Proyecto entrega #3

Members:

Juan Pablo Rueda Vera

Sebastian Vasquez Saldarriaga

Prevention of sexual harassment through a route search algorithm using near search

How can we determine the safety of the routes?

Once again, working on our path from simple solutions to more complex solutions, we can:

-Determine the overall safety of the route only by the risk score associated with the destination

-Calculate the average of risk scores based on the coverage of the straight-line route grid from origin to destination

-Calculate the average of the risk scores based on the network coverage of each step of the route to reach the destination

-here we use a linearized method and assume that there is no curvature between the points for simplicity

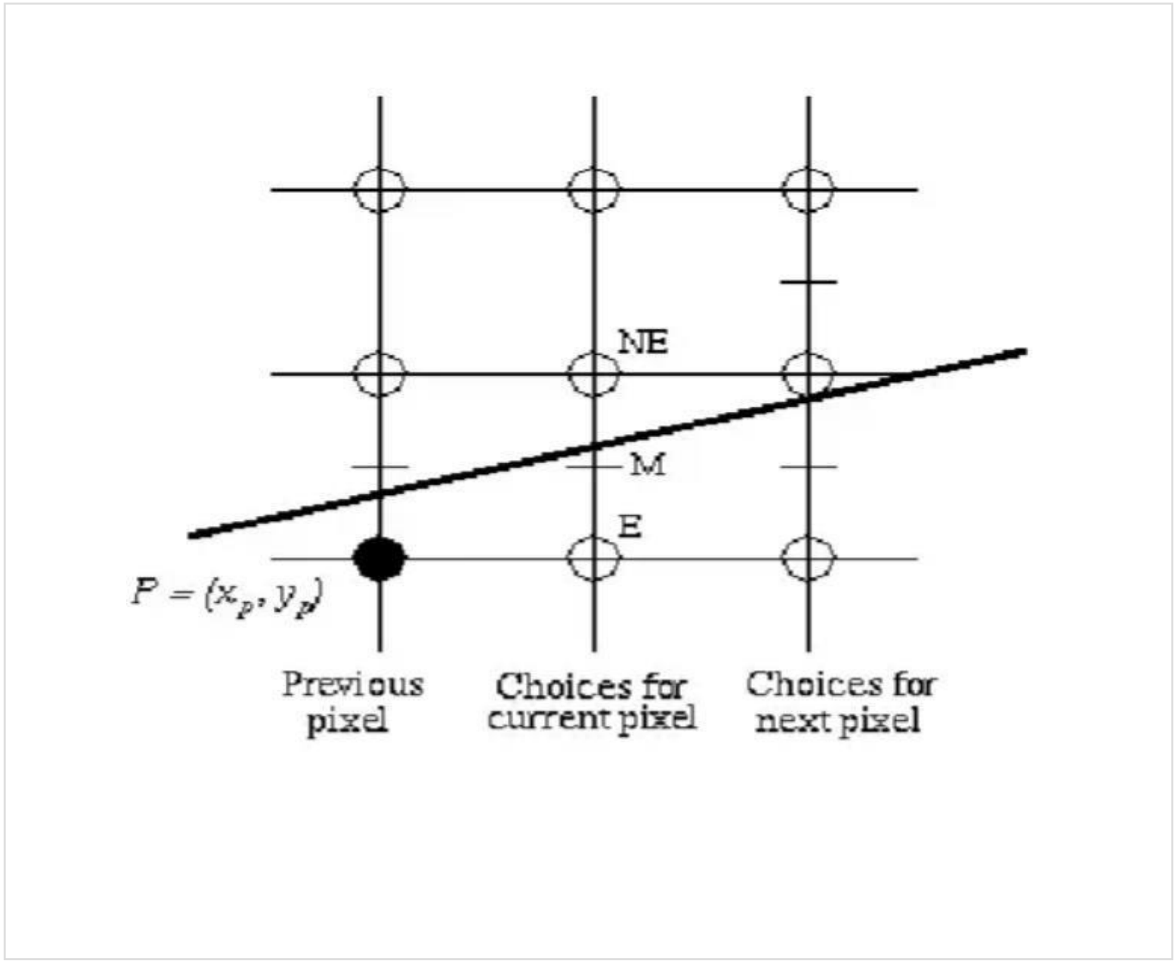
$$d(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_n - b_n)^2}$$

Taking on the second and third solutions requires leveraging external resources: we can reframe this problem as one that involves determining the line of sight.

