Multiple Traveling Salesperson

Exploring the MTSP problem using Genetic Algorithms

David Leonard

Date: May 1, 2016

Contents

1 Introduction **2**
1.1 Problem Variations 2

2 Genetic Algorithm **2**
2.1 Initial Population 2
2.2 Fitness Function 2
2.3 Selection 2
2.4 Crossover 3
2.5 Mutation 3
2.6 Results 3

3 Source Code **4**

1 Introduction

One of the most iconic problems in any Algorithms class is the **Traveling Salesperson** problem, in which a salesman must travel to n cities in a Hamiltonian Cycle while minimizing the distance traveled. In this project we explore a harder variation of this problem - the **Multiple Traveling Salesperson** problem. Here, multiple salesman n must travel across m cities in such a way that each salesman travels a Hamiltonian Path without intersecting with the other Hamiltonian Paths.

Put more specifically, we have $n=5$ salesman who must travel across $m=150$ cities while minimizing the total sum of their trips. The end goal of this problem is to divide these cities among all 5 salesman in such a way that the sum of their distances traveled is minimized.

1.1 Problem Variations

There are several variations to this problem, in which trips may start at a single ***depot*** - which is the starting location to which each salesman must return to at the end of their trip. By approaching the problem in this manner, we can effectively reduce the MTSP into 5 separate instances of the Traveling Salesperson problem. Another (and somewhat more difficult) approach is to consider the case in which each salesman has their own depot to return to. In this project, we consider this approach and explore how to solve it using **Genetic Algorithms**.

2 Genetic Algorithm

A genetic algorithm is one which is based on evolutionary traits from nature - we represent genes and chromosomes by a series of strings, which will represent our initial population. We then apply a series of genetic operators to iteratively evolve our population, which we hope will have a better solution to the problem as their predecessors.

2.1 Initial Population

In order to generate our initial population, we must determine how to represent a possible solution to our problem. We begin by generating an array which is enumerated from [1, 2, ..., 149, 150], and then generate a random tour by shuffling this array randomly. Next, we create another array which will contain the **partitions** of this trip to each of the 5 salesman. The initial array has the form [30, 30, 30, 30, 30], meaning that the first 30 cities is part of Salesman 1's (S1) tour, the next 30 are part of S2's tour, and so on. As we will see later on, the partitioning of these cities across the salesman will change. With this out of the way, we generate an **initial population** of 20.

2.2 Fitness Function

With our initial population created, we need to determine how to rank how well a given tour solves our problem. To do so, we derive what is known as the **fitness function** - where members (strings) of our population with the highest fitness correspond to better solutions to our problem. After some experimentation, the fitness function chosen for this project is simply 1 divided by the sum of the euclidean distances for each trip by each salesman.

2.3 Selection

While we may get some improvement over our initial fitnesses in our first population, it is highly unlikely that this generation provides the optimal solution. We must **select** the individuals which will reproduce and form the $n+1$ generations, through the process of **selection**. In order to help preserve good solutions, we choose a combination of **Elitist Selection** and **Tournament Selection**. With **Elitist Selection**, we specify how many individuals from our current population to bring over to our new population - in this case, we always carry over the individual which has the best fitness function. Using **Tournament Selection**, we

select 5 individuals based on a ***tournament size***, which we then select the two best individuals ranked by their fitness functions. These two individuals will then be used to form a new child through **Crossover**.

2.4 Crossover

Crossover is the action of taking the genetic material (in this case, arrays representing tours) from two individuals and create a new child (tour). The idea is that iteratively repeating this process will generate children with increasingly beneficial fitnesses. For this project, **Ordered Crossover** was used to generate new children. In order to create a child tour from two parent tours, we take the mother tour and copy a random subsection of this tour into the corresponding locations into the child. Next, we iterate over the father tour and populate the empty indices of the child tour from left to right - while making sure that we do not copy any genetic material from the father that was already given by the mother. By doing this, we preserve the invariant of the problem in which no salesman may revisit the same city twice in a tour (unless going back to their starting location). We also re-perform the partitioning of cities among the 5 salesman here, so the partition array which was once [30, 30, 30, 30, 30] will now change based on which cites were spliced around.

2.5 Mutation

Once a child has been successfully created, we now randomly mutate the child in such a way that cities are randomly swapped. This will preserve the problem invariant, and once again we re-perform the partitioning of cities among the 5 salesman (i.e, if a city belonging to S4 was given to S1, etc). One **improvement** to this would be to perform **two-point crossover**, in which we select two segments randomly and check if swapping them would result in a smaller euclidean distance, however we'll stick to the basic mutation method for this project. As a side note, during my experimentation it turned out that two-point mutation lead to amazing results, and often times performed better on its own without needing crossover at all. Once a child has been mutated, we repeat Selection, Crossover and Mutation until we have formed a new population of 20.

2.6 Results

We repeat this process 100 times - thereby creating 100 generations each consisting of 20 individuals in each population. In order to properly visualize the solution, we visualization component was also built for this project. Also, we start by observing the genetic algorithm in action with a smaller instance of this problem - using 50 cities as opposed to 150.

