Artificial intelligence techniques to solve problems in drones.

Supervised by:
Dr Haitham El-Wash

Computer Science Department
graduation project 2021

Graduation Project submitted to the faculty of computers & information in partial fulfillment of the requirements for the degree of Bachelor in Computer Science.

Project team:

Mohamed Ashraf Rady
Abdelrahman Hamza El deeb
Abdelrahman Samir Mohamed
Ahmed Tarek Helal
Abdelrahman Ibrahim Mohamed
Mahmoud Mohamed Abed

Table Of Contents

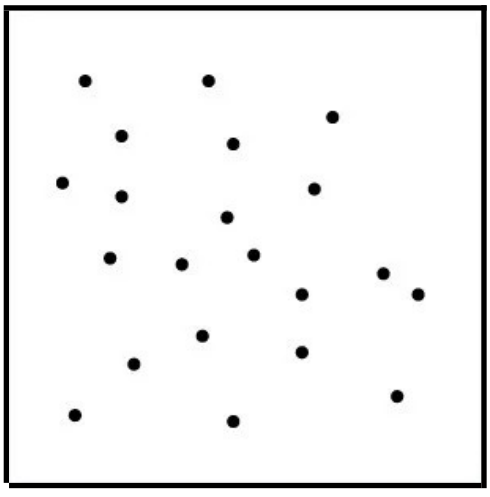
Chapter 3: Implementation.

Chapter 1
Introduction

1.1 Traveling salesman problem:

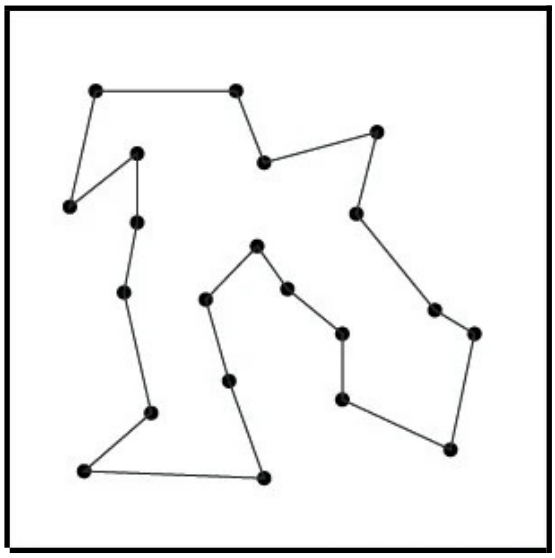
The traveling salesman problem (TSP) is one of the most studied problems in computational intelligence and operations research. Since its first formulation, a myriad of works has been published proposing different alternatives for its solution. Additionally, a plethora of advanced formulations have also been proposed by the related practitioners, trying to enhance the applicability of the basic TSP.

The Travelling Salesman Problem (TSP) is the challenge of finding the shortest yet most efficient route for a person to take given a list of specific destinations. It is a well-known algorithmic problem in the fields of computer science and operations research.There are obviously a lot of different routes to choose from, but finding the best one—the one that will require the least distance or cost—is what mathematicians and computer scientists have spent decades trying to solve for.



TSP has commanded so much attention because it’s so easy to describe yet so difficult to solve. In fact, TSP

belongs to the class of combinatorial optimization problems known as NP-complete. This means that TSP is classified as NP-hard because it has no “quick” solution and the complexity of calculating the best route will increase when you add more destinations to the problem.



The problem can be solved by analyzing every round-trip route to determine the shortest one. However, as the number of destinations increases, the corresponding number of roundtrips surpasses the capabilities of even the fastest computers. With 10 destinations, there can be more than 300,000 roundtrip permutations and combinations. With 15 destinations, the number of possible routes could exceed 87 billion.

1.2 Popular Travelling Salesman Problem Solutions:

Here are some of the most popular solutions to the Traveling Salesman Problem:

1.2.1 The Brute-Force Approach:

The Brute Force approach, also known as the Naive Approach, calculates and compares all possible permutations of routes or paths to determine the shortest unique solution. To solve the TSP using the Brute-Force approach, you must calculate the total number of routes and then draw and list all the possible routes. Calculate the distance of each route and then choose the shortest one—this is the optimal solution.

1.2.2 The Branch and Bound Method:

This method breaks a problem to be solved into several sub-problems. It’s a system for solving a series of sub-problems, each of which may have several possible solutions and where the solution selected for one problem may have an effect on the possible solutions of subsequent sub-problems. To solve the TSP using the Branch and Bound method, you must choose a start node and then set it bound to a very large value (let’s say infinity). Select the cheapest arch between the unvisited and current node and then add the distance to the current distance. Repeat the process while the current

distance is less than the bound. If the current distance is less than the bound, you’re done. You may now add up the distance so that the bound will be equal to the current distance. Repeat this process until all the arcs have been covered.

1.2.3 The Nearest Neighbor Method:

This is perhaps the simplest TSP heuristic. The key to this method is to always visit the nearest destination and then go back to the first city when all other cities are visited. To solve the TSP using this method, choose a random city and then look for the closest unvisited city and go there. Once you have visited all cities, you must return to the first city.

1.3 Vehicle Routing Problem with Drones:

Drones have a distinct advantage in delivering high-priority packages due to their light weight, high travel speeds, and limited travel restrictions. In early 2016, the Workhorse Group Inc. successfully utilized the HorseFly drone to deliver from a Workhorse truck simultaneously while the truck completed its normal delivery route [7]. In 2015, Murray and Chu [6] introduced a new type of TSP problem called "Flying Sidekick Traveling Salesman Problem'' (FSTSP). They proposed the idea of having a drone move along with the truck and fly from the truck to make a delivery while the truck continues to serve its

customers in different locations. Once the drone completes its service for one customer, it needs to fly back to the truck at the current delivery location or along the route to the next delivery location. As such, the FSTSP is a variant of the Vehicle Routing Problem (VRP) with synchronization constraints [4]. The objective, in the FSTSP, is to develop a route such that the time to complete the deliveries is minimized for the truck and drone. A greedy heuristic procedure was proposed to solve the problem [6]. In addition to FSTSP, there are other emerging truck-drone routing problems in the literature. Agatz et al. [1] defined a problem called the TSP-D, in which trucks and drones are assumed to be on the road network. The model identifies the drone service to deliver a package from the departure node to the customer, and back to the arrival node as an operation . TSP-D is different from FSTSP in the sense that it allows the drone to be launched and landed at the same node. Recently, Kitjacharoenchai et al. [5] proposed the Multiple Traveling Salesman Problem with Drones (mTSPD) which has the same operation like FSTSP but utilizing multiple trucks and drones and allowing drones to be retrieved by any truck that is nearby and not necessarily the same truck that it is launched from.

The Vehicle Routing Problem (VRP) and its variants are well-studied problems in Operations Research. They are related to many
real-world applications. Recently, several companies like Amazon, UPS,
and Deutsche Post AG showed interest in the integration of autonomous
drones in delivery of parcels. This motivates researchers to extend the

classical VRP to the Vehicle Routing Problem with Drones (VRPD),
where a drone works in tandem with a vehicle to reduce delivery times.

1.4 Drones in general:

Construction of the drone, which the elements are frame, propellers, engine, system of power, the electronic control and communication system. A drone is powered by batteries, which is the major drawback, because it is exhausted quickly, causing a decreased drone on the ground. The lithium-polymer batteries are used for powering the drones. Civil drones are driven by electric motors. Next there were shown the possibilities of using drones. They can be used in industry, for taking photos and filming, in delivering shipments. The main danger of using drones is the fall of a drone from a great height, which may be due to discharge of the battery, damage caused by weather conditions (low air temperature, precipitation), hitting an obstacle (tree, building). Currently a lot of projects related to the development of power for drones are conducted like batteries of graphene, pure lithium anodes, and fuel cells. A very important risk associated with the extensive use of civilian drones is related to privacy and the rights of citizens.

Drones or Unmanned Aerial Systems (UAV - Unmanned Aerial Vehicle or UAS - Unmanned Aerial Systems) are the aircrafts, which are able to fly without a pilot and passengers on board. Drone Controlling is performed remotely by radio waves or autonomously (with a predetermined route). Drones do not have a specific size or type of a drive. They are often equipped with

accessories used for surveillance and monitoring, in the form of the optoelectronic heads. A significant advantage is the extremely short reaction time when it comes to commissioning and preparing the unit for a flight.

The first countries that started research on UAVs were the United States, the United Kingdom, Russia, and Germany. The first time an unmanned flying vehicle was used by the Austrians in August of 1849.

One of the first creators of drones was Charles Kettering, who in collaboration with Elmer Sperrym, Orville Wright and Robert Milikanem created in 1915, the aircraft named “Kettering Bug”. It was a primitive automatic plane, which on the basis of sensors defined its height, the distance traveled and the position. In contrast, the first civilian aircraft was produced only in the 80s of the twentieth century in Japan. Public drones differ from the military in the size and the drive. They are smaller and they are driven by an electric motor (military are driven by an internal combustion engine). They are mainly used for photographing, filming, and in delivering shipments.

1.5 Drone construction:

Drone is composed of two major systems:

- Movement system
- Control system.

