(self.mainFrame,text="ALGORITHM",font=("Poppins",15))

        self.algorithmLabel.grid(row=4, column=0, sticky="w", padx=10, pady=5)

        # Algorithm drop down Combobox

        self.algorithm = ttkb.StringVar()

        self.algorithmCombobox = ttkb.Combobox(self.mainFrame,textvariable=self.algorithm, values = ["Breadth For Search", "Bellman Ford", "Dijkstra"])

        self.algorithmCombobox.grid(row=4, column=1, sticky="w", padx=10, pady=5)

        # Checkbox to compare Algorithms

        self.compareAlgoVar = ttkb.BooleanVar()

        self.compareAlgo = ttkb.Checkbutton(self.mainFrame,text="Compare",variable=self.compareAlgoVar,command=self.compareVisible)

        self.compareAlgo.grid(row=5, column=0, columnspan=2, sticky="w", padx=10, pady=5)

        # Checkbox to provide input size range

        self.inputRangeVar = ttkb.BooleanVar()

        self.inputRange = ttkb.Checkbutton(self.mainFrame,text="Size Range",variable=self.inputRangeVar,command=self.activateInputRange)

        self.inputRange.grid(row=5, column=1, columnspan=2, sticky="w", padx=10, pady=5)

        # Algorithms to be selected for comparision

        # Algorithm1 Label

        self.algorithmLabel1 = ttkb.Label(self.mainFrame,text="ALGORITHM1",font=("Poppins",15))

        self.algorithmLabel1.grid(row=6, column=0, sticky="w", padx=10, pady=5)

        self.algorithmLabel1.grid\_remove()

        # Algorithm1 Dropbox

        self.algorithm1 = ttkb.StringVar()

        self.algorithmCombobox1 = ttkb.Combobox(self.mainFrame,textvariable=self.algorithm1, values = ["Breadth For Search", "Bellman Ford", "Dijkstra"])

        self.algorithmCombobox1.grid(row=6, column=1, sticky="w", padx=10, pady=5)

        self.algorithmCombobox1.grid\_remove()

        # Algorithm2 Label

        self.algorithmLabel2 = ttkb.Label(self.mainFrame,text="ALGORITHM2",font=("Poppins",15))

        self.algorithmLabel2.grid(row=7, column=0, sticky="w", padx=10, pady=5)

        self.algorithmLabel2.grid\_remove()

        # Algorithm2 Dropbox

        self.algorithm2 = ttkb.StringVar()

        self.algorithmCombobox2 = ttkb.Combobox(self.mainFrame,textvariable=self.algorithm2, values = ["Breadth For Search", "Bellman Ford", "Dijkstra"])

        self.algorithmCombobox2.grid(row=7, column=1, sticky="w", padx=10, pady=5)

        self.algorithmCombobox2.grid\_remove()

        # Algorithm3 Label

        self.algorithmLabel3 = ttkb.Label(self.mainFrame,text="ALGORITHM3",font=("Poppins",15))

        self.algorithmLabel3.grid(row=8, column=0, sticky="w", padx=10, pady=5)

        self.algorithmLabel3.grid\_remove()

        # Algorithm3 Dropbox

        self.algorithm3 = ttkb.StringVar()

        self.algorithmCombobox3 = ttkb.Combobox(self.mainFrame,textvariable=self.algorithm3, values = ["Breadth For Search", "Bellman Ford", "Dijkstra"])

        self.algorithmCombobox3.grid(row=8, column=1, sticky="w", padx=10, pady=5)

        self.algorithmCombobox3.grid\_remove()

        # Custom Style for Run button

        self.buttonStyle = ttkb.Style()

        self.buttonStyle.configure('dark.TButton',font=("Poppins",20))

        # Run button

        # bootstyle="dark", style="dark.TButton, outline",

        self.runButton = ttkb.Button(self.mainFrame,text="Run", style="outline",command=self.runButton)

        self.runButton.grid(row=9, column=0, sticky="w", padx=10, pady=5)

compareVisible – Used to make Algorithm, Algorithm1, Algorithm2, Algorithm3 fields visible/hide based on the user input of compare checkbox.

