Kafrelsheikh University
Faculty of Computers and Information
Computer Science Department

“Flying Box” Drone Delivery System

Graduation Project

By:

Abdelrahman Mohammed Ebied
Ahmed Emad Abdelhafez
Alaa-Elden Abdelfattah ElSayed
Hassan Ahmed ElDesouky
Mahmoud Abdullah Ibrahim
Norhan Ehab Mabrouk
Yahia Ashraf Aglan

Under Supervision of:

Dr. Haitham Elwahsh

June, 2021

Acknowledgment

Working on this project has been a truly life-changing experience for us and it would not have been possible to do without the support and guidance that We received from many people.

We would like to first say a very big thank you to our supervisor Dr. Haitham Elwahsh for all the support and encouragement he gave us, during both the long months We spent undertaking our labwork in University and also the time We spent on the remote. Without his guidance and constant feedback, this project of our B.Sc. would not have been achievable.

Many thanks also to Dr. Enjy A. for all her guidance, she provided us with many insights and useful examples, which proved to be of immense help in the successful completion of this project.

This B.Sc. study would not have been possible without the corporation and support extended by the staff of the CS Department. Their patience during the numerous focus group discussions as well the school courses that We undertook between 2017 to 2021 is very much appreciated. We are especially grateful to them for letting us undertake the Drone-Delivery project during a period that was incredibly difficult for them.

We also express our gratitude to Dr. Osama M. Abu Zaid, without him this project would not have been possible, and also for his friendship and the warmth he extended to us during our time in the college and for always making us feel so welcome.

We are also very grateful to our colleagues who all helped us with invaluable advice and feedback on our research and for always being so supportive of our work in numerous ways during various stages of our study, and of course a big thank you to A. Samir, especially with the mammoth task of doing the final formatting and printing of this thesis.

Abstract

The Internet has profoundly changed the way we buy things, but the online shopping of today is likely not the end of that change; after each purchase we still need to wait multiple days for physical goods to be carried to our doorstep.

Also during the current pandemic, it's not safe to interact directly with others, So when we buy things online, It's expected to receive the product by taking it from the delivery person's hands with a massive human interaction which is not likely now.

This is where drones come in autonomous, electric vehicles delivering online purchases. Flying, so never stuck in traffic. As drone technology improves every year, there remains a major issue: how do we manage, coordinate, and populate all those drones?

the drones in order to make the delivery process as efficient and speed as we can.

The solution includes two successive optimizations. The first one is an assignment problem that assigns warehouses to service orders. The second one is a vehicle routing problem that provides a sequence of customers to serve.

A combination of participatory methods and conventional methods was used. We first examined the major warehouses and drones, thereafter assigned the targets between households within and between warehouses and between houses respondents using Machine Learning.

Finally, OR-Tools analyses were used to relate the coordinates and warehouses matching indices to a number of explanatory variables such as the location of customers. My findings suggest that in terms of resource management, the use of indices and tools such as those developed under this study could prove to be useful in respect to better targeting the far groups among all the houses. This in turn would contribute towards the overall success and long-term sustainability of delivery management by drones.

