**COURSE CODE : INT-254**

**PROJECT REPORT**

**ON**

**TRAVELLING SALESMAN PROBLEM USING**

**ANT-COLONY OPTIMIZATION**

Submitted in partial fulfilment of the requirements for the assignment

Of

**B. Tech** (INT-254)

In

**Computer Science & Engineering** (Hons.)

Submitted to:

**LOVELY PROFESSIONAL UNIVERSITY**

A picture containing shape

Description automatically generated

**Submitted by: Under the Guidance of**

1. E.RAKESH (12012967) **Dr. Rajan Kakkar** .

2. K.PURNA CHANDU (12014045)

**CANDIDATE’S DECLARATION**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

We hereby certify that the work, which is being presented in this report entitled, **Travelling Salesman**, in partial fulfillment of the requirements for the degree of **Bachelor of Technology, INT-254** submitted in the **Computer Science and Engineering** , Lovely Professional University, Phagwara, Punjab; by **K.Purna Chandu (12014045), E.Rakesh (12012067),** is the authentic record of our own work carried out under the supervision of **Dr. Rajan Kakkar**, **Professor, Computer Science and Engineering**, Lovely Professional University, Phagwara, Punjab.

We further declare that the matter embodied in this report has not been submitted by us for the award of any other degree.

**Candidate(s) Signature**

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

**Signature of Students Signature of Supervisor**

**E. Rakesh (12012967). Dr. Rajan Kakka.**

**K. Purna Chandu (12014045).**

**Date :**

# ACKNOWLEDGMENT

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

It is our pleasure to acknowledge the contributions of all who have helped us and supported us during this Project report.

First, we thank God for helping us in one way or another and providing strength and endurance to us. We wish to express my sincere gratitude and indebtedness to our supervisor Dr. Rajan Kakkar, Professor, Computer Science and Engineering, Lovely Professional University, Phagwara, Punjab; for his intuitive and meticulous guidance and perpetual inspiration in completion of this report. In spite of his busy schedule, he rendered help whenever needed, giving useful suggestions and holding informal discussions. His invaluable guidance and support throughout this work cannot be written down in few words. We also thank her for providing facilities for my work in the department.

We are also humbly obliged by the support of our group members and friends for their love and caring attitude. The sentimental support they rendered to us is invaluable and everlasting. They have helped us through thick and thin and enabled us to complete the work with joy and vigor. We thank the group members for entrusting in each other and following directions, without them this report would never have been possible.

We are also thankful to our parents, elders and all family members for their blessing, motivation and inspiration throughout our work and bearing with us even during stress and bad temper. They have always provided us a high moral support and contributed in all possible ways in completion of this Capstone report

Thank You!

# ABSTRACT

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Transportation refers to the movement of everything from raw materials to finished goods between various facilities in the supply chain. In transportation, the exchange between responsiveness and efficiency is manifested in the choice of transportation modes. Because transportation costs can be as much as one third of the supply chain operating costs, the decision made here is very important. Traveling Salesman Problem is one of the best-known NP-hard problems where there is no precise algorithm to solve it in polynomial time. The ACO algorithm has good potential for problem solving and recent research that has attracted a lot of attention, in particular is the case of solving NP-Hard problems. One of the earliest best works is completing TSP using ACS (Ant Colony System). Ant System is the first ACO algorithm with its main characteristics being that at each iteration, the pheromone values are updated by all (m) ants who have built a solution in the iteration itself. From the arrangement of the shipping routes that have been implemented by the Ant Colony Optimization on the Traveling Salesman Problem, the amount of savings in the transportation mode of trucks to mileage is 37%.

Vehicle traffic congestion leads to air pollution, driver frustration, and costs billions of dollars annually in fuel consumption. Finding a proper solution to vehicle congestion is a considerable challenge due to the dynamic and unpredictable nature of the network topology of vehicular environments, especially in urban areas. Ant colony optimization (ACO) has been widely used for different combinatorial optimization problems. Ant Colony Optimization (ACO) as a heuristic algorithm has been proven a successful technique and applied to a number of combinatorial optimization (CO) problems. The traveling salesman problem (TSP) is one of the most important combinatorial problems. We present a bio-inspired algorithm, food search behavior of ants, which is a promising way of solving the Travel Salesman Problem. in this paper, we investigate ACO algorithms with respect to their runtime behavior for the traveling salesperson (TSP) problem. Simulation results conducted using MATLAB suggests that the proposed system would perform consistently despite increase of vehicles with in the given area.

**Keywords:** Traveling Sales Salesman Problem(TSP), Ant Colony Optimization(ACO), Numpy, Congestion, Huristic Algorithm, MATLAB.

# TABLE OF CONTENTS

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Title PageNo.**

**CANDIDATE’S DECLARATION AND CERTIFICATE…...................................2**

**ACKNOWLEDGMENT……………………………………………………………..3**

**ABSTRACT……………………………………………………….…………………..4**

**TABLE OF CONTENTS…………………………………….……………………….5-6**

**S No. Title Page No.**

* **INTRODUCTION………………………………………………..7**

1. **PYTHON…………………………………………………………..9**
   1. **Scripting Language………………………………………………10**
   2. **Object Oriented Programming Language…………………..10**
   3. **History………………………………………………………………11**
   4. **Data Type……………………………………………………………11**
   5. **Variable……………………………………………………………...12**
   6. **Tuples………………………………………………………………..14**
   7. **List…………………………………………………………………...15**
   8. **Loop definition…………………………………………………….18**
   9. **Conditional Statements…………………………………………..20**

**S No. Title Page No.**

* 1. **Function……………………………………………………………..21**

1. **NUMPY…………………………………………………………….22**
   1. **WHY NUMPY IS FAST ? ……………………………………………24**
   2. **WHO ELSE USES NUMPY ? ………………………………………..24**
2. **MATPLOT LIBRARY …………………………………………..25**
   1. **Matplotlib UI Menu…………………………………………………...26**
   2. **Matplotlib and NumPy………………………………………………..26**
   3. **Matplotlib and Pandas………………………………………………..27**
   4. **Matplotlib Line Plot…………………………………………………..28**
   5. **Matplotlib Pie Plot…………………………………………………….29**
   6. **Matplotlib Bar Plot……………………………………………………30**
3. **Implementing Ant Colony Optimization (ACO) algorithm**

**for a given Symmetric traveling salesman problem (TSP) …………………………………………………………………………..31**

1. **Implementation……………………………………………………..34**
2. **Extras………………………………………………………………….38**
3. **Conclusion…………………………………………………………….42**
4. **Referrence……………………………………………………………..43**

**INTRODUCTION**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Transportation refers to the movement of everything from raw materials to finished goods between various facilities in the supply chain. In transportation, the exchange between responsiveness and efficiency is manifested in the choice of transportation modes. Because transportation costs can be as much as one third of the supply chain operating costs, the decision made here is very important. The objective of the transportation system aims to minimize the distance and cost. ensure that all vehicles are not allowed to carry more than their capacity, and choose better route for transportation process. So that the transportation system running with optimal and efficient condition to meet right place, right time, right quantity, right quantity [1]. Due to various modes of transportation and location of facilities in the supply chain, managers need to design routes and networks to move products. Route is the path where the product moves and the network consists of a collection of lines and facilities that are connected by that path. As a general rule, the higher the value of a product (such as electronic components or drugs), the more the transportation network must emphasize responsiveness and the lower the value of a product (such as bulk commodities such as grain or wood), the greater the network must emphasize efficiency [2]. The main components of transportation that must be analyzed by the company when designing and operating a Supply Chain [3].