 # Hide/Display fields based on compare checkbox

    def compareVisible(self):

        if self.compareAlgoVar.get():

            self.algorithmLabel1.grid()

            self.algorithmCombobox1.grid()

            self.algorithmLabel2.grid()

            self.algorithmCombobox2.grid()

            self.algorithmLabel3.grid()

            self.algorithmCombobox3.grid()

            self.algorithmLabel.grid\_remove()

            self.algorithmCombobox.grid\_remove()

        else:

            self.algorithmLabel1.grid\_remove()

            self.algorithmCombobox1.grid\_remove()

            self.algorithmLabel2.grid\_remove()

            self.algorithmCombobox2.grid\_remove()

            self.algorithmLabel3.grid\_remove()

            self.algorithmCombobox3.grid\_remove()

            self.algorithmLabel.grid()

            self.algorithmCombobox.grid()

activateInputRange – Hide/Display the range fields based on users input in “range” checkbox

# Hide/Display fields based on Size Range Checkbox

    def activateInputRange(self):

        if self.inputRangeVar.get():

            self.inputSizeRangeLowEntry.grid()

            self.inputSizeRangeHighEntry.grid()

            self.inputSizeRangeLabel.grid()

            self.inputSizeRangeStepSizeLabel.grid()

            self.inputSizeRangeStepSizeEntry.grid()

            self.inputSizeLabel.grid\_remove()

            self.inputSizeEntry.grid\_remove()

        else:

            self.inputSizeRangeLowEntry.grid\_remove()

            self.inputSizeRangeHighEntry.grid\_remove()

            self.inputSizeRangeLabel.grid\_remove()

            self.inputSizeRangeStepSizeLabel.grid\_remove()

            self.inputSizeRangeStepSizeEntry.grid\_remove()

            self.inputSizeLabel.grid()

            self.inputSizeEntry.grid()

runButton – This does basic validation on few input fields and if validation is successful triggers the program which runs the shortest path algorithms.

def runButton(self):

        # Validations

        if self.inputRangeVar.get() and self.compareAlgoVar.get():

            Messagebox.show\_warning("Range option is not available in compare mode!!")

        elif self.inputRangeVar.get() and (self.inputSizeRangeLowEntry.get() == "" or self.inputSizeRangeHighEntry.get() == "" or self.inputSizeRangeStepSizeEntry.get() == ""):

            Messagebox.show\_warning("Please enter the input size range and step size!")

        elif self.compareAlgoVar.get() and ((self.algorithmCombobox1.get() == "" and self.algorithmCombobox2.get() == "") or (self.algorithmCombobox1.get() == "" and self.algorithmCombobox3.get() == "") or (self.algorithmCombobox2.get() == "" and self.algorithmCombobox3.get() == "")):

            Messagebox.show\_warning("Please enter atleast two algorithms to compare!")

        elif ( self.compareAlgoVar.get() == False and self.algorithm.get() not in ["Breadth For Search", "Bellman Ford", "Dijkstra"] ) or ( self.compareAlgoVar.get() == True and ((self.algorithm1.get() and self.algorithm1.get() not in ["Breadth For Search", "Bellman Ford", "Dijkstra"]) or (self.algorithm2.get() and self.algorithm2.get() not in ["Breadth For Search", "Bellman Ford", "Dijkstra"]) or (self.algorithm3.get() and self.algorithm3.get() not in ["Breadth For Search", "Bellman Ford", "Dijkstra"]))):

            Messagebox.show\_warning("Please enter valid Algorithm")

        else:

            # Trigger the logic of Shortest Path Algorithms

            clearCache(dir=".")

            algoLogic.main(int(self.inputSizeEntry.get()),int(self.sourceVertexEntry.get()),self.algorithm.get(),self.compareAlgoVar.get(),self.algorithm1.get(),self.algorithm2.get(),self.algorithm3.get(),int(self.inputSizeRangeLowEntry.get()),int(self.inputSizeRangeHighEntry.get()),int(self.inputSizeRangeStepSizeEntry.get()),self.inputRangeVar.get())

\_\_main\_\_ - Main function of the program to trigger the ttkboostrap and GUI

root = ttkb.Window(themename="cyborg")

    shortestPathAlgorithmsApp(root)

    root.mainloop()

Breadth-first search

# Breadth First Search Algorithm to find the shortest path

def breadthFirstSearch(n,graph,sourceVertex):

    # BFS for specific vertex

    def bfsForVertex(vertex):

        for col in range(0,n):

            if col != vertex and graph[vertex][col] != 0 and col not in verticesPathDistance:

                verticesPathDistance[col] = verticesPathDistance[vertex] + 1

                vertQueue.put(col)

            elif col == vertex and col not in verticesPathDistance:

                verticesPathDistance[col] = 0

    def relaxQueue():

        while not(vertQueue.empty()):

            tempVertex = vertQueue.get()

            bfsForVertex(tempVertex)

