Evolutionary Traveling Sales Person Project

Using Amazon Web Services (AWS) lambda functions, DynamoDB, IAM roles, and REST APIs.

By Robert Beane
University of Minnesota – Morris
Spring Semester 2021
CSCI 4601

Table of Contents

Overview

Purpose

The purpose of this application was to allow for an evolutionary process to find the best solution to the “Traveling Salesperson Problem,” a problem that figures out the shortest route from a given set of locations and distances. This was accomplished by using the skills we learned throughout the Processes, Programming, and Languages: Programming for Cloud Computing course offered by the University of Minnesota – Morris. These skills included many technologies offered by Amazon's Web Services (AWS) including Lambda functions, DynamoDB Databases, and some others (all technologies are discussed and explained later in this write-up).

Solution

The applications architecture consists of an HTML web app that displays the evolutions at work. Behind the scenes, several Lambdas using JavaScript are called upon via a REST API which then returns the requested data to the web app. Meanwhile, certain lambdas are also responsible for adding certain “routes” to the DynamoDB database which is used throughout the application. IAM roles are also utilized to give permissions to the lambdas to write and read from the databases. Each of these technologies and how they are used are talked about in more detail in the Technical Details section of this write-up.

User Documentation

Using the web app should be a pretty easy process. Upon opening the page, you will be greeted with something like this.

Evo-TSP

By Robert Beane

"Global" parameters

Run ID: This is auto generated! Population size: 100 Number of parents to keep: 20

Best so far

- Best routeId:
- Best length (in meters):
- Best path:
- Current threshold (in meters):

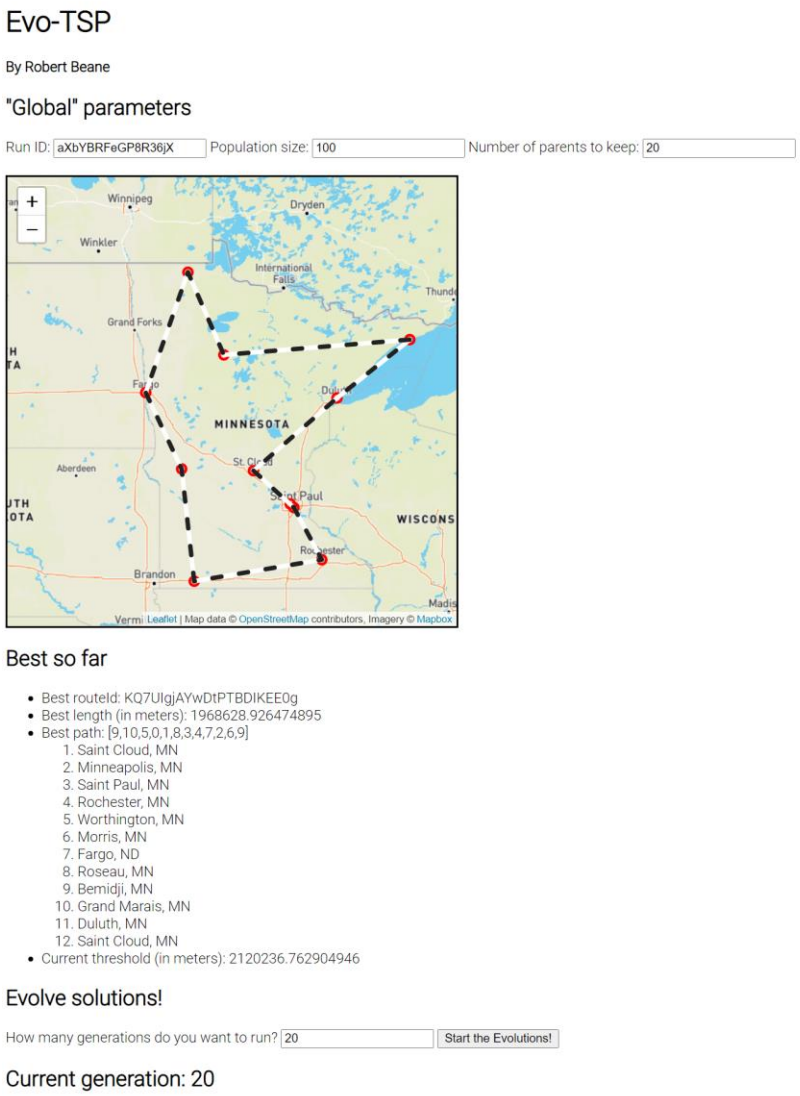
Evolve solutions!

How many generations do you want to run? 20 Start the Evolutions!

Current generation:

Best routes from previous generation

You can see that several input boxes already have “starting point” numbers, these help determine the initial population size, the number of “parents” we want to keep from each generation, and the number of generations we want to run. There is also a “Run ID” input box, however, this ID is randomly generated after you press the “Start the Evolutions!” button so we don’t need to give it one. Once you have determined the numbers you want to run with you press the “Start the Evolutions!” button and watch the web app go to work. After letting it run through all of your generations, a popup will show up saying it is done and you should see a map with a route and info about that route. You are also able to see the best route from previous generations.



Technical Details

API Details

This application utilized an API Gateway via AWS. For this case, we used a REST API with five resources.

- **/best**

  This resource used a GET method that called the **getBestRoutes()** lambda and was in charge of returning a certain number of the best routes from a certain run. It required a runId, a generation, and a number of how many you wanted to return. When called, it returned that number of best routes in a stringified JSON form.

- **/city-data**

  This resource used a GET method that called the **getCityInfo()** lambda to return all of the city info used with mapping. It required no arguments as we assumed that the user only wanted data for the Minnesota region. In the future, if more regions wanted to be added, an argument with the region could be taken but this was unnecessary for our use case.

- **/mutateroute**

  This resource used a POST method that called the **mutateRoute()** lambda. This lambda took a routeId, the lengthStoreThreshold, and the number of children that were wanted. This lambda did the majority of the work for this application as it mutated the routes finding the best ones.

- **/routes**

  This resource used a POST method that called the **generateRandomRoute()** lambda. It

took a runId and a generation that was used in the creation of the random routes. When it is called, it generates a random route and pushes that to the database to be used later.

- **/routes/{routeId}**

  This resource used a GET method to get a specific route using the **getRouteById()** lambda. It took a single routeId and returned the information for that specific route.

