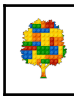
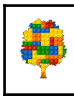
Section H
Unit #7
TSP
GA



Content

Section H Unit #7 TSP Genetic Algorithm



Introduction to Unit #7 (TSP) Genetic Algorithm

We have looked at how to solve this problem using brute force approach ie working though every possible solution in order to check for the shortest distance. This is fine for up to 10 cities but more than that it becomes computationally challenging.

Genetic algorithm gives us an opportunity to bring a solution that offers a result which is more accessible and less computationally demanding.

We need:
Population
Fitness
Mutation
Crossover

To recap we have two arrays

1. An array of cities which are fixed vectors (x, y) and so the array will look like this [(x1, y1), (x2, y2), (x3, y3), . . . (xn, yn)] where n is the number of cities

2. An array of order eg [0, 1, 2, 3, 4] and we will want a population of these [1, 3, 2, 4, 0] etc for a particular population size

Sketch H7.1 remove the permutations

First we remove a lot of the stuff related to the permutations (including count), factorial() function, and text on screen. There is quite a bit to remove so go through it patiently and find it. Also I have combined the cities within the same loop. I have highlighted and commented out all the code to be removed.

sketch.js

```
let cities = []
let totalCities = 5
let recordDistance
let bestEver
let order = []
// let totalPermutations
// let count = 1

function setup()
{
  createCanvas(400, 800)
  for (let i = 0; i < totalCities; i++)
  {
    let v = createVector(random(width), random(height / 2))
    cities[i] = v
    order[i] = i
  }
  let d = calcDistance(cities, order)
  recordDistance = d
  bestEver = order.slice()
  // totalPermutations = factorial(totalCities)
}

function draw()
```

```
{
  background(220)
  // for (let i = 0; i < cities.length; i++)
  // {
  //   stroke(0)
  //   fill(0)
  //   circle(cities[i].x, cities[i].y, 5)
  // }
  // beginShape()
  // for (let i = 0; i < order.length; i++)
  // {
  //   noFill()
  //   strokeWeight(1)
  //   let n = order[i]
  //   vertex(cities[n].x, cities[n].y)
  // }
  // endShape()
  beginShape()
  for (let i = 0; i < order.length; i++)
  {
    stroke(200, 0, 0)
    noFill()
    strokeWeight(3)
    let n = bestEver[i]
    vertex(cities[n].x, cities[n].y)
    circle(cities[i].x, cities[i].y, 5)
  }
  endShape()
  let d = calcDistance(cities, order)
  if (d < recordDistance)
  {
    recordDistance = d
    bestEver = order.slice()
```

```
  }
  // textSize(32)
  // noStroke()
  // fill(0)
  // let percent = 100 * (count / totalPermutations)
  // text(nf(percent, 0, 2) + '% completed', 20, 3 * height / 4)
  nextOrder()
}

function swap(a, i, j)
{
  let temp = a[i]
  a[i] = a[j]
  a[j] = temp
}

function calcDistance(points, order)
{
  let sum = 0
  for (let i = 0; i < order.length - 1; i++)
  {
    let cityAIndex = order[i]
    let cityA = points[cityAIndex]
    let cityBIndex = order[i + 1]
    let cityB = points[cityBIndex]
    let d = dist(cityA.x, cityA.y, cityB.x, cityB.y)
    sum += d
  }
  return sum
}

function nextOrder()
{
```

```
  // count++
  let largestI = -1
  for (let i = 0; i < order.length - 1; i++)
  {
    if (order[i] < order[i + 1])
    {
      largestI = i
    }
  }
  if (largestI == -1)
  {
    noLoop()
  }
  let largestJ = -1
  for (let j = 0; j < order.length; j++)
  {
    if (order[largestI] < order[j])
    {
      largestJ = j
    }
  }
  swap(order, largestI, largestJ)
  let endArray = order.splice(largestI + 1)
  endArray.reverse()
  order = order.concat(endArray)
}

// function factorial(n)
// {
//   if (n == 1)
//   {
//     return 1
//   }
```

```
//   else
//   {
//     return n * factorial(n-1)
//   }
// }
```