    # Queue to hold vertices to be visited

    vertQueue = Queue(n)

    # Dictionary to hold distance of vertices

    verticesPathDistance = defaultdict(lambda: 0)

    # For Vertex sourceVertex check the shortest path for vertices connected and add connected vertices to queue

    bfsForVertex(sourceVertex)

    # Working with all the vertices which were connected to vertex 0 and continuing with BFS

    relaxQueue()

**Below is the list of functions used in program for all the shortesPathAlgorithms, code and the purpose of each function.**

**createGraph** – Used to generate graph of n vertices using networkx library and return the graph in 2D array format

# Fucntion Generating Graph using networkx Library to be used to run and test the algorithms

def createGraph(n):

    # Using Complete Graph from networkx to generate a graph of n vertices

    graph = genGraph.complete\_graph(n)

    graph2D = [[0 for \_ in range(n)] for \_ in range(n)]

    # Update weight for the graph edges randomly generated between 1 and 20

    for edge in graph.edges():

        graph[edge[0]][edge[1]]['weight'] = random.randrange(1,20)

    # Generate a 2D matrix from the generated graph

    for row in range(0,n):

        for col in graph.adj[row]:

            graph2D[row][col] = graph.adj[row][col]['weight']

    count = n//2

    while(count > 0):

        count -= 1

        row = random.randrange(1,n)

        col = random.randrange(1,n)

        graph2D[row][col] = 0

    # return the generated 2D Matrix of Graph and it's weight

    return graph2D

**plotComparisionSameInputSize** – To Generate Plot for an input size against time taken by multiple algorithms, comparing performance between algorithms.

# Generate Plot for an input size against time taken by multiple algorithms, comparing performance between algorithms

def plotComparisionSameInputSize(n,shortestPathAlgos,timeTaken):

    algoplot.plot(shortestPathAlgos,timeTaken,marker='x')

    algoplot.xlabel('Shortest Path Algorithms')

    algoplot.ylabel('Time Taken in milli seconds')

    title = 'Comparision of time taken between shortest path algorithms for ' + str(n) + ' Vertices.'

    algoplot.title(title)

    algoplot.show()

**plotForDifferentInputSizes** – To Generate Plot for multiple input sizes with time taken for each size by an algorithm.

# Generate Plot for multiple input sizes with time taken for each size by an algorithm

def plotForDifferentInputSizes(inputSizes,algorithm,timeTaken):

    algoplot.plot(inputSizes,timeTaken,marker='x')

    algoplot.xlabel('Input Sizes')

    algoplot.ylabel('Time Taken in milli seconds')

    title = 'Comparision of time taken by ' + algorithm + ' for different input sizes.'

    algoplot.title(title)

    algoplot.show()

**plotForInputSize** – Generate Plot for an input size against time taken by an algorithm, it would just be a point on a graph, using graph instead of printing

# Generate Plot for an input size against time taken by an algorithm, it would just be a point

# on a graph, using graph instead of printing

def plotForInputSize(inputSize,algorithm,timeTaken):

    algoplot.plot(inputSize,timeTaken,marker='x')

    algoplot.xlabel('Input Size')

    algoplot.ylabel('Time Taken in milli seconds')

    title = 'Time taken by ' + algorithm + ' for input size ' + str(inputSize[0]) + '.'

    algoplot.title(title)

    algoplot.show()

**prepareOut** - Printing graph just to use it to do testing.

def prepareOut(graph):

    print("\n")

    print("Generated Graph:")

    print(graph)

    print("\n")

**bfsTiming** - Function to check the time taken by BFS algorithm to find shortest path of all vertices from

source vertex. Calling the BFS Algorithm function “breadthFirstSearch” to trigger the algorithm and using time.monotonic\_ns() to get time in nanosecond level

# Function to check the time taken by BFS algorithm to find shortest path of all vertices from

# source vertex

def bfsTiming(n,graph,sourceVertex):

    print("Breadth First Search: ")

    bfsStartTime = time.monotonic\_ns()

    breadthFirstSearch(n,graph,sourceVertex)

    bfsEndTime = time.monotonic\_ns()

    timeTaken = (bfsEndTime-bfsStartTime)/1000000

    print(type(timeTaken))

    print("Time taken in milli seconds to find the distance of all other nodes from source node: ", timeTaken)

    return timeTaken

**djikstraTiming -** Function to check the time taken by Djikstra algorithm to find shortest path of all vertices from source vertex. Calling the Djikstra Algorithm function “djikstrasShortestPath” to trigger the algorithm and using time.monotonic\_ns() to get time in nanosecond level.