Lambdas

This application uses five lambda functions, each of which were run on AWS. This allowed for less strain on the users' browser and making it easier to make adjustments on a live page (don't have to wait for your cache to catch up to updates).

getBestRoutes()

This lambda took a runId, generation, and a numToReturn. It then queries the **routes** database for the best routes with the given runId and generation. It then returns the top best routes based on the number that you want to be returned with a callback. The body of this callback is the stringified JSON of the best routes.

getRouteById()

This lambda simply took a single routeId. It used this routeId to do a GET within the **routes** database to return all the information tied to that single routeId. Because each routeId was unique, the lambda can only return one route.

mutateRoute()

This lambda takes a "parent" routeId, the lengthStoreThreshold, and the number of "children" you want. With these, it will generate children using the 2-opt mutation

method. It also adds one to the generation of the runId so it's easier to see within the database. Once the children are generated, the function compares the lengths of the new routes to the lengthThreshold to determine if they should be kept. The “good” ones are the ones that we batchWrite to the database. Using batchWrite, we can post at most 25 items at once instead of using a for loop to repeatedly do the same thing.

getCityInfo()

This lambda took no arguments and simply returned the “Minnesota” region from the “distance_data” database. The lambda used a get call to get the database and returned it with a callback where it stringified the JSON. This lambda was built specifically to get only the Minnesota region, it could easily be manipulated to accept any region if it were added to the database.

generateRandomRoute()

This lambda took a runId and a generation. This lambda was in charge of creating a random route by scrambling the array and computing the distance of that route. Once all the elements of the new route are created, the lambda then utilizes a put to add the new route to the “routes” database. Because this lambda function only generates one route, the evotsp.js file is where the number created is determined and handled.

IAM Roles

For this application, we created one role that was used by all of our lambda functions and consisted of 4 actions; GetItem, Query, BatchWriteItem, and PutItem. GetItem was used when we needed to get something from a certain database. Query was used when we needed to query for something within a database, we used it when finding the best routes. BatchWriteItem was

used within **mutateRoute** to write a bunch of new children to the database. Finally, PutItem was used when we needed to put something to a database, for example, we used it in **generateRandomRoute** to put the newly generated route to the database.

Table Structures

For this application, we utilized two tables, one for the distance data and one to store our routes.

Routes

The routes table was used for our generated routes. Each route that was generated was put here, allowing for queries and gets to later be used to find certain routes. This table had four fields; **routeId** (the routeId of the route, useful for finding a specific route), **len** (the length of the route), **route** (the array of the specific route), and the **runGen** (a combination of the runId and Generation separated by a #, useful for finding the best routes in a given runGen). The routes table also had a primary partition key of **routeId** with no sort key. It also had a secondary index with a partition key of **runGen** and a sort key of **len**. This secondary index allowed us to get back routes with a certain runGen and get the best routes by length (len).

Distance_Data

The distance_data table contained our region information. This table had 3 fields; region (the region of which the cities are, in our case, the region was “Minnesota”), cities (this contained the majority of the information including the city name, index number, location (long, lat), and the zip code), and distances (this had the distances from each index to each other, this was used in calculating total distance). This table also only had a primary partition key which was the region.

Leaflet Details

In my application, I utilized Mapbox as my tile provider which then incorporated leaflet. Leaflet was used to display each city on the map. Once a route was generated, leaflet was also used to map the route with a dotted line. I chose a dotted line over a solid line because in most cases it's easier to distinguish a dotted line than a single solid line, plus it just looks cool. I chose black and white as they are the most contrasting colors so it should be easy for anyone to see.

Appendix I: Lambda Functions

GetBestRoutes

```
const AWS = require('aws-sdk');
const ddb = new AWS.DynamoDB.DocumentClient();

exports.handler = (event, context, callback) => {
    const queryStringParameters = event.queryStringParameters;
    const runId = queryStringParameters.runId;
    const generation = queryStringParameters.generation;
    const numToReturn = queryStringParameters.numToReturn;

    getBestRoutes(runId, generation, numToReturn)
        .then(dbResults => {
            const bestRoutes = dbResults.Items;
            callback(null, {
                statusCode: 201,
                body: JSON.stringify(bestRoutes),
                headers: {
                    'Access-Control-Allow-Origin': '*'
                }
            });
        })
        .catch(err => {
            console.log(`Problem getting best runs for generation
${generation} of ${runId}.`);
            console.error(err);
            errorResponse(err.message, context.awsRequestId, callback);
        });
};

function getBestRoutes(runId, generation, numToReturn) {
```

```
    const runGen = runId + "#" + generation;
    console.log(runGen);
    return ddb.query({
        TableName: 'routes',
        IndexName: 'runGen-len-index',
        ProjectionExpression: "routeId, len, runGen, route",
        KeyConditionExpression: 'runGen = :runGen',
        ExpressionAttributeValues: {
                ':runGen': runGen,
            },
        Limit: numToReturn,
    }).promise();
}

function errorResponse(errorMessage, awsRequestId, callback) {
  callback(null, {
    statusCode: 500,
    body: JSON.stringify({
      Error: errorMessage,
      Reference: awsRequestId,
    }),
    headers: {
      'Access-Control-Allow-Origin': '*',
    },
  });
}
```

getRouteById

```
const AWS = require('aws-sdk');
const ddb = new AWS.DynamoDB.DocumentClient();

exports.handler = (event, context, callback) => {
    const pathParameters = event.pathParameters;
    const givenRouteId = pathParameters.routeId;


    getRouteById(givenRouteId)
        .then(dbResults => {
            const routeFromId = dbResults.Item;
            callback(null, {
                statusCode: 201,
                body: JSON.stringify(routeFromId), //returns found route by
id
                headers: {
                    'Access-Control-Allow-Origin': '*'
                }
            });
        })
```

```
        .catch(err => {
            console.log(`Problem getting certain route with id:
${givenRouteId}`);
            console.error(err);
            errorResponse(err.message, context.awsRequestId, callback);
        });
}

function getRouteById(givenRouteId) {
    return ddb.get({
        TableName: 'routes',
        Key: {
            "routeId": givenRouteId
        },
    }).promise();
}

function errorResponse(errorMessage, awsRequestId, callback) {
  callback(null, {
    statusCode: 500,
    body: JSON.stringify({
      Error: errorMessage,
      Reference: awsRequestId,
    }),
    headers: {
      'Access-Control-Allow-Origin': '*',
    },
  });
}
```