1.5.1 Movement system:

The basic element of a drone is a frame, which should be maximum light. The classification of frame construction is mainly based on the number of arms. Due to the number

of arms and the motors used the drones can be divided into:

1. Bicopters – two engines,

2. Tricopters – three engines,

3. Quadcopters – four engines,

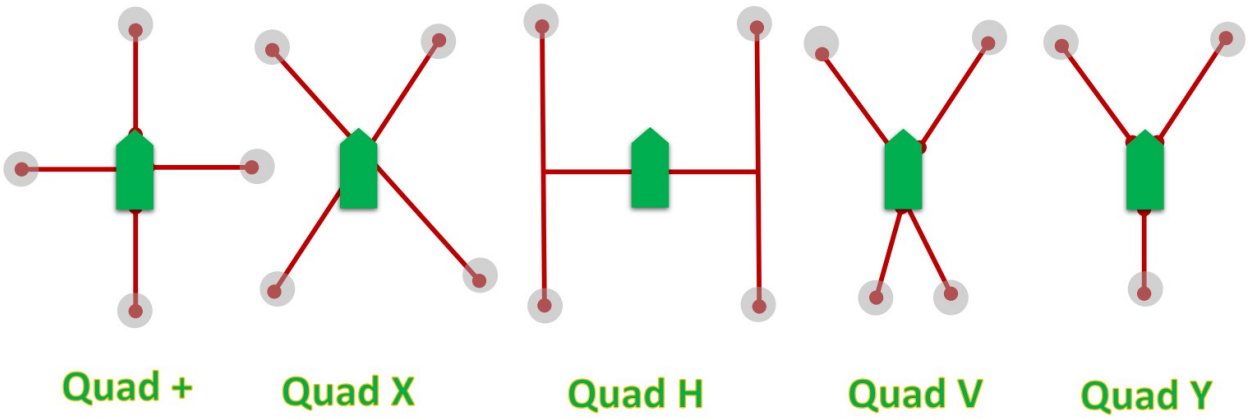
4. Hexacopters – six engines,

5. Octocopters – eight engines.

The frame is made of carbon cloth 3K. Propellers and engine: The next components of a drone are engine and propellers. They constitute the main propulsion system of a drone and are subjected to the highest loads, therefore their durability is very important. The propellers change a torque for a work used for lifting the vehicle in the air. Due to the propeller system in relation to the flight direction it can be divided into the following types:

- \+ – One is the leading propeller (at least four propellers),
- X – The most common construction, in which two propellers are leading (with an even number of propellers),
- H – A very rare arrangement where the construction is based on the H-shaped with two propellers leading.
- V – Very rare arrangement in which two propellers lead onto outstretched arms,

- Y – Three arms stacked in the Y, where one or two arms can be leading,



In each of the construction can be mounted double propellers (at the top and in the bottom), which significantly increases the strength of the drone, and does not require the addition of another arm. Double propellers mounted on a smaller number of arms increases the strength of a drone allowing more lift capacity. Thus, the material costs are falling and the drone can carry a heavier load.

The wings are made of carbon fiber, plastic or aluminum, and are attached to each other by lamination (also used for attaching the drone extremities), which ensures performance between the weight of the entire construction and mechanical durability. Because the engine and propellers must be replaced as their consumption, the periodic preventive inspections are carried out. The size of the wings is very important. The larger is the diameter; the lower is the speed, which contributes to a reduction of drone volatility. The larger the wing blades, the greater aerodynamic lift generated, also the pressure exerted on the propeller hub also

increases and the forces deforming propellers are getting bigger. The bigger are the propeller blades the stronger must also be the engine to cope torque, which is required to propel propellers, into motion. In addition, it is important to balance each propeller before use to minimize vibrations generated by the unequal operation of the system. It is very important to choose the engine and propellers in such a way that drones could for as long as possible lift a given load.

The power of a drone: The residence time of the object flying in the air depends both on the type of drive and the type of power supply. A drone is powered by batteries, which is the major drawback, because it is exhausted quickly, causing a decreased drone on the ground. In general, batteries are the sets of two or more voltaic cells of the same type, providing a current that is stronger than a single cell. These can be divided into disposable batteries and electric accumulators that can be unloaded and loaded many times. In the batteries and accumulators complex chemical reactions occur in which, depending on the type of battery, many chemical elements. As a result of chemical reactions the chemical energy contained in their active substances is converted into electrical energy. Batteries are defined as chemical current sources. The set of active substances and the electrolyte is the basis of chemical current sources action. This set functions in the form of a cell containing positive and negative electrodes and an electrolyte in an individual sealed enclosure in the batteries and accumulators.

The cells are a source of direct current and in depending on the type of chemical reaction can be divided into:

- Primary cells in which electricity generation followed by an irreversible chemical reaction.
- They are not designed to be charged by any other electricity source.
- Secondary cells in which electricity generation occurs by reversible chemical reaction and are designed to be charged by other electricity sources. The battery is a source of electrical energy generated by direct transformation of chemical energy, which consists of one or more primary cells which are not reusable, including the housing, the ends and marking. Accumulator (electric) is a chemical power source allowing the multiple storage and release of electricity as a result of reversible transformations of energy. This is a source of electrical energy generated by direct transformation of chemical energy, which consists of one or more secondary reusable materials.

1.5.2 Control system:

The control system is responsible for the drone flying up, down, rotating, for his reaction to the emerging forces and for stability. Most of the control systems are equipped with the same set of sensors with the difference in the speed of calculations and in algorithms used. The control system consists of:

1. Flight controller, responsible for machine control capabilities,

2. Electronic Speed Control (ESC) –the unit responsible for engine rpm,

3. Supplying plate, separating the power supply for regulators turnovers and motors,

4. Sim module, which allows the transmission of telemetry data,

5. Proximity camera - an element of anti-collision system,

6. The numeric keypad to enter the customer PIN codes.

Controllers engines are used to ensure maximum performance and the highest level of fail safety. Controls should be selected, so that their parameters correspond to the maximum current consumption of the motor.Some controllers have additional exit type BEC (Battery Eliminator Circuits), thus it is possible to supply the control system with the voltage of 5V and efficiency of 2A. This provides good working conditions for the control system. Limited is also the complexity of the control board. The controller also controls the condition of the battery. When the battery voltage drops to low levels, it forces a reduction in engine RPM, preventing the damage of the battery. It provides additional security when the control unit will not work. Programming the controllers is carried out using a programming card, thanks to which it is possible to change the parameters of the devices - the voltage level of a power cut, and how to start up the engine. During the designing process of the control system must first pay attention to the correct power to the controller and establish communication by the programmer. In a further step it is needed to include the

necessary filtration power (capacitors and choke) and protection of the pin analog-to-digital converter. The digital transducer serves as a voltage meter and its conversion into digital form, understandable for the controller. By measuring the voltage at every cell we can gain information about the battery charge. In study for controlling the motor rotary speed a mechanism was used PWM (Pulse Width Modulation) or pulse width modulation. It is possible using this method to obtain different average voltages. At this project we use Bluetooth communication between the controller and smartphone with Android operating system. Android is now the most popular operating system in the world for mobile devices. It is distinguished by openness, small hardware requirements, simple configuration and easy transfer between different mobile devices. This has contributed to the preparation of an application that allows controlling the drone from the level of a smartphone. This allows for data transmission up to 100 m. The gyroscope allows tracking the flight of the object, and accelerometer allows the drift of the module and determines the absolute point of reference. Drone control is done by varying the speed of the respective motors. The control algorithm is necessary to stabilize the machine in air. Civilian drones on selected examples.

Civil drones: One example is used for photography, video filming, and delivering shipments. The mass of the airplane battery is 1160 g. A lithium-polymer cell with a capacity of 5200 mAh for driving the four rotors allows for 25 minutes of continuous flight with the recording. The control is performed over the air waves with a frequency of 5.8 GHz using the remote control. The effective control

range is 300 m, and using signal amplifiers is even 1000 m. Thanks to the Wi-Fi module the synchronization of the device with a phone or tablet is possible, which increases the possibility of modifying settings for drone in flight, such as the size or resolution of the recording multimedia, information about the status of the machine (battery status, connection to GPS altitude speed). It is also possible to preview the camera view “live”, recording and downloading videos and photos during the flight. The GPS receiver with software provides a standalone return to the starting point in case of losing the connection with the controller. Drone also recognizes areas where flights are prohibited (proximity to the airport) and informs the controller about it. The heart of the drone is a camera with photo resolution of 14 Megapixels, filming of 1080p, diagonal of the matrix 1/2.3”and the field of view of 110°/ 85°. Photos are saved in formats .JPEG and .RAW, which facilitates their interpretation. After equipping the drone with a camera of different type and the appropriate software, the camera can be used to map areas difficult to access.

1.6 Possibilities of using the drones:

Unmanned units are the ideal devices to patrol large areas, so they can be used to protect property and the protection of state borders. They can also perform aerial photographs used for geodesy, archaeological, advertising purposes etc. With its small dimensions and high maneuverability they can operate the flights between obstacles, buildings, and even are able to fly to rooms, through the open gates, windows and doors. Models equipped with thermal and night vision cameras

(using the infrared active or reinforcing starlight) can be used as prospecting machines in rescue operations, with a daily patrolling of the chosen area and can operate round the clock above the woody areas. They transmit an image in real time allowing an immediate reaction of relevant services in case of emergency, an accident or a crisis situation requiring intervention. They can be used by the following services, industry and companies:

- Advertising Businesses:
  - ➔ Drones have also a bigger use in delivering shipments.
  - ➔ Promotional materials.
- Police:
  - ➔ Communication disasters service.
  - ➔ Patrolling a designated area.
  - ➔ Traffic congestion documentation and traffic jams.
  - ➔ Operation and monitoring of mass events.
- Army:
  - ➔ Reconnaissance and surveillance area.
  - ➔ Direct support for fighting and training tasks.
  - ➔ Conducting the shares of intelligence.
  - ➔ Tracking a moving target.
  - ➔ The fight against terrorism.
- Geodesy companies:
  - ➔ Fast visualization and control of area,
  - ➔ Mapping.