# Function to check the time taken by Djikstra algorithm to find shortest path of all vertices from

# source vertex

def djikstraTiming(n,graph,sourceVertex):

    print("Djikstra: ")

    djikstraStartTime = time.monotonic\_ns()

    djikstrasShortestPath(n,graph,sourceVertex)

    djikstraEndTime = time.monotonic\_ns()

    timeTaken = (djikstraEndTime-djikstraStartTime)/1000000

    print("Time taken in milli seconds to find the distance of all other nodes from source node: ", timeTaken)

    return timeTaken

**bellmanFordTiming -** Function to check the time taken by Bellman Ford algorithm to find shortest path of all vertices from source vertex. Calling the Bellman Ford Algorithm function “bellmanFordAlgo” to trigger the algorithm and using time.monotonic\_ns() to get time in nanosecond level.

# Function to check the time taken by Bellman Ford algorithm to find shortest path of all vertices from

# source vertex

def bellmanFordTiming(n,graph,sourceVertex):

    print("Bellman Ford: ")

    bellmanFordStartTime = time.monotonic\_ns()

    bellmanFordAlgo(n,graph,sourceVertex)

    bellmanFordEndTime = time.monotonic\_ns()

    timeTaken = (bellmanFordEndTime-bellmanFordStartTime)/1000000

    print("Time taken in milli seconds to find the distance of all other nodes from source node: ", timeTaken)

    return timeTaken

**Bellman Ford Algorithm(bellmanFordAlgo) –** Holds the Bellman Ford algorithm to find the shortest paths. Also includes logic to check if there is a negative cycle which results in wrong shortest paths, if exists provides an error message.

**relaxEdges –** To relax all the edges for the graph, calling this in loop and relaxing V-1 times. This is within the Bellman Ford function.

# Bellman Ford Algorithm to find the shortest path

def bellmanFordAlgo(n,graph,sourceVertex):

    # Relax all the edges

    def relaxEdges(n,graph,negativeCycleCheck):

        for source in range(0,n):

            for dest in range(0,n):

                if dest != source and graph[source][dest] != 0 and verticesDistanceDict[dest]>(verticesDistanceDict[source] + graph[source][dest]):

                    if negativeCycleCheck == True:

                        return True

                    verticesDistanceDict[dest] = verticesDistanceDict[source] + graph[source][dest]

    # Dictionary to hold the list of vertices and their distances

    verticesDistanceDict = {}

    # Build the dictionary with weight as 0 for source vertex and inf for all other vertices

    verticesDistanceDict[sourceVertex] = 0

    for i in range(0,n):

        if i == sourceVertex:

            continue

        verticesDistanceDict[i] = float('inf')

    # Relax the edges V-1 times, V-Vertices

    counter = 1

    while counter < n:

        relaxEdges(n,graph,False)

        counter += 1

    # Check if there exits a negative cycle

    hasNegativeCycle = relaxEdges(n,graph,True)

    if hasNegativeCycle == True:

        print("Cannot determine shortest path, negative cycle exists")

**Breadth First Search Algorithm (breadthFirstSearch) -** Holds Breadth First Search Algorithm to check the shortest path.

**bfsForVertex –** Within the breadthFirstSearch function, used to find all the vertices connected to the Vertex and update their distances. Using “Queue” from “queue” as stack to keep stock of vertices to be further visited and pop the visited vertices.

**relaxQueue –** This will relax all the vertices connected to source vertex. These were added to queue when source vertex was relaxed. Now further all these vertices will be relaxed by calling “bfsForVertex” for each vertex of queue and then inside “bfsForVertex” queue will be updated with new vertices to be visited.

# Breadth First Search Algorithm to find the shortest path

def breadthFirstSearch(n,graph,sourceVertex):

    # BFS for specific vertex

    def bfsForVertex(vertex):

        for col in range(0,n):

            if col != vertex and graph[vertex][col] != 0 and col not in verticesPathDistance:

                verticesPathDistance[col] = verticesPathDistance[vertex] + 1

                vertQueue.put(col)

            elif col == vertex and col not in verticesPathDistance:

                verticesPathDistance[col] = 0

    def relaxQueue():

        while not(vertQueue.empty()):

            tempVertex = vertQueue.get()

            bfsForVertex(tempVertex)

    # Queue to hold vertices to be visited

    vertQueue = Queue(n)

    # Dictionary to hold distance of vertices

    verticesPathDistance = defaultdict(lambda: 0)

    # For Vertex sourceVertex check the shortest path for vertices connected and add connected vertices to queue

    bfsForVertex(sourceVertex)

    # Working with all the vertices which were connected to vertex 0 and continuing with BFS

    relaxQueue()

**Djikstra Algorithm (djikstrasShortestPath) –** Holds the Djikstra Algorithm logic to find the shortest path. Calls “relaxation” function to relax the sourcevertex.

