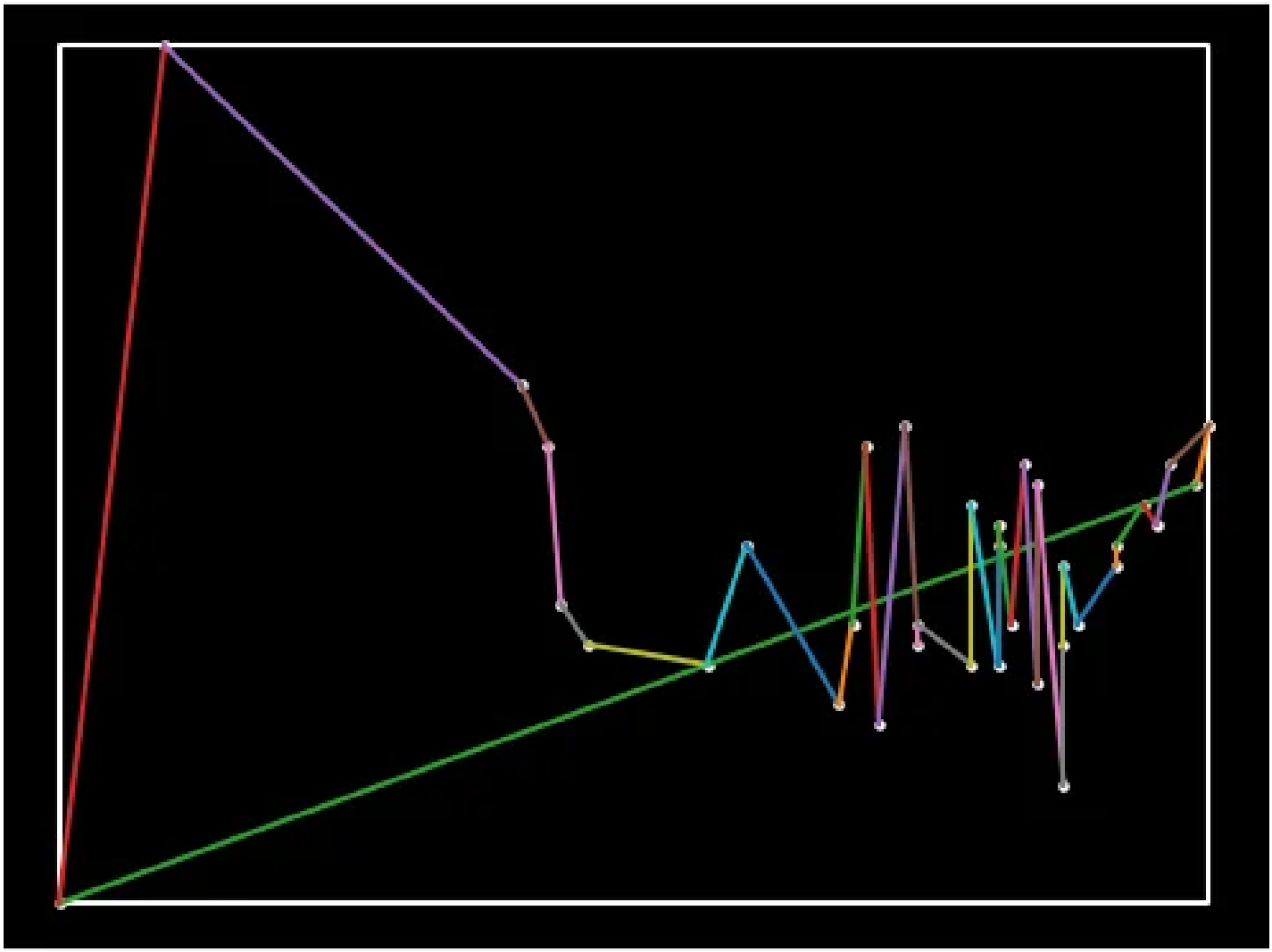
r/ThingsYouWillNeed • 1 mo. ago
TheStocksGuy

# Traveling Sales Problem - (Door to door salesperson does want to return to the same door twice, SOLVED)



Traveling Salesperson Problem Visual Art

### A Thesis on Efficient Route Optimization and Data Handling

**Abstract:** This thesis delves into the intricate process of optimizing travel routes between multiple cities. With an emphasis on efficiency and precision, this document outlines the methodologies employed to determine the most efficient routes, ensure robust data handling, and avoid redundant calculations. The culmination of this work is a testament to the time, dedication, and thoughtful consideration poured into solving these challenges.

