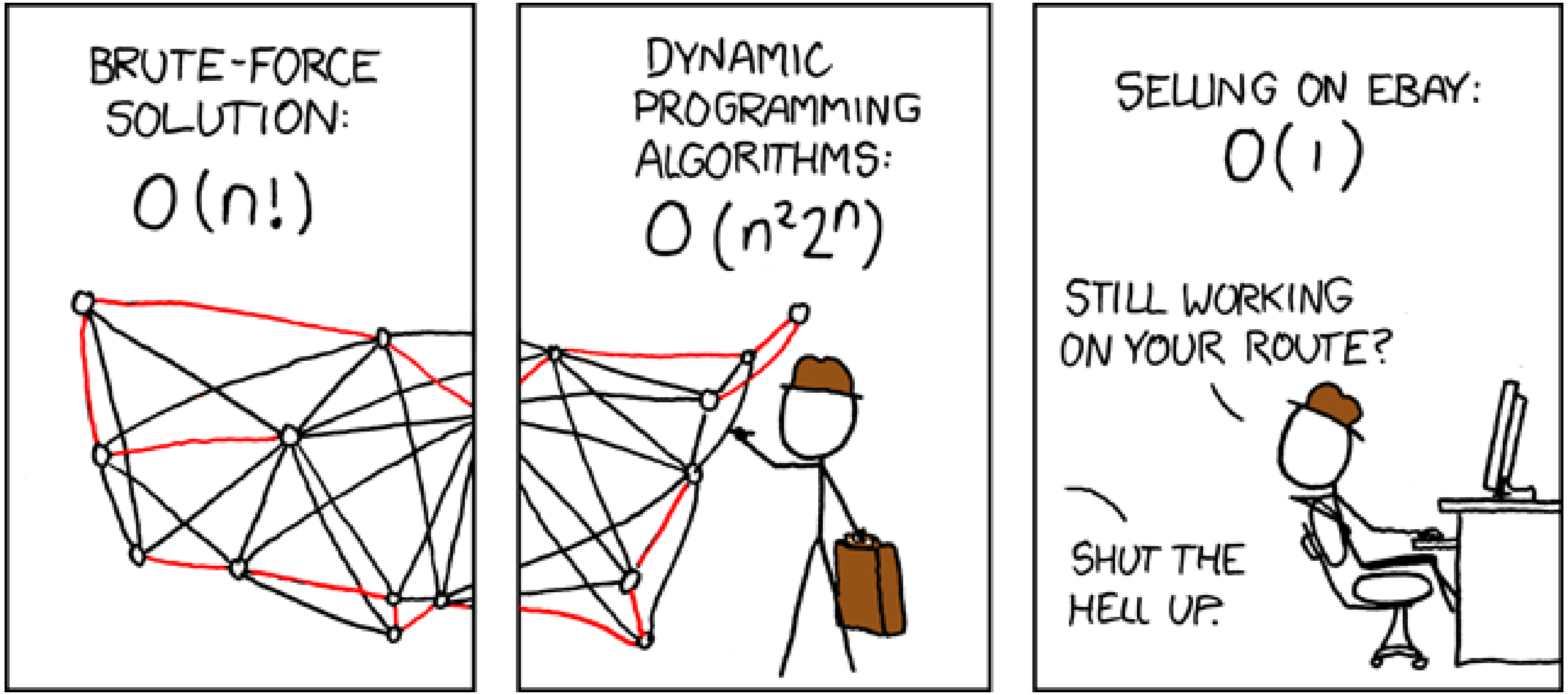
Traveling Salesman Problem

John Franklin Crenshaw



The Traveling Salesman Problem (TSP)

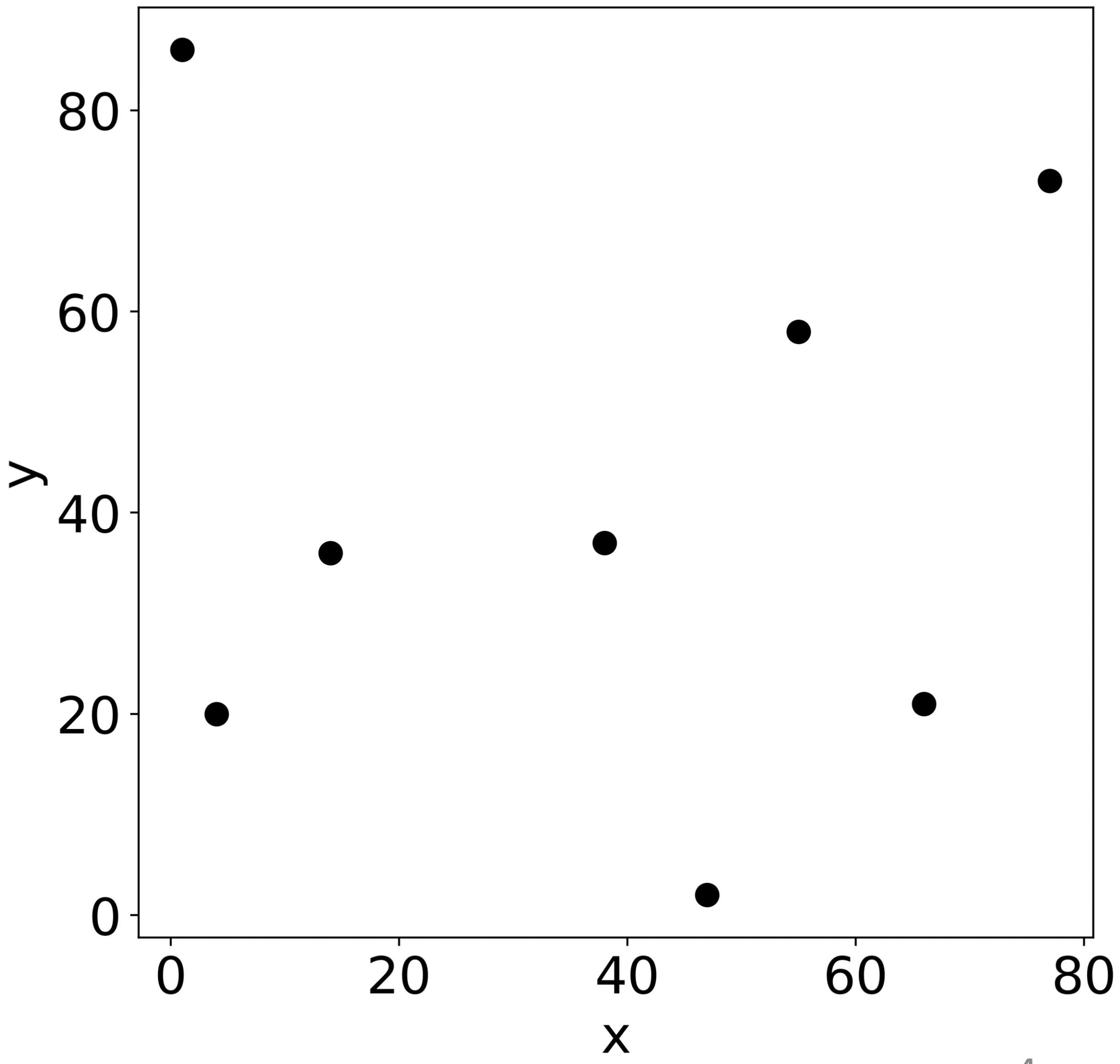
- Given a list of locations, what is the shortest round trip?
  - This is a discrete optimization problem

Outline

- Introduce the basic vocabulary and code
- Examine the simple (but slow) solution
- Look at two (faster) methods for approximating the solution
  - Simulated Annealing
  - Genetic Algorithms
- Brief look at other possible methods
- Applications
  - Scientific applications of the TSP
  - Scientific applications of these optimization methods

Vocabulary

- <u>Cities</u> – locations our traveling salesman needs to visit. Characterized by a location (x,y)