• Transportation Network Design

• Transportation Mode Alternatives

Traveling Salesman Problem is one of the best known NP-hard problems where there is no precise algorithm to solve it in polynomial time. TSP is defined as a permutation problem with the goal of finding the path with the shortest length or minimum cost. The TSP was then studied by mathematicians Karl Menger at Vienna, Harvard and Hassler Whitney and Merrill Flood at Princeton in 1930 and wrote their general form. Because the growth of the algorithm is increasing from year to year, starting in 1950 the completion of the CSR program was solved by computers. In 1954 the important growth of CSR was researched by researchers namely: Dantzig, Fulkerson and Jhonson who continued to develop new methods for solving CSR problems [4]. The Ant Colony Optimization (ACO) Heuristic Method is inspired by the behavior of real ants in finding the shortest path between nests and food. This is achieved by a substance called pheromone which shows ant traces. In its search, the ant uses heuristic information which is its own knowledge about where the smell of food comes from and other ant's decisions about the path to food with pheromone information [5].

ACO-TSP research like Dorigo, Stutzle and Birattari in the year applied the shortest route determination using ACO-TSP theoretically. Research on fuel costs (fuel cost) from public vehicles to get satisfactory results from ACO-TSP in vehicle efficiency [6]. ACO can also be used to improve the performance of a TSP in minimizing the transportation costs of the resulting route. [7]

The ACO algorithm has good potential for problem solving and recent research that has attracted a lot of attention, in particular is the case of solving NP-Hard problems. One of the earliest best works is completing TSP using ACS (Ant Colony System). ACS algorithm is used to solve TSP and claimed that ACS outperforms other algorithms inspired by nature such as simulated annealing and evolutionary computation [8]. Ant System is the first ACO algorithm with its main characteristics being that at each iteration, the pheromone values are updated by all (m) ants who have built a solution in the iteration itself [9,10]. Traveling Salesman Problem is well known in operations research to minimize travel costs or distance. Some linear programming concepts used with MATLAB, researchers have previously described the implementation of primal infeasible double interior point algorithm for large-scale linear programming under the MATLAB environment [11].

**Research Methodology**

**Ant System**

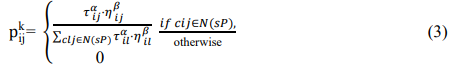
Ant System is the first ACO algorithm with its main characteristics being that at each iteration, the pheromone values are updated by all (m) ants who have built a solution in the iteration itself. Updated Pheromone τij can be seen in the following formulations.

Whereas :

⍴ = Evaporation Rate

m = The number of artificial ants

∆τij^k = Pheromone quantity at edge (i, j) by ant k

 Where Q is constant and Lk is the length of the tour trip by ant k. In the stage of finding a solution, the ant chooses the prospective city to be traversed using a stochastic mechanism. Where ant k is in city i and builds a partial solution Sp so that the probability to visit city j can be formulated as follows:

Where N (sp) is a set of feasible components, where in nodes (i, l), l is a city that has not been visited by ants k. The parameters of α and β are control variables that are relatively dependent on how important the pheromone value is, which is inversely proportional to the heuristic information ηij, so the formulation is as follows:

Where dij is the distance between city i and city j.

**Parameter Any Colony Optimization**

The ACO algorithm involves a number of parameters that need to be set precisely. Of these, the α and β parameters are used to weigh the relative influence of pheromones and heuristic values in the construction of ant solutions. The role of these parameters in refracting the search for ants. Higher α values emphasize differences in pheromone values, and higher β values have the same effect on heuristic values. The initial value of the pheromone, τ0, has a significant effect on the convergence speed of the algorithm.

However, the recommended settings depend on the specific ACO algorithm. The evaporation rate parameter, ρ, 0≤ ρ ≤1, regulates the rate of decline in the pheromone pathway. If ρ is low, the effect of pheromone values will last longer, while a high ρ value allows quick forgetting of choices that were previously very interesting and resulted in allowing more rapid focus on new information entered into the pheromone matrix. Another parameter is the number of ants in the colony (m). For a given computing budget, such as the maximum calculation time, the number of ants is an important parameter to determine the exchange between the number of iterations that can be made and how extensive the search is in each iteration. Ant Colony's performance is very dependent on its intensity parameter settings. To find the appropriate settings for the algorithm parameters is considered a nontrivial task. Traditionally, parameter settings are mostly done by Trial and Error.

This algorithm is done using the following requirements:

* Python (<https://python.org>)
* Numpy (<https://numpy.org>)
* Matplotlib ( <https://matplotlib.org> )

1. **PYTHON :**

Python is a widely used high-level, general-purpose, interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library. Python interpreters are available for installation on many operating systems, allowing Python code execution on a wide variety of systems.

**1.1 Scripting Language** :

A scripting or script language is a programming language that supports scripts, programs written for a special run-time environment that automate the execution of tasks that could alternatively be executed one-by-one by a human operator.

Scripting languages are often interpreted (rather than compiled). Primitives are usually the elementary tasks or API calls, and the language allows them to be combined into more complex programs. Environments that can be automated through scripting include software applications, web pages within a web browser, the shells of operating systems (OS), embedded systems, as well as numerous games.

A scripting language can be viewed as a domain-specific language for a particular environment; in the case of scripting an application, this is also known as an extension language. Scripting languages are also sometimes referred to as very high-level programming languages, as they operate at a high level of abstraction, or as control languages.

* 1. **Object Oriented Programming Language :**

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A distinguishing feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self").

In OO programming, computer programs are designed by making them out of objects that interact with one another. There is significant diversity in objectoriented programming, but most popular languages are class-based, meaning that objects are instances of classes, which typically also determines their type.

* 1. **History** :

Python was conceived in the late 1980s, and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands as a successor to the ABC language (itself inspired by SETL) capable of exception handling and interfacing with the Amoeba operating system. Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, benevolent dictator for life (BDFL).



