Project 7.7 Heuristics for traveling salespeople

Imagine that you drive a delivery truck for one of the major package delivery companies. Each day, you are presented with a list of addresses to which packages must be delivered. Before you can call it a day, you must drive from the distribution center to each package address, and back to the distribution center. This cycle is known as a ***tour***. Naturally, you wish to minimize the total distance that you need to drive to deliver all the packages, i.e., the total length of the tour. For example, Figures 3 and 4 show two different tours.

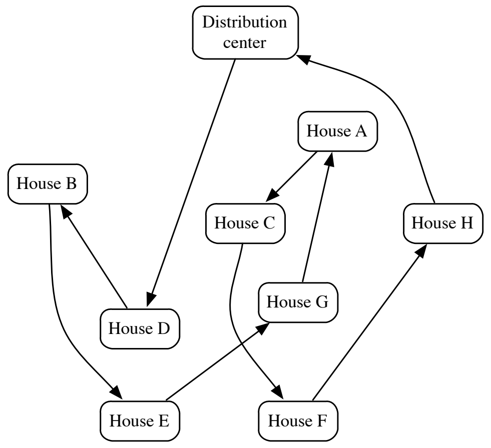


Figure 3 An inefficient tour.

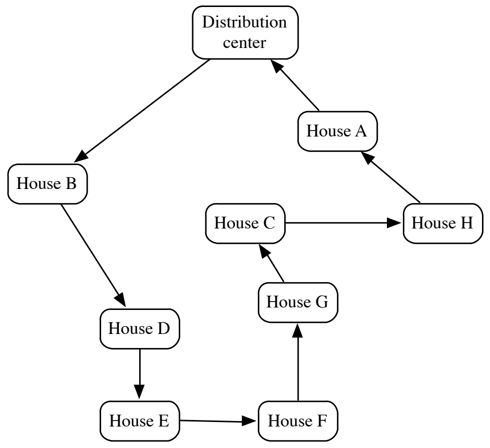


Figure 4 A more efficient tour.

This is known as the *traveling salesperson problem (TSP)*, and it is notoriously difficult. In fact, as far as anyone knows, the only way to come up with a guaranteed correct solution is to essentially enumerate all possible tours and choose the best one. But since, for n locations, there are $n!$ (n factorial) different tours, this is practically impossible.

Unfortunately, the TSP has many important applications, several of which seem at first glance to have nothing at all to do with traveling or salespeople, including circuit board drilling, controlling robots, designing networks, x-ray crystallography, scheduling computer time, and assembling genomes. In these cases, a heuristic must be used. A heuristic does not necessarily give the best answer, but it tends to work well in practice

For this project, you will design your own heuristic, and then work with a genetic algorithm, a type of heuristic that mimics the process of evolution to iteratively improve problem solutions.

Part 1: Write some utility functions

Each point on your itinerary will be represented by (x, y) coordinates, and the input to the problem is a list of these points. A tour will also be represented by a list of points; the order of the points indicates the order in which they are visited. We will store a list of points as a list of tuples.

The following function reads in points from a file and returns the points as a list of tuples. We assume that the file contains one point per line, with the x and y coordinates separated by a space.

```
def readPoints(filename):
    inputFile = open(filename, 'r')
    points = []
    for line in inputFile:
        values = line.split()
        points.append((float(values[0]), float(values[1])))
    return points
```

To begin, write the following three functions. The first two will be needed by your heuristics, and the third will allow you to visualize the tours that you create. To test your functions, and the heuristics that you will develop below, use the example file containing the coordinates of 96 African cities (`africa.tsp`) on the book website.

1. `distance(p, q)` returns the distance between points `p` and `q`, each of which is stored as a two-element tuple.
2. `tourLength(tour)` returns the total length of a tour. The tour is stored as a list of tuples. Remember to include the distance from the last point back to the first point.
3. `drawTour(tour)` draws a tour using turtle graphics. Use the `setworldcoordinates` method to make the coordinates in your drawing window more closely match the coordinates in the data files you use. For example, for the `africa.tsp` data file, the following will work well:

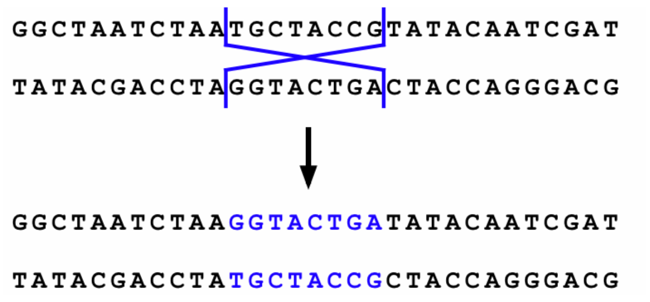
   ```
   screen.setworldcoordinates(-40, -25, 40, 60)
   ```

Part 2: Design a heuristic

Now design your own heuristic to find a good TSP tour. There are many possible ways to go about this. Be creative. Think about what points should be next to each other. What kinds of situations should be fixed? Use the `drawTour` function to visualize your tours and help you design your heuristic.