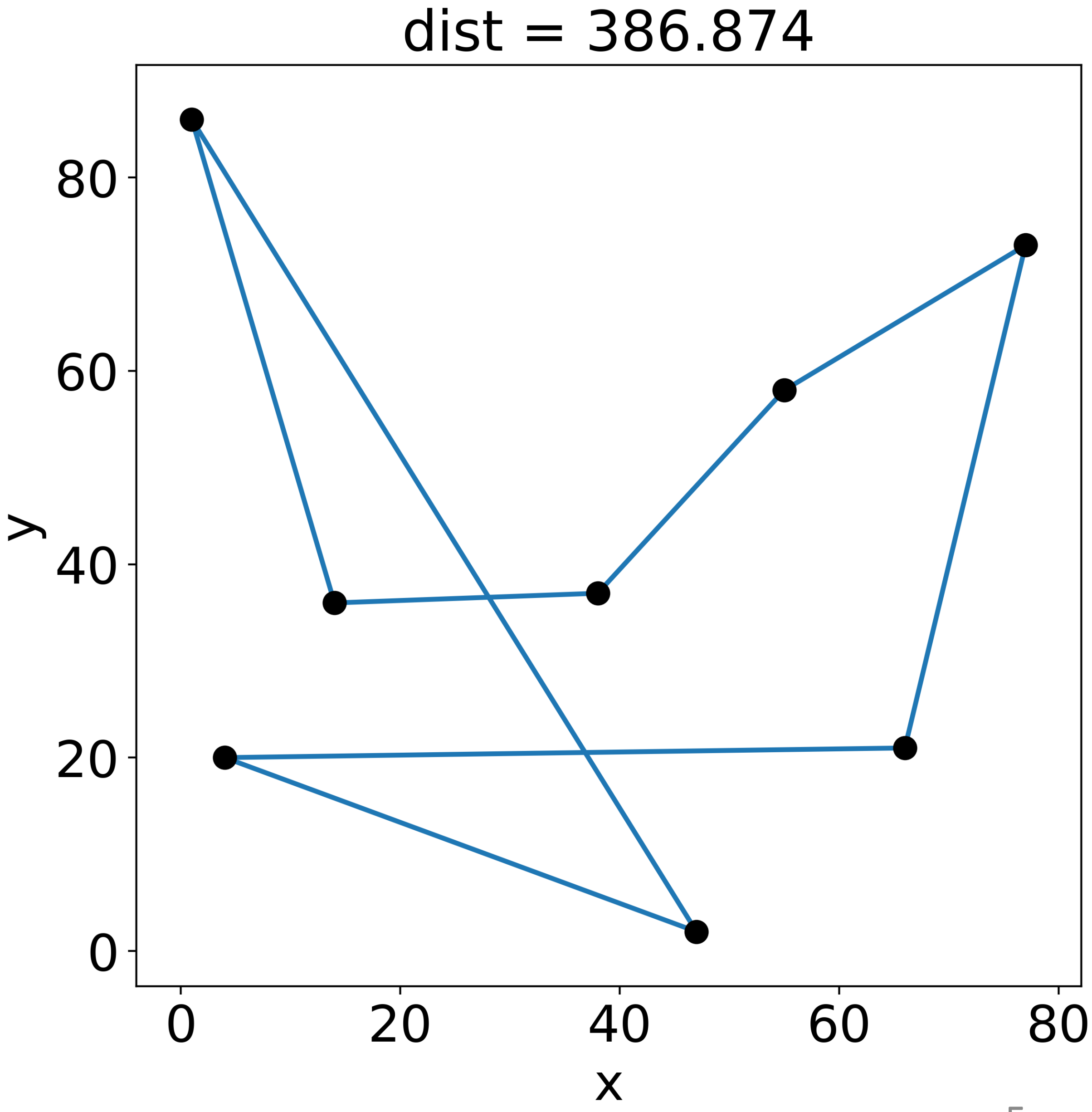


Vocabulary

- Cities – locations our traveling salesman needs to visit. Characterized by a location (x,y)

- Route – any round-trip path through a set of cities. Characterized by a ordered list:

  [city1, city7, city2, city3, …]



Translating the vocab into code: cities

```python
class City:
    """Class defines a City object, which has a location and a method
    to calculate distance to another city.
    """

    def __init__(self,x,y):
        # Location of the city
        self.x = x
        self.y = y

    def dist(self,city):
        # calculate distance to another city
        dx = abs(self.x - city.x)
        dy = abs(self.y - city.y)
        distance = np.sqrt(dx**2 + dy**2)
        return distance

    def __repr__(self):
        # what the city looks like when printed
        return "City(" + str(self.x) + "," + str(self.y) + ")"
```

Using the City class

```
city1 = City(3,2)
print("Our first city is",city1)
print("city1.x =",city1.x)
print("city1.y =",city1.y)

city2 = City(7,5)
print("Our second city is",city2)

print("The distance between them is",city1.dist(city2))
```

```
Our first city is City(3,2)
city1.x = 3
city1.y = 2
Our second city is City(7,5)
The distance between them is 5.0
```

Translating the vocab into code: cities

Now the x and y positions are treated as the real and imaginary parts of a complex number

```
class City(complex):
    """Class defines a City object, which inherits from complex.
    Cities have x coord (= real) and y coord (= imag). There is
    also a method which calculates the distance to another city.
    """

    @property
    def x(self):
        # define x coord
        return self.real

    @property
    def y(self):
        # define y coord
        return self.imag

    def dist(self,city):
        # calculate distance to another city
        distance = abs(self - city)
        return distance
```

Translating the vocab into code: routes

```
class Route(MutableSequence):

    def __init__(self,citylist):
        self.list = citylist

    def __len__(self):
        return len(self.list)

    def __getitem__(self,i):
        return self.list[i]

    def __delitem__(self,i):
        del self.list[i]

    def __setitem__(self,i,val):
        self.list[i] = val

    def insert(self,i,val):
        self.list.insert(i,val)

    def append(self,val):
        self.list.append(val)

    ...
```

Required code to make sure it behaves like a list

Translating the vocab into code: routes

```python
class Route(MutableSequence):
    ...

    def dist(self):
        # calculate route distance
        ncities = len(self.list)
        dist = 0
        for i in range(ncities):
            city1 = self.list[i-1]
            city2 = self.list[i]
            dist += city1.dist(city2)
        return dist

    def fitness(self):
        return 1/self.dist()

    def plot(self):
        # plot the route
        x = [city.x for city in self.list]
        y = [city.y for city in self.list]
        fig, ax = plt.subplots(1,1)
        ax.plot(x,y,c='C0')
        ax.plot([x[0],x[-1]],[y[0],y[-1]],c='C0')
        ax.scatter(x,y,c='k',zorder=10)
        ax.set_xlabel("x")
        ax.set_ylabel("y")
        ax.set_title("dist = {0:.3f}".format(self.dist()))
        return fig,ax
```

Method to calculate route distance

Method to plot the route

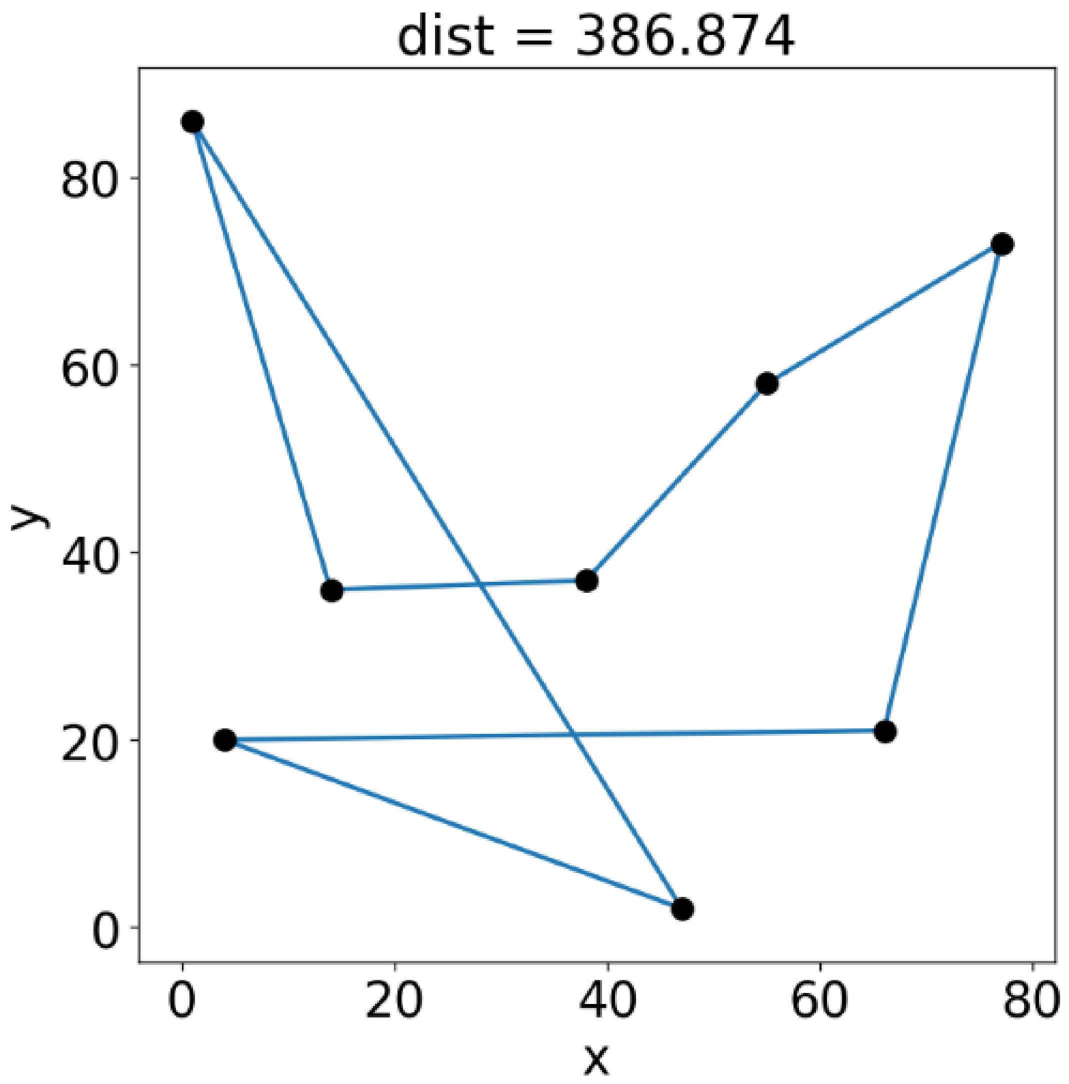
Using the Route class

```
# generate a random list of cities
cities = randomCities(8,seed=274)
print("Here is a random list of 8 cities:\n"+str(cities)+"\n")

# generate a random route through these cities
random.seed(274)
shuffled = random.sample(cities,len(cities))
route = Route(shuffled)
print("Here is a random route through these cities:\n"+str(route)+"\n")

print("The route distance is",route.dist())
```

```
Here is a random list of 8 cities:
[City(14,36), City(4,20), City(66,21), City(77,73), City(1,86), City(47,2), City(38,37), City(55,58)]

Here is a random route through these cities:
[City(66,21), City(77,73), City(55,58), City(38,37), City(14,36), City(1,86), City(47,2), City(4,20)]

The route distance is 386.87355608400077
```