How do we know if a route touches a grid point? Equipped with the knowledge that there is more or less an algorithm for everything, we turned to Bresenham's line drawing method. is a line drawing algorithm that determines the points of an n-dimensional raster that must be selected to form an approximation close to a straight line between two points.

Computer graphics work in units of pixels, so we can see how this path algorithm originated from the need to shade graphic operations in pixels. The general idea behind this algorithm is: given an initial endpoint of a line segment, the next point on the grid that crosses for reaching the other endpoint is determined by evaluating where the line segment crosses relative to the midpoint (above or below) of the two possible grid point options as an attempt to walk through this method.

Now we will study and show the function of the following diagram:



-Here, we assume that the points refer to each grid point (which are the center point for each grid frame)

Suppose we have a line segment that extends from left to right in the form of an upward slope (smaller (x, y) to larger (x, y) coordinates. Let's also say that the starting point of the P grid is shaded and has already been determined to be traversed through it.

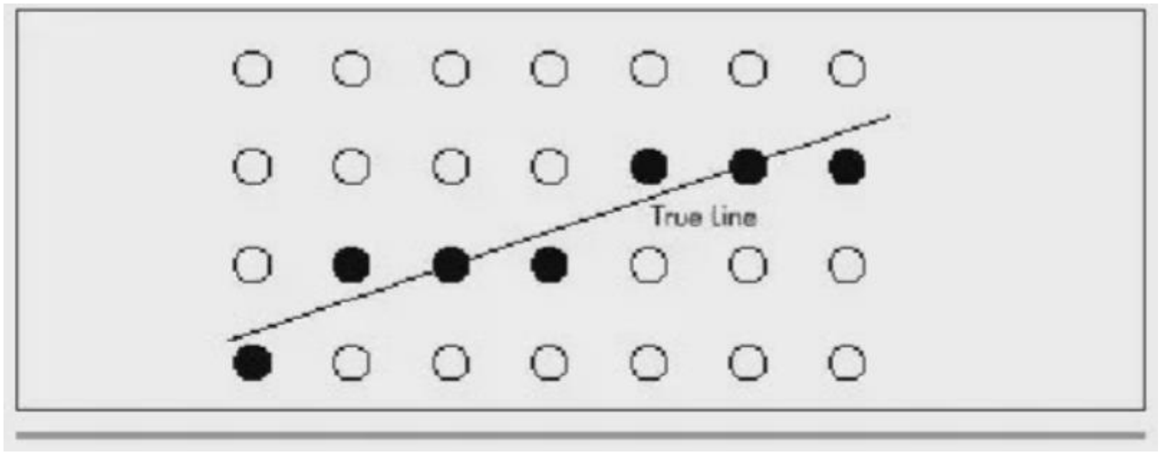
-Consider the point P as the point of the previous grid, then we must determine the point of the current grid to be shaded. To do this, we also consider grid points E and NE (for the east and northeast of the current grid point, respectively)

-M is the midpoint between E and NE

- Based on where the line segment between the line between E and NE and the relative intersection point M (above or below M) intersects, we can make a decision about the point of the current grid that the line crosses, which is NE in this case, since the line intersects above M