Python is an experiment in how much freedom programmers need. Too much freedomand nobody can read another's code; too little and expressiveness is endangered.”

- Guido van Rossum

**1.4 Data Type** (This is called dynamic typing) :

Data types determine whether an object can do something, or whether it just would not make sense. Other programming languages often determine whether an operation makessense for an object by making sure the object can never be stored somewhere where the operation will be performed on the object (this type system is called static typing). Python does not do that. Instead, it stores the type of an object with the object, and checks when the operation is performed whether that operation makes sense for that object Python has many native data types. Here are the important ones:

**Booleans** are either True or False.

**Numbers** can be integers (1 and 2), floats (1.1 and 1.2), fractions (1/2 and 2/3), or even complex numbers.

**Strings** are sequences of Unicode characters, e.g., an HTML document.

**Bytes and byte arrays**, e.g., a JPEG image file.

**Lists** are ordered sequences of values.

**Tuples** are ordered, immutable sequences of values.

**Sets** are unordered bags of values

**1.5 Variable :**

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Ex: counter = 100 # An integer

assignment miles = 1000.0 # A floating

point name= "John" # A string

**String** :

In programming terms, we usually call text a string. When you think of a string as a collection of letters, the term makes sense.

All the letters, numbers, and symbols in this book could be a string. For that matter, your name could be a string, and so could your address.

**Creating Strings** In Python, we create a string by putting quotes around text. For example, we could take our otherwise useless

• "hello"+"world" "helloworld" # concatenation

• "hello"\*3 "hellohellohello" # repetition

• "hello"[0] "h" # indexing

• "hello"[-1] "o" # (from end)

• "hello"[1:4] "ell" # slicing

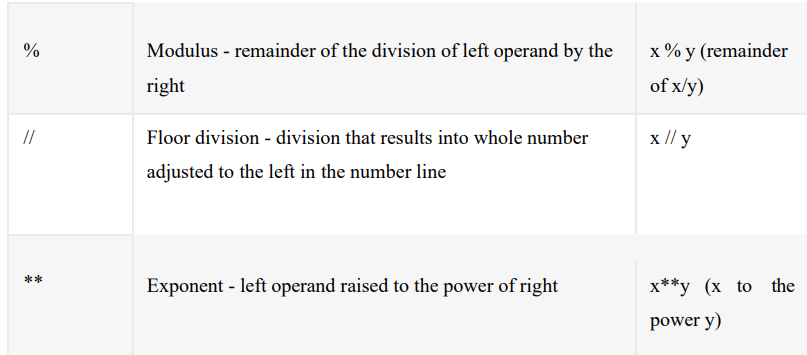
• len("hello") 5 # size

• "hello" < "jello" 1 # comparison

• "e" in "hello" 1 # search

**Operator Arithmetic :**

**Graphical user interface, application, table

Description automatically generated**

**Comparison Operator :**

Graphical user interface, text, application, email

Description automatically generated

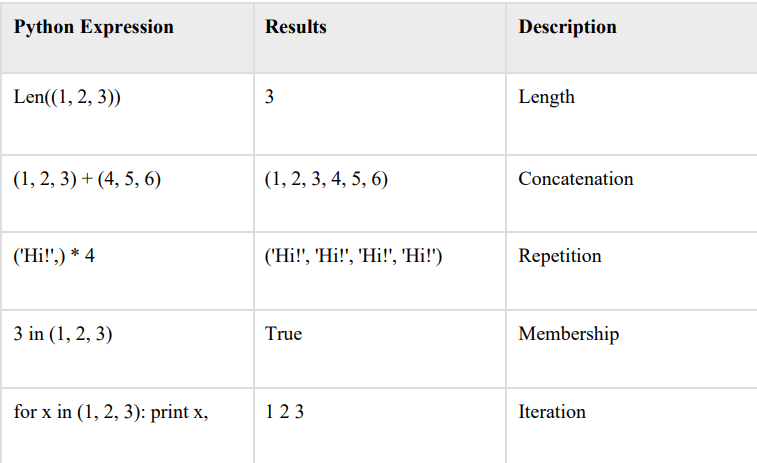
**1.6 Tuples** :

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses.

**Accessing Values in Tuples**:

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example − tup1 = ('physics', 'chemistry', 1997, 2000); tup2 = (1, 2, 3, 4, 5, 6, 7); print "tup1[0]: ", tup1[0] print "tup2[1:5]: ", tup2[1:5] When the above code is executed, it produces the following result − tup1[0]: physics tup2[1:5]: [2, 3, 4, 5].

Basic TuplesOperation Tuples respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string. In fact, tuples respond to all of the general sequence operations we used on strings in the prior pages



**Built-in Tuple Functions**

Python includes the following tuple functions –

1. **cmp(tuple1, tuple2)** Compares elements of both tuples.
2. **len(tuple)** Gives the total length of the tuple.
3. **max(tuple)** Returns item from the tuple with max value.
4. **min(tuple)** Returns item from the tuple with min value. 5
5. **tuple(seq)** Converts a list into tuple.

**1.7 List** :

The list is a most versatile datatype available in Python which can be written as a list of commas- separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type. Creating a list is as simple as putting different comma-separated values between square brackets. For example − list1 = ['physics', 'chemistry', 1997, 2000]; list2 = [1, 2, 3, 4, 5]; list3 = ["a", "b", "c", "d"];

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

**Accessing Values in Lists:** To access values in lists, use the square brackets forslicing along with the index or indices to obtain value available at that index.

For example − list1 = ['physics', 'chemistry', 1997, 2000];

list2 = [1, 2, 3, 4, 5, 6, 7];

print "list1[0]: ", list1[0]

print "list2[1:5]: ", list2[1:5]

Output: list1[0]: physics

list2[1:5]: [2, 3, 4, 5]

**Update**: list = ['physics', 'chemistry', 1997, 2000];

print "Value available at index 2: "

print list [2] list [2] = 2001;

print "New value available at index 2: "

print list [2]

Output: Value available at index 2 :

1997 New value available at index 2:

2001

**Delete:** list1 = ['physics', 'chemistry', 1997, 2000];

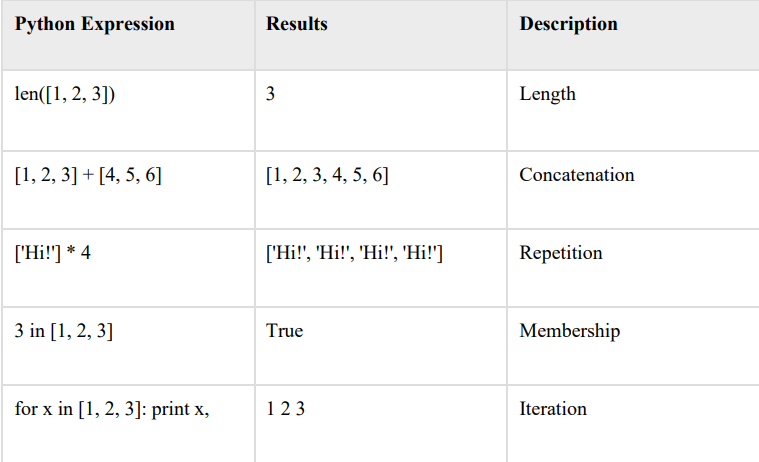
print list1 del list1[2];

print "After deleting value at index 2: "

print list1 ['physics', 'chemistry', 1997, 2000]