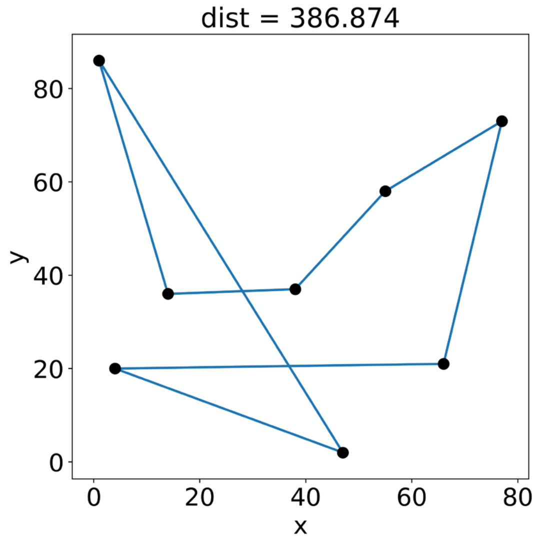
Using the Route class

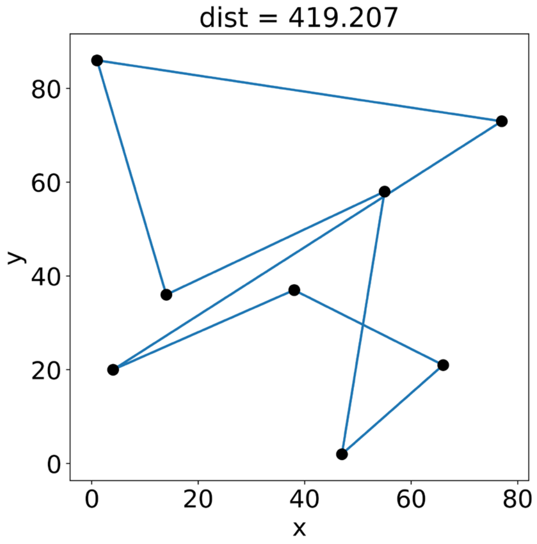
```
route.plot();
```

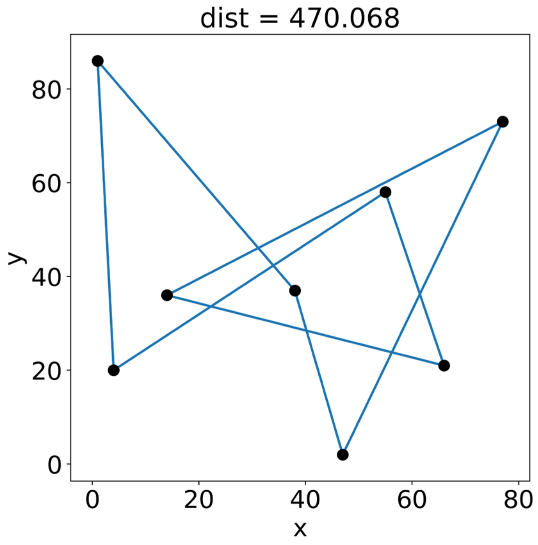


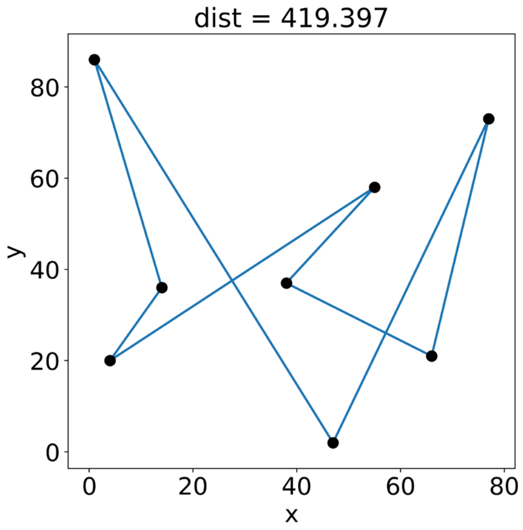
Solving the TSP

- Fortunately, the solution to the TSP is trivial!
  - Given a set of cities, there are only a finite number of possible routes
  - Check them all, and the shortest is the solution





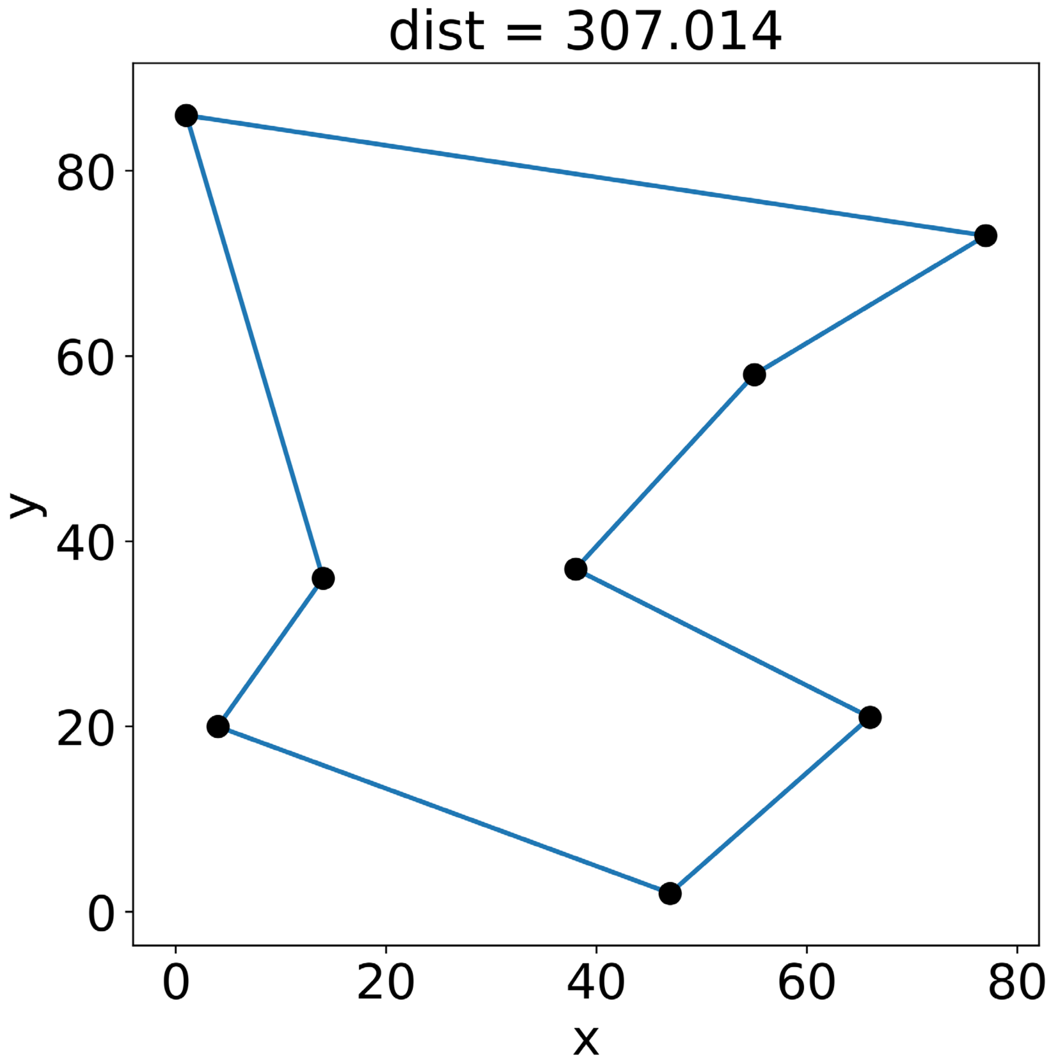




Solving the TSP

```
%%time
cities = randomCities(8,seed=274)
route = bestRoute(cities)
route.plot()
```