Part 3: A genetic algorithm

A ***genetic algorithm*** attempts to solve a hard problem by emulating the process of evolution. The basic idea is to start with a population of feasible solutions to the problem, called individuals, and then iteratively try to improve the fitness of this population through the evolutionary operations of recombination and mutation. In genetics, recombination is when chromosomes in a pair exchange portions of their DNA during meiosis. The illustration below shows how a crossover would affect the bases in a particular pair of (single stranded) DNA molecules.


GGCTAATCTAATGCTACCGTATACAATCGAT
TATACGACCTAGGTACTGACTACCAGGGACG
GGCTAATCTAAGGTACTGATATACAATCGAT
TATACGACCTATGCTACCGCTACCAGGGACG

Mutation occurs when a base in a DNA molecule is replaced with a different base or when bases are inserted into or deleted from a sequence. Most mutation is the result of DNA replication errors but environmental factors can also lead to mutations in DNA.

To apply this technique to the traveling salesperson problem, we first need to define what we mean by an individual in a population. In genetics, an individual is represented by its DNA, and an individual's fitness, for the purposes of evolution, is some measure of how well it will thrive in its environment. In the TSP, we will have a population of tours, so an individual is one particular tour — a list of cities. The most natural fitness function for an individual is the length of the tour; a shorter tour is more fit than a longer tour.

Recombination and mutation on tours are a bit trickier conceptually than they are for DNA. Unlike with DNA, swapping two subsequences of cities between two tours is not likely to produce two new valid tours. For example, suppose we have two tours `[a, b, c, d]` and `[b, a, d, c]`, where each letter represents a point. Swapping the two middle items between the tours will produce the offspring `[a, a, d, d]` and `[b, b, c, c]`, neither of which are permutations of the cities. One way around this is to delete from the first tour the cities in the portion to be swapped from the second tour, and then insert this portion from the second tour. In the above example, we would delete points `a` and `d` from the first tour, leaving `[b, c]`, before inserting `[a, d]` in the middle. Doing likewise for the second tour gives us children `[b, a, d, c]` and `[a, b, c, d]`. But we are not limited in genetic programming to recombination that more or less mimics that found in nature. A recombination operation can be anything that creates new offspring by somehow combining two parents. A large part of this project involves brainstorming about and experimenting with different techniques.

We must also rethink mutation since we cannot simply replace an arbitrary city with another city and end up with a valid tour. One idea might be to swap the positions of two randomly selected cities instead. But there are other possibilities as well.

Your mission is to improve upon a baseline genetic algorithm for TSP. Be creative! You may change anything you wish as long as the result can still be considered a genetic algorithm. To get started, download the baseline program from the book website. Try running it with the 96-point instance on the book website. Take some time to understand how the program works. Ask questions. You may want to refer to the Python documentation if you don't recall how a particular function works. Most of the work is performed by the following four functions:

- `makePopulation(cities)`: creates an initial population (of random tours)
- `crossover(mom, pop)`: performs a recombination operation on two tours and returns the two offspring
- `mutate(individual)`: mutates an individual tour
- `newGeneration(population)`: update the population by performing a crossover and mutating the offspring

Write the function

```
histogram(population)
```

that is called from the `report` function. (Use a Python dictionary.) This function should print a frequency chart (based on tour length) that gives you a snapshot of the diversity in your population. Your histogram function should print something like this:

```
Population diversity
   1993.2714596455853 : ****
   2013.1798076309087 : **
   2015.1395212505120 : ****
   2017.1005248468230 : ********************************
   2020.6881282400334 : *
   2022.9044855489917 : *
   2030.9623523675089 : *
   2031.4773010231959 : *
   2038.0257926528227 : *
   2040.7438913120230 : *
   2042.8148398732630 : *
   2050.1916058477627 : *
```

This will be very helpful as you strive to improve the algorithm: recombination in a homogeneous population is not likely to get you very far.

Brainstorm ways to **improve the algorithm**. Try lots of different things, ranging from tweaking parameters to completely rewriting any of the four functions described above. You are free to change anything, as long as the result still resembles a genetic algorithm. Keep careful records of what works and what doesn't to include in your submission.

On the book website is a link to a very good reference [51] that will help you think of new things to try. Take some time to skim the introductory sections, as they will give you a broader sense of the work that has been done on this problem. Sections 2 and 3 contain information on genetic algorithms; Section 5 contains information on various recombination/crossover operations; and Section 7 contains information on possible mutation operations. As you will see, this problem has been well studied by researchers over the past few decades! (To learn more about this history, we recommend *In Pursuit of the Traveling Salesman* by William Cook [10].)