1.7 Risks associated with the use of drones:

The use of drones on a large scale entails a high risk. The main danger is the fall of a drone from a great height, which may be due to:

- ➔ Discharge of the battery,
- ➔ Damage caused by weather conditions (low air temperature, precipitation),
- ➔ Hitting an obstacle (tree, building, high-voltage line).

These risks can be predicted; therefore the action should be taken to prevent their uprising. The battery status and other telemetry data, including temperature can be controlled remotely by the system.

In case of exceeding one of the parameters the alarm should be launched. This will allow action, such as emergency recall of the drone to a branch. However, the sensors and software that based on the flight path and on the detected obstacles continuously update the route are responsible for the avoidance of obstacles. A serious threat to the drone, due to its value is the people. It can be stolen. In this situation, it may be helpful for the localization function and recognizing the situation. Change of the machine course can indicate theft. In this case, the drone can begin to take pictures using cameras (sensors) and give a beep to deter the thief and focus the attention of witnesses.

A very important risk associated with the extensive use of civilian drones is related to privacy. These devices have the ability to follow the tracked object and to observe it from many different perspectives. They can be equipped

with cameras, night vision devices and various sensors, facilitating snooping. Potential risks associated with the widespread use of drones require the use of complex solutions and the introduction of deliberate regulation aiming at effective protection of citizens’ privacy.

Chapter 2
Methodology

2.1 Clustering:

2.1.1 Introduction:

Imagine that we have a specific number of drones and a specific number of points and we want to let the drone pass by all the points that we have without passing by the same point more than once then we have to divide the points into a number of groups based on the number of drones.

Eventually each group should represent a path for a specific drone and a vehicle and that's why we use clustering.

We need to use it to handle a large unstructured dataset and that helps us to organize the dataset into something useful and more usable in order to move into the next step.

Clustering:

Clustering is one of the most popular unsupervised classification techniques. The task of dividing data points into a number of groups such that data points in the same groups are more similar to other data points in the same group than those in other groups.

In simple words, the aim is to segregate groups with

similar traits and assign them into clusters in order to handle every group individually and determine every point’s group. It is a main task of exploratory data analysis, and a common technique for statistical data analysis.

Cluster analysis as such is not an automatic task, but an iterative process of knowledge discovery or interactive multi-objective optimization that involves trial and failure. It is often necessary to modify data preprocessing and model parameters until the result achieves the desired properties.

2.1.2 Clustering Types:

Broadly speaking, clustering can be divided into two subgroups :

- **Hard Clustering:** In hard clustering, each point should belong to a single group or may not belong to any at all.
- **Soft Clustering**: In soft clustering, instead of putting each data point into a separate cluster, a probability or likelihood of that data point to be in those clusters is assigned.
  So a single point may belong to different clusters at the same time and we have to maintain the degree

of relation between each node and all clusters it belongs to.

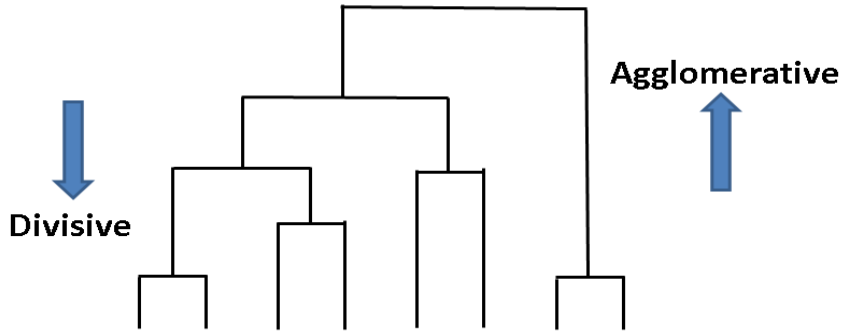
2.1.3 Types of Clustering Algorithms:

It can be achieved by various algorithms that differ significantly in their understanding of what constitutes a cluster and how to efficiently find them

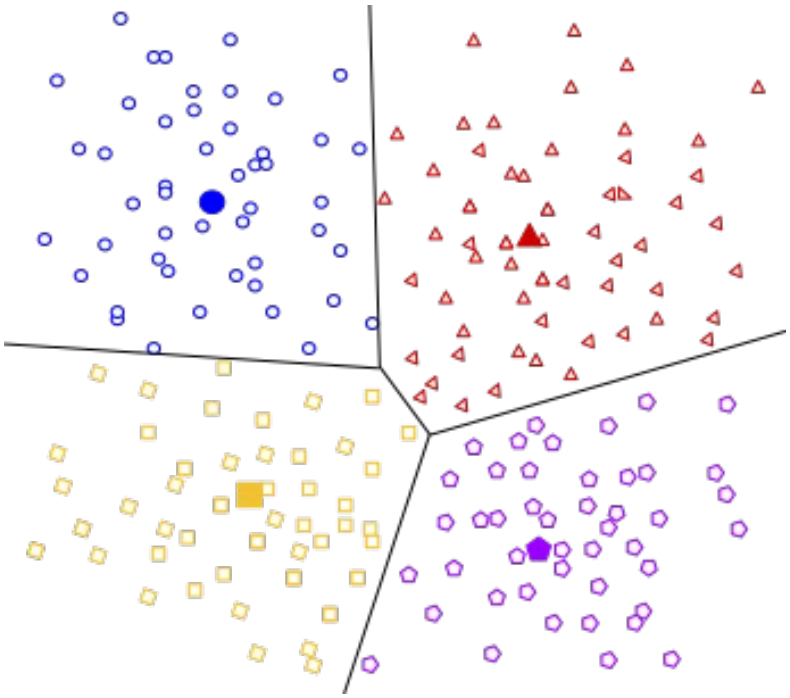
1. **Connectivity-based clustering:** also known as hierarchical clustering and as the name suggests, It's based on the core idea of objects being more related to nearby objects than to objects farther away. These algorithms connect "objects" to form "clusters" based on their distance.
   The clusters formed in this method form a tree-type structure based on the hierarchy. New clusters are formed using the previously formed one.It is divided into two categories:
   - **Agglomerative** (*bottom up approach*): they start with classifying all data points into separate clusters & then aggregating them as the distance decreases.
   - **Divisive** (*top down approach*): all data points are classified as a single cluster and then partitioned as the distance increases.Also, the choice of distance function is subjective. These models are very easy to interpret but lack

scalability for handling big datasets.

Example of these models is *hierarchical clustering algorithm*.

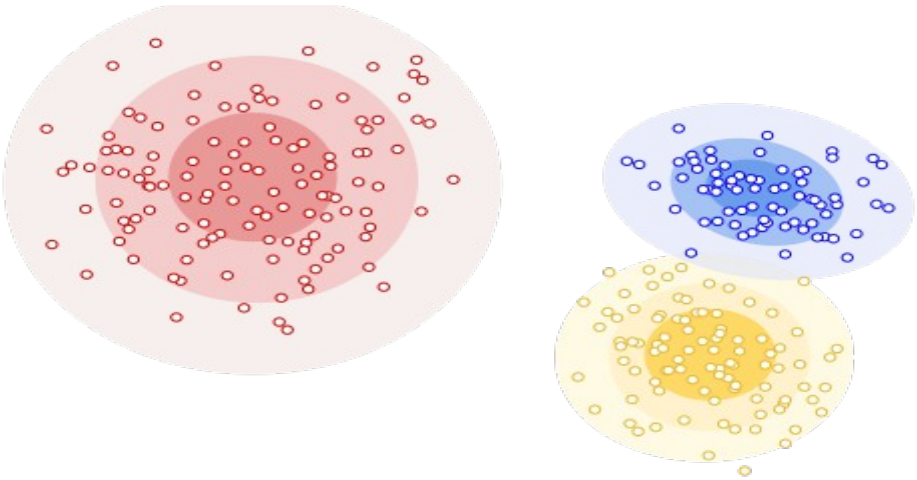


2. **Centroid-based clustering:** These are iterative clustering algorithms in which the notion of similarity is derived by the closeness of a data point to the centroid of the clusters. K-Means clustering algorithm is a popular algorithm that falls into this category. In these models, the no. of clusters required at the end have to be mentioned beforehand, which makes it important to have prior knowledge of the dataset. These models run iteratively to find the local optima.



Example of centroid based clustering

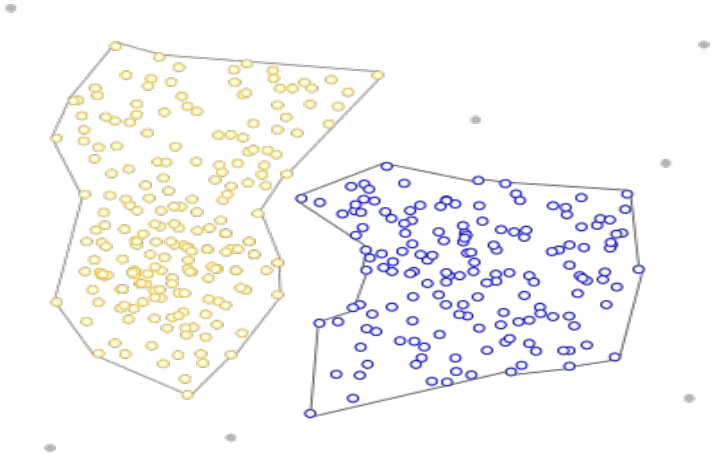
3. **Distribution-based clustering:** These clustering models are based on the notion of how probable is it that all data points in the cluster belong to the same distribution (For example: Normal, Gaussian). These models often suffer from overfitting.
   A popular example of these models is the *Expectation-maximization algorithm* which uses multivariate normal distributions.