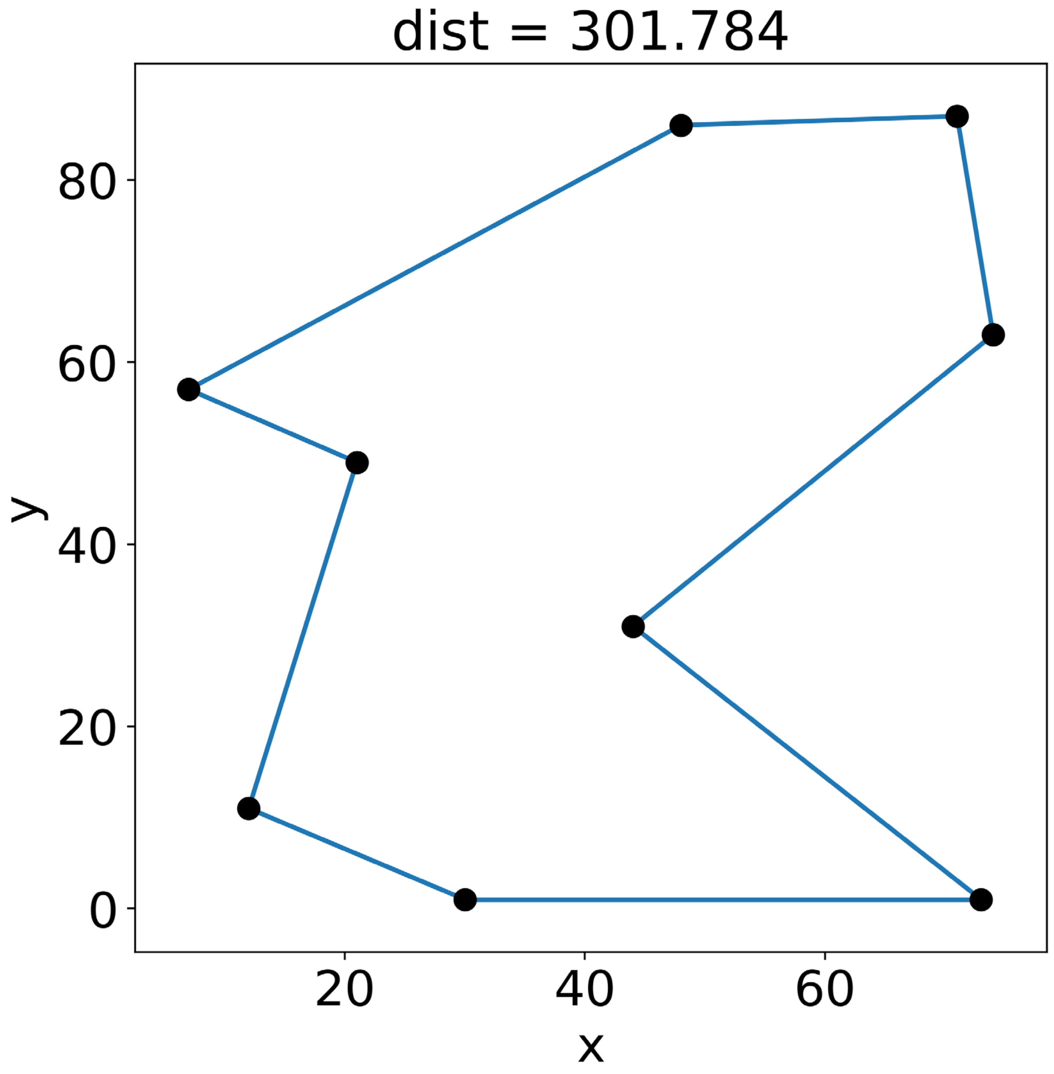
```
Wall time: 396 ms
```



Solving the TSP

```
%%time
cities = randomCities(9,seed=444)
route = bestRoute(cities)
route.plot()
```

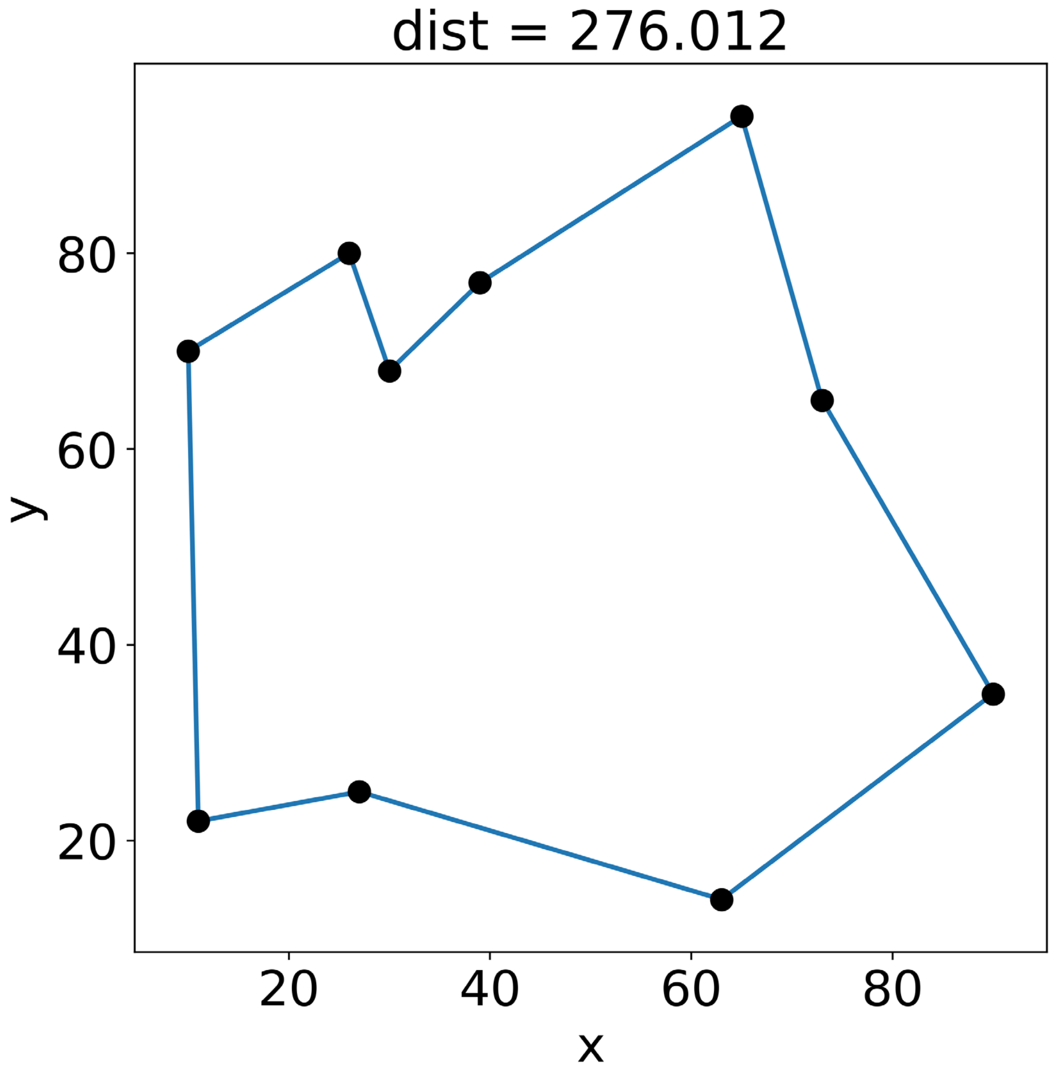
```
Wall time: 4.12 s
```



Solving the TSP

```
%%time
cities = randomCities(10,seed=14)
route = bestRoute(cities)
route.plot()
```

```
Wall time: 41.7 s
```