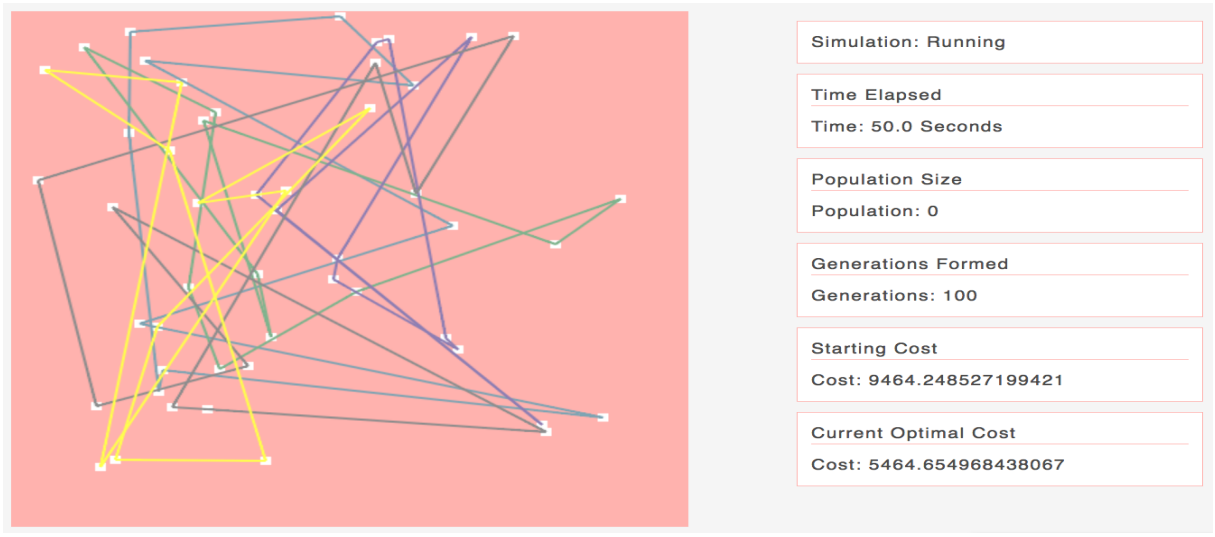

Figure 1: MTSP with 50 Cities and 5 Salesman

Based on the image above, we can see that the starting cost of the tours was 9464, and at the end of the simulation we have nearly cut that value in half, resulting in a total distance traveled of 5464. In the visualization, each color denotes the path taken by each salesman. Next, we observe the results when the number of cities is equal to 150:

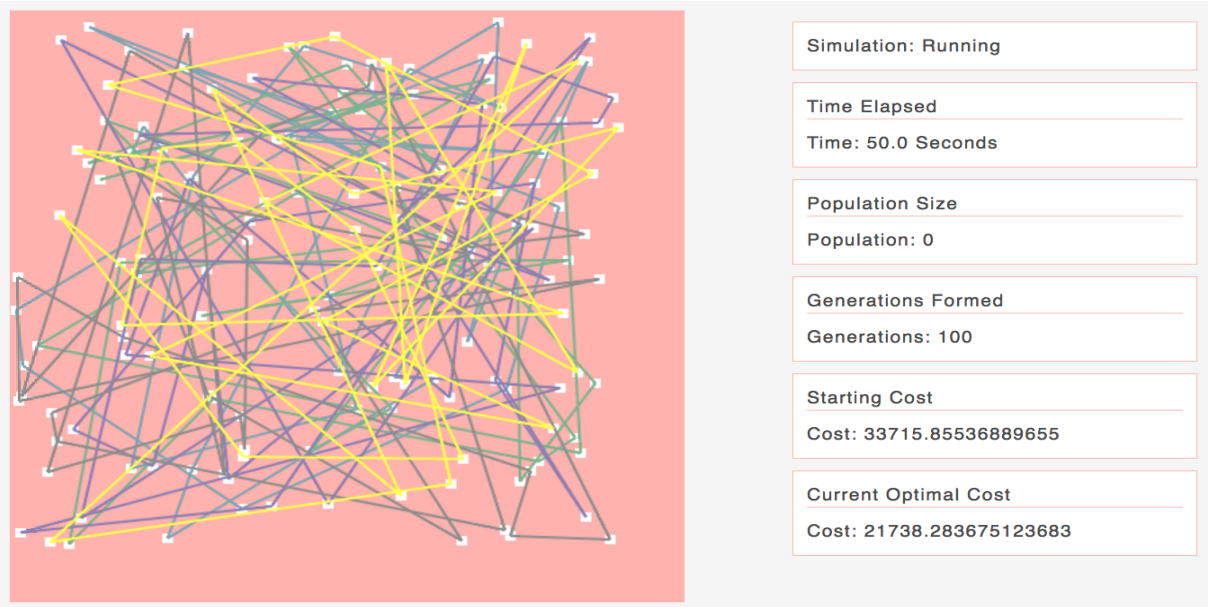

Figure 2: MTSP with 150 Cities and 5 Salesman

In this instance, we can see that the initial starting cost is 33715, and at the end of the simulation we have improved this solution by nearly 30% - leading to a final cost of 21738. An interesting pattern that occurs often is the optimal trip ends up needing only 4 salesman - as morbid as that may imply, one salesman ends up with zero cities in his trip. Another observation is the seemingly chaotic trips shown in the simulation with 150 cities. While intuition may lead to expecting nice patterns which are seemingly clustered, the random generation of cities as well as the smaller-scale visualization may play a role in this. Moreso, this algorithm can be greatly improved by generating initial trips using a greedy method such as a clustering algorithm using nearest neighbors as opposed to randomly generating the initial population.