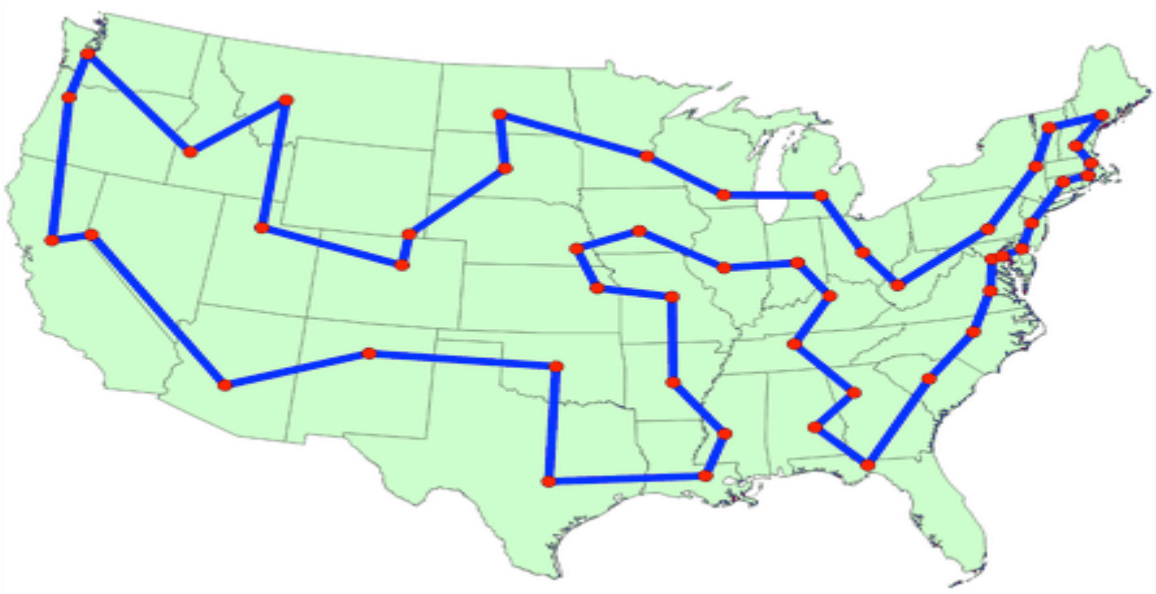
Introduction

Study Of Traveling Salesman

In recent years, TSPs (Traveling Salesman Problem) have become one of the most intensively investigated optimization problems, and many consider it to be the prototypical combinatorial optimization problem that has made it possible to develop new search algorithms, develop a new complexity theory, and analyze solution space and search space.

The traveling salesman problem asks the following question:
given a set of cities and a cost to travel from one city to another, seeks to identify the tour that will allow a salesman to visit each city only once, starting and ending in the same city, at the minimum cost.

Despite this simple problem statement, solving the TSP is difficult since it belongs to the class of NP-complete problems.



In addition to its theoretical appeal, the TSP has a variety of practical applications. Typical applications in operations research include vehicle routing, computer wiring, cutting wallpaper, and job sequencing.

The origins of the traveling salesman problem are obscure, it is mentioned in an 1832 manual for the traveling salesman, which included example tours of 45 German cities but gave no mathematical consideration.

Hamilton and Thomas Kirkman devised mathematical formulations of the problem in the 1800s, The general form was first studied in Vienna and Harvard in the 1930s by Karl Menger.

The problem became increasingly popular in the 1950s and 1960s. George Dantzig, Delber R. Fulkerson, and Selmer M. Johnson at the RAND Corporation in Santa Monica, California, resolved the 48 state problem by formulating it as a linear programming problem.

In 1959, Jillian Beardwood, J.H. Halton, and John Hammersley published an article entitled "The Shortest Path Through Many Points" in the journal of the Cambridge Philosophical Society. The Beardwood–Halton–Hammersley theorem provides a practical solution to the traveling salesman problem. The authors derived an asymptotic formula to determine the length of the shortest route for a salesman who starts at a home or office and visits a fixed number of locations before returning to the start.

In the following decades, the problem was studied by many researchers from mathematics, computer science, chemistry, physics, and other sciences. In the 1960s, however, a new approach was created, that instead of seeking optimal solutions would produce a solution whose length is provably bounded by a multiple of the optimal length.

In 1976, Christofides and Serdyukov independently of each other made a big advance in this direction, the Christofides-Serdyukov algorithm yields a solution that, in the worst case, is at most 1.5 times longer than the optimal solution. The algorithm was simple and quick, so many hoped it would provide a near-optimal solution method. However, the hoped-for improvement was not immediately realized, and Christofides-Serdyukov remained the best worst-case scenario method until 2011 when an algorithm with (only very) slight improvements was developed for the subset of "graphical" TSPs.
This tiny improvement was extended to include the full TSP (metric) in 2020.

Vehicle Routing Problem's Solutions

The Simplest Solution: Brute Force

The most direct solution would be to try all permutations (ordered combinations) and see which one is the cheapest (using brute-force search). The running time for this approach lies within a polynomial factor of O(n!), the factorial of the number of cities, so this solution becomes impractical even for only 20 cities.
The Brute-Force Algorithm is optimal but inefficient. It is guaranteed to find a solution, but it may take an unreasonably long time to do so.

Dynamic Programming Approach

In this approach, we take a subset of N the required cities that need to be visited, distance among the cities $dist$, and starting city as inputs. Each city is identified by a unique city id like $\{1, 2, 3,...., n\}$.

Initially, all cities are unvisited, and the visit starts from the city s. We assume that the initial traveling cost is equal to 0. Next, the TSP distance value is calculated based on a recursive function. If the number of cities in the subset is two, then the recursive function returns their distance as a base case.

On the other hand, if the number of cities is greater than 2, then we'll calculate the distance from the current city to the nearest city, and the minimum distance among the remaining cities is calculated recursively.

Finally, the algorithm returns the minimum distance as a TSP solution.
In this approach for TSP, the number of possible subsets can be at most $N \times 2^N$. Each subset can be solved in $O(N)$ times. Therefore, the time complexity of this algorithm would be $O(N^2 \times 2^N)$.

Machine Learning Approach

As Machine Learning (ML) and deep learning have popularized, several research groups have started to use ML to solve combinatorial optimization problems, such as the Travelling Salesman Problem (TSP). Based on deep (reinforcement) learning, new models and architecture for the TSP have been successively developed and have gained increasing performances.