Example of distribution based clustering

4. **Density-based clustering:** These models search the data space for areas of varied density of data points in the data space. It isolates various different density regions and assigns the data points within these regions in the same cluster.

   Popular examples of density models are *DBSCAN* and *OPTICS*.



Example of density-based clustering

2.1.4 K-Means:

K-means clustering is an unsupervised learning technique to classify unlabeled data by grouping them by features, rather than predefined categories. The variable K represents the number of groups or categories created. The goal is to split the data into K different clusters and report the location of the center of mass for each cluster. Then, a new data point can be assigned a cluster (class) based on the closed center of mass. The big advantage of this approach is that the human bias is taken out of the equation. Instead of having a researcher create classification groups, the machine creates its own clusters based upon empirical proofs, rather than assumptions.

How it works:

Let $X = \{x_1, x_2, x_3, \ldots\ldots\ldots, x_n\}$ be the set of data points and $V = \{v_1, v_2, \ldots\ldots\ldots, v_c\}$ be the set of centers.

1. Randomly select ‘c’ cluster centers.
2. Calculate the Euclidean distance between each data point and cluster centers.
3. Assign the data point to the cluster center whose distance from the cluster center is the minimum of all the cluster centers.
4. Recalculate the new cluster center using:

$$v_i = (1/c_i) \sum_{j=1}^{c_i} x_i$$ where, ‘ci’ represents the number of data points in i-th cluster.

5. If the new obtained cluster centers are still the same then stop, else repeat the steps from step (2) with the new obtained cluster centers.

Why did we use K-Means specifically?

- Choosing K manually:
  We have to set the number of groups based on the certain number of drones.
- Strong sensitivity to outliers and noise:
  Centroids can be dragged by outliers, or outliers might get their own cluster instead of being ignored.
- Relatively simple to implement.
- Scales to large data sets.
- Can warm-start the positions of centroids.

2.2 (2-Approximation) Heuristic:

In this section we will discuss how our heuristic works in detail. Our heuristic is called (2-Approximation using MST) since we use the Minimum Spanning Tree to build a tour with a total distance lay between [best answer, 2 * best answer].

And these are the heuristic steps:

1- Determine our starting and ending vertex.

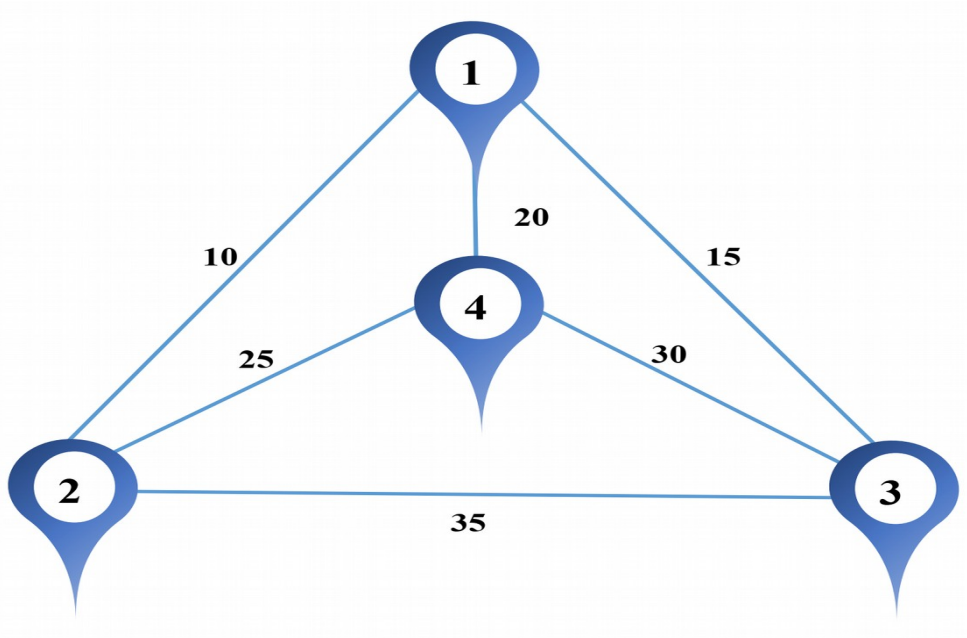
2- Construct a MST using Kruskal's algorithm.

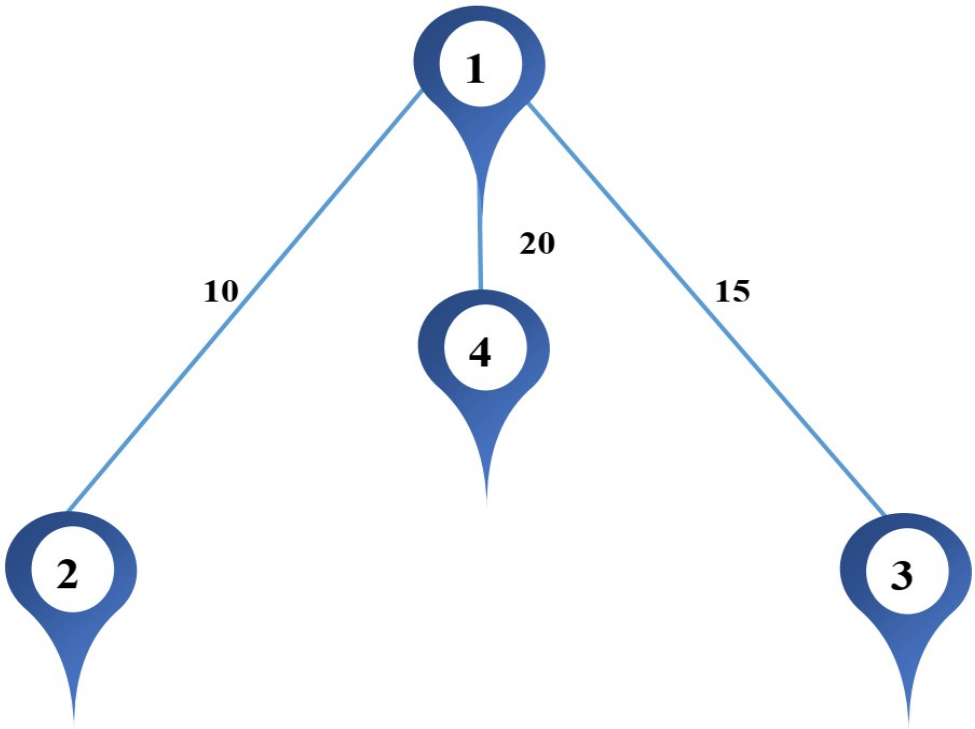
3- Listing the nodes in a pre-order walk and then adding 1 to the end or using the Euler tour.

How the heuristic works in details:

1- We can just determine our starting and ending point.

2- To construct a MST we need to know what MST is and How to build it. MST is a tree with minimum total cost which connects all vertices with each other (In other words there's always a simple path between any node to any other node).





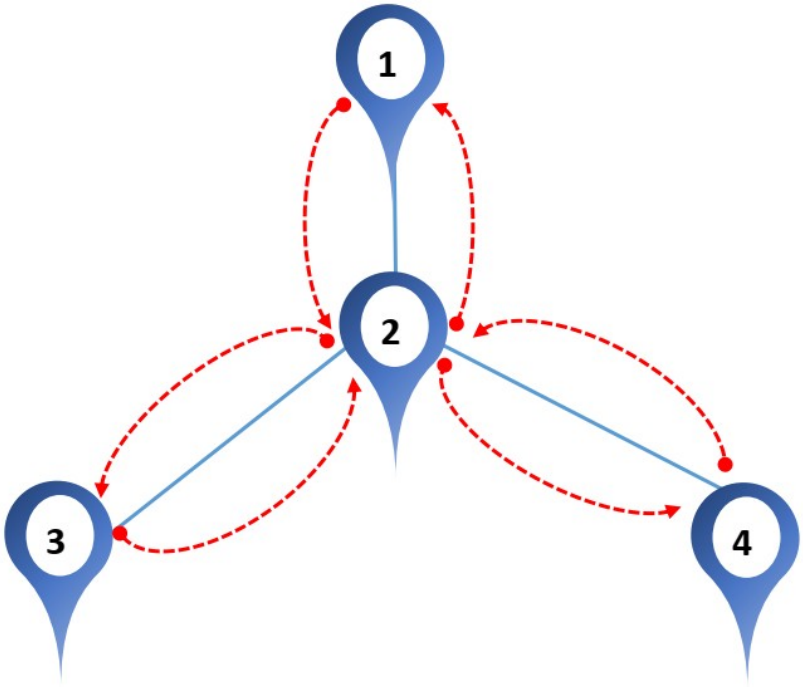
Disjoint and union set (DSU) → This is a data structure which is used to build the MST, and its main purpose is to check if two nodes are in the same forest or not in almost constant time O(1).