3 Source Code

The source code in it's entirely can be found below.

```
// city.js

'use strict';

import {generateRandomInt} from './utils.js'

class City {
  /**
   * Create a city at location x, y.
   * @param {number} x - The x-coordinate.
   * @param {number} y - The y-coordinate.
  */
  constructor(x, y) {
    this.x = x;
    this.y = y;
    this.owner = null;
```

```
  }

  /**
   * Create a city at a random x, y location.
  */
  city() {
    this.x = generateRandomInt(0, 500);
    this.y = generateRandomInt(0, 500);
  }

  /**
   * Get the x-coordinate of this city.
   * @returns {number} The x-coordinate of this city.
  */
  getX() {
    return this.x;
  }

  /**
   * Get the y-coordinate of this city.
   * @returns {number} The y-coordinate of this city.
  */
  getY() {
    return this.y;
  }

  setOwner(owner) {
    this.owner = owner;
  }

  getOwner() {
    return this.owner;
  }

  /**
   * Computes the Euclidean Distance between two cities.
   * @param {object} city - The city object to compute distance to.
   * @returns {number} The euclidean distance from this city to the given city.
  */
  euclideanDistance(city) {
    const dx = this.x - city.x;
    const dy = this.y - city.y;
    return Math.sqrt(dx * dx + dy * dy);
  }

  /**
   * Prints the x, y position of this city.
   * @returns {undefined}
  */
  toString() {
    console.log(this.x + ', ' + this.y);
  }
}

export default City;

// destinations.js

'use strict';
```

```
/**
 * Stores all destination cities.
*/

class Destinations {
  /**
   * Store the list of destination cities.
   * @param {array} path - List of cities we can travel to.
  */
  constructor() {
    this.destinations = [];
  }

  /**
   * Adds a city to list of destinations.
   * @param {object} city - The city to add to destinations.
   * @returns {undefined}
  */
  addCity(city) {
    this.destinations.push(city);
  }

  /**
   * Gets a city from list of destinations.
   * @param {number} index - The index of the city to get.
   * @returns {object} undefined - The city at given index.
  */
  getCity(index) {
    return this.destinations[index];
  }

  /**
   * Gets the length of total destination cities.
   * @returns {number} undefined - Total number of destinations.
  */
  numberOfDestinations() {
    return this.destinations.length;
  }
}

export default Destinations;

// draw.js

'use strict';

class Draw {
  /**
   * Initializes references to the canvas element.
  */
  constructor() {
    this.canvas = document.getElementById('grid');
    this.ctx = this.canvas.getContext('2d');
    this.ctx.fillStyle = '#ffffff';
    this.ctx.lineWidth = 2;
    this.width = 500;
    this.height = 500;
  }
```

```
  /**
   * Draws a rectangle on the canvas.
   * @param {number} x - The x-coordinate to draw at.
   * @param {number} y - The y-coordinate to draw at.
  */
  drawPoint(x, y) {
    this.ctx.fillRect(x, y, 8, 8);
  }

  drawLine(x, y, toX, toY, strokeColor) {
    this.ctx.beginPath();
    this.ctx.strokeStyle = strokeColor;
    this.ctx.moveTo(x + 4, y + 4);
    this.ctx.lineTo(toX + 4, toY + 4);
    this.ctx.stroke();
  }

  clearCanvas() {
    this.ctx.clearRect(0, 0, this.width, this.height);
  }
}

export default Draw;

// genetic.js

'use strict';

import Trip from './trip.js';
import Population from './population.js';
import {containsObject, generateRandomInt} from './utils.js';

class Genetic {
  /**
   * Constructs the Genetic Algorithm parameters.
   * @param {array} destinations - Instance of the Destinations class.
  */
  constructor(destinations) {
    this.destinations = destinations;
    this.mutationRate = 0.01;
    this.tournamentSize = 5;
    this.elitism = true;
  }

  /**
   * Evolves a population using Genetic Operators.
   * @param {object} population - Instance of Population class.
   * @returns {array} newPopulation - The new generation of individuals.
  */
  evolvePopulation(population) {
    let newPopulation = new Population(this.destinations, population.populationSize(), false);
    let elitismOffset = 0;
    if (this.elitism) {
      newPopulation.saveTrip(0, population.getFittest());
      elitismOffset = 1;
    }

    for (var i = elitismOffset; i < newPopulation.populationSize(); i++) {
```

```
    let parent1 = this.tournamentSelection(population);
    let parent2 = this.tournamentSelection(population);
    let child   = this.crossover(parent1, parent2);
    newPopulation.saveTrip(i, child);
  }

  for (var i = elitismOffset; i < newPopulation.populationSize(); i++) {
    this.mutate(newPopulation.getTrip(i));
  }

  return newPopulation;
}

/**
 * Generates a new child from two parents.
 * @param {array} parent1 - Instance of Trip class.
