r/ThingsYouDidntKnow • 2 mo. ago
TheStocksGuy

Solving the Traveling Salesman Problem in Python with Nearest Neighbor and 2-Opt Optimization

Solving the Traveling Salesman Problem (TSP) — the challenge of finding the shortest route that visits every city exactly once and returns to the starting point.

The code implements two main strategies for tackling TSP:

1. **Nearest Neighbor Heuristic** - Quickly finds an initial, approximate solution by always moving to the nearest unvisited city.
2. **2-Opt Optimization** - Refines the initial route by swapping city pairs to minimize total distance further.

Setting Up Your Cities

Each city should have:

- A unique name (e.g., `'City1'`, `'City2'`).
- An x and y coordinate to represent its position on a 2D plane.
  cities = [{'name': 'City1', 'x': 0, 'y': 0}, {'name': 'City2', 'x': 2, 'y': 4}, {'name': 'City3', 'x': 3, 'y': 1}, # Add more cities here]

Step 1: Calculating Distances

The function `calculate_distance` finds the Euclidean distance between two cities:

```
def calculate_distance(city1, city2):
    return math.hypot(city1['x'] - city2['x'], city1['y'] - city2['y'])
```

Using a memoized dictionary, this distance is stored after calculation to avoid redundant computations.

Step 2: The Nearest Neighbor Heuristic

The nearest neighbor function constructs an initial path by:

1. Starting from an initial city.
2. Repeatedly adding the closest unvisited city to the path.
3. Returning to the starting city to complete the loop.

Here's the `tsp_nearest_neighbor` function:

```
    path = [cities[0]]
    current_city = cities[0]
    while unvisited:
        unvisited.discard(current_city['name'])
        closest, closest_distance = None, float('inf')
        for city in cities:
            if city['name'] in unvisited:
                dist = calculate_distance(current_city, city)
                if dist < closest_distance:
                    closest_distance, closest = dist, city
        if closest is None:
            break
        path.append(closest)
        current_city = closest
    # Ensure path returns to start to form a complete tour
    path.append(path[0])
    distance = total_distance(path)
    return {'path': path, 'distance': distance}
```

Step 3: 2-Opt Optimization

The 2-Opt algorithm improves the initial path by finding shorter subpaths. It achieves this by:

1. Selecting two cities in the path.
2. Reversing the order of cities between them if this reduces total path distance.
   def two_opt(path): improvement_threshold = 1e-6 improved = True while improved: improved = False for i in range(1, len(path) - 2): for j in range(i + 1, len(path) - 1): before_swap = calculate_distance(path[i - 1], path[i]) + calculate_distance(path[j], path[(j + 1) % len(path)]) after_swap = calculate_distance(path[i - 1], path[j]) + calculate_distance(path[i], path[(j + 1) % len(path)]) if after_swap + improvement_threshold < before_swap: path[i:j + 1] = reversed(path[i:j + 1]) improved = True if path[-1] != path[0]: # Ensure the path returns to the start path.append(path[0]) return path

Step 4: Running the Solution and Optimization

The main `solve_tsp` function integrates these components. It calculates the initial route and distance, optimizes it with 2-Opt, and validates the result:

```
def solve_tsp(cities):
    initial_solution = tsp_nearest_neighbor(cities)
    initial_path, initial_distance = initial_solution['path'], initial_solution['distance']
    optimized_path = two_opt(initial_path)
    optimized_distance = total_distance(optimized_path)
    # Validate that the path visits each city once and returns to origin
    if validate_path(optimized_path, cities):
```

Step 5: Timing and Results

The code measures the time taken for both the initial solution and optimization to understand efficiency:

```
start_initial = time.time()
initial_solution = tsp_nearest_neighbor(cities)
initial_time = time.time() - start_initial

start_optimized = time.time()
optimized_path = two_opt(initial_solution['path'])
optimized_time = time.time() - start_optimized

print("Initial Distance:", initial_solution['distance'])
print("Optimized Distance:", total_distance(optimized_path))
print("Initial Solution Time:", initial_time)
print("Optimization Time:", optimized_time)
```

Expected Output Example

When run, the script will print:

- **Initial and optimized paths**.
- **Distances** for both paths.
- **Time** taken for each step.

Github : https://github.com/BadNintendo/tsp/blob/main/traveler.py

Results: Path validation successful: Each city is visited once, and path returns to origin. Initial Distance: 1257.0209779547552 Optimized Distance: 1051.0785415861549 Initial Solution Time (seconds): 0.0009884834289550781 Optimization Time (seconds): 0.005000591278076172 Initial Path: ['City0', 'City1', 'City2', 'City4', 'City7', 'City9', 'City12', 'City14', 'City17', 'City19', 'City22', 'City24', 'City27', 'City29', 'City32', 'City34', 'City37', 'City39', 'City42', 'City44', 'City47', 'City49', 'City48', 'City46', 'City45', 'City43', 'City41', 'City40', 'City38', 'City36', 'City35', 'City33', 'City31', 'City30', 'City28', 'City26', 'City25', 'City23', 'City21', 'City20', 'City18', 'City16', 'City15', 'City13', 'City11', 'City10', 'City8', 'City6', 'City5', 'City3', 'City0'] Optimized Path: ['City0', 'City1', 'City2', 'City4', 'City7', 'City9', 'City12', 'City14', 'City17', 'City19', 'City22', 'City24', 'City27', 'City29', 'City32', 'City34', 'City37', 'City39', 'City42', 'City44', 'City47', 'City49', 'City48', 'City46', 'City45', 'City43', 'City41', 'City40', 'City38', 'City36', 'City35', 'City33', 'City31', 'City30', 'City28', 'City26', 'City25', 'City23', 'City21', 'City20', 'City18', 'City16', 'City15', 'City13', 'City11', 'City10', 'City8', 'City6', 'City5', 'City3', 'City0'] Initial Distance: 1257.0209779547552 Optimized Distance: 1051.0785415861549 Initial Solution Time (seconds): 0.0009884834289550781 Optimization Time (seconds): 0.005000591278076172

Create

Post Insights

Only the post author and moderators can see this

172	100%	1	0
Total Views	Upvote Rate	Comments	Total Shares

Hourly views for first 48 hours

Some insights are no longer available because this post is older than 45 days

Add a comment

Sort by: **Best**

Search Comments

TheStocksGuy OP • 2mo ago

I'll be implementing the Morton Order approach in the next big update to enhance the results further. I've been working hard to provide the best possible outcomes, but the information overload from the past few days has made it challenging to calculate the results for 1 million requests, which I estimate to be under 90 seconds.

Please note: if you create a test array, make sure to time how long it takes to generate it before using it for testing. The arrays need to be prefabricated, as shown in the example, for accurate processing time calculations. Any routes provided will get the best solution, then refined in real-time. Enjoy!

zedashaw on twitch gave me the morton suggestion

1 ···

Approved 2 months ago