Kruskal's Algorithm The idea of this algorithm is simple, we will follow a greedy approach. For example: if we want to construct a Tree with N nodes then we must use

N – 1 edges by definition so obviously we will try to take the least N – 1 edges in our graph (based on their weight), and that's the idea behind Kruskal's Algorithm. we should sort the edges based on their weights then we use the DSU to merge the forest. We consider each vertex as a forest of its own and we try to make a forest of size N. How do we merge two forests We are iterating

over all the edges in ascending order, and each edge have a 2 vertices (start and end) so we use the DSU to know if these 2 nodes are in the same forest or not, if yes then we cannot take this edge because it may cause a cycle later otherwise we take the edge and merge them, at the end of the processing we should have an MST for a connected component.

3- Building the tour We can build this tour by either doing a pre-order walk on the MST or building the Euler tour and removing all the cycles in the tour except for the last cycle. We used the Euler tour in our code



Euler Walk: 1 – 2 – 3 – 2 – 4 – 2 – 1

Valid Walk: 1 – 2 – 3 – 4 – 1

The main goal of the TSP problem is to find any minimum Hamilton Cycle in our graph.

Proof Section:

We will be using the triangle inequality to proof that shorten the path of the Euler walk will never give us a longer answer,

Triangle inequality:

- d(u, v) >= 0
- d(u, v) = d(v, u)

→ d(u, w) + d(w, v) >= d(u, v)

So in our walk (1→ 2→ 3→ 2→ 4→ 2→ 1) moving from 3→ 4 instead of 3→2→4 will never make the length path longer.

We will refer to the cost of the best Hamilton cycle as C(H), the cost of our MST is C(T) and the cost of our tour is C(C) and the cost of our tour after passing nodes as C(**C** $^{/}$) and removing edges as e.

- We know that C(C) = 2 * C(T) since we are traversing every edge twice

→ Then C(**C** $^{/}$) <= 2 * C(T) (1)

We know that by removing a single edge from the Hamilton cycle we will get a spanning tree but not necessary MST

→ Then C(H) >= C(H - e) >= C(T) (2)

From 1 and 2 we conduct that C(**C** $^{/}$) <= 2 * C(H)
(Since we know from (2) that C(H) >= C(T) we can safely assume that
C(**C** $^{/}$) <= 2 * C(H) in (1)).

2.3 Genetic Algorithms(GAs)

Are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.
Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to the next generation. In simple words, they simulate “survival of the fittest” among individuals of consecutive generations for solving a problem. Each generation consists of a population of individuals and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

Genetic algorithms are based on an analogy with genetic structure and behavior of chromosomes of the population.

Following is the foundation of GAs based on this analogy:

- Individual in population compete for resources and mate
- Those individuals who are successful (fittest) then mate to create more offspring than others
- Genes from “fittest” parents propagate throughout the generation, that is sometimes parents create offspring which are better than either parent.
- Thus each successive generation is more suited for their environment.

Search space

The population of individuals is maintained within search space. Each individual represents a solution in search space for a given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).

Fitness

A Fitness Score is given to each individual which shows the ability of an individual to “compete”. The individual having optimal fitness score (or near optimal) are sought.

The GAs maintains the population of n individuals (chromosome/solutions) along with their fitness scores.The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and produce better offspring by combining chromosomes of parents. The population size is static so the room has to be created for new arrivals. So, some individuals die and get replaced by new arrivals, eventually creating a new generation when all the mating opportunities of the old population are exhausted. It is hoped that over successive generations better solutions will arrive while least fit die.

Each new generation has on average more “better genes” than the individual (solution) of previous generations. Thus each new generation has better “partial solutions” than previous generations. Once the offspring produced have no significant difference than offspring produced by previous populations, the

population converges.

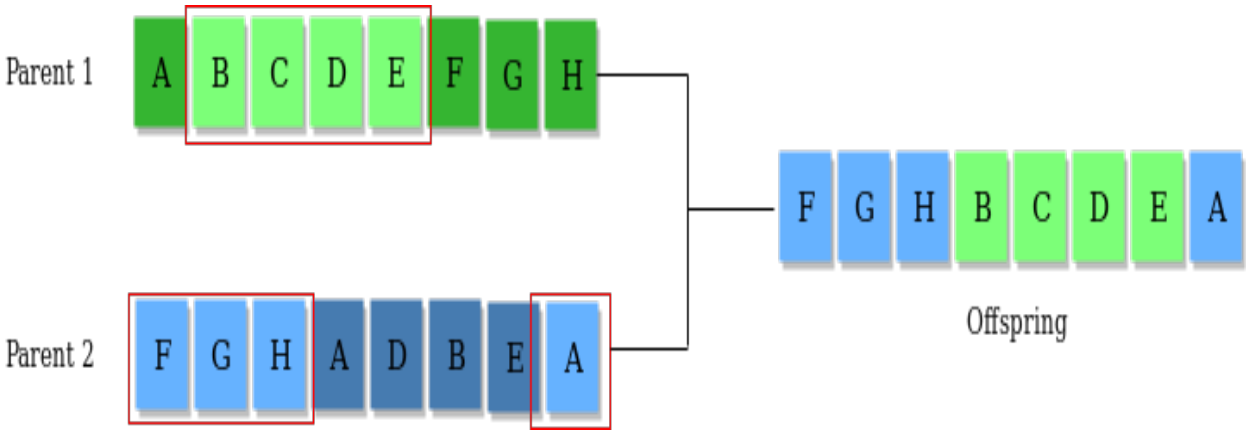
The algorithm is said to be converged to a set of solutions for the problem.

Operators of Genetic Algorithms

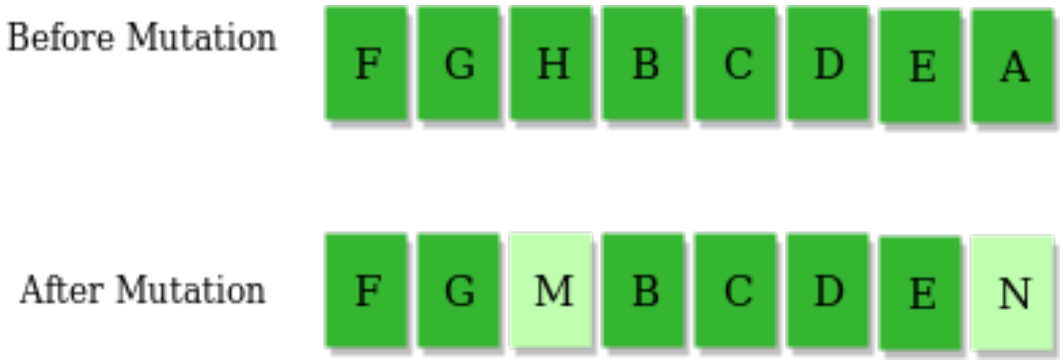
1) Selection Operator:

   Is a process based on objective function (fitness function) of each string.The objective function identifies how “good” a string is. String with a higher fitness value has a bigger probability of contributing offspring to the next generation.

2) Crossover Operator: Here, genes are exchanged or mixed between the chosen solutions or the “Survived Solutions”. That is, every two solutions or two parents will mate in the “mating pool”, thus exchanging genes and producing an “offspring / new child / new solution”. There are several methods of "crossover", but the most widespread is the **"crossing based on point"** method, where genes are exchanged between the solutions chosen to play the role of parents based on the "cut point", but the programmer can determine the "crossover" method according to the problem.



3) Mutation Operator: After obtaining the "offspring / new child" from all the two solutions between the exchange of genes, changes can be made to the new solution obtained by a process known as "Mutation". Means, for example, can be divided into "First gene" of "New child" on 2 and so on. The purpose of this process is to be able to improve the solution obtained after mating or exchanging genes. And after "Crossover" work for all "Parents" and "Mutation" for all "Children", we will get a new generation known as "Generation 1". The same steps are continued until “GA” is reached for the “Optimum solution”.The key idea is to insert random genes in offspring to maintain the diversity in population to avoid premature convergence.

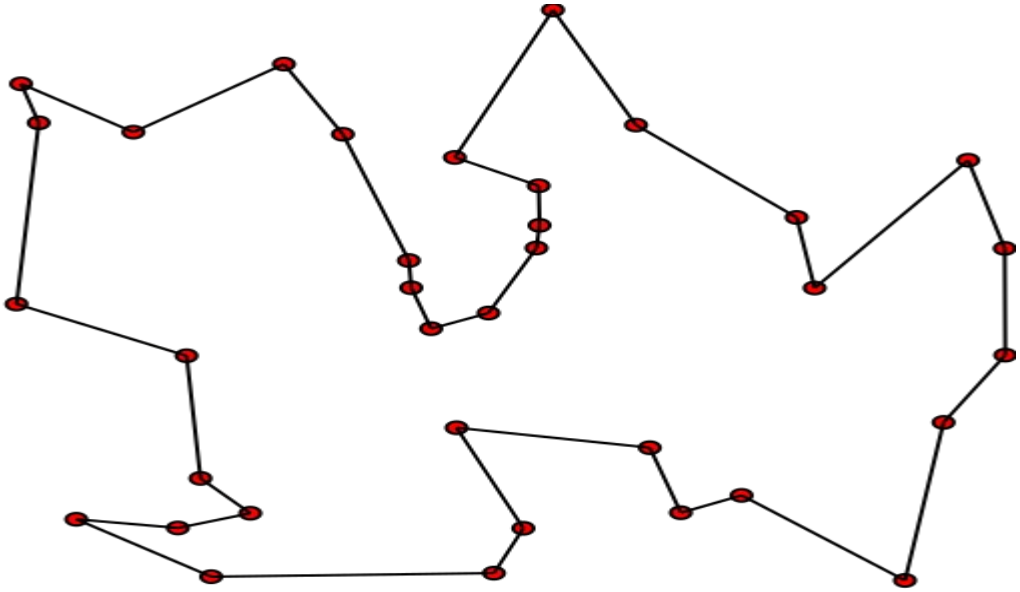


The whole algorithm can be summarized as:

1) Randomly initialize populations p
2) Determine fitness of population
3) Until convergence repeat:
    a) Select parents from population
    b) Crossover and generate new population
    c) Perform mutation on new population
    d) Calculate fitness for new population

The problem

We'll be using a GA to find a solution to the traveling salesman problem (TSP). The TSP is described as follows: -“Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?”