**Relaxation -** Starting from source vertex and further relaxing vertices based on their current distance from source vertex. Selecting the next vertex connected to source vertex with minimum distance and calling the “relaxation” function recursively.

# Djikstra Algorithm to find the shortest path

def djikstrasShortestPath(n,graph,sourceVertex):

    # Relaxation of all the vertices

    def relaxation(vertex):

        if len(unvisitedVertices) <= 0:

            return

        if vertex in visitedVertices:

            return

        unvisitedVertices.remove(vertex)

        visitedVertices.append(vertex)

        for i in range(0,n):

            if vertex != i and graph[vertex][i] > 0 and verticesPathDistance[i] > (verticesPathDistance[vertex] + graph[vertex][i]):

                verticesPathDistance[i] = verticesPathDistance[vertex] + graph[vertex][i]

        minVertex = 0

        minDistance = float('inf')

        for i in range(0,n):

            if vertex != i and graph[vertex][i] > 0:

                if minDistance > verticesPathDistance[i]:

                    minVertex = i

                    minDistance = verticesPathDistance[i]

        relaxation(minVertex)

    # Dictionary to hold distance of vertices

    verticesPathDistance = {}

    # List to maintain unvisited vertices

    unvisitedVertices = []

    # List to maintain visited vertices

    visitedVertices = []

    # Considering sourceVertex as the source node, so assiging distance as 0 and rest of the vertices will be

    # assigned float(inf) to represent it as infinity. Keeping list of all possible vertices

    # Also update unvisitedVertices list with all the vertices

    verticesPathDistance[sourceVertex] = 0

    unvisitedVertices.append(sourceVertex)

    for i in range(0,n):

        if i == sourceVertex:

            continue

        verticesPathDistance[i] = float('inf')

        unvisitedVertices.append(i)

    # Begin relaxation of all vertices

    relaxation(sourceVertex)

**handleCompare:** Used to check the algorithm selected by user for the compare run mode and trigger the right algorithm. Take the time taken by the algorithm and return the name of algorithm and time taken. This function is called once for each variable.

# When Compare Checkbox is selected on GUI, algorithms whose run time needs to be compared will be

# passed, based on that, running the algorithms and updating the time taken and xaxis name to be

# considered on plot

def handleCompare(algorithm,xaxis,yaxis,n,graph,sourceVertex):

    if algorithm == "Breadth For Search":

        yaxis.append(bfsTiming(n,graph,sourceVertex))

        xaxis.append('BFS')

    elif algorithm == "Bellman Ford":

        yaxis.append(bellmanFordTiming(n,graph,sourceVertex))

        xaxis.append('Bellman Ford')

    elif algorithm == "Dijkstra":

        yaxis.append(djikstraTiming(n,graph,sourceVertex))

        xaxis.append('Djikstra')

**handleSinglerun:** Used to check the algorithm selected by user for the normal run mode and trigger the right algorithm. Take the time taken by the algorithm and return the name of algorithm and time taken.

# When running one algorithm for a specific input size, running that algorithm and displaying

# the plot which shows the runtime for that algorithm

def handleSinglerun(algorithm,xaxis,yaxis,n,graph,sourceVertex):

    xaxis.append(n)

    if algorithm == "Breadth For Search":

        yaxis.append(bfsTiming(n,graph,sourceVertex))

    elif algorithm == "Bellman Ford":

        yaxis.append(bellmanFordTiming(n,graph,sourceVertex))

    elif algorithm == "Dijkstra":

        yaxis.append(djikstraTiming(n,graph,sourceVertex))

**inputSizeRange:** Used to check the algorithm selected by user for the vertices range mode and trigger the right algorithm. Take the time taken by the algorithm and return the name of algorithm and time taken.