At the time of writing, state-of-the-art models provide solutions to TSP instances of 100 cities that are roughly 1.33% away from optimal solutions. However, despite these apparently positive results, the performances remain far from those that can be achieved using a specialized search procedure.

The Nearest Neighbour Algorithm

The nearest neighbor algorithm was one of the first algorithms used to solve the traveling salesman problem approximately. In that problem, the salesman starts at a random city and repeatedly visits the nearest city until all have been visited.

The nearest neighbor algorithm is easy to implement and executes quickly, but it can sometimes miss shorter routes which are easily noticed with human insight, due to its "greedy" nature. As a general guide, if the last few stages of the tour are comparable in length to the first stages, then the tour is reasonable; if they are much greater, then it is likely that much better tours exist.
In the worst case, the algorithm results in a tour that is much longer than the optimal tour.

The Branch and Bound Method

To solve a problem, this method breaks it into several subproblems. It involves solving a series of subproblems, each of which may have several possible solutions, and in which the solution chosen for one problem may affect the possible solutions for subsequent subproblems. The Branch and Bound method requires that you choose a start node before setting bound to a large value (such as infinity).

Choose the cheapest arch between the unvisited and current node and 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.

Unmanned Aerial Vehicle 'Drone'

What is a Drone

A drone, in technological terms, is an unmanned aircraft. Drones are more formally known as unmanned aerial vehicles (UAVs) or unmanned aircraft systems (UASes). Essentially, a drone is a flying robot that can be remotely controlled or fly autonomously through software-controlled flight plans in their embedded systems, working in conjunction with onboard sensors and GPS.

In the recent past, UAVs were most often associated with the military, where they were used initially for anti-aircraft target practice, intelligence gathering, and then, more controversially, as weapons platforms.

Drones are now also used in a wide range of civilian roles ranging from search and rescue, surveillance, traffic monitoring, weather monitoring, and firefighting, to personal drones and business drone-based photography, as well as videography, agriculture, and even delivery services.

History Of Drones

Many trace the history of drones to 1849 Italy, when Venice was fighting for its independence from Austria. Austrian soldiers attacked Venice with hot-air, hydrogen- or helium-filled balloons equipped with bombs.

The first pilotless radio-controlled aircraft were used in World War I.
In 1918, the U.S. Army developed the experimental Kettering Bug, an unmanned "flying bomb" aircraft, which was never used in combat.

The first generally used drone appeared in 1935 as a full-size retooling of the de Havilland DH82B "Queen Bee" biplane, which was fitted with a radio and servo-operated controls in the back seat.
The plane could be conventionally piloted from the front seat, but generally, it flew unmanned and was shot at by artillery gunners in training.

The term drone dates to this initial use, a play on the "Queen Bee" nomenclature.
UAV technology continued to be of interest to the military, but it was often too unreliable and costly to put into use. After concerns about the shooting down of spy planes arose, the military revisited the topic of unmanned aerial vehicles. Military use of drones soon expanded to play roles in dropping leaflets and acting as spying decoys.

Military drone use solidified in 1982 when the Israeli Air Force used UAVs to wipe out the Syrian fleet with minimal loss of Israeli forces, The Israeli UAVs acted as decoys, jammed communication, and offered real-time video reconnaissance.

Drones have continued to be a mainstay in the military, playing critical roles in intelligence, surveillance and force protection, artillery spotting, target following and acquisition, battle damage assessment and reconnaissance, as well as for weaponry.

A Wall Street Journal report claims widespread drone use began in 2006 when the U.S. Customs and Border Protection Agency introduced UAVs to monitor the U.S. and Mexico border.

In late 2012, Chris Anderson, editor in chief of Wired magazine, retired to dedicate himself to his drones company, 3D Robotics, Inc. (3DR), The company, which started off specializing in hobbyist personal drones, now markets its UAVs to aerial

photography and film companies, construction, utilities, and telecom businesses, and public safety companies, among others.

In late 2013, Amazon CEO Jeff Bezos announced a plan to use commercial drones for delivery activities. However, in July 2016, Reno-based startup Flirtey beat Amazon to the punch, successfully delivering a package to a resident in Nevada via a commercial drone. Other companies have since followed suit. For example, in September 2016, Virginia Polytechnic Institute and State University began a test with Project Wing, a unit of Google owner Alphabet, Inc., to make deliveries, starting with burritos produced at a local Chipotle restaurant. Then in December 2016, Amazon delivered its first Prime Air package in Cambridge, England. In March of 2017, it demonstrated a Prime Air drone delivery in California.

Drone education is also expanding, Embry-Riddle Aeronautical University, long a training ground for the aviation industry, now offers a Bachelor of Science in unmanned systems applications, a Master of Science in unmanned systems, and an undergraduate minor in unmanned aerial systems.

How do drones work?

While drones serve a variety of purposes, such as recreational, photography, commercial and military, their two basic functions are flight and navigation.