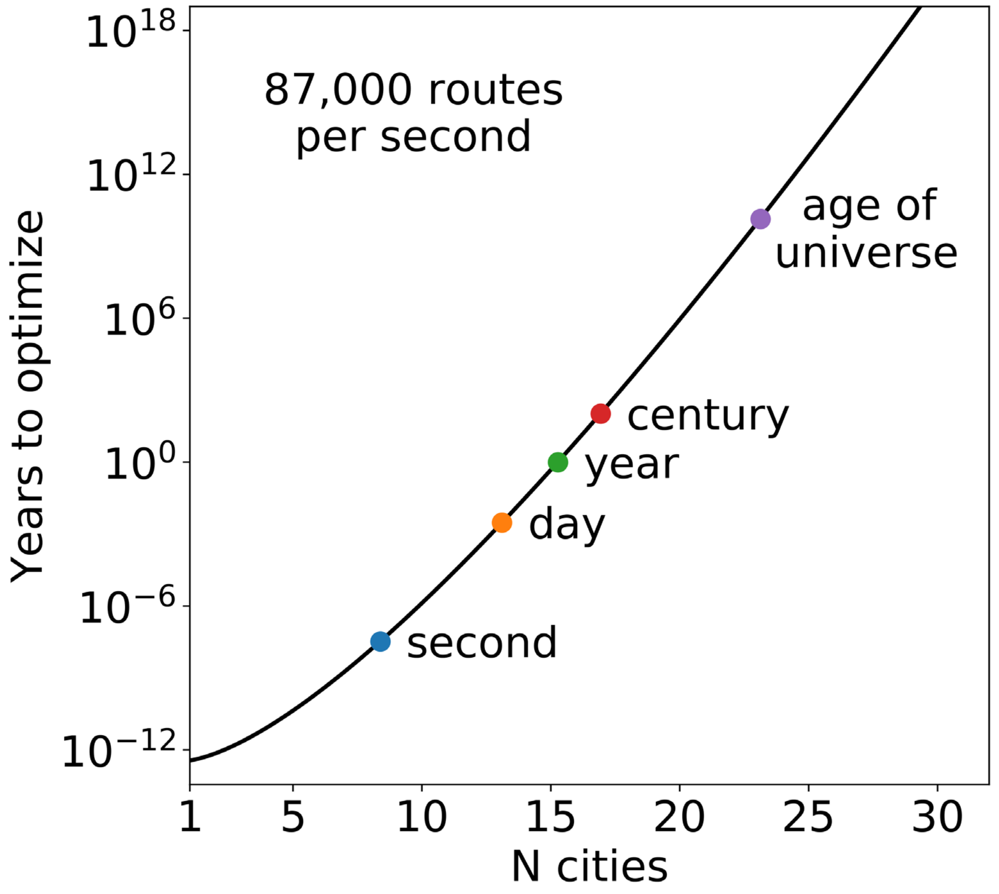


Exact solution is computationally expensive

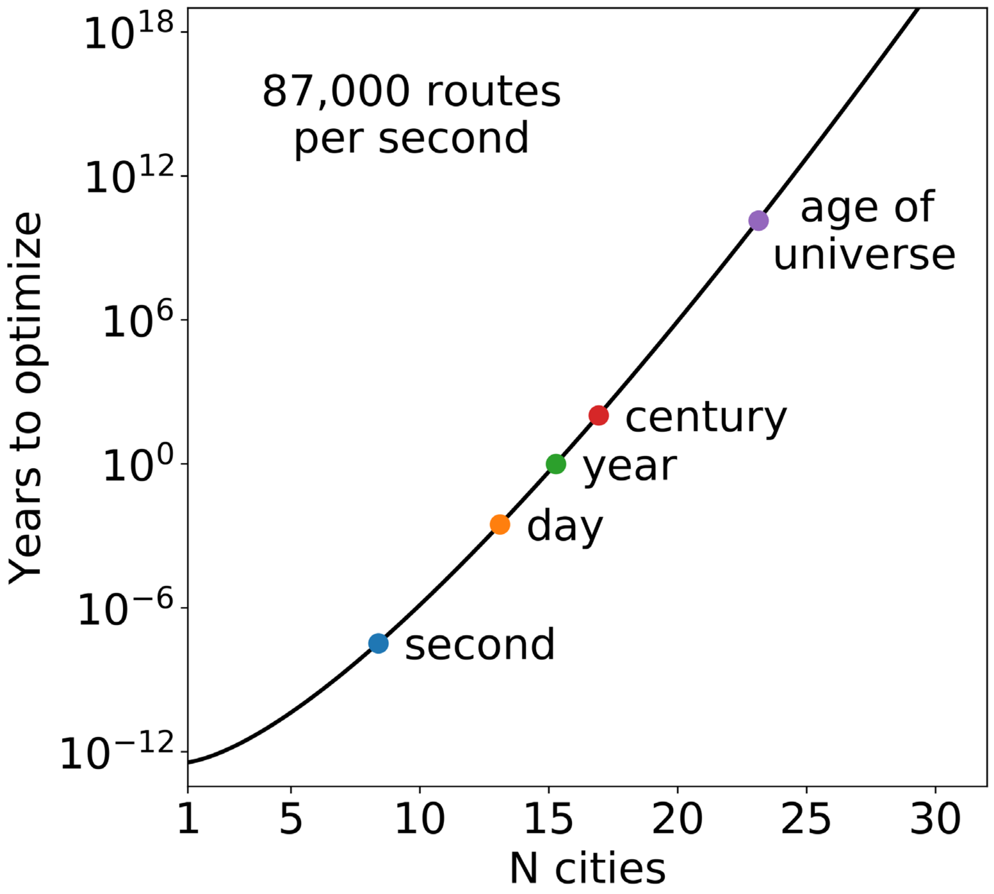
- To find the exact solution for N cities, you have to check N! routes
- Factorial grows quicker than exponential – it is going to get really expensive, really really fast
- We can estimate how quickly:
  - On the N = 10 example, my computer checked 10! routes in 41.7s

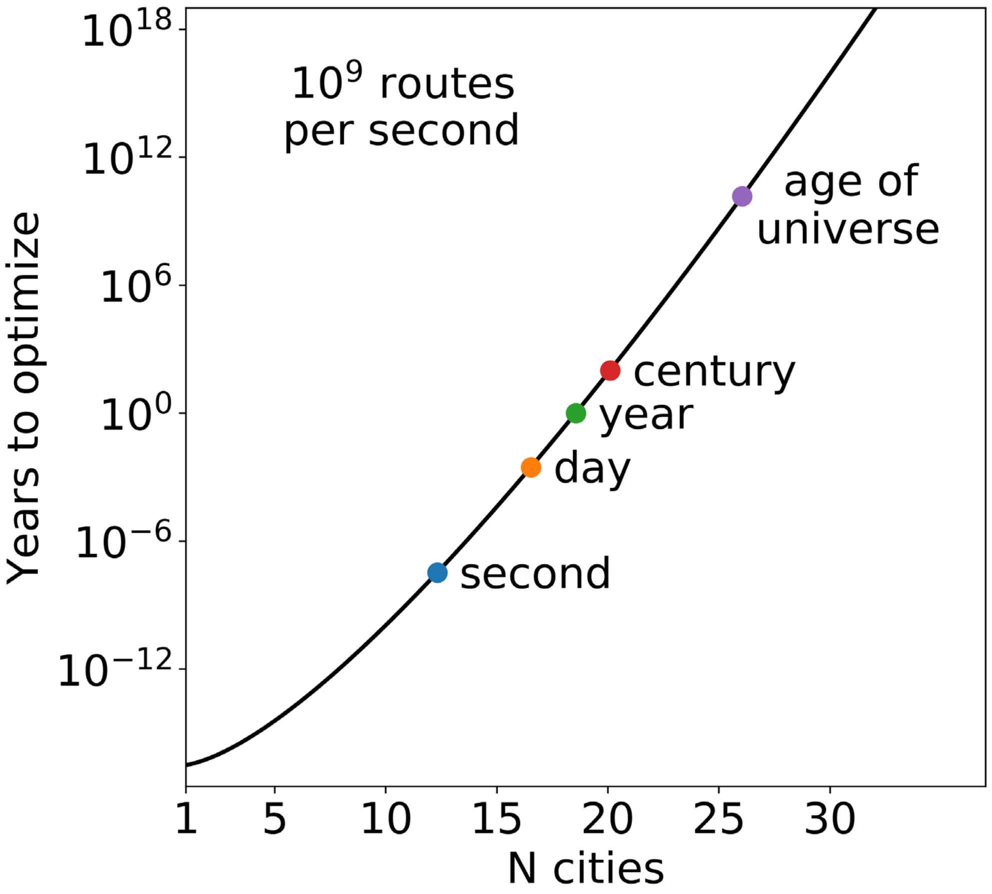
$$\frac{10!}{41.7\ s} = \frac{3628800}{41.7\ s} \approx 87{,}000\ s^{-1}$$

Exact solution is computationally expensive



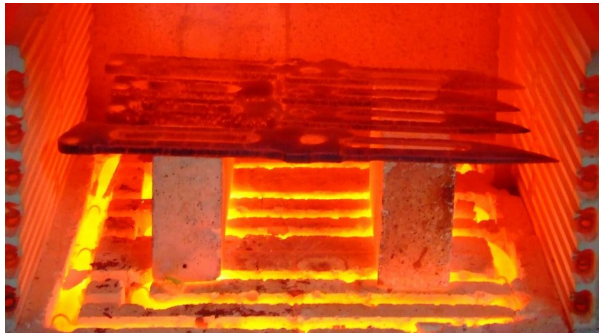
Exact solution is computationally expensive

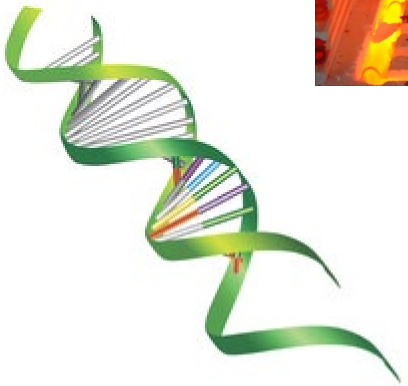




Better methods

- This method is infeasible for routes with more than a dozen cities
- Instead we can hope to quickly find *approximate solutions*
  - There are ***<u>many</u>*** ways to do this
  - We will focus on two
    - Simulated annealing
    - Genetic algorithms