mutateRoute

```
const AWS = require('aws-sdk');
const ddb = new AWS.DynamoDB.DocumentClient();
const randomBytes = require('crypto').randomBytes;

/*
 * Parts of this are already in working order, and
 * other parts (marked by "FILL THIS IN") need to be
 * done by you.
 *
 * For reference, here's a list of all the functions that
 * you need to complete:
 * - `getDistanceData()` - done
 * - `getRouteById()` - done
 * - `generateChildren()` - done?
 * - `addOneToGen()` - done
 * - `recordChildren()`
 * - `returnChildren`
```

```
 * - `computeDistance` - done?
 */

// This will be called in response to a POST request.
// The routeId of the "parent" route will be
// provided in the body, along with the number
// of "children" (mutations) to make.
// Each child will be entered into the database,
// and we'll return an array of JSON objects
// that contain the "child" IDs and the length
// of those routes. To reduce computation on the
// client end, we'll also sort these by length,
// so the "shortest" route will be at the front
// of the return array.
//
// Since all we'll get is the routeId, we'll need
// to first get the full details of the route from
// the DB. This will include the generation, and
// we'll need to add one to that to create the
// generation of all the children.
exports.handler = (event, context, callback) => {
    const requestBody = JSON.parse(event.body);
    const routeId = requestBody.routeId;
    const numChildren = requestBody.numChildren;
    let lengthStoreThreshold = requestBody.lengthStoreThreshold;
    if (lengthStoreThreshold == null) {
        lengthStoreThreshold = Infinity;
    }

    // Batch writes in DynamoDB are restricted to at most 25 writes.
    // Because of that, I'm limiting this Lambda to only only handle
    // at most 25 mutations so that I can write them all to the DB
    // in a single batch write.
    //
    // If that irks you, you could create a function that creates
    // and stores a batch of at most 25, and then call it multiple
    // times to create the requested number of children.
    if (numChildren > 25) {
        errorResponse("You can't generate more than 25 mutations at a time",
context.awsRequestId, callback);
        return;
    }

    // Promise.all makes these two requests in parallel, and only returns
    // it's promise when both of them are complete. That is then sent
    // into a `.then()` chain that passes the results of each previous
    // step as the argument to the next step.
    Promise.all([getDistanceData(), getRouteById(routeId)])
        .then(([distanceData, parentRoute]) =>
generateChildren(distanceData.Item, parentRoute.Item, numChildren))
        .then(children => recordChildren(children, lengthStoreThreshold))
        .then(children => returnChildren(callback, children))
        .catch(err => {
```

```
            console.log("Problem mutating given parent route");
            console.error(err);
            errorResponse(err.message, context.awsRequestId, callback);
        });
};

// Get the city-distance object for the region 'Minnesota'.
function getDistanceData() {
    return ddb.get({
        TableName: 'distance_data',
        Key: { region: 'Minnesota' }
    }).promise();
}

// Get the full info for the route with the given ID.
function getRouteById(routeId) {
    return ddb.get({
        TableName: 'routes',
        Key: {
            "routeId": routeId
        },
    }).promise();
}

// Generate an array of new routes, each of which is a mutation
// of the given `parentRoute`. You essentially need to call
// `generateChild` repeatedly (`numChildren` times) and return
// the array of the resulting children. `generateChild` does
// most of the heavy lifting here, and this function should
// be quite short.
function generateChildren(distanceData, parentRoute, numChildren) {
    let children = [];

    for (let i = 0; i < numChildren; i++){
        children.push(generateChild(distanceData, parentRoute));
    }
    //return Array(numChildren).fill().map(generateChild(distanceData,
parentRoute)); //---- Tried doing this method but kept getting 500 errors
and couldnt firgure out why
    return children;

    // You could just use a for-loop for this, or see
    // https://stackoverflow.com/a/42306160 for a nice description of
    // how to use of Array()/fill/map to generate the desired number of
    // children.
}

// This is complete and you shouldn't need to change it. You
// will need to implement `computeDistance()` and `addOneToGen()`
// to get it to work, though.
function generateChild(distanceData, parentRoute) {
    const oldPath = parentRoute.route;
    const numCities = oldPath.length;
```

```
    // These are a pair of random indices into the path s.t.
    // 0<=i<j<=N and j-i>2. The second condition ensures that the
    // length of the "middle section" has length at least 2, so that
    // reversing it actually changes the route.
    const [i, j] = genSwapPoints(numCities);
    // The new "mutated" path is the old path with the "middle section"
    // (`slice(i, j)`) reversed. This implements a very simple TSP mutation
    // technique known as 2-opt (https://en.wikipedia.org/wiki/2-opt).
    const newPath =
        oldPath.slice(0, i)
            .concat(oldPath.slice(i, j).reverse(),
                    oldPath.slice(j));
    const len = computeDistance(distanceData.distances, newPath);
    const child = {
        routeId: newId(),
        runGen: addOneToGen(parentRoute.runGen),
        route: newPath,
        len: len,
    };
    return child;
}

// Generate a pair of random indices into the path s.t.
// 0<=i<j<=N and j-i>2. The second condition ensures that the
// length of the "middle section" has length at least 2, so that
// reversing it actually changes the route.
function genSwapPoints(numCities) {
    let i = 0;
    let j = 0;
    while (j-i < 2) {
        i = Math.floor(Math.random() * numCities);
        j = Math.floor(Math.random() * (numCities+1));
    }
    return [i, j];
}

// Take a runId-generation string (`oldRunGen`) and
// return a new runId-generation string
// that has the generation component incremented by
// one. If, for example, we are given 'XYZ#17', we
// should return 'XYZ#18'.
function addOneToGen(oldRunGen) {
    var runId = oldRunGen.split('#')[0]; //splits the runGen and gets the
runId
    const gen = oldRunGen.split('#')[1]; //splits the runGen and gets the
generation
    const toInt = parseInt(gen, 10); //turns the string into a number so
addition works
    const newGen = toInt + 1;
    return runId + '#' + newGen; //combines the new runGen
}

// Write all the children whose length
```

```
// is less than `lengthStoreThreshold` to the database. We only
// write new routes that are shorter than the threshold as a
// way of reducing the write load on the database, which makes
// it (much) less likely that we'll have writes fail because we've
// exceeded our default (free) provisioning.
function recordChildren(children, lengthStoreThreshold) {
    // Get just the children whose length is less than the threshold.
    const childrenToWrite
        = children.filter(child => child.len < lengthStoreThreshold);

        var childrenJSON = {
            RequestItems:{
                'routes':[
                    ]}
        };

        var i;
        for (i = 0; i < childrenToWrite.length; i++){

            //this should push new items into the JSON object to be
batchWritten
            childrenJSON.RequestItems['routes'].push({
                PutRequest: {
                    Item: childrenToWrite[i]
                },
            });
        }

        ddb.batchWrite(childrenJSON, function(err, data) {
            if (err) console.log(err);
            else console.log(data);
        }); //this writes our newly created json file
        return childrenToWrite;
    // FILL IN THE REST OF THIS.
    // You'll need to generate a batch request object (described
    // in the write-up) and then call `ddb.batchWrite()` to write
    // those children to the database.

    // After the `ddb.batchWrite()` has completed, make sure you
    // return the `childrenToWrite` array.
    // We only want to return _those_ children (i.e., those good
    // enough to be written to the DB) instead of returning all
    // the generated children.

}

// Take the children that were good (short) enough to be written
// to the database.
//
//   * You should "simplify" each child, converting it to a new
//     JSON object that only contains the `routeId` and `len` fields.
//   * You should sort the simplified children by length, so the
//     shortest is at the front of the array.
```

```
//    * Use `callback` to "return" that array of children as the
//      the result of this Lambda call, with status code 201 and
//      the 'Access-Control-Allow-Origin' line.
function returnChildren(callback, children) {
    var childrenSorted = children.sort(sortByProperty("len")); //sorts based
on len

    var i;
    for (i=0; i < childrenSorted.length; i++){
        delete childrenSorted[i].route;
        delete childrenSorted[i].runGen;
        // delete childrenSorted[i].generation;
        // delete childrenSorted[i].runId;
    }

    callback(null, {
            statusCode: 201,
            body: JSON.stringify(childrenSorted),
            headers: {
                'Access-Control-Allow-Origin': '*'
            }
        });
}

// Compute the length of the given route.
function computeDistance(distances, route) {
    let totalDistance = 0;
    for (let i = 0; i < route.length; i++){
        if (i == route.length-1){
            const finalStop = route[i];
            const startCity = route[0];
            const firstLastDistance = distances[finalStop][startCity];
            totalDistance = totalDistance + firstLastDistance;
        } else {
            const cityA = route[i];
            const cityB = route[i+1];
            const abDistance = distances[cityA][cityB];
            totalDistance = totalDistance + abDistance;
        }
    } return totalDistance;
}

function newId() {
    return toUrlString(randomBytes(16));
}

function toUrlString(buffer) {
    return buffer.toString('base64')
        .replace(/\+/g, '-')
        .replace(/\//g, '_')
        .replace(/=/g, '');
}
```

```
function errorResponse(errorMessage, awsRequestId, callback) {
  callback(null, {
    statusCode: 500,
    body: JSON.stringify({
      Error: errorMessage,
      Reference: awsRequestId,
    }),
    headers: {
      'Access-Control-Allow-Origin': '*',
    },
  });
}

//Sourced from this site https://medium.com/@asadise/sorting-a-json-array-according-one-property-in-javascript-18b1d22cd9e9 This is used for sorting our children array based on length (len)
function sortByProperty(property){
   return function(a,b){
      if(a[property] > b[property])
         return 1;
      else if(a[property] < b[property])
         return -1;

      return 0;
   };
}
```

