**حل مشكلة البائع المتجول (TSP) باستخدام Python وDjango**

**الاستضافة على github**

[**https://github.com/Awad-cyber/tsp\_solver\_project\_awad.git**](https://github.com/Awad-cyber/tsp_solver_project_awad.git)

**رابط استضافة**

[**https://awad165083.pythonanywhere.com/**](https://awad165083.pythonanywhere.com/)

**مقدمة**

مشكلة البائع المتجول (**Travelling Salesman Problem - TSP**) هي واحدة من أشهر مشاكل التحسين في علوم الحاسوب والرياضيات التطبيقية. تتلخص المشكلة في الآتي:

"لدى بائع مجموعة من المدن التي يجب عليه زيارتها. هدفه هو إيجاد أقصر مسار يمر بجميع المدن مرة واحدة فقط ويعود في النهاية إلى المدينة التي بدأ منها."

**أهمية المشكلة:**

تظهر مشكلة البائع المتجول في العديد من التطبيقات العملية، خاصة في المجالات التي تتطلب تحسين المسارات أو إدارة الموارد، مثل:

* **خدمات النقل والتوصيل:** تحسين مسارات السائقين لتقليل الوقت والتكلفة.
* **الخدمات اللوجستية:** تقليل تكلفة النقل بين المخازن والموزعين.
* **مجال الروبوتات:** تحديد مسارات الحركة للروبوتات لتغطية منطقة معينة بكفاءة.
* **مجال الطاقة:** تحسين خطوط نقل الطاقة لتقليل الفاقد.

**تحديات المشكلة:**

* **عدد الحلول الممكنة:** عند وجود n مدن، فإن عدد المسارات الممكنة هو (n−1)! وبالتالي، مع زيادة عدد المدن، تصبح المشكلة أكثر تعقيدًا بشكل أُسّي.
* **البحث عن الحل الأمثل:** إيجاد أقصر مسار ممكن (Optimal Path) يتطلب وقتًا طويلًا باستخدام الطرق التقليدية.
* **القيود الإضافية:** في بعض الحالات، قد تكون هناك قيود إضافية مثل توقيت التسليم أو عدد السائقين، مما يزيد من تعقيد المشكلة.

**الهدف من المشروع:**

الهدف من هذا المشروع هو:

1. **طرح مشكلة عملية (تحسين مسار توصيل الطرود).**
2. **تطوير خوارزمية فعّالة لحلها (الجار الأقرب كخطوة أولى).**
3. **بناء نظام برمجي تفاعلي لتطبيق الحل.**
4. **عرض النتائج من خلال واجهة مستخدم سهلة الاستخدام.**

**الأدوات والتقنيات المستخدمة**

1. **لغة البرمجة** Python
2. **إطار العمل**  Django
3. **مكتبات Python:**
4. **الهيكلية:**
   * تطبيق ويب باستخدام Django MVC (Model-View-Controller)
   * قوالب HTML لعرض النتائج.

**الخطوات التنفيذية**

**تصميم المشروع**

* تم تقسيم المشروع إلى ثلاثة مكونات رئيسية:
  + **الخوارزمية:** خوارزمية الجار الأقرب لحل مشكلة TSP
  + **التصور:** تصوير المسار الناتج باستخدام الرسوم البيانية.
  + **واجهة المستخدم:** استلام البيانات من المستخدم وعرض النتائج.

**إعداد البيئة**

* إنشاء بيئة افتراضية باستخدام venv:
  1. Python -m venv venv
  2. .\venv\Scripts\activate
* تثبيت الحزم اللازمة:

pip install django networkx matplotlib

**إنشاء مشروع Django**

* إنشاء مشروع جديد:

django-admin startproject tsp\_project

* إنشاء تطبيق داخل المشروع:

python manage.py startapp tsp\_solver

**تنفيذ خوارزمية الجار الأقرب**

* تمت كتابة خوارزمية nearst\_neighbor\_tspلحساب المسار والتكلفة:
  + تستخدم مصفوفة المسافات لحساب أقرب مدينة.
  + تضيف المدن إلى المسار مع حساب التكلفة الإجمالية.
  + تعود إلى نقطة البداية بعد زيارة جميع المدن.