Given this, there are two important rules to keep in mind:

1. Each city needs to be visited exactly one time.
2. We must return to the starting city, so our total distance needs to be calculated accordingly.

The approach

Let’s start with a few definitions, rephrased in the context of the TSP:

- **Gene:** a city (represented as (x, y) coordinates)
- **Individual (aka “chromosome”):** a single route satisfying the conditions above
- **Population:** a collection of possible routes (i.e., collection of individuals)
- **Parents:** two routes that are combined to create a new route
- **Mating pool:** a collection of parents that are used to create our next population (thus creating

the next generation of routes)

- **Fitness:** a function that tells us how good each route is (in our case, how short the distance is)
- **Mutation:** a way to introduce variation in our population by randomly swapping two cities in a route
- **Elitism:** a way to carry the best individuals into the next generation

Our GA will proceed in the following steps:

1. Create the population
2. Determine fitness
3. Select the mating pool
4. Breed
5. Mutate
6. Repeat

2.4 Using the drone with greedy approach:

- We will follow a greedy technique to use the drones in order to optimize our path with the vehicle only
- The approach is just to check every 3 consecutive cities for example (1 -> 2 -> 3) if the distance(1, 2) + distance (2, 3) < drone_limit then we can use the drone to make it go to city with index 2.
- We can decide if it’s better to send it or not based on the time waste as well since the car and drone may not be with the same speed so something may wait for the other to arrive.

Chapter 3
Implementation

3.1 K-Means:

We have created the kmeans function which is customized to do what k-means in sklearn do but with just what we need in our project.

While each part of our clustering is built from scratch, we'll use a few standard packages to make things easier:

- Numby: to use randomization and the argmin function.
- Pandas: to use data frame
- Scipy.spatial: to use cdist function

```
1  import numpy as np
2  import pandas as pd
3  from scipy.spatial import distance
```

First, it takes three parameters:

1. X which represents our n-dimension data matrix.
2. k which represents the number of clusters/groups and sets its default to 1.
3. max_iterations which represents the maximum number of iterations which is allowed to do.

We generate random centroids from the dataset as in line (8).

We determine the euclidean distance between these centroids and the entire data points using *distance.cdist* and put the minimum distance in an array using the *argmin* method as in line (10).

```
6  def kmeans(X, k=3, max_iterations=100):
7      if isinstance(X, pd.DataFrame): X = X.values
8      idx = np.random.choice(len(X), k, replace=False)
9      centroids = X[idx, :]
10     P = np.argmin(distance.cdist(X, centroids, 'euclidean'), axis=1)
```

Then we start to iterate and each time we try to:

- Calculate the new centroids using the mean of each cluster as in line (12).
- Find the cluster of each point and assign it in the tmp array as in line (13).
- Check if the tmp array equals the P array and both arrays represent the new and previous clustering array as in line (14,15).
    - If P equals tmp , then its value will remain the same and we break the loop and return the array P
    - If P does not equal tmp then we will assign the value of tmp to P and continue the iteration and so on

```
11     for _ in range(max_iterations):
12         centroids = np.vstack([X[P == i, :].mean(axis=0) for i in range(k)])
13         tmp = np.argmin(distance.cdist(X, centroids, 'euclidean'), axis=1)
14         if np.array_equal(P, tmp): break
15         P = tmp
16     return P
```

When we call this method to use it we assign its return clustering array to a variable so we can make use of it in the other steps and chapters.

3.2 (2- Approximation Heuristic)

- First we will choose our root which should be given as an input
- Secondly we should build our MST by using kruskal algorithm and DSU.

Building DSU:

```
# This function is responsible to query which forest this node belong to in
# almost constant time

def find(self, parent, i):
    if parent[i] == i:
        return i

    # this is called Path compression to make the query in O(1)
    parent[i] = self.find(parent, parent[i])
    return parent[i]


# This function is responsible for making the union  operation or merging
# The two forest together by rank

def apply_union(self, parent, rank, x, y):
    xroot = self.find(parent, x)
    yroot = self.find(parent, y)
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    else:
        parent[yroot] = xroot
        rank[xroot] += 1
```

- Find function is used to return the ID of the forest that contains this node in almost constant time O(1) by using path compression technique.
- Apply_union used to merge 2 forests into one larger forest by using union by rank technique.

Kruskal's Algorithm to find a MST:

- We sort the edges according to their weights and go greedly by picking the lowest edge weight and trying to take that edge into our current MST.

```
#  This function return the MST by applying kruskal algorithm.
# Sorting the edge list then using the DSU to check the forest of u, v for each

def kruskal_algo(self, cnt):
    result = []
    i, e = 0, 0
    self.graph = sorted(self.graph, key = lambda item: item[2])
    parent = []
    rank = []

    for node in range(self.V):
        parent.append(node)
        rank.append(0)


    while e < cnt - 1:
        u, v, w = self.graph[i]
        i = i + 1
        x = self.find(parent, u)
        y = self.find(parent, v)
        if x != y:
            e = e + 1
            result.append([u, v, w])
            self.apply_union(parent, rank, x, y)
    return result
```

Building the Euler tour:

- We will do a normal DFS and remove the duplicate nodes and maintain the starting vertex in the end to make the cycle valid.

```
# This DFS function will build us the Euler tour instead of doing pre-order
# Walk then removing the cycles from it except of the last one.
def DFS(self, u, parent, Euler, new_graph, V):
    if(u == parent):
        return
    if(V[u] == True):
        return
    V[u] = True

    Euler.append(u)

    for v in new_graph[u]:
        self.DFS(v, u, Euler, new_graph, V)
        Euler.append(u)
    return
```

We apply the same steps for each cluster and save the answer in the adjacency list to send it to the next step which is applying Genetic Algorithm for each group we answered.

3.3 Genetic Algorithm:

While each part of our GA is built from scratch, we'll use a few standard packages to make things easier:

```
import numpy as np, random, operator, pandas as pd, matplotlib.pyplot
as plt
```

Create two classes: City and Fitness

- We first create a City class that will allow us to create and handle our cities. These are simply our (x, y) coordinates. Within the City class, we add a distance calculation (making use of the Pythagorean theorem) in line 13 and a cleaner way to output the cities as coordinates with __repr__ in line 19.

```
 8  class City:
 9      def __init__(self, x, y):
10          self.x = x
11          self.y = y
12
13      def distance(self, city):
14          xDis = abs(self.x - city.x)
15          yDis = abs(self.y - city.y)
16          distance = np.sqrt((xDis ** 2) + (yDis ** 2))
17          return distance
18
19      def __repr__(self):
20          return "(" + str(self.x) + "," + str(self.y) + ")"
```

Creating a Fitness function

- In our case, we'll treat the fitness as the inverse of the route distance. We want to minimize route distance, so a larger fitness score is better. Based on Rule #2, we need to start and end at the same place, so this extra calculation is accounted for in line 35 of the distance calculation.

```
23  class Fitness:
24      def __init__(self, route):
25          self.route = route
26          self.distance = 0
27          self.fitness = 0.0
28
29      def routeDistance(self):
30          if self.distance == 0:
31              pathDistance = 0
32              for i in range(0, len(self.route)):
33                  fromCity = self.route[i]
34                  toCity = None
35                  if i + 1 < len(self.route):
36                      toCity = self.route[i + 1]
37                  else:
38                      toCity = self.route[0]
39                  pathDistance += fromCity.distance(toCity)
40              self.distance = pathDistance
41          return self.distance
42
43      def routeFitness(self):
44          if self.fitness == 0:
45              self.fitness = 1 / float(self.routeDistance())
46          return self.fitness
```

Create the population

- We now can make our initial population (aka first generation). To do so, we need a way to create a function that produces routes that satisfy our conditions (Note: we'll create our list of cities when we actually run the GA at the end of the tutorial). To create an individual, we randomly select the order in which we visit each city:

```
50  def createRoute(cityList):
51      route = random.sample(cityList, len(cityList))
52      return route
```

- This produces one individual, but we want a full population, so let's do that in our next function. This is as simple as looping through the createRoute function until we have as many routes as we want for our population.

```
55  def initialPopulation(popSize, cityList):
56      population = []
57
58      for i in range(0, popSize):
59          population.append(createRoute(cityList))
60      return population
```

- Note: we only have to use these functions to create the initial population. Subsequent generations will be

produced through breeding and mutation.

Determine fitness

- Next, the evolutionary fun begins. To simulate our "survival of the fittest", we can make use of **Fitness** to rank each individual in the population. Our output will be an ordered list with the route IDs and each associated fitness score.

```
63  def rankRoutes(population):
64      fitnessResults = {}
65      for i in range(0, len(population)):
66          fitnessResults[i] = Fitness(population[i]).routeFitness()
67      return sorted(fitnessResults.items(), key=operator.itemgetter(1), reverse=True)
68
```

Select the mating pool

- There are a few options for how to select the parents that will be used to create the next generation. The most common approaches are either fitness proportionate selection (aka "**roulette wheel selection**") or **tournament selection**:

  1- **Fitness proportionate selection (the version implemented below):** The fitness of each individual relative to the population is used to assign a probability of selection. Think of this as the fitness-

weighted probability of being selected

2-**Tournament selection:** A set number of individuals are randomly selected from the population and the one with the highest fitness in the group is chosen as the first parent. This is repeated to choose the second parent.