getCityInfo

```
const AWS = require('aws-sdk');

const ddb = new AWS.DynamoDB.DocumentClient();

exports.handler = (event, context, callback) => {

    getCityData().then(dbResults => {
        const cities = dbResults.Item.cities;
            callback(null, {
                statusCode: 201,
                body: JSON.stringify(cities),
                headers: {
                    'Access-Control-Allow-Origin': '*'
                }
            });
        }).catch(err => {
            console.log(`Problem collecting the City data!`);
            console.error(err);
            errorResponse(err.message, context.awsRequestId, callback);
        });  //calls the actual method for the post
```

```
}


function getCityData(callback) {
    return ddb.get({
        TableName: 'distance_data',
        Key: { region: 'Minnesota' },
        ProjectionExpression: "cities"
    }).promise()
}

function errorResponse(errorMessage, awsRequestId, callback) {
  callback(null, {
    statusCode: 500,
    body: JSON.stringify({
      Error: errorMessage,
      Reference: awsRequestId,
    }),
    headers: {
      'Access-Control-Allow-Origin': '*',
    },
  });
}
```

generateRandomRoute

```
const randomBytes = require('crypto').randomBytes;

const AWS = require('aws-sdk');

const ddb = new AWS.DynamoDB.DocumentClient();

exports.handler = (event, context, callback) => {
    const requestBody = JSON.parse(event.body);
    const runId = requestBody.runId;
    const generation = requestBody.generation;
    console.log(runId+ " run id");
    console.log(generation+" generation");
    const partitionKey = runId + '#' + generation;
    console.log(partitionKey);

    generateRandomRoute(runId, generation, callback, partitionKey);  //calls
the actual method for the post

}

function generateRandomRoute(runId, generation, callback, partitionKey) {
    getData().then(routeData =>{
```

```
        const routeObjectData = JSON.stringify(routeData.Item.cities);
        const cityDistances = routeData.Item.distances;

        const cityArray = new Array();
        //gets the number of cities in the db. Sourced from this
stackoverflow discussion https://stackoverflow.com/questions/13782698/get-
total-number-of-items-on-json-object
        const numOfCities = Object.keys(routeData.Item.cities).length;
        console.log("There are " + numOfCities + " cities in this route");
        fillCityArray(cityArray, numOfCities); // fills the array with the
index of cities
        shuffleCities(cityArray); // shuffles the city array


        const routeId = toUrlString(randomBytes(16));
        const routeDistance = calculateRouteDistance(cityArray,
cityDistances);

        return ddb.put(
            {TableName: 'routes',
                Item: {
                    runGen: partitionKey,
                    routeId: routeId,
                    route: cityArray,
                    len: routeDistance,
                    // runId: runId,
                    // generation: generation
                },

            }).promise().then(dbResults => {
            callback(null, {
                statusCode: 201,
                body: JSON.stringify({
                    routeId: routeId,
                    len: routeDistance,
                }),
                headers: {
                    'Access-Control-Allow-Origin': '*'
                }
            });
        });

    });
}

function getData() {
    return ddb.get({
        TableName: 'distance_data',
        Key: { region: 'Minnesota' }
    }).promise();
}
```

```
//Shuffle example comes from https://javascript.info/task/shuffle
function shuffleCities(array) {
    for (let i = array.length - 1; i > 0; i--) {
        let j = Math.floor(Math.random() * (i + 1)); // random index from 0
to i

        // swap elements array[i] and array[j]
        // we use "destructuring assignment" syntax to achieve that
        // you'll find more details about that syntax in later chapters
        // same can be written as:
        // let t = array[i]; array[i] = array[j]; array[j] = t
        [array[i], array[j]] = [array[j], array[i]];
    }
}

function fillCityArray(array, numOfCities) {
    for (let i = 0; i < numOfCities; i++){
        array[i] = i;
    }
}

function calculateRouteDistance(array, distances){
    let totalDistance = 0;
    for (let i = 0; i < array.length; i++){
        if (i == array.length-1){
            const finalStop = array[i];
            const startCity = array[0];
            const firstLastDistance = distances[finalStop][startCity];
            totalDistance = totalDistance + firstLastDistance;
        } else {
            const cityA = array[i];
            const cityB = array[i+1];
            const abDistance = distances[cityA][cityB];
            totalDistance = totalDistance + abDistance;
        }
    } return totalDistance;
}

function randomRoute (runId, generation, routeId, route, length) {
    return ddb.put({
        TableName: 'routes',
        Item: {
            //runId: 'runId',
            //generation: generation,
            partitionKey: runId + '#' + generation,
            routeId: routeId,
            route: route,
            sortKey: length
        },
    }).promise();
}
```

```
//From the "requestUnicorn" lambda function from the WildRydes demo created
by Amazon
function toUrlString(buffer) {
    return buffer.toString('base64')
        .replace(/\+/g, '-')
        .replace(/\//g, '_')
        .replace(/=/g, '');
}

//From the "requestUnicorn" lambda function from the WildRydes demo created
by Amazon
function errorResponse(errorMessage, awsRequestId, callback) {
    callback(null, {
        statusCode: 500,
        body: JSON.stringify({
            Error: errorMessage,
            Reference: awsRequestId,
        }),
        headers: {
            'Access-Control-Allow-Origin': '*',
        },
    });
}
```