 * @param {array} parent2 - Instance of Trip class.
 * @returns {array} child - The new child trip.
*/
crossover(parent1, parent2) {
  let child = new Trip(this.destinations);
  let start = generateRandomInt(0, parent1.getTripSize() - 1);
  let end   = generateRandomInt(0, parent1.getTripSize());
  let startPosition = Math.min(start, end);
  let endPosition   = Math.max(start, end);
  let cities = [];

  // Cities taken from mother chromosome
  let motherCities1 = 0;
  let motherCities2 = 0;
  let motherCities3 = 0;
  let motherCities4 = 0;
  let motherCities5 = 0;

  // Cities taken from father chromosome
  let fatherCities1 = 0;
  let fatherCities2 = 0;
  let fatherCities3 = 0;
  let fatherCities4 = 0;
  let fatherCities5 = 0;

  for (var x = 0; x < child.getTripSize(); x++) {
    cities.push(null);
  }

  for (var i = 0; i < child.getTripSize(); i++) {
    // Gets cities within range
    // Example:
    // Start position = 4
    // End position = 8;
    // Will have cities at index 5, 6, 7
    if ((startPosition < endPosition) && (i > startPosition) && (i < endPosition)) {
      cities[i] = parent1.getCity(i);
      let owner = parent1.getCity(i).getOwner();

      if (owner == 1) {
        motherCities1++
      }
```

```
    if (owner == 2) {
      motherCities2++;
    }

    if (owner == 3) {
      motherCities3++;
    }

    if (owner == 4) {
      motherCities4++;
    }

    if (owner == 5) {
      motherCities5++;
    }
  }
}

for (var i = 0; i < parent2.getTripSize(); i++) {
  // If parent 2 city not already in cities array...
  if (!containsObject(parent2.getCity(i), cities)) {
    // Loop over cities array
    for (var j = 0; j < cities.length; j++) {
      // If the entry is null, we insert here!
      if (cities[j] == null) {
        cities[j] = parent2.getCity(i);
        let owner = parent2.getCity(i).getOwner();

        if (owner == 1) {
          fatherCities1++
        }

        if (owner == 2) {
          fatherCities2++;
        }

        if (owner == 3) {
          fatherCities3++;
        }

        if (owner == 4) {
          fatherCities4++;
        }

        if (owner == 5) {
          fatherCities5++;
        }

        break;
      }
    }
  }
}

let partition1 = motherCities1 + fatherCities1;
let partition2 = motherCities2 + fatherCities2;
let partition3 = motherCities3 + fatherCities3;
let partition4 = motherCities4 + fatherCities4;
let partition5 = motherCities5 + fatherCities5;
```

```
    let childPartition  = [partition1, partition2, partition3, partition4, partition5];

    // Copy all cities into the child
    for (var i = 0; i < child.getTripSize(); i++) {
      child.setCity(i, cities[i]);
    }

    child.setPartition(childPartition);
    return child;
  }

  /**
   * Random mutates a trip, swaps positions of two locations in trip.
   * @param {array} trip - Instance of Trip class.
  */
  mutate(trip) {
    for (var i = 0; i < trip.getTripSize(); i++) {
      if (Math.random() < this.mutationRate) {
        let splice = Math.floor(trip.getTripSize() * Math.random());
        let city1  = trip.getCity(i);
        let city2  = trip.getCity(splice);
        trip.setCity(splice, city1);
        trip.setCity(i, city2);
      }
    }
  }

  /**
   * Tournament Selection for selecting fittest individual for next generation.
   * @param {array} population - Instance of Population class.
   * @returns {array} fittest - The fittest individual trip from the tournament.
  */
  tournamentSelection(population) {
    let tournament = new Population(this.destinations, this.tournamentSize, false);
    for (var i = 0; i < this.tournamentSize; i++) {
      let random = Math.floor(Math.random() * population.populationSize());
      tournament.saveTrip(i, population.getTrip(random));
    }
    let fittest = tournament.getFittest();
    return fittest;
  }
}

export default Genetic;

// main.js 'use strict';

import Draw from './draw.js';
import Trip from './trip.js';
import City from './city.js';
import Stats from './stats.js';
import Genetic from './genetic.js';
import Population from './population.js';
import Simulation from './simulation.js';
import Destinations from './destinations.js';
import {generateUniqueNumbers} from './utils.js';

// Store destination cities
let destinations = new Destinations();
```

```
// Generate destination cities
let xCoordinates = generateUniqueNumbers(150, 475);
let yCoordinates = generateUniqueNumbers(150, 475);
for (var j = 0; j < 150; j++) {
  let city = new City(xCoordinates[j], yCoordinates[j]);
  destinations.addCity(city);
}

// Setup Simulation class and event handlers
let simulation = new Simulation();
simulation.attachEventListners();

// Initialize Draw class