To achieve flight, drones consist of a power source, such as battery or fuel, rotors, propellers, and a frame. The frame of a drone is typically made of lightweight, composite materials, to reduce weight and increase maneuverability during flight.

Drones require a controller, which is used remotely by an operator to launch, navigate and land it. Controllers communicate with the drone using radio waves, including Wi-Fi.

Drones Features and Components

Drones contain a large number of technological components, including:
Electronic Speed Controllers (ESC), an electronic circuit that controls a motor's speed and direction, Flight controller, GPS module, Battery, Antenna, Receiver, Cameras, Sensors, including ultrasonic sensors and collision avoidance sensors, Accelerometer, which measures the speed and Altimeter, which measures altitude.

Any discussion about drone features is closely tied to the type and use case of the drone, including recreational, photography, commercial, and military uses.
Examples of features include:

Camera type, video resolution, megapixels, and media storage format, Maximum flight time, such as how long the drone can remain in the air, Maximum speeds, including ascent and descent, Hover accuracy, Obstacle sensory range, Altitude hold, which keeps the drone at a fixed altitude, Live video feed, and Flight logs.

Navigational systems, such as GPS, are typically housed in the nose of a drone. The GPS on a drone communicates its precise location with the controller. If present, an onboard altimeter can communicate altitude information. The altimeter also helps keep the drone at a specific altitude, if commanded by the controller.

Drones can be equipped with a number of sensors, including distance sensors (ultrasonic, laser, lidar), time-of-flight sensors, chemical sensors, and stabilization and orientation sensors, among others. Visual sensors offer still or video data, with RGB sensors collecting standard visual red, green and blue wavelengths, and multispectral sensors collecting visible and non-visible wavelengths, such as infrared and ultraviolet. Accelerometers, gyroscopes, magnetometers, barometers, and GPS are also common drone features.

For example, thermal sensors can be integral in surveillance or security applications, such as livestock monitoring or heat-signature detection. Hyperspectral sensors can help identify minerals and vegetation and are ideal for use in crop health, water quality, and surface composition.

Some drones employ obstacle detection and collision avoidance sensors. Initially, the sensors were designed to detect objects in front of the drive. Some drones now provide obstacle detection in all six directions: front, back, below, above, and side to side.

For the purpose of landing, drones employ visual positioning systems with downward-facing cameras and ultrasonic sensors. The ultrasonic sensors determine how close the drone is to the ground.

Types Of Drones

Drone platforms have two main types: rotor, including single-rotor or multi-rotor (such as tricopters, quadcopters, hexacopters, and octocopters), or fixed-wing, which include the hybrid VTOL (vertical takeoff and landing) drones that don't require runways.
Drones can be categorized as either personal/hobbyist or commercial/enterprise.

Many personal drones are now available for consumer use, offering HD video or still camera capabilities, or simply fly around. These drones often weigh anywhere from less than a pound to 10 pounds.

Stronger, more capable drones are also available for use in commercial settings. For example, Insitu, a Boeing company, offers the ScanEagle, which has a 10-foot wingspan and weighs 35 pounds. The company also builds the Integrator, an 80-pound aircraft with a 16-foot wingspan. Insitu drones do not take off from runways, as an airplane would; rather they are VTOL as they take off and are recovered from the company's SkyHook launchers. Sensors available include electro-optic imagers, mid-wave infrared imagers, infrared markers, and laser rangefinders.

In 2018, Boeing announced it had prototyped an unmanned electric VTOL cargo air vehicle (CAV) capable of transporting up to a 500-pound payload.

Tethered drones are another option, though with the obvious limitation that they are physically tethered to a base station. Certain tethered drones can solve the challenge many drones face when it comes to the power supply if the tether provides a direct power supply. The Safe-T tethering station for drones from Elistair, for example, offers 2.5 kW power and can fly to heights of more than 200 feet, with data transfer rates of up to 200 Mb/s.

Commercial drone manufacturers include 3D Robotics, DJI, Elistair, Hubsan Identified Technologies, Insitu, Measure, Parrot, PrecisionHawk, and Yuneec.

Drone Applications

The use of drones outside the military has grown tremendously over the past decade. Beyond surveillance and delivery applications, UAVs are used in drone journalism, search and rescue, disaster response, asset protection, wildlife monitoring, firefighting, communications relay, healthcare, and agriculture.

The integration of drones and internet of things (IoT) technology has created numerous enterprise use cases. Drones working with on-ground IoT sensor networks can help agricultural companies monitor land and crops; energy companies survey power lines and operational equipment, and insurance companies monitor properties for claims and policies.

A 2015 experiment in Austin, Texas, showed how drones can potentially "connect the dots" using IoT. A security tech company teamed up with a drone startup to hunt for Zigbee beacons to try to provide an overview of what IoT networks were present in residential and business areas of the city. The companies reported that the results were quick and instructive.