Appendix II: JavaScript

```
(function evoTSPwrapper($) {
  const baseUrl =
    "https://sbfud4h6bd.execute-api.us-east-1.amazonaws.com/prod";

  /*
   * This is organized into sections:
   *  - Declaration of some global variables
   *  - The `runEvolution` function and helpers
   *  - The `runGeneration` function and helpers
   *  - The Ajax calls
   *  - The functions that update the HTML over time
   *  - The functions that keep track of the best route
   *  - The functions that initialize the map and plot the best route
   *
   * _Most_ of this is complete. You have to:
   *
   *  - Fill in all the Ajax/HTTP calls
   *  - Finish up some of the HTML update functions
   *
   * We gave you all the evolution stuff and the mapping code, although what
we gave
```

```
 * you is pretty crude and you should feel free to fancy it up.
 */

// Will be populated by `populateCityData`
var cityData;

// No routes worse than this length will be stored in the
// database or returned by methods that create new
// routes.
var lengthStoreThreshold = Infinity;

// `best` stores what we know about the best route we've
// seen so far. Here this is set to to "initial"
// values, and then then these values are updated as better
// routes are discovered.
var best = {
  runID: "", // The ID of the best current path
  bestPath: [], // The array of indices of best current path
  len: Infinity, // The length of the best current path
  coords: [], // The coordinates of the best current path
  lRoute: [[], []], // best route as lat-long data
};

////////////////////////////////////////////////////////////
// BEGIN OF RUN EVOLUTION //////////////////////////////////
////////////////////////////////////////////////////////////

// This runs the evolutionary process. This function and it's
// helper functions are all complete and you shouldn't have to
// change anything here. Some of these functions do call functions
// "outside" this, some of which you'll need to write. In particular
// you'll need to implement `randomRoute()` below in this section.
function runEvolution() {
  // Generate a new runId and set the current generation to 0
  const runId = generateUID(16);
  const initialGeneration = 0;
  $("#runId-text-field").val(runId);
  $("#current-generation").text(initialGeneration);

  // `async.series` takes an array of (asynchronous) functions, and
  // calls them one at a time, waiting until the promise generated by
  // one has been resolved before starting the next one. This is similar
  // to a big chain of f().then().then()... calls, but (I think) cleaner.
  //
  // cb in this (and down below in runGeneration) is short for "callback".
  // Each of the functions in the series takes a callback as its last
  // (sometimes only) argument. That needs to be either passed in to a
  // nested async tool (like `asyn.timesSeries` below) or called after
  // the other work is done (like the `cb()` call in the last function).
  async.series([
    initializePopulation, // create the initial population
    runAllGenerations,    // Run the specified number of generations
    showAllDoneAlert,     // Show an "All done" alert.
```

```
    ]);

    function initializePopulation(cb) {
      const populationSize = parseInt($("#population-size-text-
field").val());
      console.log(
        `Initializing pop for runId = ${runId} with pop size
${populationSize}, generation = ${initialGeneration}`
      );
      $("#new-route-list").text("");
      async.times(
        populationSize,
        (counter, rr_cb) => randomRoute(runId, initialGeneration, rr_cb),
        cb
      );
    }

    function runAllGenerations(cb) {
      // get # of generations
      const numGenerations = parseInt($("#num-generations").val());

      // `async.timesSeries` runs the given function the specified number
      // of times. Unlike `async.times`, which does all the calls in
      // "parallel", `async.timesSeries` makes sure that each call is
      // done before the next call starts.
      async.timesSeries(
        numGenerations,
        runGeneration,
        cb
      );
    }

    function showAllDoneAlert(cb) {
      alert("All done! Check the map for the most optimal route!");
      cb();
    }

    // Generate a unique ID; lifted from
https://stackoverflow.com/a/63363662
    function generateUID(length) {
      return window
        .btoa(
          Array.from(window.crypto.getRandomValues(new Uint8Array(length *
2)))
            .map((b) => String.fromCharCode(b))
            .join("")
        )
        .replace(/[+/]/g, "")
        .substring(0, length);
    }
  }

  function randomRoute(runId, generation, cb) {
```

```
    $.ajax({
      method: 'POST',
      url: baseUrl + '/routes',
      data: JSON.stringify({
        runId: runId,
        generation: generation,
        lengthStoreThreshold: lengthStoreThreshold
      })})

    // If the Ajax call succeeds, return the newly generated route.
    .done((newRoute) => { cb(null, newRoute); })
    // If the Ajax call fails, print a message and pass the error up through
    // the callback function `cb`.
    .fail((jqHXR, textStatus, err) => {
      console.error("Problem with randomRoute AJAX call: " + textStatus);
      cb(err);
    });
  }

  ////////////////////////////////////////////////////////////
  // END OF RUN EVOLUTION ////////////////////////////////////
  ////////////////////////////////////////////////////////////

  ////////////////////////////////////////////////////////////
  // BEGIN OF RUN GENERATION /////////////////////////////////
  ////////////////////////////////////////////////////////////

  // This runs a single generation, getting the best routes from the
  // specified generation, and using them to make a population of
  // new routes for the next generation via mutation. This is all
  // complete and you shouldn't need to change anything here. It
  // does, however, call things that you need to complete.

  function runGeneration(generation, cb) {
    const popSize = parseInt($("#population-size-text-field").val());
    console.log(`Running generation ${generation}`);

    // `async.waterfall` is sorta like `async.series`, except here the
value(s)
    // returned by one function in the array is passed on as the argument(s)
    // to the _next_ function in the array. This essentially "pipes" the
functions
    // together, taking the output of one and making it the input of the
next.
    //
    // The callbacks (cb) are key to this communication. Each function needs
to
    // call `cb(…)` as it's way of saying "I'm done, and here are the values
to
    // pass on to the next function". If one function returns three values,
    // like `cb(null, x, y, z)`, then those three values will be the
arguments
    // to the next function in the sequence.
```

```
    //
    // The convention with these callbacks is that the _first_ argument is
an
    // error if there is one, and the remaining arguments are the return
values
    // if the function was successful. So `cb(err)` would return the error
`err`,
    // while `cb(null, "This", "and", "that", 47)` says there's no error
(the `null`
    // in the first argument) and that there are four values to return
(three
    // strings and a number).
    //
    // Not everything here has value to pass on or wants a value. Several
are
    // just there to insert print statements for logging/debugging purposes.
    // If they don't have any values to pass on, they just call `cb()`.
    //
    // `async.constant` lets us insert one or more specific values into the
    // pipeline, which then become the input(s) to the next item in the
    // waterfall. Here we'll inserting the current generation number so it
will
    // be the argument to the next function.

    async.waterfall(
      [
        wait5seconds,
        updateGenerationHTMLcomponents,
        async.constant(generation), // Insert generation into the pipeline
        (gen, log_cb) => logValue("generation", gen, log_cb), // log
generation