**إنشاء الرسم البياني**

* تمت كتابة وظيفة visualize\_tsp لتصور المسار الناتج:
  + استخدام مكتبة networkx لإنشاء العقد والحواف.
  + تلوين المسار الناتج باللون الأحمر لتمييزه.
  + حفظ الرسم كصورة بتنسيق Base64 لعرضها في واجهة المستخدم.

**إنشاء واجهة المستخدم**

* تم إنشاء قالب HTML لعرض:
  + المسار الناتج والتكلفة الإجمالية.
  + الرسم البياني كصورة Base64

#### ****الخوارزمية المستخدمة****

استخدمنا **خوارزمية الجار الأقرب (Nearest Neighbor Algorithm)** لحل المشكلة:

* **الهدف:** إيجاد مسار مقبول بسرعة (وليس بالضرورة الأمثل).
* **كيف تعمل:**
  1. تبدأ من نقطة معينة.
  2. تبحث عن أقرب نقطة لم يتم زيارتها بعد.
  3. تنتقل إلى تلك النقطة وتكرر العملية.
  4. تعود إلى نقطة البداية بعد زيارة جميع النقاط.

#### ****الخطوات التفصيلية للكود****

##### **أ. كتابة خوارزمية الجار الأقرب**

تم تنفيذ خوارزمية الجار الأقرب في الدالة: nearest\_neighbor\_tsp

* **المدخلات:**
  + قائمة بأسماء الأماكن مثل ["A", "B", "C", "D"])
  + مصفوفة تمثل المسافات بين جميع الأزواج الممكنة من النقاط.
* **الخطوات:**
  + قم بإنشاء قائمة visited لتتبع النقاط التي تمت زيارتها.
  + ابدأ من النقطة الأولى (current = 0)
  + أضف النقطة الحالية إلى المسار (path)
  + ابحث عن أقرب نقطة لم تتم زيارتها.
  + أضف المسافة إلى التكلفة الإجمالية (cost)
  + كرر الخطوات حتى يتم زيارة جميع النقاط.
  + أضف المسافة من النقطة الأخيرة إلى نقطة البداية.
* **المخرجات:**
  + path قائمة تمثل الترتيب الذي يتم فيه زيارة النقاط.
  + cost التكلفة الإجمالية (المسافة).

**الكود**

import networkx as nx

import matplotlib.pyplot as plt

import io

import base64

from django.shortcuts import render

# TSP Solver using Nearest Neighbor

def nearest\_neighbor\_tsp(locations, distances):

    n = len(locations)

    visited = [False] \* n

    path = []

    cost = 0

    current = 0

    for \_ in range(n):

        visited[current] = True

        path.append(current)

        next\_city = min(

            [(distances[current][j], j) for j in range(n) if not visited[j]],

            default=(None, None)

        )[1]

        if next\_city is None:

            break

        cost += distances[current][next\_city]

        current = next\_city

    cost += distances[current][0]  # Return to starting point

    path.append(0)

    return {'path': path, 'cost': cost}

def visualize\_tsp(locations, distances, path):

    G = nx.DiGraph()  # Directed graph for visualizing the path

    # Add nodes

    for loc in locations:

        G.add\_node(loc)

    # Add edges with weights

    for i in range(len(distances)):

        for j in range(len(distances[i])):

            if i != j:

                G.add\_edge(locations[i], locations[j], weight=distances[i][j])

    # Highlight the optimal TSP path

    path\_edges = [(locations[path[i]], locations[path[i + 1]]) for i in range(len(path) - 1)]

    path\_edges.append((locations[path[-1]], locations[path[0]]))  # Close the loop

    pos = nx.circular\_layout(G)  # Circular layout

    plt.figure(figsize=(8, 8))

    # Draw the full graph

    nx.draw(G, pos, with\_labels=True, node\_size=800, node\_color="skyblue", font\_size=10, font\_weight="bold")

    edge\_labels = nx.get\_edge\_attributes(G, "weight")

    nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels, font\_size=8)

    # Highlight the TSP path

    nx.draw\_networkx\_edges(G, pos, edgelist=path\_edges, edge\_color="red", arrows=True, width=2)