**Introduction:** In our increasingly connected world, finding the most efficient routes between cities has become an essential task. This thesis presents a comprehensive solution to this problem, developed with meticulous attention to detail and a profound appreciation for the intricacies involved. The goal is to provide a clear understanding of the processes and principles that underpin the optimization of travel routes and the management of associated data.

position, providing a foundational framework for further analysis.

**Chapter 2: Sorting and Selecting Cities** The initial step in route optimization involves sorting the cities based on their coordinates. By organizing the cities from the highest to the lowest values, we established an order that guided the connection process. The first city in this sorted list served as our starting point, marking the beginning of our journey.

**Chapter 3: Establishing Connections** Connecting the cities involved a meticulous process of distance calculation and path selection. From the current city, distances to all other cities were calculated. The city with the shortest distance that did not intersect any previous paths was selected as the next destination. This iterative process continued until all cities were connected. Finally, the journey looped back to the starting city, completing the route.

**Chapter 4: Efficient Data Handling** To ensure efficiency, all routes and paths were stored in a file. This file served as a repository of information, recording the order of cities and the connecting paths. Additionally, the file contained smaller routes within specific regions, such as routes confined to states within the USA. This storage system allowed for quick access and reduced the need for redundant calculations.

**Chapter 5: Requesting and Updating Data** Each time route information was needed, the file was checked first. If the required data was available, it was retrieved directly, avoiding unnecessary recalculations. If the data was not available, the route was calculated and subsequently stored for future use. This system ensured that all new paths were added to the file, continuously updating the repository with the latest information.

**Chapter 6: Avoiding Redundancy** One of the core principles of this solution was to avoid redundant reads and calculations. Before making any new calculations, the existing storage was thoroughly checked. This preemptive step prevented repeating the same calculations and saved valuable time and resources.

**Chapter 7: Ensuring Path Integrity** To maintain the integrity of the paths, each new path was carefully checked for intersections with existing paths. This ensured that no paths crossed each other, maintaining a clear and efficient route. Every consideration was given to the sequence of connections, ensuring a logical and coherent journey.

**Conclusion:** The process of optimizing travel routes between multiple cities is a complex and nuanced task. Through careful planning, innovative methodologies, and a deep commitment to efficiency, this solution provides a robust framework for route optimization. The dedication and time invested in developing this solution reflect a genuine love for the subject and a desire to make meaningful contributions to the field.

By following these steps, we ensure an organized and efficient way of connecting cities, storing routes, and avoiding redundant calculations.

Pythton Generates a Path for Visual aspects but not needed.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation
import matplotlib.colors as mcolors
import json
import os
```

```
    value = (value | (value << 4)) & 0x0F0F0F0F
    value = (value | (value << 2)) & 0x33333333
    value = (value | (value << 1)) & 0x55555555
    return value

def morton_code(city):
    x, y = city['x'], city['y']
    x = int(x * 10000)
    y = int(y * 10000)
    return bit_spread(x) | (bit_spread(y) << 1)

def is_path_intersect(new_path, paths):
    for path in paths:
        path = [np.array(p) for p in path]
        new_path = [np.array(p) for p in new_path]
        if len(path[0]) == 2:
            path[0] = np.append(path[0], 0)
        if len(path[1]) == 2:
            path[1] = np.append(path[1], 0)
        if len(new_path[0]) == 2:
            new_path[0] = np.append(new_path[0], 0)
        if len(new_path[1]) == 2:
            new_path[1] = np.append(new_path[1], 0)
        cross1 = np.cross(path[1] - path[0], new_path[0] - path[0])
        cross2 = np.cross(path[1] - path[0], new_path[1] - path[0])
        cross3 = np.cross(new_path[1] - new_path[0], path[0] - new_path[0])
        cross4 = np.cross(new_path[1] - new_path[0], path[1] - new_path[0])
        if (np.all(cross1 * cross2 <= 0) and np.all(cross3 * cross4 <= 0)):
            return True
    return False

def connect_cities(cities):
    if not cities:
        print("No cities to process.")
        return []
    cities_sorted = sorted(cities, key=lambda c: (c['x'], c['y']), reverse=True)
    sorted_route = [cities_sorted.pop(0)]
    paths = []
    while cities_sorted:
        current_city = sorted_route[-1]
        current_coords = [current_city.get('x', 0), current_city.get('y', 0)]
        if 'z' in current_city:
            current_coords.append(current_city['z'])
        min_distance = float('inf')
        closest_city_idx = -1
        for i, city in enumerate(cities_sorted):
```