From logistics to agriculture to security, unmanned aerial vehicles and IoT are frequently part of the same discussion; offering a component in ubiquitous connectivity and interactivity.

Other examples of drone applications and functions include:

- Drones can assist farmers by measuring and recording the height of crops. They use a remote sensing technology called Lidar that illuminates the crop with a laser and calculates the distance by measuring what is reflected back.

- Drones with biological sensors can fly to unsafe areas to take air quality readings and check for the presence of specific microorganisms or atmospheric elements.

- During wildfires, drones can survey the extent of the affected areas and determine how quickly the fires are spreading. Images taken can provide details of the damage in specific areas.

- Drones are used by television sport networks to capture sporting event footage, such as taped and live flyover footage, that would otherwise be difficult to acquire. The use of drones must comply with regulations from the FAA, the sports leagues, the venue, and local law enforcement.

Drone Delivery

Amazon CEO Jeff Bezos created a buzz in the world of e-commerce in 2013 when he revealed what most expected to become the future of product delivery. Their Prime Air service would feature small drones that were capable of carrying up to 5 pounds of cargo that could be delivered to your front door in 30 minutes or less.

It was an idea that seemed to be something that came from science fiction. If you were to visit the Prime Air page on Amazon today, you would find that the promotional video for the system has not received an update since 2016. There is a legitimate need to feature rapid parcel delivery, but will unmanned aerial vehicles be able to offer delivery services that provide meaningful results?

The Associated Press followed up on the lack of delivery drones in the skies from Amazon in 2018 to see how the industry was progressing. Their research discovered that almost 110,000 commercial drones currently operate in the United States. That number is expected to top 500,000 within the next four years. Most of these unmanned vehicles are utilized today for industrial purposes, such as surveying, instead of using them for delivery.

That means the delivery drone industry is still trying to evolve. It is happening much more slowly than the hype from 2013 predicted.

Amazon hasn't abandoned its quest for Prime Air because it sees the value in the pros and cons of delivery drones.

The Pros of delivery drones include:

1. They can deliver products quickly to virtually any location.
   The purpose of a delivery drone is to provide services for delivering objects or commodities rapidly and in out-of-the-way locations. These unmanned aerial vehicles can be programmed to deliver items from their headquarters to a designated area – the address of the individual who ordered the product. It is a service that can provide a bulk transfer of materials when consumers need to receive their items quickly and without damage.

2. It improves time management for all parties. When delivery drones are working to provide services for an organization, then consumers and the employees involved in the process both benefit from the increased efficiency. It allows people to focus on other essential items of the purchasing process. With accurate locating programs, this service offers the potential of a lower error margin assuming that the addresses submitted through a shopping cart are accurate. Consumers receive their goods faster, companies can increase their turnover rate, and that leads to higher levels of productivity.

3. Delivery drones conserve energy, The emissions that a delivery drone is responsible for are far fewer than standard packages using traditional delivery mechanisms. There would no longer be a need for airplanes to transport some goods, delivery trucks to offer home delivery, and other fossil fuel costs because warehouses would be conveniently located in most urban areas. That reduces the price of shipping and handling because there are fewer logistics to complete. Although this process could reduce some job opportunities, there would be an increase in positions related to drone programming and maintenance.

4. It is a technology that helps us to save time as well. Delivery drones allow organizations to divert their current human resources toward creative, innovative pursuits that can expand their product lines or services even

further. Workers will have more time to see to the regular operations of the company while ensuring a higher quality of the product being received by their customers. Instead of paying for repetitive activities that keep production levels artificially low, this technology makes it possible to spend more time and money on ideas that could one day change the world.

5. Delivery drones offer a safer delivery system. Drivers that bring packages to a consumer's home are assuming a level of risk because of their road-based activities. There is always a chance that they could be involved in an accident, run out of fuel, or encounter delays due to road construction. Delivery personnel are sometimes exposed to dangerous environments as well, such as a slippery set of entryway stairs. Delivery drones create a safer system for delivery because they remove many of these factors from the equation. There is always the added risk that a drone could fall from the sky or accidentally release its package, but the comparative perspective places this issue in the advantage category.

6. They would offer a higher level of efficiency. Thanks to GPS technologies, delivery drones could automatically navigate to a specific address when a product is ordered by a consumer. This technology has a higher success rate compared to humans when performing the same task. Although there will always be homes that are not accurately charted by these systems (like a GPS guide telling someone to turn right, which would lead them into a building), this issue is one that will eventually resolve as the technology continues evolving.

7. It would have a positive effect on shareholders. One of the key benefits that a company would experience when using an automated delivery system that includes drones is an improvement in efficiency. That would translate into a cost-savings process that could reduce expenses for the organization. Shareholders would benefit from a potential increase in dividends, while senior managers might receive rewards for their creativity with a bump in their pay. There would be fewer workers potentially needed as well, which would further increase these compensation factors.