let draw = new Draw();

// Initialize Stats class
let stats = new Stats();

// Generate the initial population
let population = new Population(destinations, 20, true);

stats.setStartingDistance(population.getFittest().computeDistance());
stats.showStartingDistance();
stats.setOptimalDistance();
stats.showOptimalDistance();

// Instantiate Genetic Algorithm
let ga = new Genetic(destinations);
population = ga.evolvePopulation(population);

// Evolve over 100 generations
for (var x = 0; x < 100; x++) {
  (function(delay) {
    setTimeout(function() {
      population = ga.evolvePopulation(population);
      stats.setCurrentDistance(population.getFittest().computeDistance());
      stats.computeOptimalDistance();
      stats.showOptimalDistance();
      stats.incrementGenerations();
      stats.showGenerations();
      stats.incrementTime(0.5);
      stats.showTime();
    }, 500 * delay);

    let s1 = [];
    let s2 = [];
    let s3 = [];
    let s4 = [];
    let s5 = [];

    // Store fittest trip of this population
    let fittest = population.getFittest().getTrip();
    for (var i = 0; i < fittest.length; i++) {
      if (fittest[i].owner == 1) {
        // s1++;
        s1.push(fittest[i]);
      }
```

```
    if (fittest[i].owner == 2) {
      // s2++;
      s2.push(fittest[i]);
    }

    if (fittest[i].owner == 3) {
      // s3++;
      s3.push(fittest[i]);
    }

    if (fittest[i].owner == 4) {
      // s4++;
      s4.push(fittest[i]);
    }

    if (fittest[i].owner == 5) {
      // s5++;
      s5.push(fittest[i]);
    }
  }

  // Draw the fittest individual
  for (var j = 0; j < fittest.length; j++) {
    setTimeout(function() {
      draw.clearCanvas();
    }, 500 * delay);

    // Draw all the points
    for (var destination = 0; destination < destinations.numberOfDestinations(); destination++)
      let x = destinations.getCity(destination).x;
      let y = destinations.getCity(destination).y;
      setTimeout(function() {
        draw.drawPoint(x, y);
      }, 500 * delay);
    }

    for (var tourIndex = 0; tourIndex < s1.length; tourIndex++) {
      let x = s1[tourIndex].x;
      let y = s1[tourIndex].y;
      // Loop around for drawing
      if (tourIndex + 1 < s1.length) {
        let toX = s1[tourIndex + 1].x;
        let toY = s1[tourIndex + 1].y;
        setTimeout(function() {
          draw.drawLine(x, y, toX, toY, '#7ea4b3');
        }, 500 * delay);
      } else {
        let toX = s1[0].x;
        let toY = s1[0].y;
        setTimeout(function() {
          draw.drawLine(x, y, toX, toY, '#7ea4b3');
        }, 500 * delay);
      }
    }

    // Trip 2
    for (var tourIndex = 0; tourIndex < s2.length; tourIndex++) {
      let x = s2[tourIndex].x;
      let y = s2[tourIndex].y;
```

```
  // Loop around for drawing
  if (tourIndex + 1 < s2.length) {
    let toX = s2[tourIndex + 1].x;
    let toY = s2[tourIndex + 1].y;
    setTimeout(function() {
      draw.drawLine(x, y, toX, toY, '#7eb38d');
    }, 500 * delay);
  } else {
    let toX = s2[0].x;
    let toY = s2[0].y;
    setTimeout(function() {
      draw.drawLine(x, y, toX, toY, '#7eb38d');
    }, 500 * delay);
  }
}

// Trip 3
for (var tourIndex = 0; tourIndex < s3.length; tourIndex++) {
  let x = s3[tourIndex].x;
  let y = s3[tourIndex].y;
  // Loop around for drawing
  if (tourIndex + 1 < s3.length) {
    let toX = s3[tourIndex + 1].x;
    let toY = s3[tourIndex + 1].y;
    setTimeout(function() {
      draw.drawLine(x, y, toX, toY, '#8d7eb3');
    }, 500 * delay);
  } else {
    let toX = s3[0].x;
    let toY = s3[0].y;
    setTimeout(function() {
      draw.drawLine(x, y, toX, toY, '#8d7eb3');
    }, 500 * delay);
  }
}

// Trip 4
for (var tourIndex = 0; tourIndex < s4.length; tourIndex++) {
  let x = s4[tourIndex].x;
  let y = s4[tourIndex].y;
  // Loop around for drawing
  if (tourIndex + 1 < s4.length) {
    let toX = s4[tourIndex + 1].x;
    let toY = s4[tourIndex + 1].y;
    setTimeout(function() {
      draw.drawLine(x, y, toX, toY, '#8d8d8d');
    }, 500 * delay);
  } else {
    let toX = s4[0].x;
    let toY = s4[0].y;
    setTimeout(function() {
      draw.drawLine(x, y, toX, toY, '#8d8d8d');
    }, 500 * delay);
  }
}

// Trip 5
for (var tourIndex = 0; tourIndex < s5.length; tourIndex++) {
  let x = s5[tourIndex].x;
```

```
        let y = s5[tourIndex].y;
        // Loop around for drawing
        if (tourIndex + 1 < s5.length) {
          let toX = s5[tourIndex + 1].x;
          let toY = s5[tourIndex + 1].y;
          setTimeout(function() {
            draw.drawLine(x, y, toX, toY, '#ffff4d');
          }, 500 * delay);
        } else {
          let toX = s5[0].x;
          let toY = s5[0].y;
          setTimeout(function() {