Output: After deleting value at index 2 :

['physics', 'chemistry', 2000]

**Basic ListOperation**

**Built-in List Functions & Methods:**

1. **cmp(list1, list2)** Compares elements of both lists.
2. **len(list)** Gives the total length of the list.
3. **max(list)** Returns item from the list with max value.
4. **min(list)** Returns item from the list with min value.
5. **list(seq)** Converts a tuple into list.

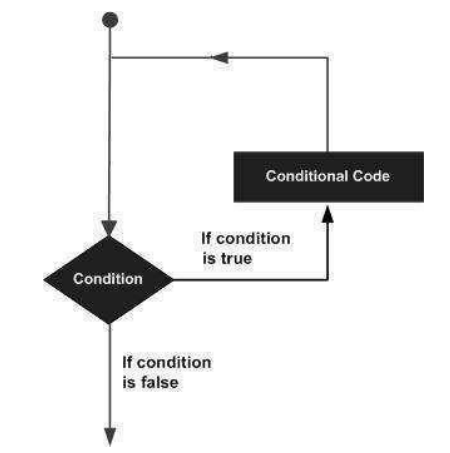
**Python includes following list methods :**

1. **list.append(obj)** Appends object obj to list
2. **list.count(obj)** Returns count of how many times obj occurs in list
3. **list.extend(seq)** Appends the contents of seq to list
4. **list.index(obj)** Returns the lowest index in list that obj appears
5. **list.insert(index, obj)** Inserts object obj into list at offset index
6. **list.pop(obj=list[-1])** Removes and returns last object or obj from list
7. **list.remove(obj)** Removes object obj from list
8. **list.reverse()** Reverses objects of list in place
9. **list.sort([func])** Sorts objects of list, use compare func if given

**1.8 Loop definition :**

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement –



Python programming language provides following types of loops to handle looping requirements.

**while loop :-** Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

**for loop ;-** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

**nested loops :-** You can use one or more loop inside any another while, for or do..while loop.

**Loop Example:**

For Loop:

>>> for mynum in [1, 2, 3, 4, 5]:

print ("Hello", mynum )

Hello 1

Hello 2

Hello 3

Hello 4

Hello 5

While Loop:

>>> count = 0 >>while(count< 4):

print 'The count is:', count count = count + 1

The count is: 0

The count is: 1

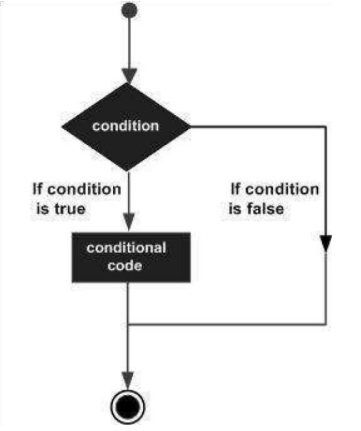
The count is: 2

The count is: 3

**1.9 Conditional Statements :**

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.



Python programming language provides following types of decision making statements. Click the following links to check their detail.

**if statements :-** An **if statement** consists of a boolean expression followed by one or more statements.

**if...else statements** :- An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.

**nested if statements** :- You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

**Example**: If Statement:

a=33

b=20

If b>a:

print(“b”)

If...Else Statement:

a=200

b=33

if b>a:

print(“b is greater than a”)

else:

print(“a is greater than b”)

**1.10 Function** :

Function blocks begin with the keyword def followed by the function name and parentheses ( ( )). Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

The first statement of a function can be an optional statement - the documentation string of the function.

The code block within every function starts with a colon (:) and is indented.

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

**Syntex:**

Def

functionname(parameters):

“function\_docstring”

Function\_suite

Return[expression]

**Example:**

Def printme(str):

“this print a passed string into this function”

print str return

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print str return;

# Now you can call printme function

printme("I'm first call to user defined function!")

printme("Again second call to the same function")

1. **NUMPY :**

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

* NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
* The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
* NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python’s built-in sequences.
* A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today’s scientific/mathematical Python-based software, just knowing how to use Python’s built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

The points about sequence size and speed are particularly important in scientific computing. As a simple example, consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data are stored in two Python lists, a and b, we could iterate over each element:

c **=** **[]**

**for** i **in** range**(**len**(**a**)):**

c**.**append**(**a**[**i**]\***b**[**i**])**

This produces the correct answer, but if a and b each contain millions of numbers, we will pay the price for the inefficiencies of looping in Python. We could accomplish the same task much more quickly in C by writing (for clarity we neglect variable declarations and initializations, memory allocation, etc.)

**for** **(**i **=** **0;** i **<** rows**;** i**++)** **{**

c**[**i**]** **=** a**[**i**]\***b**[**i**];**

**}**

This saves all the overhead involvedin interpreting the Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python. Furthermore, the coding work required increases with the dimensionality of our data. In the case of a 2-D array, for example, the C code (abridged as before) expands to

**for** **(**i **=** **0;** i **<** rows**;** i**++)** **{**

**for** **(**j **=** **0;** j **<** columns**;** j**++)** **{**

c**[**i**][**j**]** **=** a**[**i**][**j**]\***b**[**i**][**j**];**

**}**

**}**

NumPy gives us the best of both worlds: element-by-element operations are the “default mode” when an ndarray is involved, but the element-by-element operation is speedily executed by pre-compiled C code. In NumPy

c **=** a **\*** b

does what the earlier examples do, at near-C speeds, but with the code simplicity we expect from something based on Python. Indeed, the NumPy idiom is even simpler! This last example illustrates two of NumPy’s features which are the basis of much of its power: vectorization and broadcasting.

**2.1 WHY NUMPY IS FAST ?**

Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just “behind the scenes” in optimized, pre-compiled C code. Vectorized code has many advantages, among which are:

* vectorized code is more concise and easier to read
* fewer lines of code generally means fewer bugs
* the code more closely resembles standard mathematical notation (making it easier, typically, to correctly code mathematical constructs)
* vectorization results in more “Pythonic” code. Without vectorization, our code would be littered with inefficient and difficult to read for loops.