8. Delivery drones could reduce consumption levels, Over-consumption is a growing problem in our world today. It causes us to generate more trash, pollute more often (intentionally or not), and spend more instead of saving our resources. Although having more access to products purchased online might seem like it would encourage additional spending, the trends in online

shopping show the reverse is often true. Time Magazine researched the issue and found that people make fewer impulse purchases and consume less when they shop at home instead of going to a store.

The Cons of delivery drone include:

1. Delivery drones are expensive devices.
   The cost of adding delivery drones to a company's services requires a significant investment in the infrastructure of the organization. That's why there are only a few large businesses right now which are even considering this technology. The average price of a drone can be up to $500 for one that is capable of making an accurate delivery. Although smaller drones may be in the $50 price range, a company like Amazon would require tens of thousands of them (and the programming staff for them) to turn this idea into something that isn't science fiction.

2. It requires an improvement in battery technologies to be effective.
   On Amazon's Prime Air web page, they say that they are "committed to making our goal I'm delivering packages by drones in 30 minutes or less a reality." One of the reasons why this quick turnaround is necessary is because of our current battery technologies. The power requirements of the average delivery drone could cause it to lose power in 20 minutes or less. That may not offer enough flying time to get the package to its intended destination – or have the drone return to company headquarters.

   If the delivery drone failed to meet its obligation to a consumer, then there would be an increase in customer complaints, product costs, and insurance premiums. When it would not be able to return to the company, you'd still have the logistics of replacing or finding the missing drones.

3. Delivery drones require a technical familiarity with the product.
   Amazon seeks to create a technology where programming can replace a human operator. The reason behind this push for unmanned aerial vehicles is

that the cost of purchasing automatic delivery drones is cheaper than training pilots who can accurately follow a flight plan. Even when there is a technical familiarity with the product, human operators would need to spend time in the operating manual to ensure that they were flying items correctly. There would need to be a significant investment in time and effort before this service could ever get off the ground.

4. It creates a higher risk of defects during operations.
   Any vehicle that we use to deliver products around the world creates a risk of a defect changing the process for a company. Airplanes can break down on a runway unpredictably. Delivery trucks can refuse to start. This risk is something that all businesses face when managing their logistics.

   Delivery drones, using our current technologies, experience a higher risk of failure when compared to traditional options. Industrial drones have shown that their systems can malfunction within the first year of putting the device to work. Should one of these defects be present during delivery operations, then it would disrupt the logistics of the company, add more expenses to the budget, and create other potential issues as well.

5. There is a risk that the delivery drones could be stolen.
   It does not take much skill to disconnect the power system from a delivery drone. This risk is another issue that companies must face if they wish to adopt this express delivery option. Anyone could take the equipment away at any time by ordering a small product from their preferred e-commerce platform. Once this action occurs, it could be challenging for the business to have any way to recall their property. Regulations and laws would likely need an update to accommodate this risk if this service will ever become a reality.

6. Delivery drones offer a privacy risk that we must consider.
   One of the ways that Amazon and other companies would likely provide delivery services with the drone is through the use of a camera. Streaming video to the company's headquarters would allow supervisors to know if a delivery was successfully completed. This technology could help to lessen

the risk of theft in the future as well. The problem with a camera being equipped with a drone is that you can pick up video of people and properties who are not associated with the transaction. There would be no consent given by people who are caught on this stream. update in-laws and regulations would need to apply here as well.

7. It could cause delivery prices to rise exponentially.
   Automated delivery services like what a drone could provide will likely cause the cost of receiving an item to arrive, even if options for non-rush delivery become available. Amazon sees this as being a 30-minute service, so the cost would likely be similar to what express deliveries are today. If you want an overnight guarantee right now for a package shipped through the USPS, then the minimum cost is $25.50 – and that is only for "most locations." If you go out for 3 business days, then the minimum price is $7.35. Although Amazon might include this option in the Prime subscription, it would only be available to select customers, and it might increase the overall cost of membership for everyone.

8. There would be fewer job opportunities for unskilled workers.
   Whenever there has been a shift in technology that changes the way employees work, most businesses take the approach that the issue is "not their problem." Workers would need new training programs just to keep their jobs. There would also be fewer entry-level positions where no experience is necessary, which places a higher level of pressure on the educational programs that can equip people for work. Most of these structures are already incapable of serving the needs of their community at a 100% rate. Delivery drones would simply increase the issue.

9. It could offer another way to breach our data privacy.
   Is it unlikely that a company like Amazon would use the data from drone delivery to its advantage? If what we have seen from Facebook in 2018 is about privacy and data, then delivery drones could take this issue to another level. The logistics of this delivery mechanism would create new data points that could be used to upsell, advertise, or retarget consumers in ways that we have never seen before.

That information could help the criminal element in society to find new avenues of identity theft, home invasion, and more to create harm in society as well.