What you should be left with is the following

sketch.js

```
let cities = []
let totalCities = 5
let recordDistance
let bestEver
let order = []

function setup()
{
  createCanvas(400, 800)
  for (let i = 0; i < totalCities; i++)
  {
    let v = createVector(random(width), random(height / 2))
    cities[i] = v
    order[i] = i
  }
  let d = calcDistance(cities, order)
  recordDistance = d
  bestEver = order.slice()
}

function draw()
{
  background(220)
  beginShape()
  for (let i = 0; i < order.length; i++)
  {
    stroke(200, 0, 0)
    noFill()
    strokeWeight(3)
    let n = bestEver[i]
```

```
    vertex(cities[n].x, cities[n].y)
    circle(cities[i].x, cities[i].y, 5)
  }
  endShape()
  let d = calcDistance(cities, order)
  if (d < recordDistance)
  {
    recordDistance = d
    bestEver = order.slice()
  }
  nextOrder()
}

function swap(a, i, j)
{
  let temp = a[i]
  a[i] = a[j]
  a[j] = temp
}

function calcDistance(points, order)
{
  let sum = 0
  for (let i = 0; i < order.length - 1; i++)
  {
    let cityAIndex = order[i]
    let cityA = points[cityAIndex]
    let cityBIndex = order[i + 1]
    let cityB = points[cityBIndex]
    let d = dist(cityA.x, cityA.y, cityB.x, cityB.y)
    sum += d
  }
  return sum
```

```
}

function nextOrder()
{
  let largestI = -1
  for (let i = 0; i < order.length - 1; i++)
  {
    if (order[i] < order[i + 1])
    {
      largestI = i
    }
  }
  if (largestI == -1)
  {
    noLoop()
  }
  let largestJ = -1
  for (let j = 0; j < order.length; j++)
  {
    if (order[largestI] < order[j])
    {
      largestJ = j
    }
  }
  swap(order, largestI, largestJ)
  let endArray = order.splice(largestI + 1)
  endArray.reverse()
  order = order.concat(endArray)
}
```

This is what it should look like

Notes

All we are doing is drawing the bestEver as it cycles through all the permutations in order. This is to remove all the unnecessary stuff.

Sketch H7.2 adding a population

We add an empty population[] array and a population size (variable called popSize) of 10. We want to have a population of different orders, 10 different orders for the cities. The cities are vectors and they are fixed, what we are doing is visiting them in different orders.

To create different orders we take the original order and use a shuffle function to do that, I have included a console.log so that you can see it.