        getBestRoutes, // These will be passed on as the parents in the next
steps
        (parents, log_cb) => logValue("parents", parents, log_cb), // log
parents
        displayBestRoutes,    // display the parents on the web page
        updateThresholdLimit, // update the threshold limit to reduce DB
writes
        generateChildren,
        (children, log_cb) => logValue("children", children, log_cb),
        displayChildren,      // display the children in the "Current
generation" div
        updateBestRoute
      ],
      cb
    );

    // Log the given value with the specified label. To work in the
    // waterfall, this has to pass the value on to the next function, which
we do with
    // `log_cb(null, value)` call at the end.

    function logValue(label, value, log_cb) {
```

```
      console.log(`In waterfall: ${label} = ${JSON.stringify(value)}`);
      log_cb(null, value);
    }

    // Wait 5 seconds before moving on. This is really just a hack to
    // help make sure that the DynamoDB table has reached eventual
    // consistency.

    function wait5seconds(wait_cb) {
      console.log(`Starting sleep at ${Date.now()}`);
      setTimeout(function () {
        console.log(`Done sleeping gen ${generation} at ${Date.now()}`);
        wait_cb(); // Call wait_cb() after the message to "move on" through
the waterfall
      }, 5000);
    }

    // Reset a few of the page components that should "start over" at each
    // new generation.

    function updateGenerationHTMLcomponents(reset_cb) {
      $("#new-route-list").text("");
      $("#current-generation").text(generation + 1);
      reset_cb();
    }

    // Given an array of "parent" routes, generate `numChildren` mutations
    // of each parent route. `numChildren` is computed so that the total
    // number of children will be (roughly) the same as the requested
    // population size. If, for example, the population size is 100 and
    // the number of parents is 20, then `numChildren` will be 5.
    function generateChildren (parents, genChildren_cb) {
      const numChildren = Math.floor(popSize / parents.length);
      // `async.each` runs the provided function once (in "parallel") for
      // each of the values in the array of parents.
      async.concat( // each(
        parents,
        (parent, makeChildren_cb) => {
          makeChildren(parent, numChildren, generation, makeChildren_cb);
        },
        genChildren_cb
      );
    }

    // We keep track of the "best worst" route we've gotten back from the
    // database, and store its length in the "global" `lengthStoreThreshold`
    // declared up near the top. The idea is that if we've seen K routes at
    // least as good as this, we don't need to be writing _worse_ routes
into
    // the database. This saves over half the DB writes, and doesn't seem to
    // hurt the performance of the EC search, at least for this simple
problem.
    function updateThresholdLimit(bestRoutes, utl_cb) {
```

```
      if (bestRoutes.length == 0) {
        const errorMessage = 'We got no best routes back. We probably
overwhelmed the write capacity for the database.';
        alert(errorMessage);
        throw new Error(errorMessage);
      }
      // We can just take the last route as the "worst" because the
      // Lambda/DynamoDB combo gives us the routes in sorted order by
      // length.
      lengthStoreThreshold = bestRoutes[bestRoutes.length - 1].len;
      $("#current-threshold").text(lengthStoreThreshold);
      utl_cb(null, bestRoutes);
    }
  }

  ////////////////////////////////////////////////////////////
  // END OF RUN GENERATION ///////////////////////////////////
  ////////////////////////////////////////////////////////////

  ////////////////////////////////////////////////////////////
  // START OF AJAX CALLS /////////////////////////////////////
  ////////////////////////////////////////////////////////////

  // These are the various functions that will make Ajax HTTP
  // calls to your various Lambdas. Some of these are *very* similar
  // to things you've already done in the previous project.

  // This should get the best routes in the specified generation,
  // which will be used (elsewhere) as parents. You should be able
  // to use the (updated) Lambda from the previous exercise and call
  // it in essentially the same way as you did before.
  //
  // You'll need to use the value of the `num-parents` field to
  // indicate how many routes to return. You'll also need to use
  // the `runId-text-field` field to get the `runId`.
  //
  // MAKE SURE YOU USE
  //
  //    (bestRoutes) => callback(null, bestRoutes),
  //
  // as the `success` callback function in your Ajax call. That will
  // ensure that the best routes that you get from the HTTP call will
  // be passed along in the `runGeneration` waterfall.
  function getBestRoutes(generation, callback) {
    const numParents = $('#num-parents').val();
    const runId = $('#runId-text-field').val();
    //const runGen = runId + '#' + generation;
    const url =
baseUrl+`/best?runId=${runId}&generation=${generation}&numToReturn=${numPare
nts}`;
    $.ajax({
      "url": url,
      "method": "GET",
```

```
    })

    .done((bestRoutes) => {callback(null, bestRoutes); })

        .fail((jqHXR, textStatus, err) => {
          console.error("Problem with getBestRoutes AJAX call: " +
textStatus);
          callback(err);
        });
  }

  // Create the specified number of children by mutating the given
  // parent that many times. Each child should have their generation
  // set to ONE MORE THAN THE GIVEN GENERATION. This is crucial, or
  // every route will end up in the same generation.
  //
  // This will use one of the new Lambdas that you wrote for the final
  // project.
  //
  // MAKE SURE YOU USE
  //
  //   children => cb(null, children)
  //
  // as the `success` callback function in your Ajax call to make sure
  // the children pass down through the `runGeneration` waterfall.
  function makeChildren(parent, numChildren, generation, cb) {
    //const url =
baseUrl+`/mutateroute?routeId=${parent.routeId}&lengthStoreThreshold=${lengt
hStoreThreshold}&numChildren=${numChildren}`;
    const url = baseUrl+`/mutateroute`;
    $.ajax({
      "url" : url,
      "method" : "POST",
      "data" : `{\"routeId\": \"${parent.routeId}\",
\"lengthStoreThreshold\": ${lengthStoreThreshold}, \"numChildren\":
${numChildren}}`,
    }).done((children) => cb(null, children))
        .fail((jqHXR, textStatus, err) => {
            console.error("Problem with makeChildren AJAX call: " +
textStatus);
            cb(err);
        });
  }

  // Get the full details of the specified route. You should largely
  // have this done from the previous exercise. Make sure you pass
  // `callback` as the `success` callback function in the Ajax call.
  function getRouteById(routeId, callback) {
    const url = baseUrl+`/routes/${routeId}`;
    $.ajax({
      "url" : url,
      "method": "GET",
    }).done(function (data) {
```

```
      //const route = data.Item;
      callback(data);
    })
  .fail((jqHXR, textStatus, err) => {
          console.error("Problem with getRouteById AJAX call: " +
textStatus);
          callback(err);
      });
  }

  // Get city data (names, locations, etc.) from your new Lambda that
returns
  // that information. Make sure you pass `callback` as the `success`
callback
  // function in the Ajax call.
  function fetchCityData(callback) {
    const url = baseUrl+`/city-data`;
    $.ajax({
      "url" : url,
      "method" : "GET",
    }).done(function (data) {
      callback(data);
    });
  }

  ////////////////////////////////////////////////////////////
  // START OF HTML DISPLAY ///////////////////////////////////
  ////////////////////////////////////////////////////////////

  // The next few functions handle displaying different values
  // in the HTML of the web app. This is all complete and you
  // shouldn't have to do anything here, although you're welcome
  // to modify parts of this if you want to change the way
  // things look.