Broadcasting is the term used to describe the implicit element-by-element behavior of operations; generally speaking, in NumPy all operations, not just arithmetic operations, but logical, bit-wise, functional, etc., behave in this implicit element-by-element fashion, i.e., they broadcast. Moreover, in the example above, a and b could be multidimensional arrays of the same shape, or a scalar and an array, or even two arrays of with different shapes, provided that the smaller array is “expandable” to the shape of the larger in such a way that the resulting broadcast is unambiguous. For detailed “rules” of broadcasting see [Broadcasting](https://numpy.org/doc/stable/user/basics.broadcasting.html#basics-broadcasting).

* 1. **WHO ELSE USES NUMPY ?**

NumPy fully supports an object-oriented approach, starting, once again, with ndarray. For example, ndarray is a class, possessing numerous methods and attributes. Many of its methods are mirrored by functions in the outer-most NumPy namespace, allowing the programmer to code in whichever paradigm they prefer. This flexibility has allowed the NumPy array dialect and NumPy ndarray class to become the de-facto language of multi-dimensional data interchange used in Python.

**3 . MATPLOT LIBRARY**

**Matplotlib is a cross-platform, data visualization and graphical plotting library for Python and its numerical extension NumPy. As such, it offers a viable open source alternative to MATLAB. Developers can also use matplotlib’s APIs (Application Programming Interfaces) to embed plots in GUI applications.**

A Python matplotlib script is structured so that a few lines of code are all that is required in most instances to generate a visual data plot. The matplotlib scripting layer overlays two APIs:

The pyplot API is a hierarchy of Python code objects topped by matplotlib.pyplot

An OO (Object-Oriented) API collection of objects that can be assembled with greater flexibility than pyplot. This API provides direct access to Matplotlib’s backend layers.

Matplotlib and Pyplot in Python

The pyplot API has a convenient MATLAB-style stateful interface. In fact, matplotlib was originally written as an open source alternative for MATLAB. The OO API and its interface is more customizable and powerful than pyplot, but considered more difficult to use. As a result, the pyplot interface is more commonly used, and is referred to by default in this article.

Understanding matplotlib’s pyplot API is key to understanding how to work with plots:

**matplotlib.pyplot.figure: Figure** is the top-level container. It includes everything visualized in a plot including one or more **Axes**.

**matplotlib.pyplot.axes**:**Axes** contain most of the elements in a plot: **Axis, Tick, Line2D, Text,**etc., and sets the coordinates. It is the area in which data is plotted. Axes include the X-Axis, Y-Axis, and possibly a Z-Axis, as well.

For more information about the pyplot API and interface, refer to [What Is Pyplot In Matplotlib](https://www.activestate.com/resources/quick-reads/how-to-display-a-plot-in-python/)

Installing Matplotlib

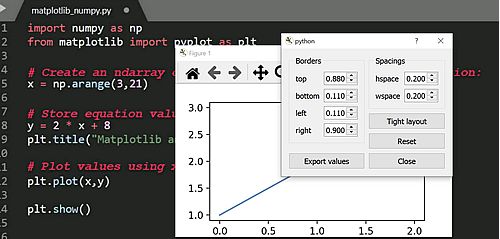
Matplotlib and its dependencies can be downloaded as a binary (pre-compiled) package from the Python Package Index (PyPI), and installed with the following command:

python -m pip install matplotlib

Matplotlib is also available as uncompiled source files. Compiling from source will require your local system to have the appropriate compiler for your OS, all dependencies, setup scripts, configuration files, and patches available. This can result in a fairly complex installation. Alternatively, consider using the [ActiveState Platform](https://platform.activestate.com/create-account) to automatically build matplotlib from source and package it for your OS.

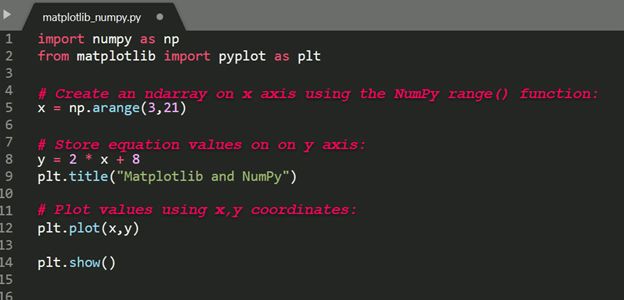
**3.1 Matplotlib UI Menu**

When matplotlib is used to create a plot, a User Interface (UI) and menu structure are generated. The UI can be used to customize the plot, as well as to pan/zoom and toggle various elements.

****

**3.2 Matplotlib and NumPy**

Numpy is a package for scientific computing. Numpy is a required dependency for matplotlib, which uses numpy functions for numerical data and multi-dimensional arrays as shown in the following code snippet:

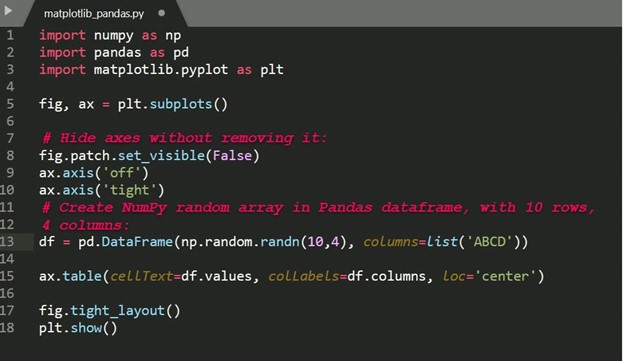


The source code for this example is available in the [Matplotlib: Plot a Numpy Array](https://www.activestate.com/resources/quick-reads/what-is-matplotlib-in-python-how-to-use-it-for-plotting/#numpyarray) section  further down in this article.

**3.3 Matplotlib and Pandas**

Pandas is a library used by matplotlib mainly for data manipulation and analysis. Pandas provides an in-memory 2D data table object called a Dataframe. Unlike numpy, pandas is not a required dependency of matplotlib.

Pandas and numpy are often used together, as shown in the following code snippet:



The source code for this example is available in the [Matplotlib: Plot a Pandas Dataframe](https://www.activestate.com/resources/quick-reads/what-is-matplotlib-in-python-how-to-use-it-for-plotting/#pandas)section  further down in this article.