sketch.js

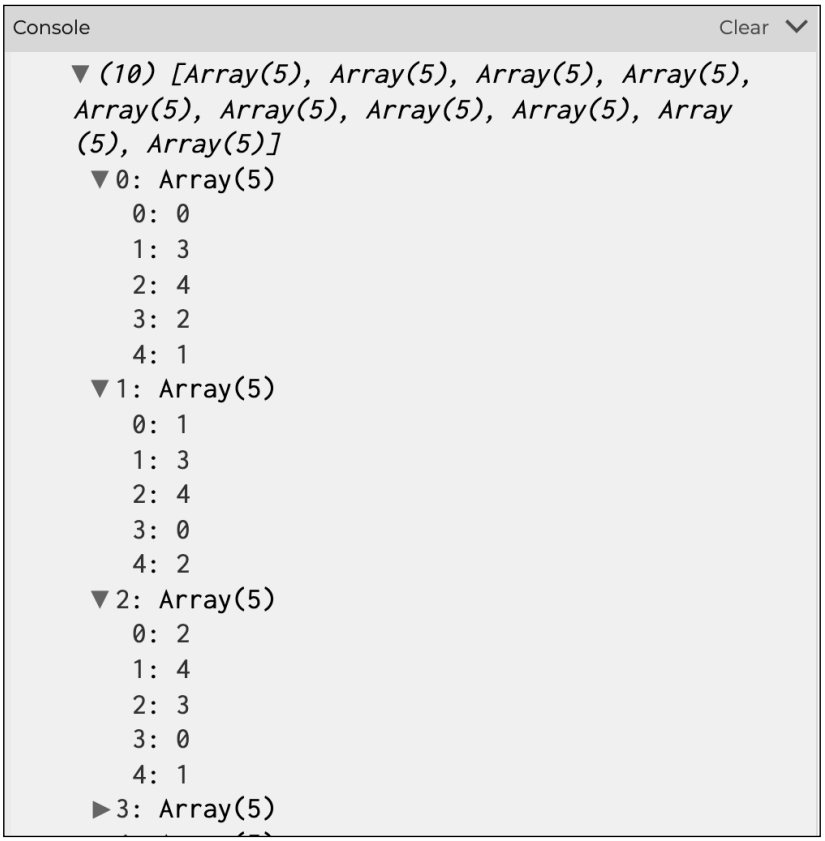
```
let cities = []
let totalCities = 5
let recordDistance
let bestEver
let order = []
let population = []
let popSize = 10

function setup()
{
  createCanvas(400, 800)
  for (let i = 0; i < totalCities; i++)
  {
    let v = createVector(random(width), random(height / 2))
    cities[i] = v
    order[i] = i
  }
  for (let i = 0; i < popSize; i++)
  {
    population[i] = shuffle(order)
  }
  console.log(population)
  let d = calcDistance(cities, order)
```

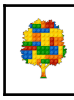
```
  recordDistance = d
  bestEver = order.slice()
}

function draw()
{
  background(220)
  for (let i = 0; i < cities.length; i++)
  {
    stroke(0)
    fill(0)
    circle(cities[i].x, cities[i].y, 5)
  }
  beginShape()
  for (let i = 0; i < order.length; i++)
  {
    stroke(0)
    noFill()
    strokeWeight(1)
    let n = order[i]
    vertex(cities[n].x, cities[n].y)
  }
  endShape()
  beginShape()
  for (let i = 0; i < order.length; i++)
  {
    stroke(200, 0, 0)
    noFill()
    strokeWeight(3)
    let n = bestEver[i]
    vertex(cities[n].x, cities[n].y)
  }
  endShape()
```

```
  let d = calcDistance(cities, order)
  if (d < recordDistance)
  {
    recordDistance = d
    bestEver = order.slice()
  }
  nextOrder()
}

function swap(a, i, j)
{
  let temp = a[i]
  a[i] = a[j]
  a[j] = temp
}

function calcDistance(points, order)
{
  let sum = 0
  for (let i = 0; i < order.length - 1; i++)
  {
    let cityAIndex = order[i]
    let cityA = points[cityAIndex]
    let cityBIndex = order[i + 1]
    let cityB = points[cityBIndex]
    let d = dist(cityA.x, cityA.y, cityB.x, cityB.y)
    sum += d
  }
  return sum
}

function nextOrder()
{
```

```
  let largestI = -1
  for (let i = 0; i < order.length - 1; i++)
  {
    if (order[i] < order[i + 1])
    {
      largestI = i
    }
  }
  if (largestI == -1)
  {
    noLoop()
  }
  let largestJ = -1
  for (let j = 0; j < order.length; j++)
  {
    if (order[largestI] < order[j])
    {
      largestJ = j
    }
  }
  swap(order, largestI, largestJ)
  let endArray = order.splice(largestI + 1)
  endArray.reverse()
  order = order.concat(endArray)
}
```

Having a look inside the shuffled population array to check that it has shuffled them





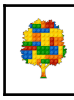
Sketch H7.3 fitness score

Next step is to give each one a fitness score and store it in an array. We will use the ga.js file as well as the sketch.js file. So in ga.js we create a function to calculate the fitness called calculateFitness(). We calculate the distance for each population[i] remember that the population is the order of the cities.