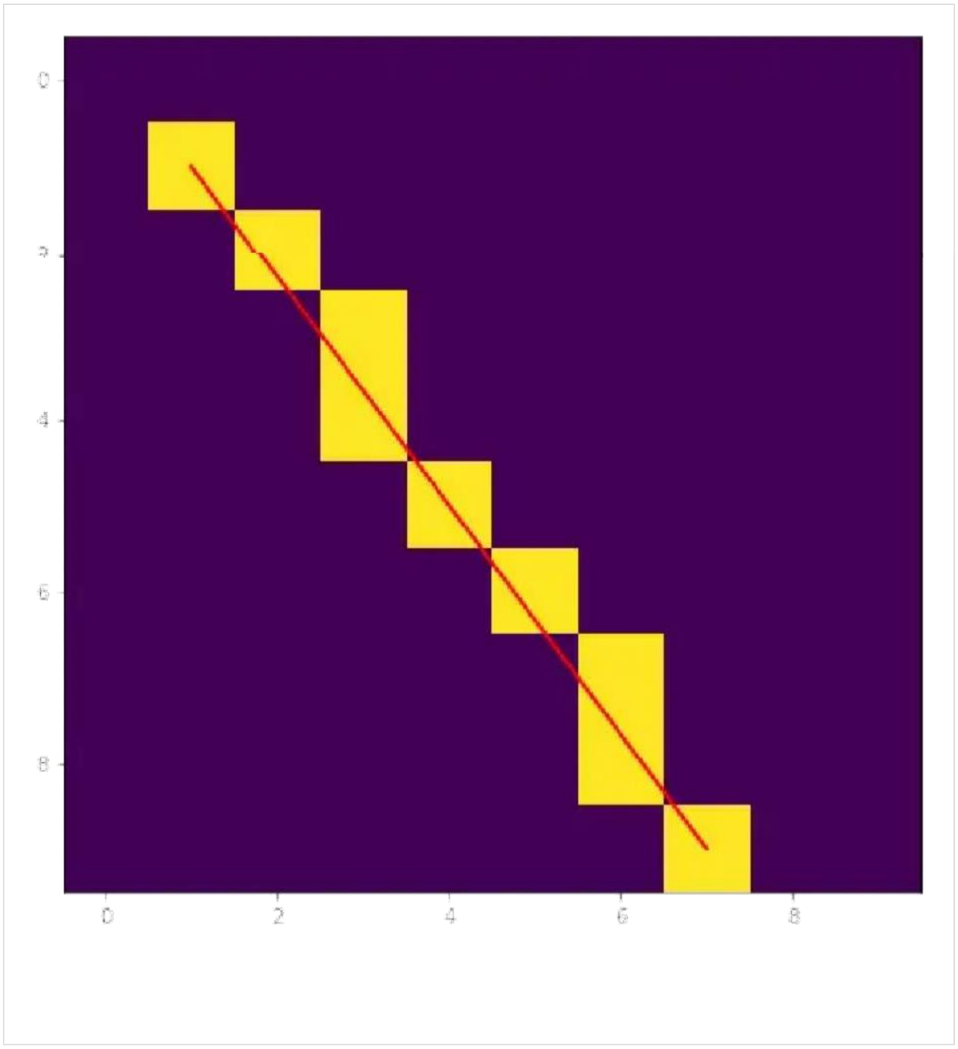
-The points of grid E and NE (and, consequently, M) are updated in reference to the point of the current grid and the same methods above are applied again and again until we reach the opposite end point of the line segment.

When we complete the pathfinding algorithm process, we come up with something like this:

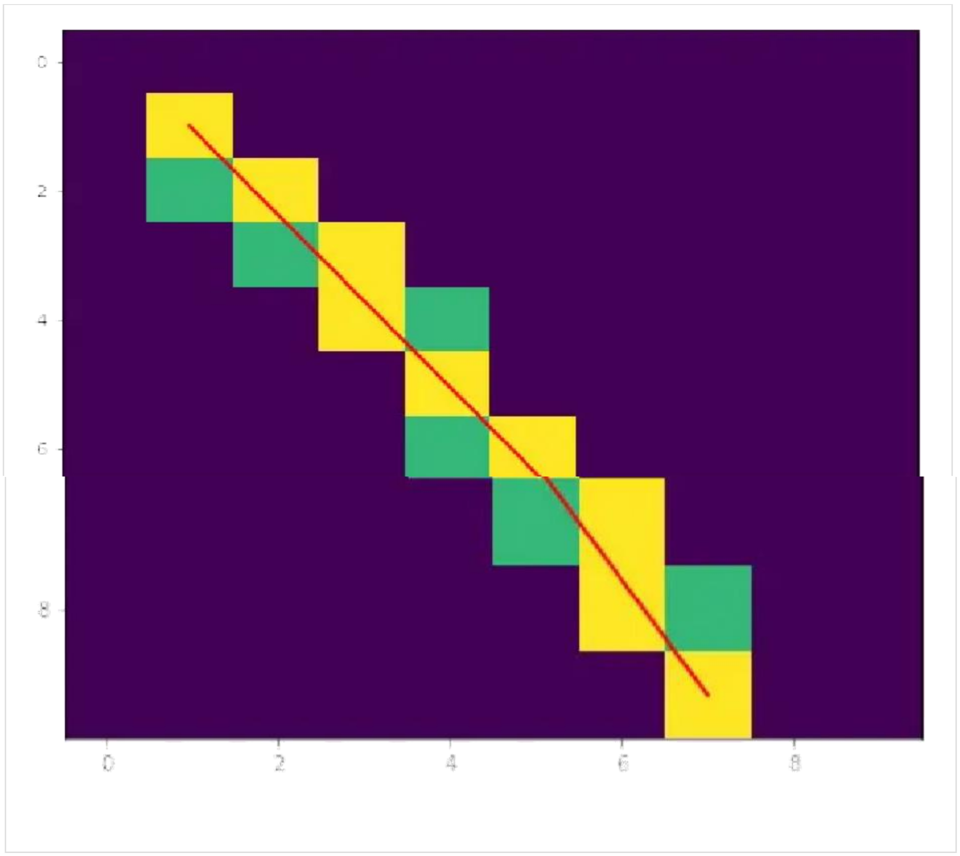


skimage already has an implementation of this: here's a little snippet of what the code looks like:

```
from skimage.draw
import line import
matplotlib.pyplot as plt
# create array of zeros
example_arr =
np.zeros((10, 10)) # set
endpoints for line
segment x0, y0 = 1, 1

x1, y1 = 7, 9

# dibuja la línea

rr, cc = línea(y0, x0, y1,
x1) example_arr[rr, cc] = 1

# plot it out
plt.figure(figsize=(8, 8))

plt.imshow(example_arr, interpolation='nearest', cmap='viridis')
plt.plot([x0, x1], [y0, y1], linewidth=2, color='r')
```



As you can see, the grid point shading approximations are imperfect, as there are definitely grid points that the line has crossed that are not shaded. So this can involve problems because we do not know very exactly the coordinates where our slope passes exactly, and these coordinates we must know them at each exact point in order to travel the entire slope without any inconvenience.Of course, there is another route search algorithm to improve this where all the points that are crossed are shaded; I won't go into detail here, but the term used is to define the supercover of the line segment:



Going back to our problem at hand here, we applied these grid coverage methods to calculate the verage of step risk scores for each route to determine it best. We prioritize route safety first, before accounting for distance (i.e., the safest route is first suggested if two routes are linked in safety, then suggest the nearest destination), using heat map analysis to prevent cases of sexual harassment.

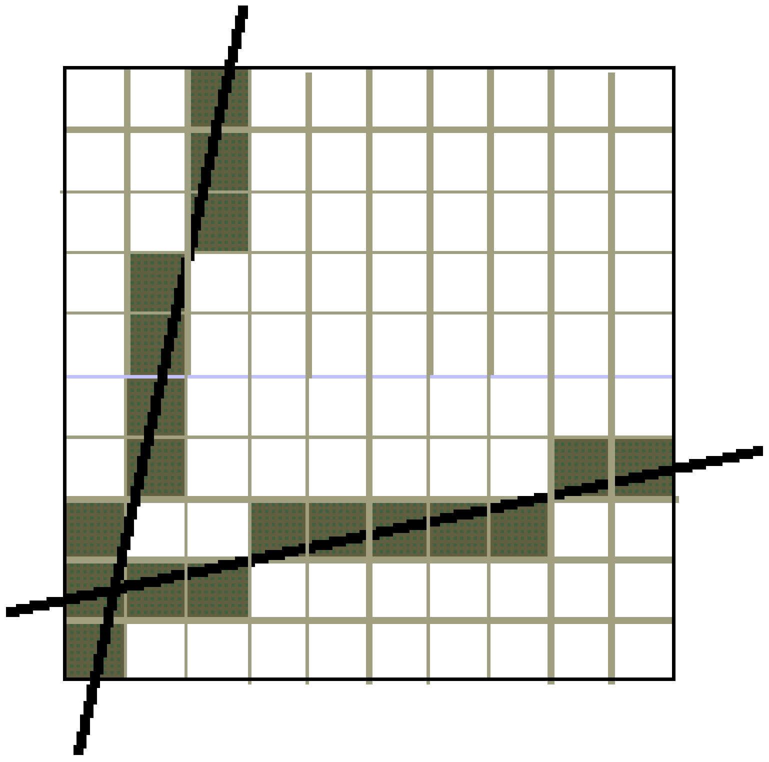
So in conclusion to know the algorithms we must follow this series of steps:

-Calculate average distance from point A to point B.

-Calculate the routes that are safest in that route first by tracing the route from point A to point B and then we can have all the information of the route and thus mark the most dangerous intermediate points that can be involved

So we use a linearized method and assume that there is no curvature between the points for simplicity.it consists of finding a linear function that can approximate a given function around a point. The first step to solving an optimization problem is to model reality with mathematical language, that is, rewrite it using variables and relationships between them.