            draw.drawLine(x, y, toX, toY, '#ffff4d');
          }, 500 * delay);
        }
      }
    }
  })(x);
}

// population.js

'use strict';

import Trip from './trip.js';

class Population {
  /**
   * Manages our population of trips.
   * @param {array} destinations - Instance of Destinations class.
   * @param {number} size - The size of the population to generate.
   * @param {boolean} initialize - Whether or not to initialize a new population.
  */
  constructor(destinations, size, initialize) {
    this.trips = [];
    this.size = size;
    for (var i = 0; i < size; i++) {
      this.trips.push(null);
    }
    if (initialize) {
      for (var i = 0; i < size; i++) {
        let newTrip = new Trip(destinations);
        newTrip.generateTrip();
        this.saveTrip(i, newTrip)
      }
    }
  }

  /**
   * Saves a trip.
   * @param {number} key - The population index to save trip to.
   * @param {number} value - The trip to save.
   * @returns {undefined}
  */
  saveTrip(key, value) {
    this.trips[key] = value;
  }

  /**
```

```
   * Gets a trip from our population.
   * @param {number} index - The individual to get from the population.
   * @returns {array} The trip from the population.
  */
  getTrip(index) {
    return this.trips[index];
  }

  /**
   * Computes the fittest individual tour.
   * @returns {number} fittest - The fitness of the best tour.
  */
  getFittest() {
    let fittest = this.trips[0];
    for (var i = 0; i < this.populationSize(); i++) {
      if (fittest.computeFitness() <= this.getTrip(i).computeFitness()) {
        fittest = this.getTrip(i);
      }
    }
    return fittest;
  }

  /**
   * Returns the number of trips in our population.
   * @returns {number} The number of individuals in the population.
  */
  populationSize() {
    return this.trips.length;
  }
}

export default Population;

// simulation.js

'use strict';

class Simulation {
  /**
   * Constructs the Simulation class.
  */
  constructor() {
    this.paused = true;
    this.running = false;
  }

  /**
   * Attaches an event listener to window object.
  */
  attachEventListners() {
    window.addEventListener('keydown', this.enableKeyboard.bind(this), false);
  }

  /**
   * Toggles pause value of Simulation class.
  */
  togglePause() {
    if (this.paused) {
      this.showRunning();
```

```
      this.paused = false;
    } else {
      this.showPaused();
      this.paused = true;
    }
  }

  /**
   * Tracks keyboard events, recognizing spacebar presses.
   * @param {object} event - The keyboard event via keypress.
   * @returns {undefined}
  */
  enableKeyboard(event) {
    if (event.which == 32) {
      event.preventDefault();
      this.togglePause();
    }
  }

  /**
   * Pauses the simulation.
  */
  pauseSimulation() {
    this.running = false;
  }

  /**
   * Resumes the simulation.
  */
  resumeSimulation() {
    this.running = true;
  }

  /**
   * Updates Simulation Pause Label to show simulation is running.
  */
  showRunning() {
    let simulationLabel = document.getElementById('simulation-pause');
    simulationLabel.textContent = 'Running';
  }

  /**
   * Updates Simulation Pause Label to show simulation is paused.
  */
  showPaused() {
    let simulationLabel = document.getElementById('simulation-pause');
    simulationLabel.textContent = 'Paused';
  }

  /**
   * Returns running state of simulation.
   * @returns {boolean} undefined - Whether the simulation is running or not.
  */
  isRunning() {
    return this.running;
  }
}

export default Simulation;
```

```
// stats.js

'use strict';

class Stats {
  /**
   * Stores parameters of the simulation.
  */
  constructor() {
    this.time = 0;
    this.generations = 0;
    this.startingDistance = 0;
    this.optimalDistance = 0;
    this.currentDistance = 0;
  }

  /**
   * Gets the time property of Stats.
   * @returns {number} Total time elapsed during simulation.
  */
  getTime() {
    return this.time;
  }

  /**
   * Increments time property by 1.
  */
  incrementTime(value) {
    this.time += value;
  }

  /**
   * Updates the Time Elapsed Label on simulation.
  */
  showTime() {
    let timeLabel = document.getElementById('simulation-time');
    if (this.getTime() % 1 === 0) {
      timeLabel.textContent = this.getTime() + '.0 Seconds';
    } else {
      timeLabel.textContent = this.getTime() + ' Seconds';
    }
  }

  /**
   * Gets the generations property of Stats.
   * @returns {number} Total number of generations formed during simulation.
  */
  getGenerations() {
    return this.generations;
  }

  /**
   * Increments generations property by 1.
  */
  incrementGenerations() {
    this.generations++;
  }
```

```