```
            distance = np.linalg.norm(np.array(city_coords) - np.array(current_coords))
            if distance < min_distance:
                new_path = [np.array(current_coords), np.array(city_coords)]
                if not is_path_intersect(new_path, paths):
                    min_distance = distance
                    closest_city_idx = i
        sorted_route.append(cities_sorted.pop(closest_city_idx))
        new_path = [np.array([current_city['x'], current_city['y'], current_city.get('z', 0)]),
                    np.array([sorted_route[-1]['x'], sorted_route[-1]['y'], sorted_route[-1].get('
        paths.append(new_path)
    sorted_route.append(sorted_route[0])
    return sorted_route

def visualize_route(ax, sorted_route, zoom=1, offset_x=0, offset_y=0):
    ax.clear()
    ax.set_facecolor('black')
    ax.set_xticks([])
    ax.set_yticks([])
    min_x = min(city.get('x', 0) for city in sorted_route)
    max_x = max(city.get('x', 0) for city in sorted_route)
    min_y = min(city.get('y', 0) for city in sorted_route)
    max_y = max(city.get('y', 0) for city in sorted_route)
    container_coords = [(min_x, min_y), (min_x, max_y), (max_x, max_y), (max_x, min_y), (min_x, mi
    ax.plot([c[0] + offset_x for c in container_coords], [c[1] + offset_y for c in container_coord
    colors = list(mcolors.TABLEAU_COLORS.values())
    for index in range(1, len(sorted_route)):
        city = sorted_route[index]
        prev_city = sorted_route[index - 1]
        color = colors[index % len(colors)]
        x_coords = [prev_city.get('x', 0) + offset_x, city.get('x', 0) + offset_x]
        y_coords = [prev_city.get('y', 0) + offset_y, city.get('y', 0) + offset_y]
        if 'z' in city and 'z' in prev_city:
            z_coords = [prev_city.get('z', 0), city.get('z', 0)]
            ax.plot(x_coords, y_coords, zs=z_coords, color=color)
        else:
            ax.plot(x_coords, y_coords, color=color)
    if 'z' in sorted_route[0]:
        ax.scatter([city.get('x', 0) + offset_x for city in sorted_route],
                   [city.get('y', 0) + offset_y for city in sorted_route],
                   [city.get('z', 0) for city in sorted_route],
                   c='white', s=5)
    else:
        ax.scatter([city.get('x', 0) + offset_x for city in sorted_route],
                   [city.get('y', 0) + offset_y for city in sorted_route],
                   c='white', s=5)
```

```
def save_data(file, primary, secondary=None):
    with open(file, 'w') as f:
        data = {'primary': primary}
        if secondary:
            data['secondary'] = secondary
        json.dump(data, f)

def load_data(file):
    if os.path.exists(file):
        with open(file, 'r') as f:
            data = json.load(f)
            if isinstance(data, list):
                return data, []
            return data.get('primary', []), data.get('secondary', [])
    return [], []

def calculate_short_paths(cities):
    # Implement your logic here
    return []

file = 'usa_states.json'
primary, secondary = load_data(file)
usa_states = primary

if not usa_states:
    print("No paths found.")
else:
    sorted_route = connect_cities(usa_states)
    short_paths = []
    if not secondary:
        short_paths = calculate_short_paths(usa_states)
        save_data('sorted_paths.json', sorted_route, short_paths)
    else:
        short_paths = secondary

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d' if 'z' in usa_states[0] else 'rectilinear')
    visualize_route(ax, sorted_route)
    ani = FuncAnimation(fig, animate_route, frames=len(sorted_route), fargs=(sorted_route, ax), in
    plt.show()
```

the json file example needed for usa_states.json