How to Create Matplotlib Plots

**This section shows how to create examples of different kinds of plots with matplotlib.**

**3.4 Matplotlib Line Plot**

In this example, pyplot is imported as plt, and then used to plot three numbers in a straight line:

import matplotlib.pyplot as plt

*# Plot some numbers:*

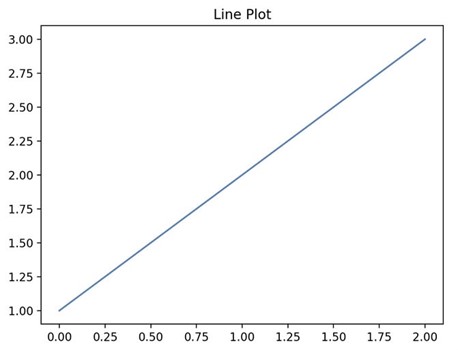
plt.plot([1, 2, 3])

plt.title(”Line Plot”)

*# Display the plot:*

plt.show()

***1.****Line plot generated by Matplotlib:*



**3.5 Matplotlib Pie Plot**

**In this example, pyplot is imported as plt, and then used to create a chart with four sections that have different labels, sizes and colors:**

import matplotlib.pyplot as plt

*# Data labels, sizes, and colors are defined:*

labels = 'Broccoli', 'Chocolate Cake', 'Blueberries', 'Raspberries'

sizes = [30, 330, 245, 210]

colors = ['green', 'brown', 'blue', 'red']

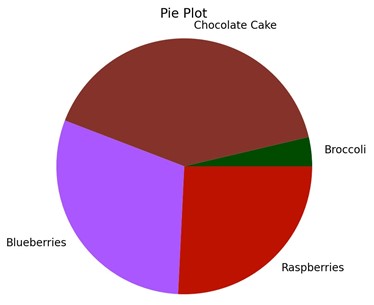
*# Data is plotted:*

plt.pie(sizes, labels=labels, colors=colors)

plt.axis('equal')

plt.title(“Pie Plot”)

plt.show()

***2.****Pie plot generated by Matplotlib:*

**3.6 Matplotlib Bar Plot**

**In this example, pyplot is imported as plt, and then used to plot three vertical bar graphs:**

import matplotlib.pyplot as plt

import numpy as np

*# Create a* [*Line2D*](https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.lines.Line2D.html#matplotlib.lines.Line2D) *instance with x and y data in sequences xdata, ydata:*

*# x data:*

xdata=['A','B','C']

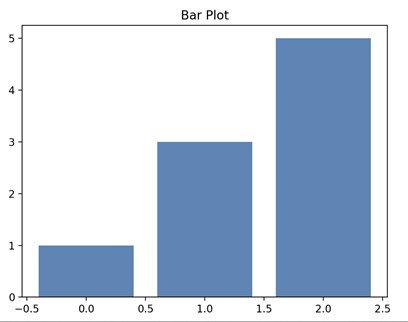
*# y data:*

ydata=[1,3,5]

plt.bar(range(len(xdata)),ydata)

plt.title(“Bar Plot”)

plt.show()

1. [](https://www.activestate.com/resources/quick-reads/what-is-matplotlib-in-python-how-to-use-it-for-plotting/#numpyarray)*Bar plot generated by Matplotlib:*
2. **Implementing Ant Colony Optimization (ACO) algorithm for a given Symmetric traveling salesman problem (TSP)**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Ant Colony Optimization (ACO)** It is an optimization algorithm used to find the shortest path between points or nodes. It is developed by observing the behaviour of ants when they follow a path to their food source. Ants are essentially blind so they follow pheromone trails left behind by other ants on the path. This algorithm follows the same approach by using the probability of going to the next node as the distance to the node and the amount of pheromones.

All over the world especially in Over the last decade, vehicle population has dramatically increased (Jabbarpour et al., 2014). This large number of vehicles leads to heavy traffic congestion, air pollution, high fuel consumption and consequent economic issues (Narzt et al., 2010). In 2010, the American people faced a lot of difficulties due to vehicle congestion which forced their government to spend 101 billion dollars on the purchase of extra fuel (Jabbarpour et al., 2014). Based on a report by Texas A&M Transportation Institute (Jabbarpour et al., 2014), it is estimated that fuel consumption will rise up to 2.5 billion gallons (from 1.9 billion gallons in 2010) with a cost of 131 billion dollars in 2015.

Building new, high-capacity streets and highways can mitigate some of the aforementioned problems. Nevertheless, this solution is very costly, time consuming and in most cases, impossible because of space limitations. On the other hand, optimal usage of the existing roads and streets capacity can lessen the congestion problem in large cities at a lower cost. Applying bio-inspired algorithm, promises to have potential in solving these problems. One such algorithm is the Ant Colony Optimization. In the case of ACO algorithms the theoretical analyses of their runtime behavior has been started only recently.

We increase the theoretical understanding of ACO algorithms by investigating their runtime behavior on the well-known traveling salesperson (TSP) problem. For ACO the TSP problem is the first problem where this kind of algorithms has been applied. Therefore, it seems to be natural to study the behavior of ACO algorithms for the TSP problem from a theoretical point of view in a rigorous manner. ACO algorithms are inspired by the behavior of ants to search for a shortest path between their nest and a common source of food. It has been observed that ants find such a path very quickly by using indirect communication via pheromones. This observed behavior is put into an algorithmic framework by considering artificial ants that construct solutions for a given problem by carrying out random walks on a so-called construction graph. The random walk (and the resulting solution) depends on pheromone values that are values on the edges of the construction graph. The probability of traversing a certain edge depends on its pheromone value. It is a relatively novel meta- heuristic technique and has been successfully used in many applications especially problems in combinatorial optimization. ACO algorithm models the behavior of real ant colonies in establishing the shortest path between food sources and nests. Ants can communicate with one another through chemicals called pheromones in their immediate environment. The ants release pheromone on the ground while walking from their nest to food and then go back to the nest. The ants move according to the amount of pheromones, the richer the pheromone trail on a path is, the more likely it would be followed by other ants. So a shorter path has a higher amount of pheromone in probability, ants will tend to choose a shorter path. Through this mechanism, ants will eventually find the shortest path. Artificial ants imitate the behavior of real ants, but can solve much more complicated problem than real ants can. Consider Fig. 1A Ants arrive at a decision point in which they have to decide whether to turn left or right (Dorigo and Gambardella, 1997).

Since they have no clue about which is the best choice, they choose randomly. It can be expected that, on average, half of the ants decide to turn left and the other half to turn right. This happens both to ants moving from left to right (those whose name begins with an L) and to those moving from right to left (name begins with a R). Figs. 1B and 1C show what happens in the immediately following instants, supposing all ants walk at approximately the same speed. The number of dashed lines is roughly proportional to the amount of pheromone that the ants have deposited on the ground. Since the lower path is shorter than the upper one, more ants will visit it on average, and therefore pheromone accumulates faster. After a short transitory period the difference in the amount of pheromone on the two path is sufficiently large so as to influence the decision of new ants coming into the system (this is shown by Fig. 1D).