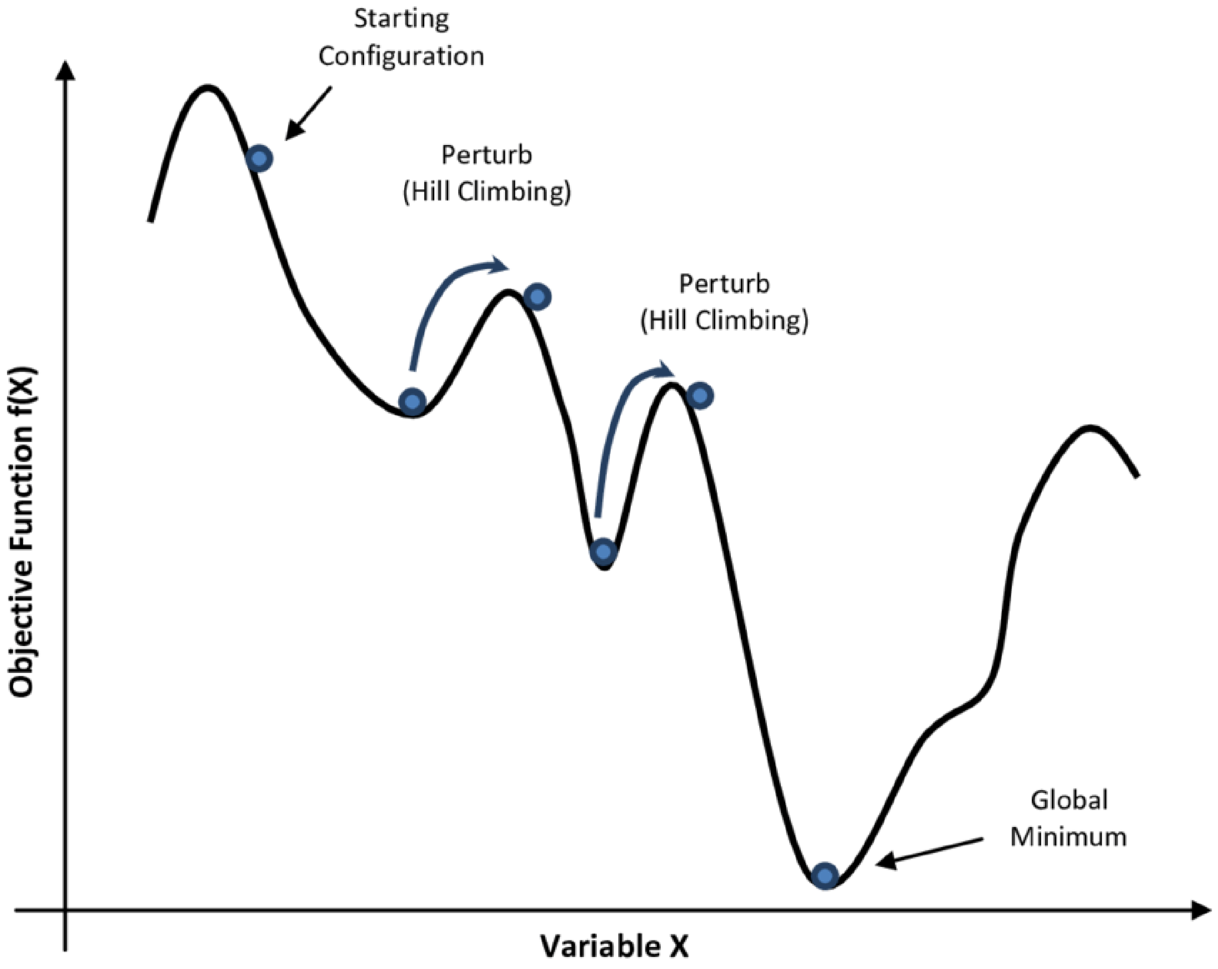


Simulated Annealing

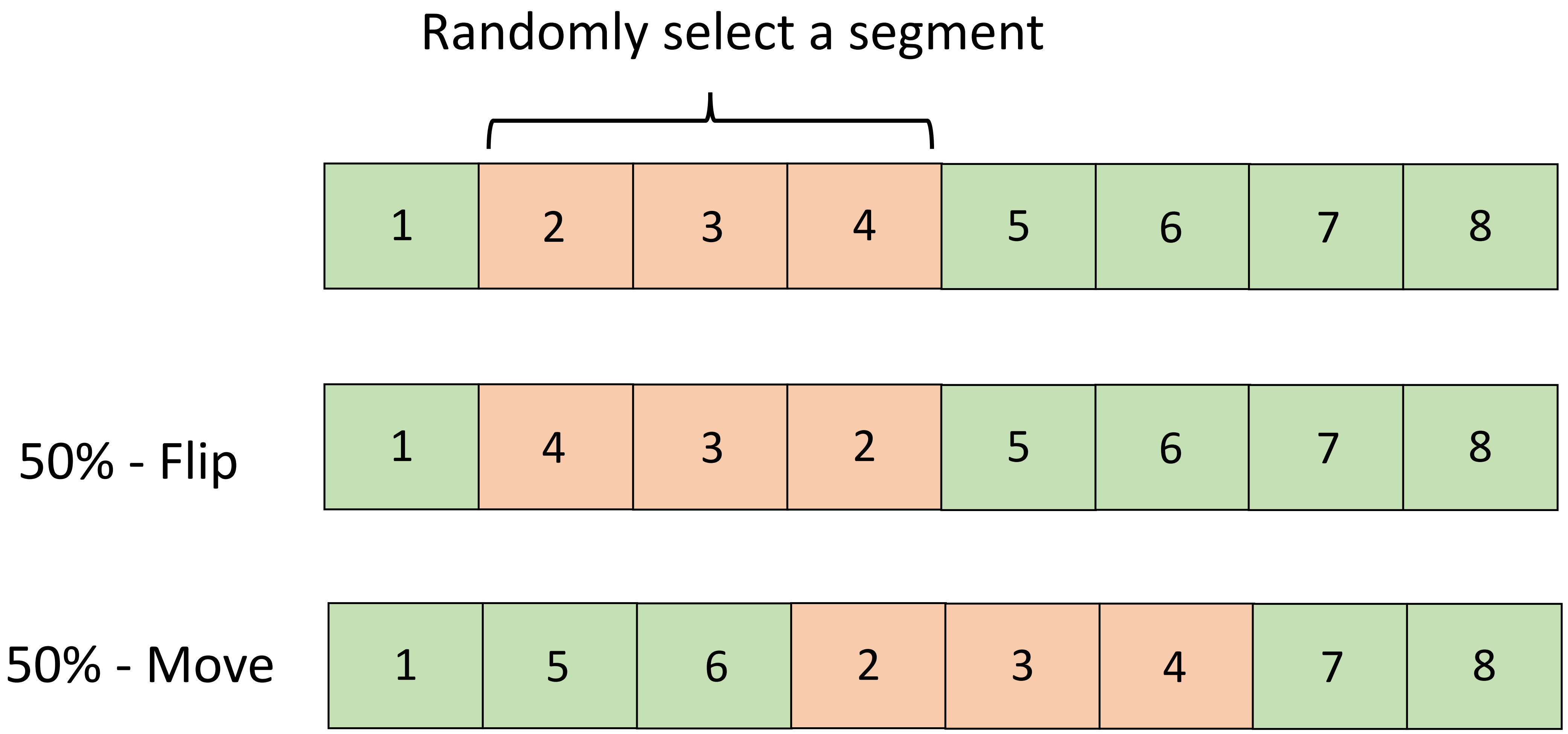
- Probabilistic technique
- Good for approximating global optima in a fixed amount of time
- Works by randomly "mutating" a solution
- The system has a "temperature" which determines the probability with which we accept worse solutions
- As the "temperature" is lowered, it is less likely that we accept worse solutions

Simulated Annealing

- Start with a random solution
- At each step, randomly “mutate” your current solution
- IF the mutated solution is better, keep it
- IF it’s worse, randomly decide with a probability that depends on the “temperature”
- Lower the temperature after each step



“Mutating” a route



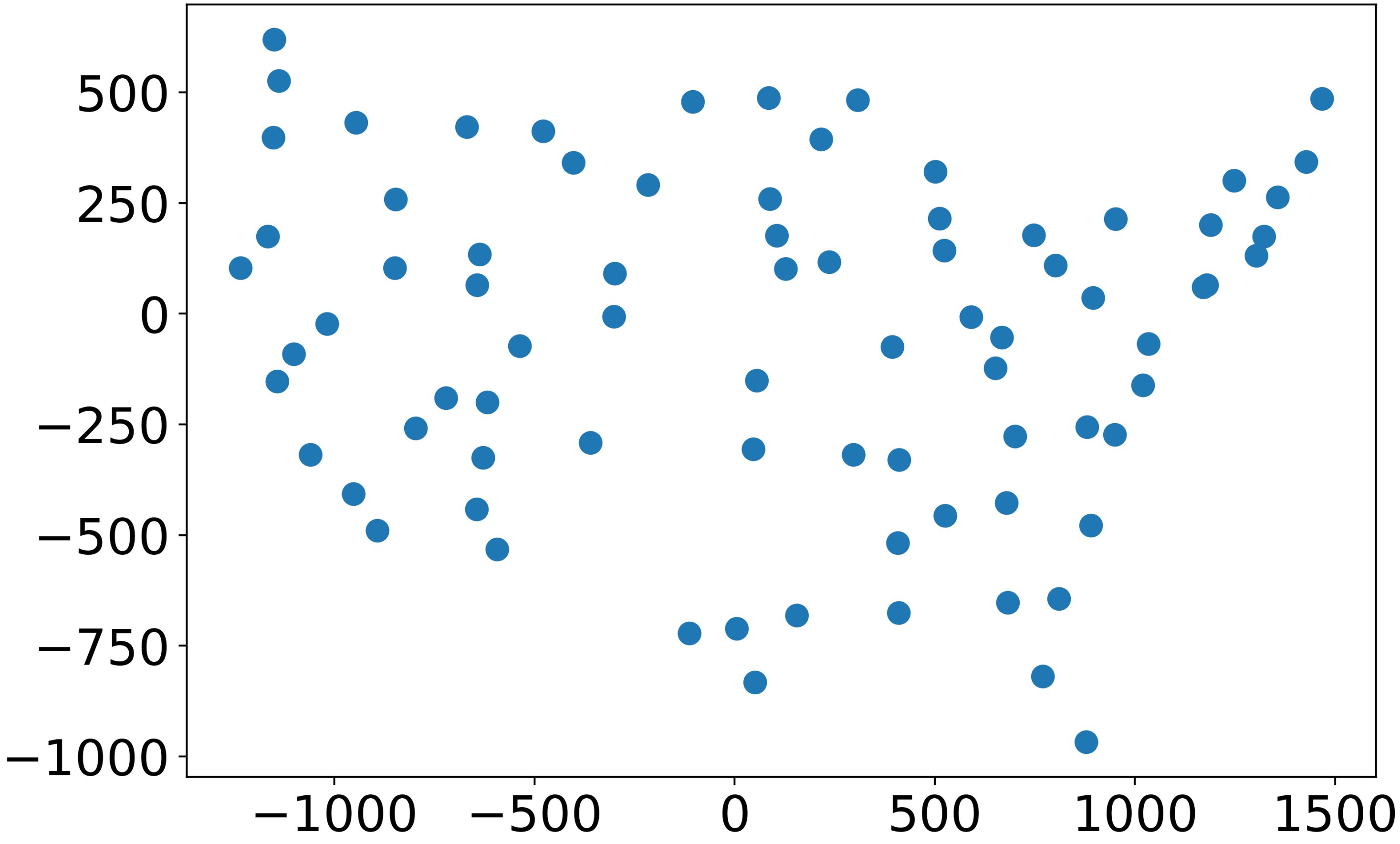
Simulated Annealing Code

```
# main loop
T = T0
while T > 1:

    # mutate until we find a suitable mutation
    prob = 0
    rand = random.random()
    for i in range(100*len(cities)):
        route2 = mutate(route)
        prob = np.exp((route.dist()-route2.dist())/T)
        if prob > rand:
            break

    if prob < rand: # if we never found one, stop annealing
        break
    else:  # else, save the data and continue

        route = route2 # update the route

        progress.append(route.dist()) # save route distance

        T *= 1-coolingRate # update the temp
```