```
    {"x": -98.0, "y": 31.0,  "name": "Texas", "size_x": 268.0, "size_y": 278.0},
    {"x": -88.0, "y": 41.0,  "name": "Illinois", "size_x": 57.0, "size_y": 55.0},
    {"x": -81.0, "y": 27.0,  "name": "Florida", "size_x": 53.0, "size_y": 58.0},
    {"x": -119.0, "y": 36.0,  "name": "California", "size_x": 163.0, "size_y": 156.0},
    {"x": -86.0, "y": 40.0,  "name": "Indiana", "size_x": 36.0, "size_y": 35.0},
    {"x": -93.0, "y": 45.0,  "name": "Minnesota", "size_x": 87.0, "size_y": 84.0},
    {"x": -105.0, "y": 39.0,  "name": "Colorado", "size_x": 104.0, "size_y": 103.0},
    {"x": -81.0, "y": 38.0,  "name": "West Virginia", "size_x": 24.0, "size_y": 31.0},
    {"x": -77.0, "y": 39.0,  "name": "Maryland", "size_x": 12.0, "size_y": 32.0},
    {"x": -92.0, "y": 34.0,  "name": "Arkansas", "size_x": 53.0, "size_y": 55.0},
    {"x": -81.0, "y": 34.0,  "name": "South Carolina", "size_x": 32.0, "size_y": 33.0},
    {"x": -83.0, "y": 32.0,  "name": "Georgia", "size_x": 59.0, "size_y": 60.0},
    {"x": -80.0, "y": 35.0,  "name": "North Carolina", "size_x": 54.0, "size_y": 55.0},
    {"x": -74.0, "y": 40.0,  "name": "New Jersey", "size_x": 9.0, "size_y": 21.0},
    {"x": -71.0, "y": 42.0,  "name": "Massachusetts", "size_x": 10.0, "size_y": 27.0},
    {"x": -122.0, "y": 47.0,  "name": "Washington", "size_x": 71.0, "size_y": 70.0},
    {"x": -70.0, "y": 45.0,  "name": "Maine", "size_x": 35.0, "size_y": 34.0},
    {"x": -149.0, "y": 64.0,  "name": "Alaska", "size_x": 665.0, "size_y": 663.0},
    {"x": -157.0, "y": 21.0,  "name": "Hawaii", "size_x": 11.0, "size_y": 10.0},
    {"x": -86.0, "y": 39.0,  "name": "Indiana", "size_x": 36.0, "size_y": 35.0},
    {"x": -95.0, "y": 30.0,  "name": "Louisiana", "size_x": 52.0, "size_y": 50.0},
    {"x": -92.0, "y": 35.0,  "name": "Mississippi", "size_x": 48.0, "size_y": 50.0},
    {"x": -117.0, "y": 34.0,  "name": "Nevada", "size_x": 110.0, "size_y": 107.0},
    {"x": -108.0, "y": 33.0,  "name": "New Mexico", "size_x": 121.0, "size_y": 122.0},
    {"x": -83.0, "y": 42.0,  "name": "Ohio", "size_x": 45.0, "size_y": 44.0},
    {"x": -96.0, "y": 44.0,  "name": "South Dakota", "size_x": 77.0, "size_y": 75.0},
    {"x": -85.0, "y": 35.0,  "name": "Tennessee", "size_x": 42.0, "size_y": 41.0},
    {"x": -120.0, "y": 44.0,  "name": "Oregon", "size_x": 98.0, "size_y": 96.0},
    {"x": -84.0, "y": 43.0,  "name": "Michigan", "size_x": 97.0, "size_y": 96.0},
    {"x": -73.0, "y": 43.0,  "name": "New York", "size_x": 55.0, "size_y": 54.0},
    {"x": -75.0, "y": 41.0,  "name": "Pennsylvania", "size_x": 46.0, "size_y": 47.0},
    {"x": -77.0, "y": 38.0,  "name": "Virginia", "size_x": 43.0, "size_y": 44.0},
    {"x": -86.0, "y": 33.0,  "name": "Alabama", "size_x": 52.0, "size_y": 53.0},
    {"x": -88.0, "y": 33.0,  "name": "Mississippi", "size_x": 48.0, "size_y": 49.0}
]
```