Another design feature to consider is the use of elitism. With elitism, the best performing individuals from the population will automatically carry over to the next generation, ensuring that the most successful individuals persist.

- For the purpose of clarity, we'll create the mating pool in two steps. First, we'll use the output from rankRoutes to determine which routes to select in our selection function. In lines 72–74, we set up the roulette wheel by calculating a relative fitness weight for each individual. In line 78, we compare a randomly drawn number to these weights to select our mating pool. We'll also want to hold on to our best routes, so we introduce elitism in line 76. Ultimately, the selection function returns a list of

route IDs, which we can use to create the mating pool in the matingPool function.

```
70  def selection(popRanked, eliteSize):
71      selectionResults = []
72      df = pd.DataFrame(np.array(popRanked), columns=["Index", "Fitness"])
73      df['cum_sum'] = df.Fitness.cumsum()
74      df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()
75
76      for i in range(0, eliteSize):
77          selectionResults.append(popRanked[i][0])
78      for i in range(0, len(popRanked) - eliteSize):
79          pick = 100*random.random()
80          for i in range(0, len(popRanked)):
81              if pick <= df.iat[i, 3]:
82                  selectionResults.append(popRanked[i][0])
83                  break
84      return selectionResults
```

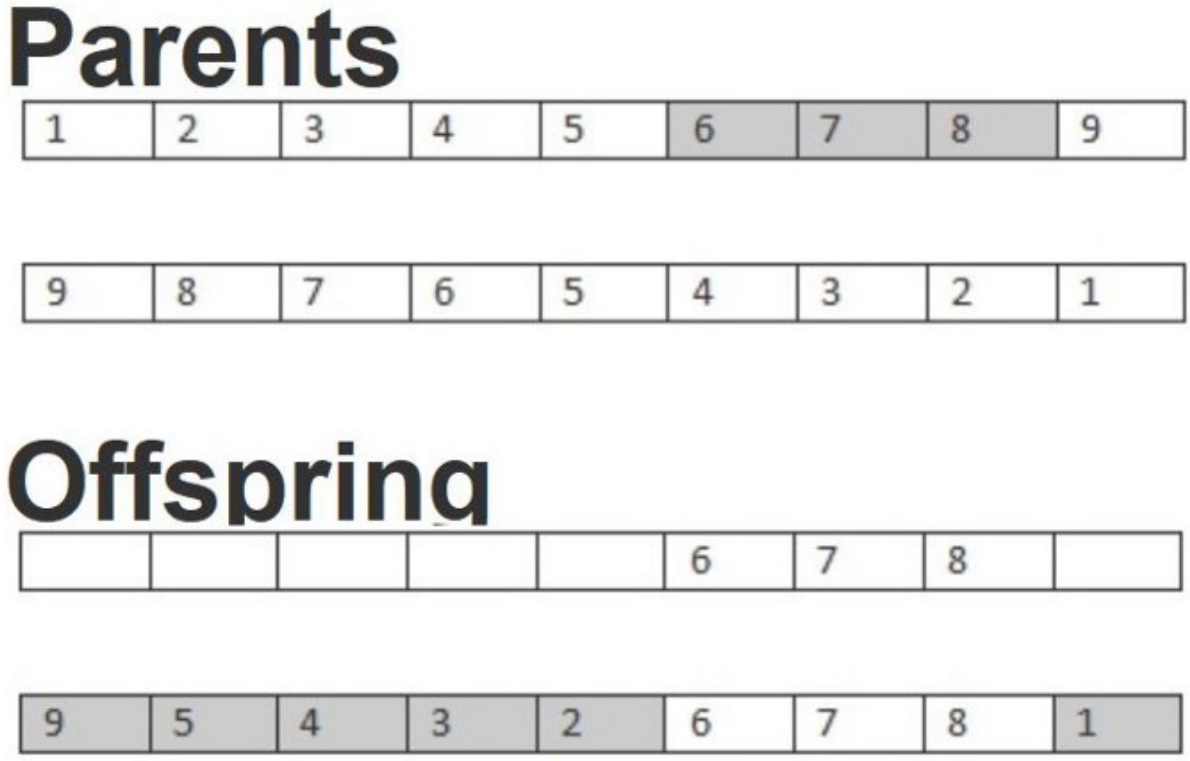
- Now that we have the IDs of the routes that will make up our mating pool from the selection function, we can create the mating pool. We're simply extracting the selected individuals from our population.

```
87  def matingPool(population, selectionResults):
88      matingpool = []
89      for i in range(0, len(selectionResults)):
90          index = selectionResults[i]
91          matingpool.append(population[index])
92      return matingpool
```

Breed

- With our mating pool created, we can create the next generation in a process called crossover (aka “breeding”). If our individuals were strings of 0s and 1s and our two rules didn’t apply (e.g., imagine we were deciding whether or not to include a stock in a portfolio), we could simply pick a crossover point and splice the two strings together to produce an offspring.

However, the TSP is unique in that we need to include all locations exactly one time. To abide by this rule, we can use a special breeding function called **ordered crossover**. In ordered crossover, we randomly select a subset of the first parent string (see line 106 in breed function below) and then fill the remainder of the route with the genes from the second parent in the order in which they appear, without duplicating any genes in the selected subset from the first parent (see line 109 in breed function below).




```
95  def breed(parent1, parent2):
96      child = []
97      childP1 = []
98      childP2 = []
99
100     geneA = int(random.random() * len(parent1))
101     geneB = int(random.random() * len(parent1))
102
103     startGene = min(geneA, geneB)
104     endGene = max(geneA, geneB)
105
106     for i in range(startGene, endGene):
107         childP1.append(parent1[i])
108
109     childP2 = [item for item in parent2 if item not in childP1]
110
111     child = childP1 + childP2
112     return child
```

Next, we'll generalize this to create our offspring

population. In line120, we use elitism to retain the best routes from the current
population. Then, in line 123, we use the breed function to fill out the rest of the next generation.

```
115  def breedPopulation(matingpool, eliteSize):
116      children = []
117      length = len(matingpool) - eliteSize
118      pool = random.sample(matingpool, len(matingpool))
119
120      for i in range(0, eliteSize):
121          children.append(matingpool[i])
122
123      for i in range(0, length):
124          child = breed(pool[i], pool[len(matingpool)-i-1])
125          children.append(child)
126      return children
```

Mutate

- Mutation serves an important function in GA, as it helps to avoid local convergence by introducing novel routes that will allow us to explore other parts of the solution space. Similar to crossover, the TSP has a special consideration when it comes to mutation. Again, if we had a chromosome of 0s and 1s, mutation would simply mean assigning a low probability of a gene changing from 0 to 1, or vice versa (to continue the example from before, a stock that was included in the offspring portfolio is now excluded).

- However, since we need to abide by our rules, we can't drop cities. Instead, we'll use **swap mutation**. This means that, with specified low probability, two cities will swap places in our route. We'll do this for one individual in our mutate function:

```
129  def mutate(individual, mutationRate):
130      for swapped in range(len(individual)):
131          if(random.random() < mutationRate):
132              swapWith = int(random.random() * len(individual))
133
134              city1 = individual[swapped]
135              city2 = individual[swapWith]
136
137              individual[swapped] = city2
138              individual[swapWith] = city1
139      return individual
```

- Next, we can extend the mutate function to run through the new population.

```
142  def mutatePopulation(population, mutationRate):
143      mutatedPop = []
144
145      for ind in range(0, len(population)):
146          mutatedInd = mutate(population[ind], mutationRate)
147          mutatedPop.append(mutatedInd)
148      return mutatedPop
```

Repeat

- We're almost there. Let's pull these pieces together to create a function that produces a new generation. First, we rank the routes in the current generation using rankRoutes. We then determine our potential parents by running the selection function, which allows us to create the mating pool using the matingPool function. Finally, we then create our new

generation using the breedPopulation function and then applying mutation using the mutatePopulation function

```
151  def nextGeneration(currentGen, eliteSize, mutationRate):
152      popRanked = rankRoutes(currentGen)
153      selectionResults = selection(popRanked, eliteSize)
154      matingpool = matingPool(currentGen, selectionResults)
155      children = breedPopulation(matingpool, eliteSize)
156      nextGeneration = mutatePopulation(children, mutationRate)
157      return nextGeneration
```

Evolution in motion

- We finally have all the pieces in place to create our GA! All we need to do is create the initial population, and then we can loop through as many generations as we desire. Of course we also want to see the best route and how much we've improved, so we capture the initial distance in line 3 (remember, distance is the inverse of the fitness), the final distance in line 8, and the best route in line 9.

```
160   def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations)
161       pop = initialPopulation(popSize, population)
162       print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))
163   
164       for i in range(0, generations):
165           pop = nextGeneration(pop, eliteSize, mutationRate)
166   
167       print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))
168       bestRouteIndex = rankRoutes(pop)[0][0]
169       bestRoute = pop[bestRouteIndex]
170       return bestRoute
179   geneticAlgorithm(population=cityList, popSize=100,
180                    eliteSize=20, mutationRate=0.01, generations=500)
```