  // A few of them are complete as is (`displayBestPath()` and
  // `displayChildren()`), while others need to be written:
  //
  // - `displayRoute()`
  // - `displayBestRoutes()`

  // Display the details of the best path. This is complete,
  // but you can fancy it up if you wish.
  function displayBestPath() {
    $("#best-length").text(best.len);
    $("#best-path").text(JSON.stringify(best.bestPath));
    $("#best-routeId").text(best.routeId);
    $("#best-route-cities").text("");
    best.bestPath.forEach((index) => {
      const cityName = cityData[index].properties.name;
      $("#best-route-cities").append(`<li>${cityName}</li>`);
    });
  }
```

```
  // Display all the children. This just uses a `forEach`
  // to call `displayRoute` on each child route. This
  // should be complete and work as is.
  function displayChildren(children, dc_cb) {
    children.forEach(child => displayRoute(child));
    dc_cb(null, children);
  }

  // Display a new (child) route (ID and length) in some way.
  // We just appended this as an `<li>` to the `new-route-list`
  // element in the HTML.
  function displayRoute(result) {
    $("#best-length").text(best.len);
    $("#best-path").text(JSON.stringify(best.bestPath));
    $("#best-routeId").text(best.routeId);
    $("#best-route-cities").text("");
    best.bestPath.forEach((index) => {
      const cityName = cityData[index].properties.name;
      $("#best-route-cities").append(`<li>${cityName}</li>`);
    });  }

  // Display the best routes (length and IDs) in some way.
  // We just appended each route's info as an `<li>` to
  // the `best-route-list` element in the HTML.
  //
  // MAKE SURE YOU END THIS with
  //
  //    dbp_cb(null, bestRoutes);
  //
  // so the array of best routes is pass along through
  // the waterfall in `runGeneration`.
  function displayBestRoutes(bestRoutes, dbp_cb) {
    $("#best-route-list").append(`<li><b>Route:</b> ${bestRoutes[0].route}.
    <b>Length (in meters):</b> ${bestRoutes[0].len}. <b>RouteId:</b>
${bestRoutes[0].routeId} </li>`);
    dbp_cb(null, bestRoutes);
  }

  ////////////////////////////////////////////////////////////
  // END OF HTML DISPLAY /////////////////////////////////////
  ////////////////////////////////////////////////////////////

  ////////////////////////////////////////////////////////////
  // START OF TRACKING BEST ROUTE ////////////////////////////
  ////////////////////////////////////////////////////////////

  // The next few functions keep track of the best route we've seen
  // so far. They should all be complete and not need any changes.

  function updateBestRoute(children, ubr_cb) {
    children.forEach(child => {
      if (child.len < best.len) {
```

```
        updateBest(child.routeId);
      }
    });
    ubr_cb(null, children);
  }

  // This is called whenever a route _might_ be the new best
  // route. It will get the full route details from the appropriate
  // Lambda, and then plot it if it's still the best. (Because of
  // asynchrony it's possible that it's no longer the best by the
  // time we get the details back from the Lambda.)
  //
  // This is complete and you shouldn't have to modify it.
  function updateBest(routeId) {
    getRouteById(routeId, processNewRoute);

    function processNewRoute(route) {
        //console.log(JSON.stringify(best));
        //console.log(JSON.stringify(route));
      // We need to check that this route is _still_ the
      // best route. Thanks to asynchrony, we might have
      // updated `best` to an even better route between
      // when we called `getRouteById` and when it returned
      // and called `processNewRoute`. The `route == ""`
      // check is just in case we our attempt to get
      // the route with the given idea fails, possibly due
      // to the eventual consistency property of the DB.
      if (best.len > route.len && route == "") {
        console.log(`Getting route ${routeId} failed; trying again.`);
        updateBest(routeId);
        return;
      }
      if (best.len > route.len) {
        console.log(`Updating Best Route for ${routeId}`);
        best.routeId = routeId;
        best.len = route.len;
        best.bestPath = route.route;
        displayBestPath(); // Display the best route on the HTML page
        best.bestPath[route.route.length] = route.route[0]; // Loop Back
        updateMapCoordinates(best.bestPath);
        mapCurrentBestRoute();
      }
    }
  }

  ////////////////////////////////////////////////////////////
  // END OF TRACKING BEST ROUTE //////////////////////////////
  ////////////////////////////////////////////////////////////

  ////////////////////////////////////////////////////////////
  // START OF MAPPING TOOLS //////////////////////////////////
  ////////////////////////////////////////////////////////////
```

```
  // The next few functions handle the mapping of the best route.
  // This is all complete and you shouldn't have to change anything
  // here.

  // Uses the data in the `best` global variable to draw the current
  // best route on the Leaflet map.

  function mapCurrentBestRoute() {
    var lineStyle = {
      dashArray: [10, 20],
      weight: 5,
      color: "#212121",
    };

    var fillStyle = {
      weight: 5,
      color: "#FFFFFF",
    };

    if (best.lRoute[0].length == 0) {
      // Initialize first time around
      best.lRoute[0] = L.polyline(best.coords, fillStyle).addTo(mymap);
      best.lRoute[1] = L.polyline(best.coords, lineStyle).addTo(mymap);
    } else {
      best.lRoute[0] = best.lRoute[0].setLatLngs(best.coords);
      best.lRoute[1] = best.lRoute[1].setLatLngs(best.coords);
    }
  }

  function initializeMap(cities) {
    cityData = [];
    for (let i = 0; i < cities.length; i++) {
      const city = cities[i];
      const cityName = city.cityName;
      var geojsonFeature = {
        type: "Feature",
        properties: {
          name: "",
          show_on_map: true,
          popupContent: "CITY",
        },
        geometry: {
          type: "Point",
          coordinates: [0, 0],
        },
      };
      geojsonFeature.properties.name = cityName;
      geojsonFeature.properties.popupContent = cityName;
      geojsonFeature.geometry.coordinates[0] = city.location[1];
      geojsonFeature.geometry.coordinates[1] = city.location[0];
      cityData[i] = geojsonFeature;
    }
```

```
    var layerProcessing = {
      pointToLayer: circleConvert,
      onEachFeature: onEachFeature,
    };

    L.geoJSON(cityData, layerProcessing).addTo(mymap);

    function onEachFeature(feature, layer) {
      // does this feature have a property named popupContent?
      if (feature.properties && feature.properties.popupContent) {
        layer.bindPopup(feature.properties.popupContent);
      }
    }

    function circleConvert(feature, latlng) {
      return new L.CircleMarker(latlng, { radius: 5, color: "#FF0000" });
    }
  }

  // This updates the `coords` field of the best route when we find
  // a new best path. The main thing this does is reverse the order of
  // the coordinates because of the mismatch between tbe GeoJSON order
  // and the Leaflet order.
  function updateMapCoordinates(path) {
    function swap(arr) {
      return [arr[1], arr[0]];
    }
    for (var i = 0; i < path.length; i++) {
      best.coords[i] = swap(cityData[path[i]].geometry.coordinates);
    }
    best.coords[i] = best.coords[0]; // End where we started
  }

  ////////////////////////////////////////////////////////////
  // END OF MAPPING TOOLS ////////////////////////////////////
  ////////////////////////////////////////////////////////////

  $(function onDocReady() {
    // These set you up with some reasonable defaults.
    $("#population-size-text-field").val(100);
    $("#num-parents").val(20);
    $("#num-generations").val(20);
    $("#run-evolution").click(runEvolution);
    // Get all the city data (names, etc.) once up
    // front to be used in the mapping throughout.
    fetchCityData(initializeMap);
  });
})(jQuery);
```