Then we turn to Bresenham's line drawing method which is a quick method for drawing lines in graphic devices, whose most appreciated quality is that it only performs calculations with integers. That will help us to draw a straight line of two points in order to know its route. And after having the route we take the distance that this has and knowing if it is the shortest also helps us to recognize several routes, and allows us to have information of each one and with this we implement which route is the safest, either opinions or recommendations of other people who have already passed said route.

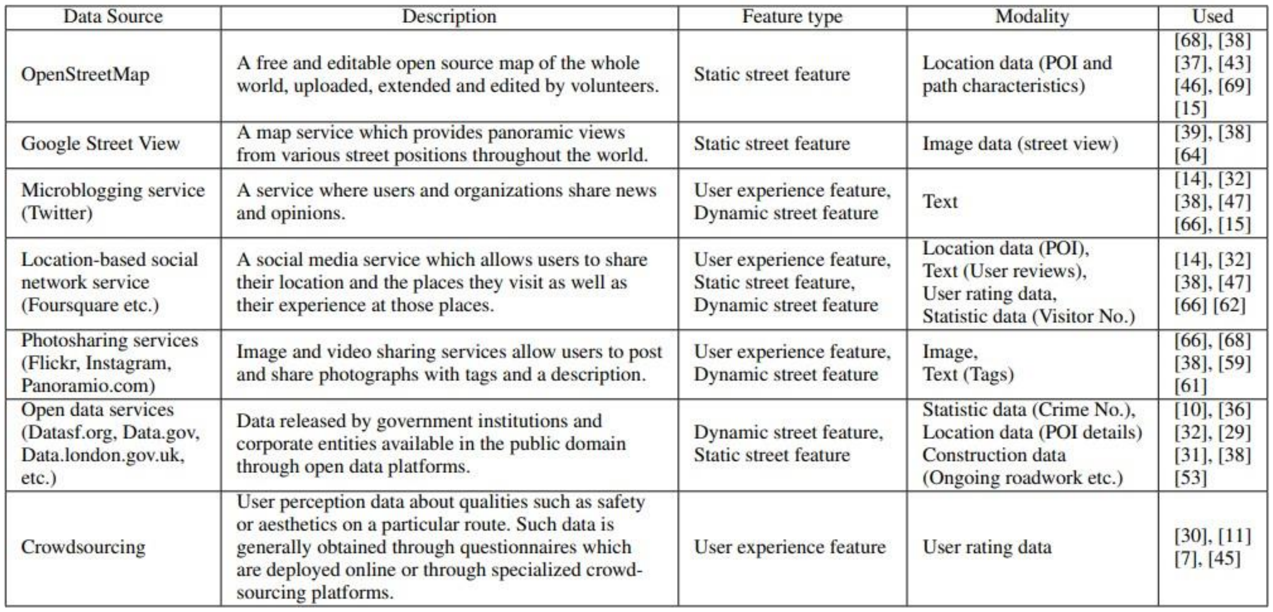


Beyond the Shortest Route: A Quality-Conscious Route Navigation Survey for Pedestrians

A summary of this system is that it provides a route that allows the user to move from starting point A to starting point B.

this system is largely based on collecting information based on police cases or stories of violence and among others.

TABLE 1. A summary of the key data sources used to implement quality-aware route recommendation systems



Data Source	Description	Feature type	Modality	Used
OpenStreetMap	A free and editable open source map of the whole world, uploaded, extended and edited by volunteers.	Static street feature	Location data (POI and path characteristics)	[68], [38] [37], [43] [46], [69] [15]
Google Street View	A map service which provides panoramic views from various street positions throughout the world.	Static street feature	Image data (street view)	[39], [38] [64]
Microblogging service (Twitter)	A service where users and organizations share news and opinions.	User experience feature, Dynamic street feature	Text	[14], [32] [38], [47] [66], [15]
Location-based social network service (Foursquare etc.)	A social media service which allows users to share their location and the places they visit as well as their experience at those places.	User experience feature, Static street feature, Dynamic street feature	Location data (POI), Text (User reviews), User rating data, Statistic data (Visitor No.)	[14], [32] [38], [47] [66] [62]
Photosharing services (Flickr, Instagram, Panoramio.com)	Image and video sharing services allow users to post and share photographs with tags and a description.	User experience feature, Dynamic street feature	Image, Text (Tags)	[66], [68] [38], [59] [61]
Open data services (Datasf.org, Data.gov, Data.london.gov.uk, etc.)	Data released by government institutions and corporate entities available in the public domain through open data platforms.	Dynamic street feature, Static street feature	Statistic data (Crime No.), Location data (POI details) Construction data (Ongoing roadwork etc.)	[10], [36] [32], [29] [31], [38] [53]
Crowdsourcing	User perception data about qualities such as safety or aesthetics on a particular route. Such data is generally obtained through questionnaires which are deployed online or through specialized crowd-sourcing platforms.	User experience feature	User rating data	[30], [11] [7], [45]