From now on, new ants will prefer in probability to choose the lower path, since at the decision point they perceive a greater amount of pheromone on the lower path (Dorigo and Gambardella, 1997). This in turn increases, with a positive feedback effect, the number of ants choosing the lower, and shorter, path. Very soon all ants will be using the shorter path. The travelling salesman problem (TSP) is the problem of finding a shortest closed tour which visits all the cities in a given set. In this article we will restrict attention to TSPs in which cities are on a plane and a path (edge) exists between each pair of cities (i.e., the TSP graph is completely connected).

Given a set of n nodes and distances for each pair of nodes, find a roundtrip of minimal total length visiting each node exactly once. The distance from node i to node j is the same as from node j to node i.

**Steps**

1. **Initialize Ants**

First its required to select a given (or arbitrary) number of ants, placed in random positions of the given TSP space.

1. **Ants moving via probability**

Then for each of the ants we complete a closed path i.e. from start, covering all the nodes and without repeating any of the nodes. To move an ant from one node to the next we use the following formula.

****

1. **Deposit of Pheromones**

When an ant moves from a node to the next, it leaves a trail for the next ant to follow, the more ants follow the same path, the stronger the pheromone trail gets. Pheromone trails are incremented by.



Where (delta\_tau) is a prameter and is the iteration number.

1. **Evaporation of Pheromones**

After each iteration the pheromones also tend to evaporate. The evaporation of pheromones are given as.

where (rho) is the evaporation rate.

1. **Ending condition**

We can use any condition to terminate the search, such as a distance below a certain threshold. In this implementation, the ending condition will be the number of iterations.

**Implementation**

Initially we import the library.py and matplotlib, as it follows:

In [1]: **from** library **import** **\***

**import** matplotlib.pyplot **as** plt

### **TSP file**

A **Symmetric traveling salesman problem (TSP)** file has the following structure:

txt

1- NAME : <string>

2- TYPE : <string>

3- COMMENT : <string>

4- DIMENSION : <integer>

5- EDGE WEIGHT TYPE : <string>

6- NODE COORD SECTION : <integer> <real> <real>

7- EOF

1- Identifies the data file

2- Specifies the type of the data (TSP: Data for a symmetric traveling salesman problem)

3- Additional comments (usually the name of the contributor or creator of the problem instance is given here).

4- For a TSP the dimension is the number of its nodes

5- Specifies how the edge weights (or distances) are given (EUC 2D: Weights are Euclidean distances in 2-D)

6- Node coordinates are given in this section. Each line is of the form

7- Terminates the input data. This entry is optional.

As some functions has been defined in libraries.py, we can call a method that reads our TSP file and store its data in order to use later. To start, we will use the **The 100-city problem A** (kroA100.tsp)

In [2]: *# Get TSP data*

TSP **=** getTspData('data/kroA100.tsp')

*# Display TSP file headers*

displayTspHeaders(TSP)

Name: kroA100

Type: TSP

Comment: 100-city

Dimension: 100

Edge Weight Type: EUC\_2D

### **Space**

We can use now its coordenates pairs to plot nodes, this representation is what is called **space**.

In [3]:

In [3]:*# Get Space*

space **=** np**.**array(TSP['node\_coord\_section'])

*# Plot nodes*

plt**.**scatter(space[:, 0], space[:, 1], s **=** 15)

*# Plot properties*

plt**.**title('Space {}'**.**format(TSP['name']))

plt**.**xlabel('Latitude')

plt**.**ylabel('Longitude')

*# Show plot*

plt**.**show()

Chart, scatter chart

Description automatically generated plt**.**close()

Then we call the algorithm as it follows. By default our algorithm parameters will be:

{numpy**.**ndarray} space **--** The space

{int} iterations {80} **--** Number of iterations (Ending condition)

{int} colony {50} **--** Number of ants **in** the colony

{float} alpha {1.0} **--** Alpha algorithm parameter, more **or** less weight to a selected distance

{float} beta {1.0} **--** Beta algorithm parameter, more **or** less weight to a selected distance

{float} del\_tau {1.0} **--** Delta Tau algorithm parameter, pheromones releasing rate

{float} rho {0.5} **--** Rho algorithm parameter, pheromones evaporation rate

But they can be changed passing them as argument to the runAcoTsp() function like:

iterations **=** 100

colony **=** 25

alpha **=** 1.2

beta **=** 1.5

del\_tau **=** 2

rho **=** 0.2

*# Call passing arguments*

min\_path, min\_distance **=** runAcoTsp(space, iterations, colony, alpha, beta, del\_tau, rho)

In this case we will run it as default, by using runAcoTsp(space).

In [4]:*# Run ACO*

min\_path, min\_distance **=** runAcoTsp(space)

*# Plot path*

plt**.**scatter(space[:, 0], space[:, 1], marker**=**'o', s**=**15)

plt**.**plot(space[min\_path, 0], space[min\_path, 1], c**=**'g',

linewidth**=**0.8, linestyle**=**"--")

*# Plot properties*

plt**.**suptitle('Mininum Path')

plt**.**title('For a minimum distance of {}'**.**format(min\_distance),

fontsize **=** 10)

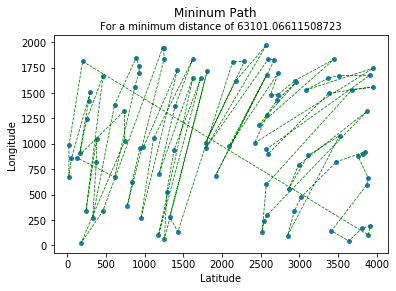
plt**.**xlabel('Latitude')

plt**.**ylabel('Longitude')

*# Show plot*

plt**.**show()

plt**.**close()



As it shows in last plot, after perform the number of iteration (our ending condition), the shortes path is represented by the green line, and it minimum distance is calculated as well.

In order to watch the algorith initial points randomness, we will perform it again, but a total of 3 times to to get an average minimum distance.

In [5]: *# Vars*

n **=** 3

average **=** 0

*# Repeat*

**for** i **in** range(n):

*# Call*

min\_path, min\_distance **=** runAcoTsp(space)

average **+=** min\_distance

*# Plot path*

plt**.**scatter(space[:, 0], space[:, 1], marker**=**'o', s**=**15)

plt**.**plot(space[min\_path, 0], space[min\_path, 1], c**=**'g', linewidth**=**0.8, linestyle**=**"--")

*# Plot properties*

plt**.**suptitle('Mininum Path for {}'**.**format(TSP['name']))

plt**.**title('Result #{} of {} for a minimum distance of {}'**.**format(i **+** 1, n, min\_distance), fontsize **=** 10)

plt**.**xlabel('Latitude')

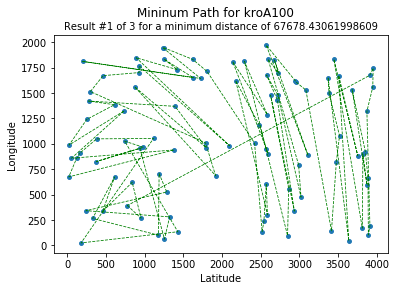
plt**.**ylabel('Longitude')

plt**.**show()

plt**.**close()

*# Show Average*

print('Min Distance Average for the last {} results is {}'**.**format(n, average**/**n))



Diagram

Description automatically generated with medium confidenceDiagram

Description automatically generated

Min Distance Average for the last 3 results is 64026.56311240327

**Extras**

The same functions can be used to and additional results for **52 locations in Berlin** (berlin52.tsp) by Groetschel but using a differente set of algorithm parameters.