# When running one algorithm for a input range, running that algorithm and displaying

# the plot which shows the runtimes for that algorithm against multiple ranges

def inputSizeRange(algorithm,sourceVertex,rangeLow,rangeHigh,rangeStep,xaxis,yaxis):

    for inSize in range(rangeLow,rangeHigh+1,rangeStep):

        xaxis.append(inSize)

        graph = createGraph(inSize)

        prepareOut(graph)

        if algorithm == "Breadth For Search":

            yaxis.append(bfsTiming(inSize,graph,sourceVertex))

        elif algorithm == "Dijkstra":

            yaxis.append(djikstraTiming(inSize,graph,sourceVertex))

        elif algorithm == "Bellman Ford":

            yaxis.append(bellmanFordTiming(inSize,graph,sourceVertex))

**main:** Main function to handle the details received from GUI and execute the right mode and algorithms and print the plot.

def main(n,sourceVertex,algorithm,compare,algorithm1,algorithm2,algorithm3,rangeLow,rangeHigh,rangeStep,rangeCheckBox):

    # Build 2d Array for Weighted Graph, a 2D array generated, initial 2D array logic

    # graph = [ [random.randrange(1,20) for i in range(0,n)] for j in range(0,n)]

    # Djikstra/BFS Test Case

    # graph = [[0,2,4,0,0,0],[0,0,1,7,0,0],[0,0,0,0,3,0],[0,0,0,0,0,1],[0,0,0,2,0,5],[0,0,0,0,0,0]]

    # Bellman/BFS Ford Test Case

    # graph = [[0,6,5,5,0,0,0],[0,0,0,0,-1,0,0],[0,-2,0,0,1,0,0],[0,0,-2,0,0,-1,0],[0,0,0,0,0,0,3],[0,0,0,0,0,0,3],[0,0,0,0,0,0,0]]

    if compare == True:

        # Generate Graph for input size provided

        graph = createGraph(n)

        # Printing graph in output for reference

        prepareOut(graph)

        # Initialize xaxis and yaxis

        xaxis = []

        yaxis = []

        # Running for each algorithms passed

        handleCompare(algorithm1,xaxis,yaxis,n,graph,sourceVertex)

        handleCompare(algorithm2,xaxis,yaxis,n,graph,sourceVertex)

        handleCompare(algorithm3,xaxis,yaxis,n,graph,sourceVertex)

        # Plot the comparision

        plotComparisionSameInputSize(n,xaxis,yaxis)

    elif rangeCheckBox == False:

        # Initialize xaxis and yaxis

        xaxis = []

        yaxis = []

        # Generate Graph for input size provided

        graph = createGraph(n)

        # Printing graph in output for reference

        prepareOut(graph)

        # Run the algorithm to get the time taken for input size

        handleSinglerun(algorithm,xaxis,yaxis,n,graph,sourceVertex)

        # Plot the output

        plotForInputSize(xaxis,algorithm,yaxis)

    elif rangeCheckBox == True:

        # Initialize xaxis and yaxis

        xaxis = []

        yaxis = []

        # Run the algorithm to get the time taken for range of input sizes

        inputSizeRange(algorithm,sourceVertex,rangeLow,rangeHigh,rangeStep,xaxis,yaxis)

        # Plot the graph for different input sizes against run time taken by an algorithm

        plotForDifferentInputSizes(xaxis,algorithm,yaxis)

**Test Cases:**

1. Normal Run Mode for all the algorithms.

**Breadth first Search**(Source Vertex - 0, Number of Vertices – 2000)

Took 906 milli seconds

A screenshot of a computer

Description automatically generated

**Djikstra Algorithm**

Took close to 0 milli seconds since it is random generated graph. The second screen shot is for 4000 vertices, for which Djikstra took 16 milli seconds.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Bellman Ford Algorithm**

Considering that the graph is very dense, using 300 vertices, source vertex as 0. Took

2859 milli seconds.

A screenshot of a computer

Description automatically generated

1. Compare run case: running the algorithms for 700 vertices and comparing the run time of all the algorithms. Source Vertex – 0, Number of Vertices – 700, selected all three algorithms for comparison.

BFS – 31milli seconds

Bellman Ford – 74391 milli seconds

Djikstra – close to 0 milli seconds

A screenshot of a computer

Description automatically generated

1. Range of Vertices run case, running the algorithms for range of vertices from 200 to 400 with step size as 20. So that we can see the time taken by each algorithm for vertices 200,220,240…

**Breadth First Search**

**A screenshot of a computer

Description automatically generated**

**Bellman Ford**

**A screenshot of a computer

Description automatically generated**

**Djikstra**(Running it from 200 to 1000 since it is really fast and auto generated graph has almost all the vertices connected to first vertex, difficult to get a case where it takes more time and we can see it on graph)

A screenshot of a computer

Description automatically generated