    # Save the graph as a base64 string

    buffer = io.BytesIO()

    plt.savefig(buffer, format="png")

    plt.close()

    buffer.seek(0)

    image\_base64 = base64.b64encode(buffer.getvalue()).decode("utf-8")

    buffer.close()

    return image\_base64

def tsp\_solver(request):

    result = None

    graph\_base64 = None

    if request.method == 'POST':

        # Input parsing

        locations = [loc.strip() for loc in request.POST.get('locations', '').split(',')]

        distances\_input = [row.strip() for row in request.POST.get('distances', '').strip().splitlines()]

        distances = [list(map(int, row.split(','))) for row in distances\_input]

        if len(locations) != len(distances):

            result = {'error': 'Number of locations and matrix size mismatch'}

        else:

            # Solve TSP

            tsp\_result = nearest\_neighbor\_tsp(locations, distances)

            path\_indices = tsp\_result['path']

            tsp\_result['path'] = [locations[i] for i in path\_indices]

            # Generate graph

            graph\_base64 = visualize\_tsp(locations, distances, path\_indices)

            result = tsp\_result

    return render(request, 'tsp\_solver/tsp\_solver.html', {'result': result, 'graph\_base64': graph\_base64})

**واجهة المستخدم**

**قالب tsp\_solver.html**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>TSP Solver</title>

</head>

<body>

    <h1>Traveling Salesman Problem Solver</h1>

    <form method="post">

        {% csrf\_token %}

        <label for="locations">Locations (comma-separated):</label><br>

        <textarea id="locations" name="locations" rows="3" cols="50" placeholder="Enter locations (e.g., A,B,C,D)"></textarea><br><br>

        <label for="distances">Distance Matrix (row by row, comma-separated):</label><br>

        <textarea id="distances" name="distances" rows="5" cols="50" placeholder="Enter distance matrix"></textarea><br><br>

        <button type="submit">Solve</button>

    </form>

    {% if result %}

        {% if result.error %}

            <h2>Error:</h2>

            <p>{{ result.error }}</p>

        {% else %}

            <h2>Result:</h2>

            <p><strong>Optimal Path:</strong> {{ result.path }}</p>

            <p><strong>Total Cost:</strong> {{ result.cost }}</p>

            <h3>Graph:</h3>

            <img src="data:image/png;base64,{{ graph\_base64 }}" alt="TSP Graph" style="max-width: 100%; height: auto;">

        {% endif %}

    {% endif %}

</body>

</html>

**النتائج**

* **المسار الأمثل:** يتم عرضه بناءً على خوارزمية الجار الأقرب.
* **الرسم البياني:** يعرض مسار البائع المتجول بشكل مرئي.

**مثال على الإدخال المتوقع من المستخدم**

**قائمة الأماكن**

يمكن للمستخدم إدخال قائمة الأماكن التي يريد حل مشكلة البائع المتجول لها. يتم فصل الأماكن بفاصلة.

**مثال:**

A, B, C, D

* **A**و **B**و **C** و**D** تمثل أسماء الأماكن.

**مصفوفة المسافات**

المستخدم يحتاج إلى إدخال مصفوفة المسافات بين الأماكن على شكل جدول حيث:

* يتم فصل القيم داخل الصف الواحد بفاصلة.
* يتم كتابة كل صف في سطر جديد.
* قيمة المسافة من مكان إلى نفسه تساوي دائمًا صفر.

**مثال:**

0, 10, 15, 20

10, 0, 35, 25

15, 35, 0, 30

20, 25, 30, 0

**شرح المثال:**

* **A → B** المسافة 10.
* **A → C** المسافة 15.
* **A → D** المسافة 20.
* وهكذا لبقية الأماكن.

**كيفية إدخال البيانات في النموذج**

**قائمة الأماكن:**

A, B, C, D

**مصفوفة المسافات:**

0, 10, 15, 20

10, 0, 35, 25

15, 35, 0, 30

20, 25, 30, 0

**النتيجة المتوقعة**

**المسار:**

['A', 'B', 'D', 'C', 'A']

**التكلفة:**

80

**الرسم البياني:**

سيتم عرض الرسم البياني الذي يُظهر المسار بين النقاط باستخدام خوارزمية الجار الأقرب.

**التنفيذ**

 