/**
 * Updates the Generations Formed Label on simulation.
*/
showGenerations() {
  let generationsLabel = document.getElementById('simulation-generation') ;
  generationsLabel.textContent = this.getGenerations();
}

/**
 * Gets the startingDistance property of Stats.
 * @returns {number} The starting distance of the simulation.
*/
getStartingDistance() {
  return this.startingDistance;
}

/**
 * Sets the startingDistance property of Stats.
 * @param {number} value - The distance value to set.
*/
setStartingDistance(value) {
  this.startingDistance = value;
}

showStartingDistance() {
  let startingDistanceLabel = document.getElementById('simulation-cost');
  startingDistanceLabel.textContent = this.getStartingDistance();
}

/**
 * Sets the currentDistance property of Stats.
 * @param {number} value - The current distance of the trip.
*/
setCurrentDistance(value) {
  this.currentDistance = value;
}

/**
 * Gets the currentDistance property of Stats.
 * @returns {number} The current distance of the trip.
*/
getCurrentDistance() {
  return this.currentDistance;
}

/**
 * Gets the optimal distance property of Stats.
 * @returns {number} The optimal distance of the simulation.
*/
getOptimalDistance() {
  return this.optimalDistance;
}

/**
 * Sets the current optimal distance = starting distance for the simulation.
*/
setOptimalDistance() {
  this.optimalDistance = this.getStartingDistance();
}
```

```
  computeOptimalDistance() {
    if (this.getCurrentDistance() < this.getOptimalDistance()) {
      this.optimalDistance = this.getCurrentDistance();
    }
  }

  showOptimalDistance() {
    let optimalDistanceLabel = document.getElementById('simulation-optimal');
    optimalDistanceLabel.textContent = this.getOptimalDistance()
  }
}

export default Stats;

// trip.js

'use strict';

import clone from 'clone';
import {shuffleArray, containsObject, generateRandomInt, arrayClone} from './utils.js';

class Trip {
  /**
   * Create a trip.
   * @param {array} destinations - Instance of Destinations class.
   * @param {number} fitness - The fitness of the trip.
   * @param {number} distance - The distance traveled on the trip.
  */
  constructor(destinations, trip = null) {
    this.destinations = destinations;
    this.trip = [];
    this.fitness = 0;
    this.distance = 0;
    this.partition = 0;
    if (trip != null) {
      this.trip = trip;
    } else {
      for (var i = 0; i < this.destinations.numberOfDestinations(); i++) {
        this.trip.push(null);
      }
    }
  }

  /**
   * Returns a city in the trip.
   * @param {number} index - Index of city in path.
   * @returns {object} undefined - The city at given index.
  */
  getCity(index) {
    return this.trip[index];
  }

  /**
   * Sets a city at specific location in the trip.
   * @param {number} index - The city index to change on the trip.
   * @param {object} city - The city to add to the trip.
   * @returns {undefined}
  */
```

```
setCity(index, city) {
  this.trip[index] = city;
  this.fitness = 0;
  this.distance = 0;
}

/**
 * Computes the fitness of the current trip.
 * @returns {number} fitness - The fitness of the current trip.
*/
computeFitness() {
  if (this.fitness == 0) {
    this.fitness = 1 / this.computeDistance();
  }
  return this.fitness;
}

/**
 * Returns the length of the current trip.
 * @returns {number} The number of cities in the current trip.
*/
getTripSize() {
  return this.trip.length;
}

/**
 * Generates a randomized trip.
 * @returns {undefined} Shuffles internal class trip.
*/
generateTrip() {
  for(var cityIndex = 0; cityIndex < this.destinations.numberOfDestinations(); cityIndex++) {
    this.setCity(cityIndex, this.destinations.getCity(cityIndex));
  }
  shuffleArray(this.trip);

  // Generate random partition
  let partition = this.generatePartition();

  // Assign partition
  this.setPartition(partition);

  // Generate ranges for cities owned by each salesman
  let ranges = [];
  let end = 0;
  for (var partitionIndex = 0; partitionIndex < this.partition.length; partitionIndex++) {
    end += this.partition[partitionIndex];
    ranges.push(end);
  }

  // Assign ownerships
  let trip1 = clone(this.trip.slice(0, ranges[0]));
  let trip2 = clone(this.trip.slice(ranges[0], ranges[1]));
  let trip3 = clone(this.trip.slice(ranges[1], ranges[2]));
  let trip4 = clone(this.trip.slice(ranges[2], ranges[3]));
  let trip5 = clone(this.trip.slice(ranges[3], ranges[4]));

  for (var i = 0; i < trip1.length; i++) {
    let city = trip1[i];
    city.owner = 1;
```

```
  }

  for (var i = 0; i < trip2.length; i++) {
    let city = trip2[i];
    city.owner = 2;
  }

  for (var i = 0; i < trip3.length; i++) {
    let city = trip3[i];