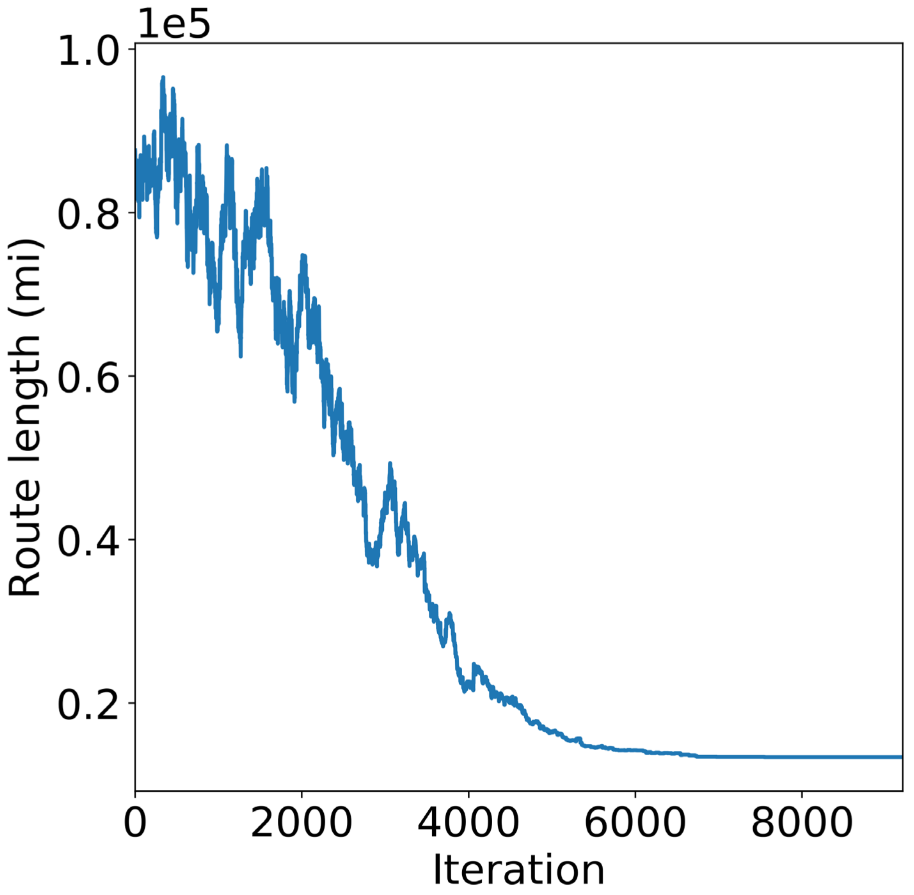
Initial temperature

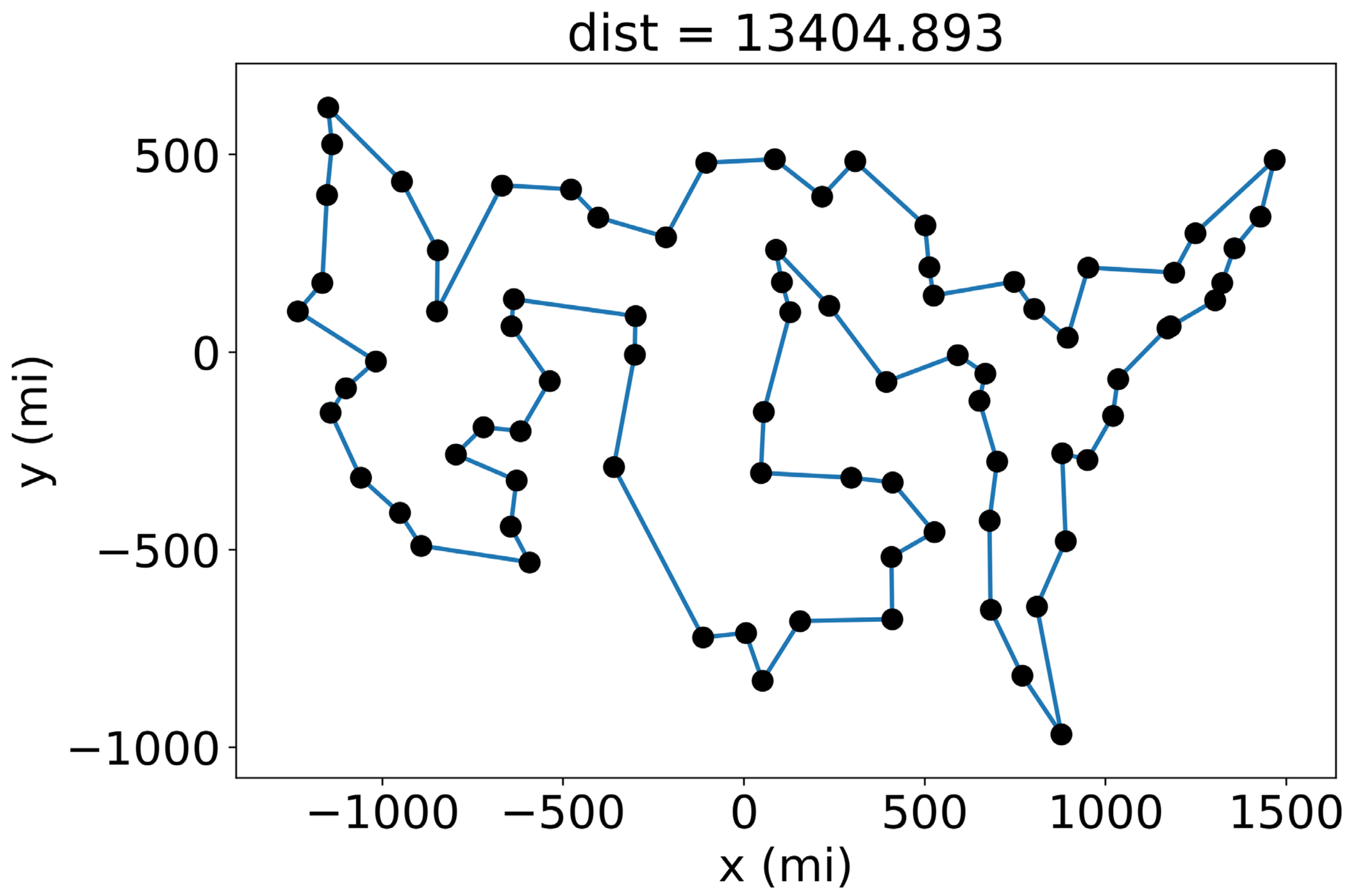
Cooling Rate

Test on 80 US cities



Simulated Annealing results





Genetic Algorithm

- Inspired by Natural Selection
- Uses mutation function from simulated annealing
- Also has crossover (i.e. “breeding”) and selection

New Terminology

- Population – a set of different routes for the same set of cities
- Breeding – a process through which two routes create a new route that has similarities to each

The process

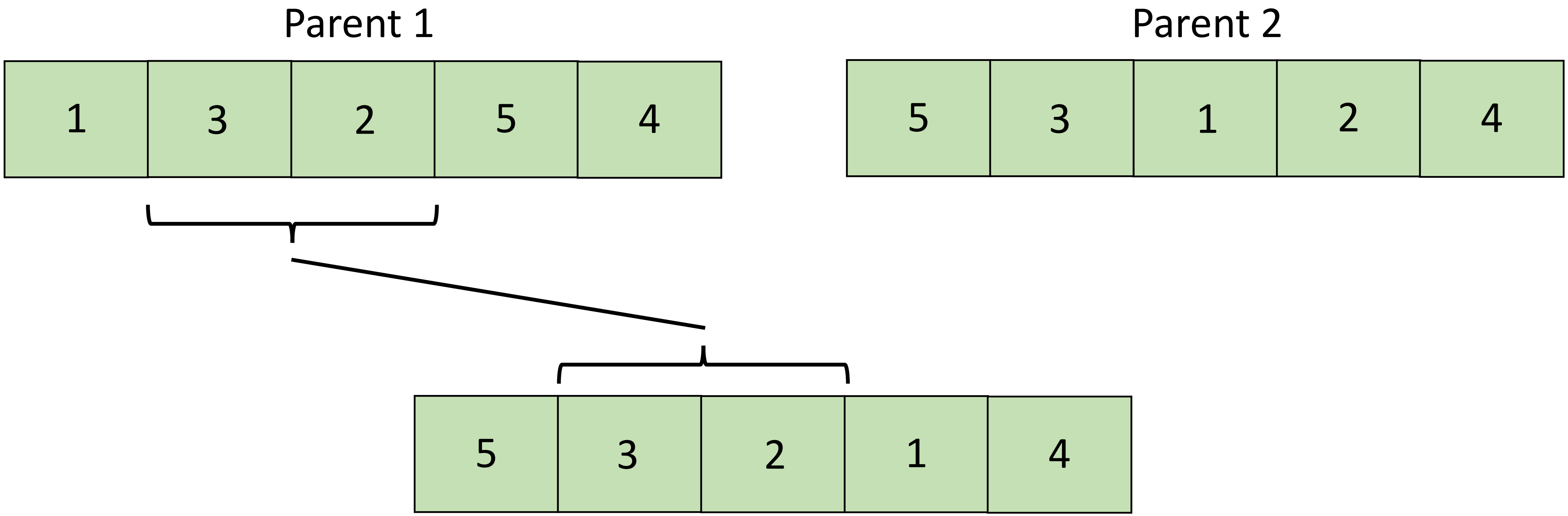
1. Create a population – take a set of cities, and make a bunch of different random routes through these cities
2. Select the mating pool – select the subset of the population that will be allowed to “breed” and make the next generation
3. Breeding – Take each pair of 2 routes, and make a new route that has similarities to each
4. Mutate – randomly alter some of the routes (using same method as mutation in simulated annealing)
5. Repeat for many generations