In [6]:*# Get TSP data*

TSP **=** getTspData('data/berlin52.tsp')

*# Display TSP file headers*

displayTspHeaders(TSP)

*# Get Space*

space **=** np**.**array(TSP['node\_coord\_section'])

*# Plot nodes*

plt**.**scatter(space[:, 0], space[:, 1], s **=** 15)

*# Plot properties*

plt**.**title('Space {}'**.**format(TSP['name']))

plt**.**xlabel('Latitude')

plt**.**ylabel('Longitude')

*# Show plot*

plt**.**show()

plt**.**close()

*# Algorithm Parameters*

iterations **=** 50

colony **=** 25

alpha **=** 1

beta **=** 1

del\_tau **=** 1.5

rho **=** 0.5

*# Vars*

average **=** 0

*# Repeat*

**for** i **in** range(n):

*# Run*

min\_path, min\_distance **=** runAcoTsp(space, iterations, colony, alpha, beta, del\_tau, rho)

average **+=** min\_distance

*# Plot path*

plt**.**scatter(space[:, 0], space[:, 1], marker**=**'o', s**=**15)

plt**.**plot(space[min\_path, 0], space[min\_path, 1], c**=**'g', linewidth**=**0.8, linestyle**=**"--")

*# Plot properties*

plt**.**suptitle('Mininum Path for {}'**.**format(TSP['name']))

plt**.**title('Result #{} of {} for a minimum distance of {}'**.**format(i **+** 1, n, min\_distance), fontsize **=** 10)

plt**.**xlabel('Latitude')

plt**.**ylabel('Longitude')

plt**.**show()

plt**.**close()

*# Show Average*

print('Min Distance Average for the last {} results is {}'**.**format(n, average**/**n))

Name: berlin52

Type: TSP

Comment: 52

Dimension: 52

Edge Weight Type: EUC\_2D

Chart, scatter chart

Description automatically generated

Graphical user interface

Description automatically generated

A picture containing chart

Description automatically generated

Chart

Description automatically generated

Min Distance Average for the last 3 results is 14708.733453124236

**CONCLUSION**

In this paper, we study the coordinated delivery system between truck and drone with ACO algorithm. We show that substantial savings are possible such as minimum delivery distance compared to the truck-only solution. This study is the first paper in the literature with the application of ACO algorithm in coordinated use of truck and drone. Therefore, the potential for future research areas is the development and implementation of ACO algorithm or other methods. The drone should arrive on the truck at the delivery beginning and end. A natural extension of the problem is to consider the possibility to recharge the drone during operations. However, it is unclear how to model a more flexible charging policy because it requires monitoring battery life in multiple operations.

**REFERENCES**

[1] Ha, Quang Minh, et al. "On the min-cost traveling salesman problem with drone." Transportation Research Part C: Emerging Technologies 86 (2018): 597-621.

[2] A. Choi-Fitzpatrick, D. Chavarria, E. Cychosz, J. P. Dingens, M. Duffey, K. Koebel, S. Siriphanh, M. Yurika Tulen, H. Watanabe, T. Juskauskas, J. Holland and L. Almquist, Up in the Air: A Global Estimate of Non-Violent Drone Use 2009-2015, Joan B. Kroc School of Peace Studies at Digital@USanDiego, University of SanDiego, 2016.

[3] R. Clarke, "Understanding the drone epidemic", Computer Law & Security Review, 30 (2014), pp. 230-246.

[4] Yurek, Emine Es, and H. Cenk Ozmutlu. "A decomposition-based iterative optimization algorithm for traveling salesman problem with drone." Transportation Research Part C: Emerging Technologies 91 (2018): 249-262.

[5] D. Esler, Drone Revolution, Business & Commercial Aviation, 2015.

[6] S. French, Drone delivery is already here — and it works, Marketwatch, 2015.

[7] O. Khazan, A Drone to Save the World, The Atlantic, 2016.

[8] M. Prigg, The ambulance drone that could save your life, Daily Mail, 2014.

[9] D. A. Raffaello, "Guest Editorial Can Drones Deliver?", IEEE Transactions on Automation Science and Engineering, 11 (2014).

[10] A. Raptopoulos, No roads? There's a drone for that, TEDGlobal 2013, 2013.Jones, personal communication, 1992.

[11] Murray, Chase C., and Amanda G. Chu. "The flying sidekick traveling salesman problem: Optimization of drone-assisted parcel delivery." Transportation Research Part C: Emerging Technologies 54 (2015): 86-109.

[12] Agatz, Niels, Paul Bouman, and Marie Schmidt. "Optimization approaches for the traveling salesman problem with drone." Transportation Science 52.4 (2018): 965-981.

[13] Ha, Q.M., Deville, Y., Pham, Q.D. and Ha, M.H., 2015. Heuristic Methods for The traveling Salesman Problem with Drone.

[14] Ha, Q.M., Deville, Y., Pham, Q.D., Ha, M.H., 2018. On the min-cost traveling salesman problem with drone. Transport. Res. Part C: Emerg. Technol. 86, 597–621.

[15] Ponza, A., 2016. Optimization of Drone-Assisted Parcel Delivery. University of Padova. IMSS’19 Sakarya University - Sakarya/Turkey, 9-11 September 2019, pp. 308-316 -315- Koç et al. Ant Colony Optimization For Solving The Traveling Salesman Proble...

[16] Wang, X., Poikonen, S., Golden, B., 2017. The vehicle routing problem with drones: several worst-case results. Optim. Lett. 11 (4), 679–697.

[17] Ferrandez, S.M., Harbison, T., Weber, T., Sturges, R., Rich, R., 2016. Optimization of a truckdrone in tandem delivery network using k-means and genetic algorithm. J. Indus. Eng. Manage., 9(2), 374–388.

[18] Stützle, Thomas, and Marco Dorigo. "ACO algorithms for the traveling salesman problem." Evolutionary algorithms in engineering and computer science (1999): 163-183.

[19] Agatz, N., Bouman, P. and Schmidt, M., 2016. Optimization Approaches for the Traveling Salesman Problem with Drone. ERIM Report Series Reference No. ERS-2015-011-LIS.