    city.owner = 3;
  }

  for (var i = 0; i < trip4.length; i++) {
    let city = trip4[i];
    city.owner = 4;
  }

  for (var i = 0; i < trip5.length; i++) {
    let city = trip5[i];
    city.owner = 5;
  }

  // Stitch the trips back together
  this.trip = trip1.concat(trip2).concat(trip3).concat(trip4).concat(trip5);
}

assignPartitions(partition, trip) {
  // Generate ranges for cities owned by each salesman
  let ranges = [];
  let end = 0;
  for (var partitionIndex = 0; partitionIndex < partition.length; partitionIndex++) {
    end += partition[partitionIndex];
    ranges.push(end);
  }

  // Assign ownerships
  let trip1 = clone(trip.slice(0, ranges[0]));
  let trip2 = clone(trip.slice(ranges[0], ranges[1]));
  let trip3 = clone(trip.slice(ranges[1], ranges[2]));
  let trip4 = clone(trip.slice(ranges[2], ranges[3]));
  let trip5 = clone(trip.slice(ranges[3], ranges[4]));

  for (var i = 0; i < trip1.length; i++) {
    let city = trip1[i];
    city.owner = 1;
  }

  for (var i = 0; i < trip2.length; i++) {
    let city = trip2[i];
    city.owner = 2;
  }

  for (var i = 0; i < trip3.length; i++) {
    let city = trip3[i];
    city.owner = 3;
  }

  for (var i = 0; i < trip4.length; i++) {
    let city = trip4[i];
```

```
    city.owner = 4;
  }

  for (var i = 0; i < trip5.length; i++) {
    let city = trip5[i];
    city.owner = 5;
  }

  // Stitch the trips back together
  trip = trip1.concat(trip2).concat(trip3).concat(trip4).concat(trip5);
  return trip;
}

/**
 * Checks whether a given city is inside the current trip.
 * @param {object} city - The city to test inclusion.
 * @returns {boolean} Whether the city is in the current trip.
*/
containsCity(city) {
  return containsObject(city, this.trip);
}

generatePartition() {
  return [30, 30, 30, 30, 30];
}

setPartition(partition) {
  this.partition = partition;
}

getPartition() {
  return this.partition;
}

getTrip() {
  return this.trip;
}

/**
 * Computes the total distance of the trip.
 * @returns {number} The distance of the entire trip.
*/
computeDistance() {
  if (this.distance == 0) {
    let start = 0;
    let end = 0;
    let totalDistance = 0;
    // console.log(this.partition);
    // Loop over each partition
    for (var partitionIndex = 0; partitionIndex < this.partition.length; partitionIndex++) {
      // Splice trip together
      end += this.partition[partitionIndex];
      let trip = this.trip.slice(start, end);
      start = end;

      // Compute the distance for this trip
      for(var cityIndex = 0; cityIndex < trip.length; cityIndex++) {
        let fromCity = trip[cityIndex];
        let destinationCity = null;
```

```
          // Get the next destination city if within bounds
          if (cityIndex + 1 < trip.length) {
            destinationCity = trip[cityIndex + 1];
          }
          // Return to starting position to complete the trip
          else {
            destinationCity = trip[0];
          }

          totalDistance += fromCity.euclideanDistance(destinationCity);
        }
      }

      this.distance = totalDistance;
    }

    return this.distance;
  }
}

export default Trip;

// utils.js

'use strict';

/**
 * Utility Methods.
 * @module js/utils
*/

/**
 * Generations a random integer between min and max (inclusive)
 * @param {number} min - The minimum desired output.
 * @param {number} max - The maximum desired output.
 * @return {number} undefined
*/
export function generateRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1) + min);
}

/**
 * Shuffles an array in O(n) time (Durstenfeld shuffle algorithm).
 * @param {array} array - The array to shuffle.
 * @returns {array} array - Shuffled input array.
*/
export function shuffleArray(array) {
  for (var i = array.length - 1; i > 0; i--) {
    let j = Math.floor(Math.random() * (i + 1));
    let temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }
  return array;
}

/**
 * Checks if an object is inside an array.
 * @param {object} obj - The object to test inclusion.
```

```
 * @param {array} array - The array to test object inclusion.
 * @returns {boolean} Whether the array contained the given object.
*/
export function containsObject(obj, array) {
  for (var i = 0; i < array.length; i++) {
    if (array[i] === obj) {
      return true;
    }
  }
  return false;
}

export function generateUniqueNumbers(length, max) {
  let arr = [];
  while (arr.length < length) {
    const randomNumber = Math.ceil(Math.random() * max);
    let found = false;
    for (var j = 0; j < arr.length; j++) {
      if (arr[j] == randomNumber) {
        found = true;
        break;
      }
    }
    if (!found) {
      arr[arr.length] = randomNumber;
    }
  }

  return arr;
}
```