10. There is a threat of property damage to consider.

    Delivery drones are not an infallible technology. There is an excellent chance that they could inadvertently damage the home while performing their function. The only way to counter this issue would be for homeowners to install landing pads for the drones at a safe location on their property. This tech option could come with an RFID signal that only activates when delivery is authorized, which could help to reduce the multiple data streams that may be available. These delivery drones pros and cons are mostly hypothetical at this stage because the technologies we use for this logistics option are still under development. This option could become the standard express delivery service for customers who shop online in the future. It could be an idea that is less than five years away from becoming a reality. If we can learn from our mistakes while continuing to fund research and development products, more companies might think about adopting a service like Amazon Air in the future.

Study Description

Variations Of The Study

Several variations of the TSP that are studied in the literature have originated from various real life or potential applications. Let us first consider some of these variations that can be reformulated as a TSP using relatively simple transformations.

1. The MAX TSP: Unlike the TSP, the objective in this case is to find a tour in G where the total cost of edges of the tour is maximum. MAX TSP can be solved as a TSP by replacing each edge cost by its additive inverse. If we require that the edge costs are to be non-negative, a large constant could be added to each of the edge-costs without changing the optimal solutions of the problem.

2. The bottleneck TSP: In the case of a bottleneck TSP, the objective is to find a tour in G such that the largest cost of edges in the tour is as small as possible. A bottleneck TSP can be formulated as a TSP with exponentially large edge costs.

3. TSP with multiple visits (TSPM): Here we want to find a routing of a traveling salesman, who starts at a given node of G, visits each node at least once and comes back to the starting node in such a way that the total distance traveled is minimized. This problem can be transformed into a TSP by replacing the edge costs with the shortest path distances in G. In the absence of negative cycles, shortest path distances between all pairs of nodes of a graph can be computed using efficient algorithms. If G contains a negative cycle, then TSPM is unbounded. Several shortest path algorithms are capable of detecting negative cycles in a graph.

4. Messenger problem: This problem is also known as the wandering salesman problem . Given two specified nodes u and v in G, we want to find a least cost Hamiltonian path in G from u to v. This can be solved as a TSP by choosing a cost of — M for the edge {v^ u) where M is a large number. If no

nodes are specified, and one wishes to find a least cost Hamiltonian path in G, it can also be achieved by solving a TSP. Modify the graph G by creating a new node and connecting it with all the nodes of G by edges (for directed graphs, introduce two arcs, one in forward and other in backward directions) of cost -M.

5. Clustered TSP: In this case, the node set of G is partitioned into clusters Vi, V2,... , Vk- Then the clustered TSP is to find a least cost tour in G subject to the constraint that cities within the same cluster must be visited consecutively. This problem can be reduced to a TSP by adding a large cost M to the cost of each inter-cluster edge.

6. Black and White TSP: This problem is a generalization of the TSP. Here, the node set of G is partitioned in to two sets, B and W. The elements of B are called black nodes and the elements of W are called white nodes. A tour in G is said to be feasible if the following two conditions are satisfied, (i) The number of white nodes between any two consecutive black nodes should not exceed a positive integer I and the distance between any two consecutive black nodes should not exceed a positive real number R. The black and white TSP is to find a minimum cost feasible tour in G. Applications of this problem include design of ring networks in the telecommunication industry. A variation of the black and white TSP with applications in the air-line industry, known as TSP with replenishment arcs.

7. Ordered Cluster TSP: This problem is a simplified version of the clustered TSP and is defined as follows. The node set of G is partitioned into k clusters Xi, X2,... , X/. where Xi is singleton representing the home location. The salesman starting and ending at the home location must visit all nodes in cluster X2, followed by all nodes in cluster X3, and so on such that the sum of the edge costs in the resulting tour is minimized.

8. Film-copy Deliverer Problem: For each node i of G, a nonnegative integer pi is prescribed. We want to find a routing of a film-copy deliverer, starting at node 1, visit each node exactly pi times and come back to node 1 in such a way that the total distance travelled is minimized . This problem generalizes TSP and some of its variations. In this case, a minimum cost cycle in G is sought that passes through each node z of G at least pi times and at most qi times.

9. Period TSP: This generalization of TSP considers a /c-day planning period, for given k. We want to find k cycles in G such that each node of G is visited a prescribed number of times during the A:-day period and the total travel cost for the entire A:-day period is minimized.

10. Remote TSP: For a given integer /c, we are interested in finding a subset 5 of y with \S\ = k such that the minimum cost of a tour in the subgraph of G induced by S is maximized [430].

11. There are several other variations that one could discuss depending on various application scenarios. These variations include traveling salesman location problem, traveling salesman problem with backhauls , /c-best TSP , min max TSP , multi-criteria TSP , stochastic TSP (different variations), fc-MST , minimum cost biconnected spanning subgraph problem , graph searching problem , 28 THE TRAVELING SALESMAN PROBLEM AND ITS VARIATIONS A:-delivery TSP , quadratic assignment problem etc.