The fitness value needs to be high for a lower value of d and vice versa so to do that we invert it and add 1 to d so that it cannot be zero otherwise it would crash the programme.

ga.js

```
function calculateFitness()
{
  for (let i = 0; i < population.length; i++)
  {
    let d = calcDistance(cities, population[i])
    if (d < recordDistance)
    {
      recordDistance = d
      bestEver = population[i]
    }
    fitness[i] = 1 / (d+1)
  }
}
```



Sketch H7.4 normalise fitness

We want to normalise the fitness so that they all add up to 100% (total to 1), so we add another function in ga.js

ga.js

```
function calculateFitness()
{
  for (let i = 0; i < population.length; i++)
  {
    let d = calcDistance(cities, population[i])
    if (d < recordDistance)
    {
      recordDistance = d
      bestEver = population[i]
    }
    fitness[i] = 1 / (d + 1)
  }
}
```

```
function normaliseFitness()
{
  let sum = 0
  for (let i = 0; i < fitness.length; i++)
  {
    sum += fitness[i]
  }
  for (let i = 0; i < fitness.length; i++)
  {
    fitness[i] = fitness[i] /sum
  }
}
```

Sketch H7.5 next generation

Next we want to create the next generation with a new population array called newPopulation[]. We want to pick one that has a high fitness score and combine it with another one that has high fitness score. We also want to keep the population size the same. So to do that we need another function called pickOne(). We pick one from the population based on its corresponding fitness

ga.js

```
function calculateFitness()
{
  for (let i = 0; i < population.length; i++)
  {
    let d = calcDistance(cities, population[i])
    if (d < recordDistance)
    {
      recordDistance = d
      bestEver = population[i]
    }
    fitness[i] = 1 / (d+1)
  }
}

function normaliseFitness()
{
  let sum = 0
  for (let i = 0; i < fitness.length; i++)
  {
    sum += fitness[i]
  }
  for (let i = 0; i < fitness.length; i++)
  {
```

```
    fitness[i] = fitness[i] /sum
  }
}
```

```
function nextGeneration()
{
  let newPopulation = []
  for (let i = 0; i < population.length; i++)
  {
    let order = pickOne(population, fitness)
  }
}
```

Sketch H7.6 pick one

Now we add the pickOne() function which takes its arguments from the population and corresponding fitness array. What this does if a greater likelihood of being picked to those whose fitness levels are higher. Although it has a random element between 0-1, there is more chance if landing on a higher score than a lower score based on their relative fitness scores.

ga.js

```
function calculateFitness()
{
  for (let i = 0; i < population.length; i++)
  {
    let d = calcDistance(cities, population[i])
    if (d < recordDistance)
    {
      recordDistance = d
      bestEver = population[i]
    }
    fitness[i] = 1 / (d+1)
  }
}

function normaliseFitness()
{
  let sum = 0
  for (let i = 0; i < fitness.length; i++)
  {
    sum += fitness[i]
  }
  for (let i = 0; i < fitness.length; i++)
  {
    fitness[i] = fitness[i] /sum
```

```
  }
}

function nextGeneration()
{
  let newPopulation = []
  for (let i = 0; i < population.length; i++)
  {
    let order = pickOne(population, fitness)
  }
}
```

```
function pickOne(list, prob)
{
  let index = 0
  let r = random(1)
  while (r > 0)
  {
    r = r - prob[index]
    index++
  }
  index--
  return list[index]
}
```

Notes

There is no improvement because it just picks the best one, what it needs is a small injection of mutation. This means that there is a greater chance of finding more optimal solutions if that mutation is a useful mutation. The mutate function is key to a genetic algorithmic (evolutionary) approach