Appendix III: HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Evo-TSP Final</title>
    <link rel="shortcut icon" type="image/jpg" href="favicon.ico"/>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="description" content="Evolving solutions to a TSP instance"
/>
    <meta name="author" content="Robert Beane" />

    <link rel="preconnect" href="https://fonts.gstatic.com">
    <link
href="https://fonts.googleapis.com/css2?family=Roboto:wght@300&display=swap"
rel="stylesheet">

    <link
      rel="stylesheet"
      href="https://unpkg.com/leaflet@1.7.1/dist/leaflet.css"
      integrity="sha512-
xodZBNTC5n17Xt2atTPuE1HxjVMSvLVW9ocqUKLsCC5CXdbqCmblAshOMAS6/keqq/sMZMZ19scR
4PsZChSR7A=="
      crossorigin=""
    />
    Make sure you put this AFTER Leaflet's CSS
    <script
      src="https://unpkg.com/leaflet@1.7.1/dist/leaflet.js"
      integrity="sha512-
XQoYMqMTK8LvdxXYG3nZ448hOEQiglfqkJs1NOQV44cWnUrBc8PkAOcXy20w0vlaXaVUearIOBhi
XZ5V3ynxwA=="
      crossorigin=""
    ></script>
  </head>

  <body style="font-family: 'Roboto', sans-serif;">
    <h1>Evo-TSP</h1>
    <h4>By Robert Beane</h4>
    <div>
      <h2>"Global" parameters</h2>

      <label for="runId-text-field">Run ID:</label>
      <input type="text" id="runId-text-field" placeholder="This is auto
generated!"/>

      <label for="population-size-text-field">Population size:</label>
      <input type="text" id="population-size-text-field" />

      <label for="num-parents">Number of parents to keep:</label>
```

```
      <input type="text" id="num-parents" />
    </div>
    <br>
    <div id="map" style="height: 500px; width: 500px; border-style:
solid"></div>
    <div id="best-run-routes">
      <h2>Best so far</h2>
      <ul>
        <li>Best routeId: <span id="best-routeId"></span></li>
        <li>Best length (in meters): <span id="best-length"></span></li>
        <li>
          Best path: <span id="best-path"></span>
          <ol id="best-route-cities"></ol>
        </li>
        <li>
          Current threshold (in meters): <span id="current-
threshold"></span>
        </li>
      </ul>
    </div>

    <div class="run-evolution">
      <h2>Evolve solutions!</h2>

      <label for="num-generations">How many generations do you want to
run?</label>
      <input type="text" id="num-generations" />

      <button id="run-evolution">Start the Evolutions!</button>
    </div>

    <div class="current-generation">
      <h2>Current generation: <span id="current-generation"></span></h2>
      <div id="new-routes">
        <ol id="new-route-list"></ol>
      </div>
    </div>

    <div class="get-best-routes">
      <h2>Best routes from previous generation</h2>
      <div id="best-routes">
        <ol id="best-route-list"></ol>
      </div>
    </div>

    <script src="js/vendor/jquery-3.6.0.min.js"></script>
    <script src="js/vendor/async.min.js"></script>
    <script src="evotsp.js"></script>
    <script>
      var mymap = L.map("map").setView([46.7296, -94.6859], 6); //automate
or import view for future

      L.tileLayer(
```

```
"https://api.mapbox.com/styles/v1/{id}/tiles/{z}/{x}/{y}?access_token={acces
sToken}",
        {
          attribution:
            'Map data &copy; <a
href="https://www.openstreetmap.org/copyright">OpenStreetMap</a>
contributors, Imagery Â© <a href="https://www.mapbox.com/">Mapbox</a>',
          maxZoom: 18,
          id: "mapbox/streets-v11",
          tileSize: 512,
          zoomOffset: -1,
          accessToken:
            "TOKEN REMOVED",
        }
      ).addTo(mymap);
    </script>
  </body>
</html>
```