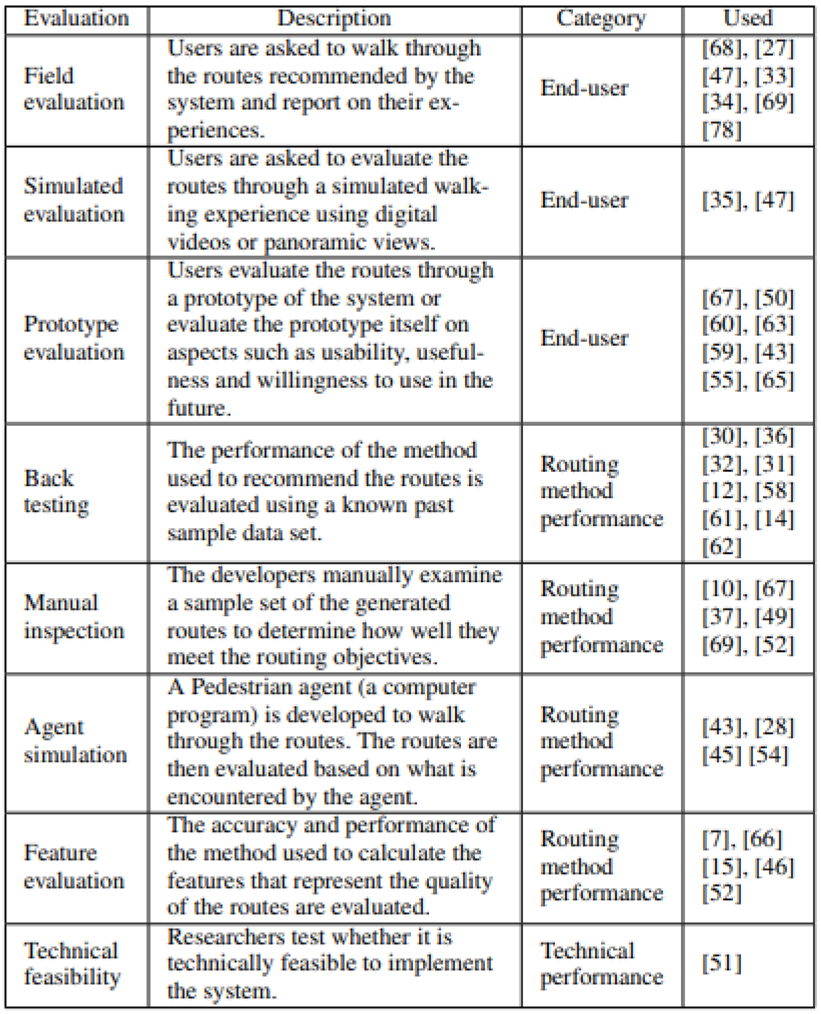
In this table they show us the information of the streets and which are constantly updated if there is any change and which remain intact since they are uploaded to the system

It also shows us the sources and a description of what we can find

It also contains the modality that in a few words are how the information is displayed.

The Routing Algorithm is a very popular algorithm from Dijkstra's as it generates as a tree of shorter paths, The algorithm maintains two sets of nodes, a set of nodes in the tree of the shortest path and a set of other nodes that have not yet been included in the tree of the shortest path. at each step of the algorithm a node is identified that is not in the shortest path tree and has a minimum distance from the initial node and moves to the node set in the shortest path tree

