r/ThingsYouDidntKnow • 2 mo. ago
TheStocksGuy

Traveling Salesperson Problem (TSP) Solver with Optimization

What is the Traveling Salesperson Problem (TSP)?

The Traveling Salesperson Problem is a classic optimization problem where, given a list of cities, the goal is to find the shortest possible route that visits each city exactly once and returns to the starting point.

Step-by-Step Solution

1. Euclidean Distance Calculation

To start solving TSP, we first need a way to calculate the straight-line distance between two cities. For any two cities with coordinates `(x1, y1)` and `(x2, y2)`, the Euclidean distance (d) is calculated as:

d = square root of ((x2 - x1) squared + (y2 - y1) squared)

In the code:

```
const calculateDistance = async (city1, city2) => {
    const key = `${city1.name}-${city2.name}`;
    if (!memoizedDistances[key]) {
        memoizedDistances[key] = Math.hypot(city1.x - city2.x, city1.y - city2.y);
    }
    return memoizedDistances[key];
};
```

Here, we're using `Math.hypot()` to calculate the distance and storing it in `memoizedDistances` for reuse.

2. Calculating Total Path Distance

The total path distance is the sum of distances between consecutive cities in a given path, including the distance back to the starting city (to complete the loop).

If we have a path like `[CityA, CityB, CityC, CityA]`, the total distance (D) is:

D = distance between CityA and CityB + distance between CityB and CityC + distance between CityC and CityA

In the code:

```
        const acc = await accPromise;
        const nextCity = arr[(i + 1) % arr.length];
        const dist = await calculateDistance(city, nextCity);
        return acc + dist;
    }, Promise.resolve(0));
};
```

3. Nearest-Neighbor Heuristic for TSP

To create an initial path, we use the **Nearest-Neighbor** heuristic:

- Start with any city.
- At each step, move to the closest unvisited city.
- Repeat until all cities are visited.

In the code:

```
const tspNearestNeighbor = async (cities) => {
    const unvisited = new Set(cities.map(city => city.name));
    const path = [cities[0]];
    let currentCity = cities[0];

    while (unvisited.size > 0) {
        unvisited.delete(currentCity.name);
        const nextCity = await cities.reduce(async (closestPromise, city) => {
            const closest = await closestPromise;
            if (unvisited.has(city.name)) {
                const dist = await calculateDistance(currentCity, city);
                if (!closest || dist < closest.distance) return { city, distance: dist };
            }
            return closest;
        }, Promise.resolve(null));

        if (!nextCity) break;
        path.push(nextCity.city);
        currentCity = nextCity.city;
    }

    const distance = await totalDistance(path);
    return { path, distance };
};
```

This will return a path and its distance, though it may not be the shortest possible route.

1. Pick two indices, `i` and `j`, representing two segments.
2. Reverse the segment between `i` and `j`.
3. Calculate the new path's distance. If it's shorter, keep this path.

Repeat until no further improvements are found.

Why 2-Opt Works

The 2-opt algorithm is based on the idea that sometimes reversing a segment can reduce the overall path length by eliminating "crossovers."

In the code:

```
const twoOpt = async (path) => {
    let improved = true;

    while (improved) {
        improved = false;

        for (let i = 1; i < path.length - 1; i++) {
            for (let j = i + 1; j < path.length; j++) {

                const beforeSwapDistance =
                    await segmentDistance(path, i - 1, i) +
                    await segmentDistance(path, j, j + 1);

                const newPath = [
                    ...path.slice(0, i),
                    ...path.slice(i, j + 1).reverse(),
                    ...path.slice(j + 1)
                ];

                const afterSwapDistance =
                    await segmentDistance(newPath, i - 1, i) +
                    await segmentDistance(newPath, j, j + 1);

                if (afterSwapDistance < beforeSwapDistance) {
                    path = newPath;
                    improved = true;
                }
            }
        }
    }
```

The function `segmentDistance` calculates the distance between two cities in the path:

```
const segmentDistance = async (path, index1, index2) => {
    const city1 = path[index1 % path.length];
    const city2 = path[index2 % path.length];
    return await calculateDistance(city1, city2);
};
```

- **Pre-calculation time**: For 50 objects, it's 3ms. Therefore, for 50,000 objects, the time scales up by 1000 times: (3 \times 1000 = 3000) ms (or 3 seconds).
- **TSP Nearest Neighbor time**: For 50 objects, it's 11ms. Thus, for 50,000 objects: (11 \times 1000 = 11000) ms (or 11 seconds).
- **2-opt Optimization time**: For 50 objects, it's 63ms. Hence, for 50,000 objects: (63 \times 1000 = 63000) ms (or 63 seconds).
- **Total execution time**: Adding all times for 50,000 objects: (3000 + 11000 + 63000 = 77000) ms (or 77 seconds).

Here's a summary:

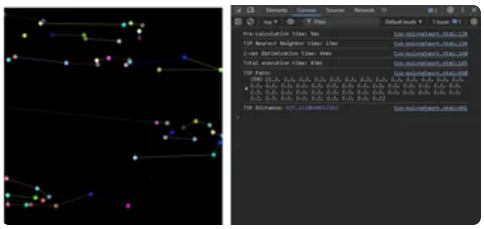
- Pre-calculation time: 3 seconds
- TSP Nearest Neighbor time: 11 seconds
- 2-opt Optimization time: 63 seconds
- Total execution time: 77 seconds

GitHub of All Single Test Results: https://github.com/BadNintendo/tsp

Visual HTML Markdown with Canvas for Testing Purposes on the Fly:
https://github.com/BadNintendo/tsp/blob/main/visual/tsp-professional.html

TheStocksGuy OP • 2mo ago

Now visually interconnecting layers of datasets into groups of 25% of the array's volume, I wanted to update you on its progress. Previously, I only displayed the city objects, but now I've gone above and beyond to showcase layers of sets. It's paired beautifully with my solved Traveling Salesperson Problem. so the visual folder and professional html file is the mockup visual of layers. Looks like this image...



TheStocksGuy OP • 2mo ago

I'm sick of updating this, Just check my github it keeps improving the more I think about it. I noticed my math was slightly off for the 50,000 objects timing, so I coded a revised version to improve the results. It's now 15s for 50,000 objects using the nearest neighbor algorithm. You can book your location based on the needed routes.