Select Mating Pool

- Need to select which routes get to reproduce to make new routes
- Elitism – top 20% automatically join the mating pool
- Rest are selected with probability: $\frac{\text{fitness}}{\text{total fitness}}$

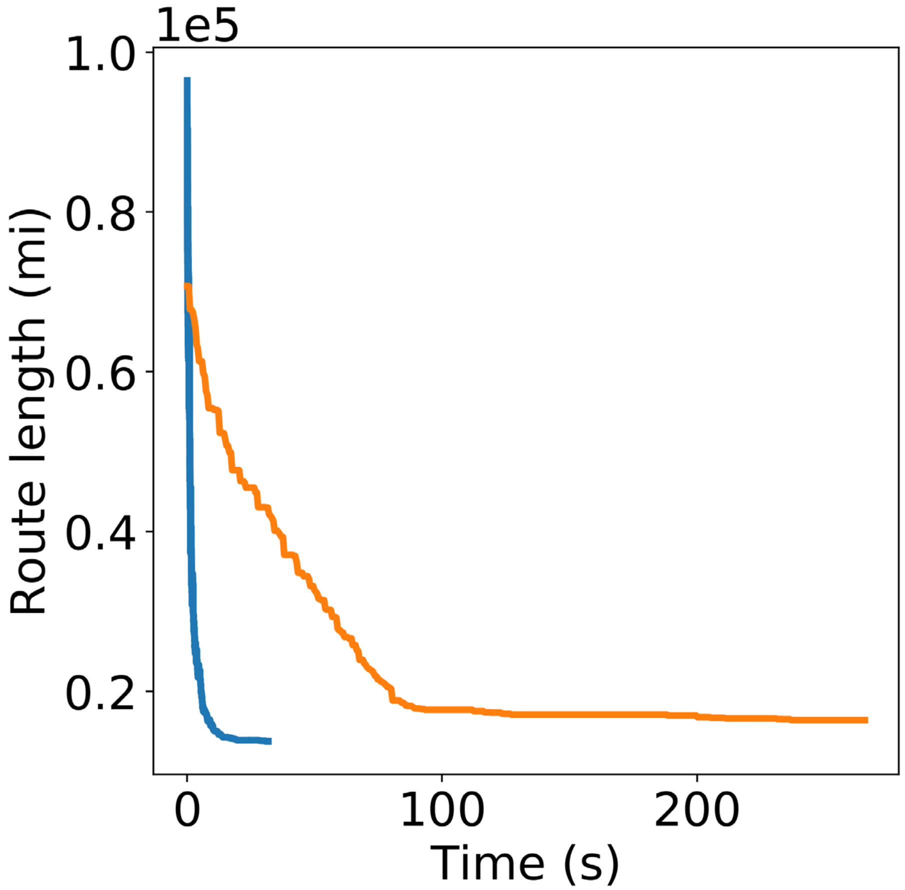
```python
# select mating pool
# first be elitist and let the top group in
matingpool = [ranked[i,0] for i in range(nElites)]
while len(matingpool) < len(ranked):
    rand = random.random()
    for j,cp in enumerate(cumProb):
        if rand <= cp:
            route = ranked[j,0]
            matingpool.append(route)
            break

return list(matingpool)
```

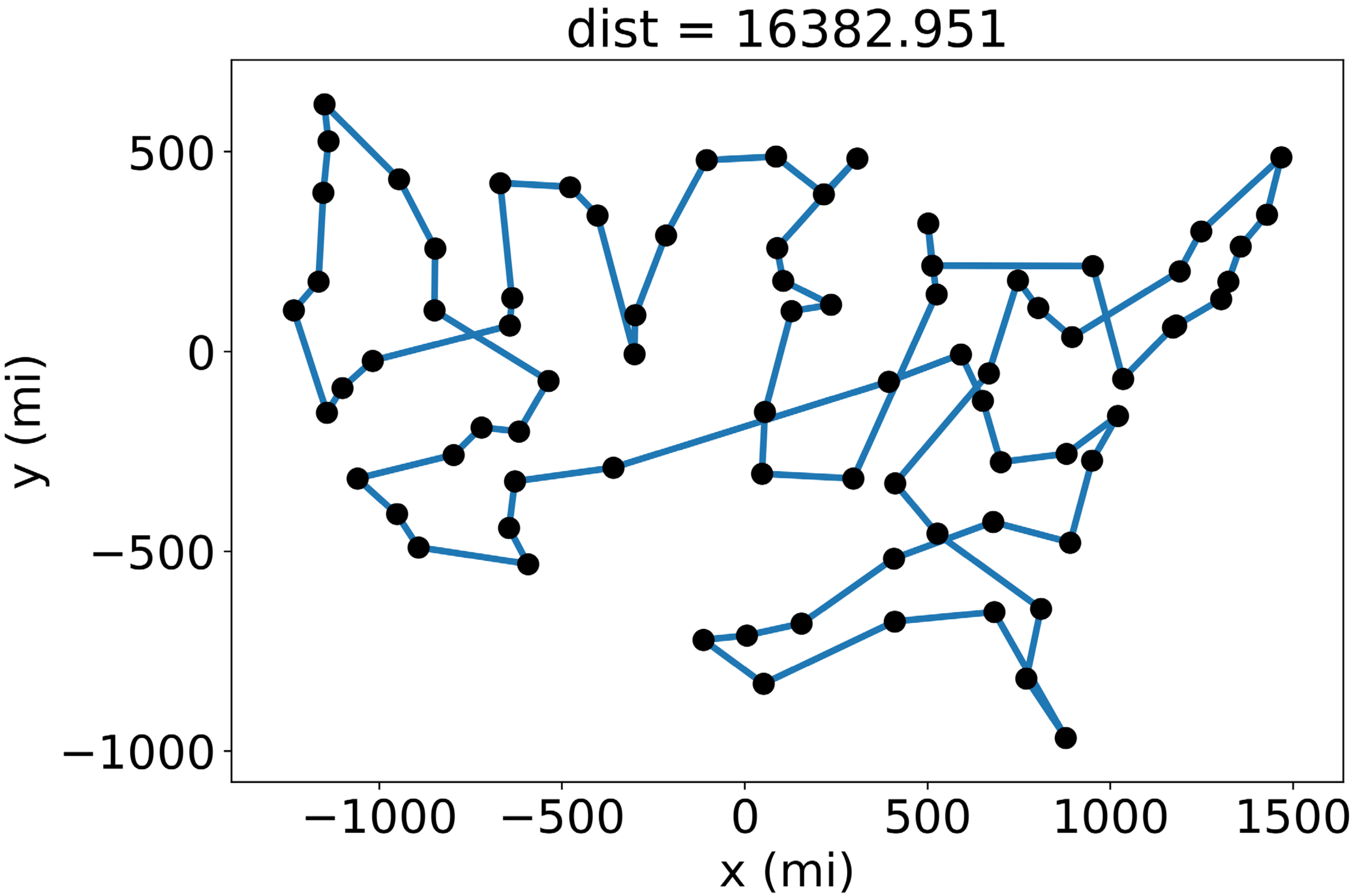
Breeding



```python
def geneticAlgorithm(cities, popSize, nElites, mutationRate,
                     generations):

    population = createPopulation(cities, popSize)

    route, dist = mostFit(population)
    progress = [dist]
    times = [0]

    start = time.time()
    for i in range(generations):
        matingpool = matingPool(population, nElites)
        children = breedPopulation(matingpool, nElites)
        population = mutatePopulation(children, mutationRate, nElites)
        route, dist = mostFit(population)
        progress.append(dist)
        now = time.time()
        times.append(now-start)

    return route, progress, times
```

Genetic algorithm results





Other possible methods

- Nearest neighbor, greedy algorithms, etc
- Ant colony optimization

Applications

- TSP – scheduling telescope observations, cabling, genetic engineering, etc
- Discreet optimization – scheduling jobs on a computer, evolved antennae

Conclusions

- The traveling salesman problem (and other problems in discrete optimization) is too computationally expensive to solve exactly
- There are often clever heuristics you can use to improve performance if you really understand the problem
- There are also probabilistic methods that aim to quickly *approximate* the optimal solution
  - These are typically good at avoiding local minima
- These methods have many applications in discrete optimization

Included in files

- TSPpres.pdf – this presentation in pdf format
- traveling_salesman.ipynb – python notebook with all of the code
- US_cities.txt – coordinates of 80 US cities used in tests

Sources

- https://en.wikipedia.org/wiki/Travelling_salesman_problem
- https://nbviewer.jupyter.org/url/norvig.com/ipython/TSP.ipynb
- https://en.wikipedia.org/wiki/Simulated_annealing
- https://www.fourmilab.ch/documents/travelling/anneal/
- http://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6
- https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35

Images

- https://xkcd.com/399/
- http://www.math.uwaterloo.ca/tsp/usa50/index.html
- https://www.machinemfg.com/annealing/
- https://esa.github.io/pagmo2/docs/cpp/algorithms/sga.html
- https://www.researchgate.net/figure/Simulated-Annealing-optimization-of-a-one-dimensional-objective-function_fig1_308786233
- https://commons.wikimedia.org/wiki/File:St_5-xband-antenna.jpg