Sketch H7.7 mutate

Adding a mutate() function and call it in the nextGeneration() function at a rate of 0.01

ga.js

```
function calculateFitness()
{
  for (let i = 0; i < population.length; i++)
  {
    let d = calcDistance(cities, population[i])
    if (d < recordDistance)
    {
      recordDistance = d
      bestEver = population[i]
    }
    fitness[i] = 1 / (d+1)
  }
}

function normaliseFitness()
{
  let sum = 0
  for (let i = 0; i < fitness.length; i++)
  {
    sum += fitness[i]
  }
  for (let i = 0; i < fitness.length; i++)
  {
    fitness[i] = fitness[i] /sum
  }
}
```

```
function nextGeneration()
{
  let newPopulation = []
  for (let i = 0; i < population.length; i++)
  {
    let order = pickOne(population, fitness)
    mutate(order, 0.01)
    newPopulation[i] = order
  }
  population = newPopulation
}

function pickOne(list, prob)
{
  let index = 0
  let r = random(1)
  while (r > 0)
  {
    r = r - prob[index]
    index++
  }
  index--
  return list[index].slice()
}

function mutate(order, mutationRate)
{
  let indexA = floor(random(order.length))
  let indexB = floor(random(order.length))
  swap(order, indexA, indexB)
}
```

Sketch H7.8 generic algorithm

We also need to make changes to sketch.js, removing the function nextOrder because we are using a generic algorithm to work through the order. There are a few other things that are now redundant. As you run it with 10 cities and a population of 100 you will notice that its response is much better. I have highlighted and commented out the bits to get rid of.

sketch.js

```
let cities = []
let totalCities = 10
let recordDistance = Infinity
let bestEver
let order = []
let population = []
let popSize = 100
let fitness = []

function setup()
{
  createCanvas(400, 800)
  for (let i = 0; i < totalCities; i++)
  {
    let v = createVector(random(width), random(height / 2))
    cities[i] = v
    order[i] = i
  }
  for (let i = 0; i < popSize; i++)
  {
    population[i] = shuffle(order)
  }
  // console.log(population)
  // let d = calcDistance(cities, order)
```

```
  // recordDistance = d
  // bestEver = order.slice()
}

function draw()
{
  background(220)
  calculateFitness()
  normaliseFitness()
  nextGeneration()
  beginShape()
  for (let i = 0; i < order.length; i++)
  {
    stroke(200, 0, 0)
    noFill()
    strokeWeight(3)
    let n = bestEver[i]
    vertex(cities[n].x, cities[n].y)
    circle(cities[i].x, cities[i].y, 5)
  }
  endShape()
  // let d = calcDistance(cities, order)
  // if (d < recordDistance)
  // {
  //   recordDistance = d
  //   bestEver = order.slice()
  // }
  // nextOrder()
}

function swap(a, i, j)
{
  let temp = a[i]
```

```
  a[i] = a[j]
  a[j] = temp
}

function calcDistance(points, order)
{
  let sum = 0
  for (let i = 0; i < order.length - 1; i++)
  {
    let cityAIndex = order[i]
    let cityA = points[cityAIndex]
    let cityBIndex = order[i + 1]
    let cityB = points[cityBIndex]
    let d = dist(cityA.x, cityA.y, cityB.x, cityB.y)
    sum += d
  }
  return sum
}

// function nextOrder()
// {
//   let largestI = -1
//   for (let i = 0; i < order.length - 1; i++)
//   {
//     if (order[i] < order[i + 1])
//     {
//       largestI = i
//     }
//   }
//   if (largestI == -1)
//   {
//     noLoop()
```

```
//   }
//   let largestJ = -1
//   for (let j = 0; j < order.length; j++)
//   {
//     if (order[largestI] < order[j])
//     {
//       largestJ = j
//     }
//   }
//   swap(order, largestI, largestJ)
//   let endArray = order.splice(largestI + 1)
//   endArray.reverse()
//   order = order.concat(endArray)
// }
```