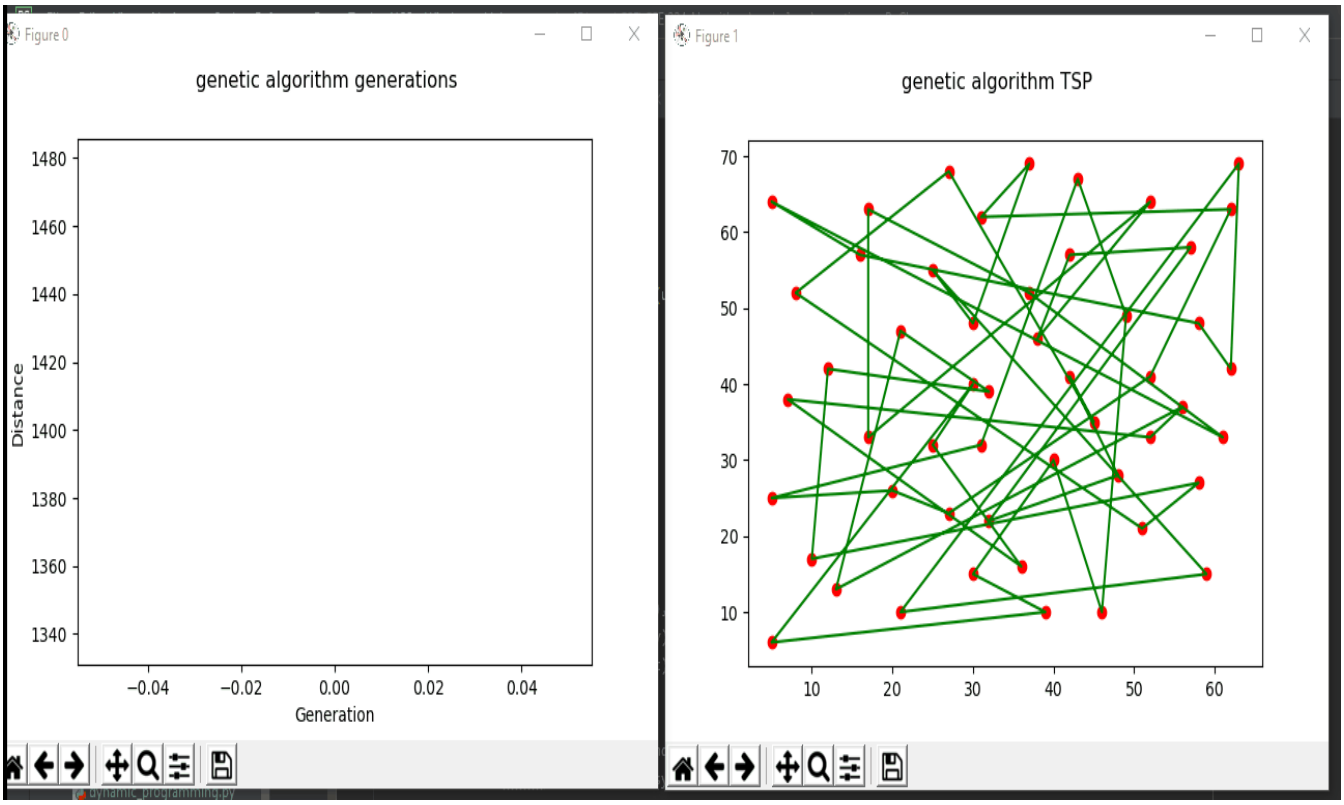
Using the drone with the greedy approach:

- We will calculate the distance between each three consecutive cities by using the Euclidean distance and we should be given the drone limit or for how long he can fly without falling, or if he can go to city or customer X or not.
- Then we can determine whether to send the drone to that customer or city or not based on multiple things, like if the drone can get there in the first place or not or if it will make the car wait for more time.

```
46  cnt_groups = 0
47  
48  for each_group_in_genetic in sol_per_group_genetic:
49      drone_for_this_group = []
50      cnt = len(each_group_in_genetic)
51      for index in range(0, cnt - 2):
52          current_city = each_group_in_genetic[index]
53          drone_city = each_group_in_genetic[index + 1]
54          next_city = each_group_in_genetic[index + 2]
55          if(distance(current_city, drone_city) + distance(drone_city, next_city) < drone_limit):
56              drone_for_this_group.append(drone_city)
57              index = index + 1
58      print("for group {0} the drone went for: ", cnt_groups);
59      for drone in drone_for_this_group:
60          print(drone)
61      cnt_groups = cnt_groups + 1
```

Simulation of the genetic algorithm:

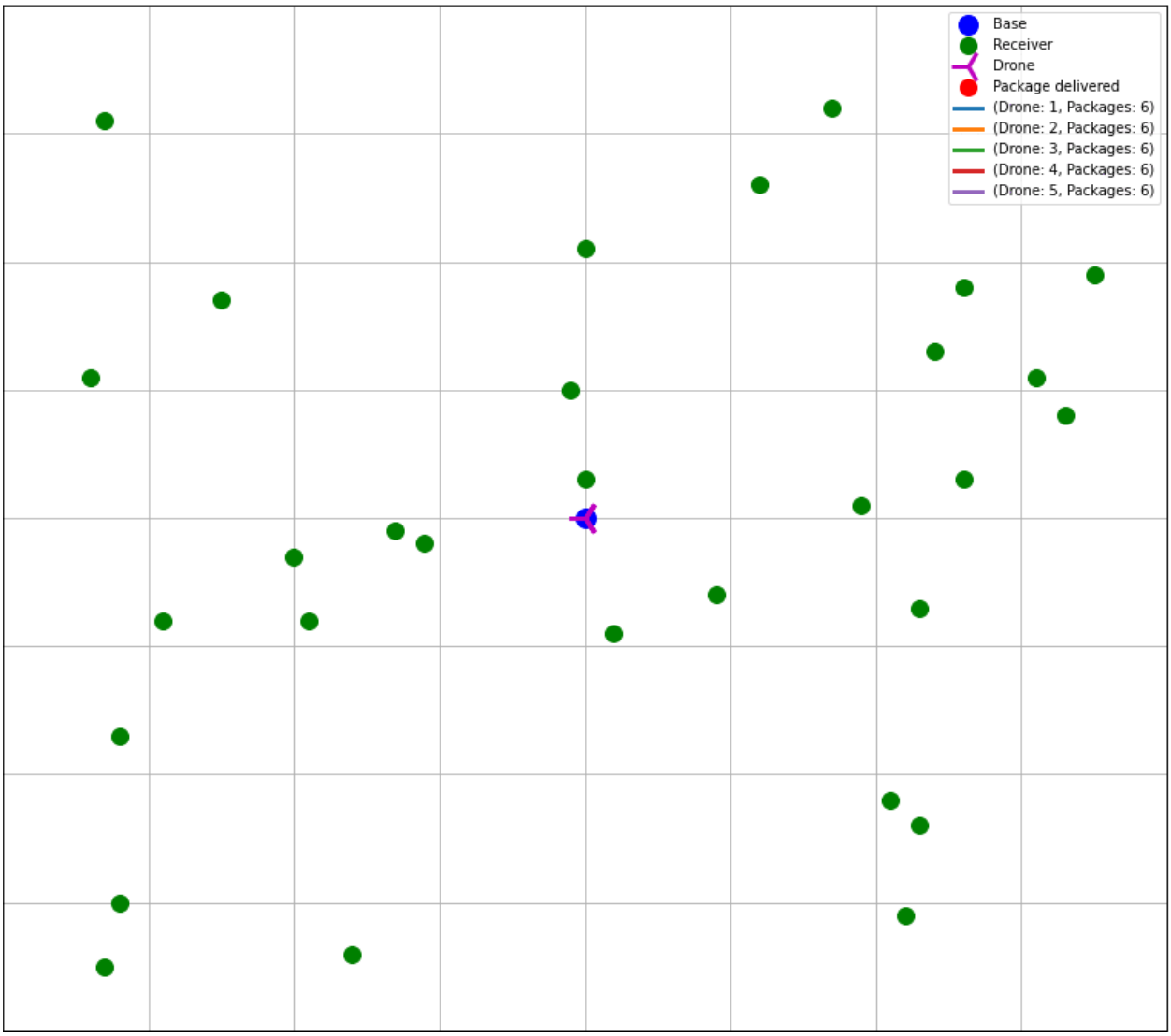
For the purpose of illustration we implemented some visual diagrams to show how fast it converges and what's our best path in each generation.



- The GIF on the left shows us the best distance we got so far after generation X
- The GIF on the right shows us the current best path and keep updating once we get a better path.

Simulation of MTSP:

- This GIF should illustrate the movement of MTSP or MTSD. It should not matter if we used a drone to deliver the package or a man as long as the drone can do it.



- This illustrate the path for each group after finishing the genetic algorithm step (So we have a path for each group using a vehicle only)

References

- http://graphics.stanford.edu/courses/cs468-06-winter/Papers/arora-tsp.pdf
- https://www.cambridge.org/core/product/identifier/S0305004100034095/type/journal_article
- https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35
- https://www.geeksforgeeks.org/genetic-algorithms/
- http://homepages.cwi.nl/~lex/files/histco.pdf
- https://github.com/rameziophobia/Travelling_Salesman_Optimization
- https://www.sciencedirect.com/science/article/abs/pii/S187449072030313X?via%3Dihub
- http://www.cs.tufts.edu/~cowen/advanced/2002/adv-lect3.pdf
- https://github.com/NPilis/Drone-Delivery-System