TABLE 2. A summary of the methods used to evaluate the navigation systems



Evaluation	Description	Category	Used
Field evaluation	Users are asked to walk through the routes recommended by the system and report on their experiences.	End-user	[68], [27] [47], [33] [34], [69] [78]
Simulated evaluation	Users are asked to evaluate the routes through a simulated walking experience using digital videos or panoramic views.	End-user	[35], [47]
Prototype evaluation	Users evaluate the routes through a prototype of the system or evaluate the prototype itself on aspects such as usability, usefulness and willingness to use in the future.	End-user	[67], [50] [60], [63] [59], [43] [55], [65]
Back testing	The performance of the method used to recommend the routes is evaluated using a known past sample data set.	Routing method performance	[30], [36] [32], [31] [12], [58] [61], [14] [62]
Manual inspection	The developers manually examine a sample set of the generated routes to determine how well they meet the routing objectives.	Routing method performance	[10], [67] [37], [49] [69], [52]
Agent simulation	A Pedestrian agent (a computer program) is developed to walk through the routes. The routes are then evaluated based on what is encountered by the agent.	Routing method performance	[43], [28] [45] [54]
Feature evaluation	The accuracy and performance of the method used to calculate the features that represent the quality of the routes are evaluated.	Routing method performance	[7], [66] [15], [46] [52]
Technical feasibility	Researchers test whether it is technically feasible to implement the system.	Technical performance	[51]

This chart is based on people's experiences in taking the desired path.

Such as what they thought or if they liked it or not in order to continue recommending it to other users

EVALUATION OF ROUTE NAVIGATION SYSTEMS

Several methods have been used to assess whether a system is effective in recommending routes to users based on different criteria (e.g. pleasure or safety). Table 2 provides a summary of these methods. The most comprehensive approach would be to ask participants to travel the routes recommended by the system in order to measure and compare the aspects of utility and cost. In this way, one could test the system with real users in a realistic context. Recommended routes are often compared to those proposed by routing methods

```
from geopy.distance import great_circle# import folium package
import folium
import math
import sys
my_map4 = folium.Map(location = [6.244203, -75.5812119],
                                    zoom_start = 12)

folium.Marker([6.244203, -75.5812119],
            popup = 'Delhi').add_to(my_map4)

folium.Marker([6.3271652, -75.571391],
            popup = 'GeeksforGeeks').add_to(my_map4)

# Add a line to the map by using line method .
# it connect both coordinates by the line
# line_opacity implies intensity of the line

folium.PolyLine(locations = [(6.244203, -75.5812119), (6.3271652, -75.571391)],
              line_opacity = 0.5).add_to(my_map4)

# Loading the lat-long data for Kolkata & Delhi
src = (6.244203, -75.5812119)
dest = (6.3271652, -75.571391)

# Print the distance calculated in km
print(great_circle(src, dest).km)



# For storing the vertex set to retrieve node with the lowest distance
class PriorityQueue:
```

```
# Based on Min Heap
def __init__(self):
    self.cur_size = 0
    self.array = []
    self.pos = {}  # To store the pos of node in array

def is_empty(self):
    return self.cur_size == 0

def min_heapify(self, idx):
    lc = self.left(idx)
    rc = self.right(idx)
    if lc < self.cur_size and self.array(lc)[0] < self.array(idx)[0]:
        smallest = lc
    else:
        smallest = idx
    if rc < self.cur_size and self.array(rc)[0] < self.array(smallest)[0]:
        smallest = rc
    if smallest != idx:
        self.swap(idx, smallest)
        self.min_heapify(smallest)

def insert(self, tup):
    # Inserts a node into the Priority Queue
    self.pos[tup[1]] = self.cur_size
    self.cur_size += 1
    self.array.append((sys.maxsize, tup[1]))
    self.decrease_key((sys.maxsize, tup[1]), tup[0])

def extract_min(self):
    # Removes and returns the min element at top of priority queue
    min_node = self.array[0][1]
    self.array[0] = self.array[self.cur_size - 1]
    self.cur_size -= 1
    self.min_heapify(1)
    del self.pos[min_node]
    return min_node

def left(self, i):
    # returns the index of left child
    return 2 * i + 1

def right(self, i):
    # returns the index of right child
    return 2 * i + 2

def par(self, i):
    # returns the index of parent
    return math.floor(i / 2)

def swap(self, i, j):
    # swaps array elements at indices i and j
    # update the pos{}
    self.pos[self.array[i][1]] = j
    self.pos[self.array[j][1]] = i
    temp = self.array[i]
    self.array[i] = self.array[j]
    self.array[j] = temp

def decrease_key(self, tup, new_d):
    idx = self.pos[tup[1]]
    # assuming the new_d is atmost old_d
    self.array[idx] = (new_d, tup[1])
    while idx > 0 and self.array[self.par(idx)][0] > self.array[idx][0]:
            self.swap(idx, self.par(idx))
            idx = self.par(idx)



class Graph:
    def __init__(self, num):
        self.adjList = {}  # To store graph: u -> (v,w)
        self.num_nodes = num  # Number of nodes in graph
        # To store the distance from source vertex
        self.dist = [0] * self.num_nodes
        self.par = [-1] * self.num_nodes  # To store the path

    def add_edge(self, u, v, w):
        #  Edge going from node u to v and v to u with weight w
        # u (w)-> v, v (w) -> u
        # Check if u already in graph
        if u in self.adjList:
            self.adjList[u].append((v, w))
        else:
            self.adjList[u] = [(v, w)]

        # Assuming undirected graph
        if v in self.adjList:
            self.adjList[v].append((u, w))
        else:
            self.adjList[v] = [(u, w)]

    def show_graph(self):
        # u -> v(w)
        for u in self.adjList:
            print(u, "->", " -> ".join(str(f"{v}({w})") for v, w in self.adjList[u]))
```

```
def dijkstra(self, src):
    # Flush old junk values in par[]
    self.par = [-1] * self.num_nodes
    # src is the source node
    self.dist[src] = 0
    q = PriorityQueue()
    q.insert((0, src))  # (dist from src, node)
    for u in self.adjList.keys():
        if u != src:
            self.dist[u] = sys.maxsize  # Infinity
            self.par[u] = -1

    while not q.is_empty():
        u = q.extract_min()  # Returns node with the min dist from source
        # Update the distance of all the neighbours of u and
        # if their prev dist was INFINITY then push them in Q
        for v, w in self.adjList[u]:
            new_dist = self.dist[u] + w
            if self.dist[v] > new_dist:
                if self.dist[v] == sys.maxsize:
                    q.insert((new_dist, v))
                else:
                    q.decrease_key((self.dist[v], v), new_dist)
                self.dist[v] = new_dist
                self.par[v] = u

    # Show the shortest distances from src
    self.show_distances(src)


def show_distances(self, src):
    print(f"Distance from node: {src}")
    for u in range(self.num_nodes):
            print(f"Node {u} has distance: {self.dist[u]}")

    def show_path(self, src, dest):
        # To show the shortest path from src to dest
        # WARNING: Use it *after* calling dijkstra
        path = []
        cost = 0
        temp = dest
        # Backtracking from dest to src
        while self.par[temp] != -1:
            path.append(temp)
            if temp != src:
                for v, w in self.adjList[temp]:
                    if v == self.par[temp]:
                        cost += w
                        break
            temp = self.par[temp]
        path.append(src)
        path.reverse()

        print(f"----Path to reach {dest} from {src}----")
        for u in path:
            print(f"{u}", end=" ")
            if u != dest:
                print("-> ", end="")

        print("\nTotal cost of path: ", cost)



if __name__ == "__main__":
    graph = Graph(10000)
    graph.add_edge((6.244203, -75.5812119),(-75.5724985, 6.2113756),(6.20020215, -
75.5784855279717))
```

```
    graph.add_edge((-75.5695566, 6.2502512), (-75.5706449, 6.2106721), (-75.5663399,
6.2099192))
    graph.show_graph()
    graph.dijkstra(1)
    graph.show_path(0, 4)


my_map4.save("my_map4.html")




import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt
from shapely import wkt

#############################################
personas = {
    "nombre": ["Dimas", "Juan", "Juana"],
    "Edad": [23, 24, 25],
    "país": ["España", "México", "Chile"]
}

#df = pd.DataFrame(personas)
#print(df)
#############################################

#datos = pd.read_csv('https://raw.githubusercontent.com/mauriciotoro/ST0245-
Eafit/master/proyecto/Datasets/calles_de_medellin_con_acoso.csv', sep=';', index_col=0)
#Podemos utilizar print(datos.head(3)) para imprimir unicamente 3 filas
#print(datos)
#load data
```

```
edges = pd.read_csv('https://raw.githubusercontent.com/mauriciotoro/ST0245-Eafit/master/proyecto/Datasets/calles_de_medellin_con_acoso.csv%27,sep=%27;%27)

edges['geometry'] = edges['geometry'].apply(wkt.loads)

edges = gpd.GeoDataFrame(edges)


area = pd.read_csv('poligono_de_medellin.csv',sep=';')

area['geometry'] = area['geometry'].apply(wkt.loads)

area = gpd.GeoDataFrame(area)


#Create plot

fig, ax = plt.subplots(figsize=(12,8))


# Plot the footprint

area.plot(ax=ax, facecolor='black')


# Plot street edges

edges.plot(ax=ax, linewidth=1, edgecolor='dimgray')


plt.tight_layout()
```