Mission

Given a fleet of drones, a list of customer orders and availability of the individual products in warehouses, schedule the drone operations so that the orders are completed as soon as possible.

Description

Area

The simulation takes place on a two-dimensional grid. The grid is not cyclic and a drone cannot fly outside of the grid. The drones can fly over all cells within the grid. Each cell is identified by a pair of integer coordinates [r , c]
($0 \leq r <$ row count, $0 \leq c <$ column count).

Products

There are a number of product types available for order. Each product type has one or more product items available in warehouses. Each product type has a fixed product weight, identical for all product items. Every product weight is guaranteed to be smaller or equal to the maximum payload that a drone can carry.

Warehouses

Product items are stored in several warehouses. Each warehouse is located in one particular cell of the grid, different for each warehouse. Each warehouse initially stocks a known number of product items of each product type. No new product items beyond the initial availability will be stocked in the warehouses during the simulation, but the drones can transport product items between the warehouses.

Any warehouse does not necessarily need to have every product type available.

Orders

Each order specifies the product items purchased by the customer. The product items in an order can be of one or multiple product types and can contain multiple product items of the same product type.

Each order specifies the cell in the grid where the product items have to be delivered. It is possible to have multiple orders with the same delivery cell. No order has the delivery cell that is a location of a warehouse. The order is considered fulfilled when all of the ordered product items are delivered.

Individual product items can be delivered in multiple steps, in any order. It is valid to deliver the individual product items of an order using multiple drones, including using different drones at the very same time.

It is guaranteed that for each product type, the total number of product items in all orders is not greater than the total availability of product items of this product type in all warehouses. It is not required to deliver all orders (see the Scoring section at the end).

Drones

Drones transport product items from warehouses to customers and between the warehouses. The drones always use the shortest path to fly from one cell in the grid to another. The distance from cell [ra, ca] to cell [rb, cb] is calculated as √|ra - rb|^2 + |c-c |^2 (two-dimensional Euclidean distance). Each drone flight takes o ne turn per one unit of distance between the start and the destination cell, rounded up to the next integer.
Multiple drones can be in the same cell at any moment they never collide, as they can fly at different altitudes.

At the beginning of the simulation, all drones are at the first warehouse (warehouse with id 0).

Commands

Each drone can be given the following basic commands:

● Load: Moves the specified number of items of the specified product type from a warehouse to the drone's inventory. If the drone isn't at the warehouse it will fly there using the shortest path before loading the product items. The requested number of items of the specified product type must be available in the warehouse. The total weight of the items in the drone's inventory after the load cannot be bigger than the drone's maximum load.

● Deliver: Delivers the specified number of items of the specified product type to a customer. If the drone isn't at the destination it will fly there using the shortest path before delivering the product items. The drone must have the requested number of items of the specified product type in its inventory. Each drone can also be given the following advanced commands. These commands are not necessary to solve the

problem, but you can use them to further improve your solution.

● Unload: Moves the specified number of items of the specified product type from drone's inventory to a warehouse. If the drone isn't at the warehouse it will fly there using the shortest path before unloading the product items. The drone must have the requested number of items of the specified product type in its inventory.

● Wait : Waits the specified number of turns in the drone's current location.

Simulation

The simulation proceeds in T turns, from 0 to . A drone executes the commands $T - 1$ issued to it in the order in which they are specified, one by one. The first command issued to the drone starts in turn 0.

The duration of a command depends on its type:

● Each Load/ Deliver/ Unload command takes d + 1 turns, where d is the distance travelled by the drone to perform the requested action (d can be 0 if the drone is already in the required location). When the command starts, the drone flies to the required location in d turns. Then, the actual action (loading, delivery or unloading) takes place and takes 1 turn.

● Each Wait command takes w turns, where w is the specified number of turns.

Methodologies

Dataset

The input data is provided as a data set file, a plain text file containing exclusively ASCII characters with lines terminated with a single '\n' character at the end of each line (UNIX style line endings). Product types, warehouses and orders are referenced by integer IDs. There are P product types numbered from 0 to $P - 1$, W warehouses numbered from 0 to $W - 1$ and C orders numbered from 0 to $C - 1$.

File format

The first section of the file describes the parameters of the simulation . This section contains a single line containing the following natural numbers separated by single spaces:

- number of rows in the area of the simulation ($1 \leq$ number of rows ≤ 10000)
- number of columns in the area of the simulation ($1 \leq$ number of columns ≤ 10000)
- D number of drones available ($1 \leq D \leq 1000$)
- deadline of the simulation ($1 \leq$ deadline of the simulation ≤ 1000000)
- maximum load of a drone ($1 \leq$ maximum load of a drone ≤ 10000)

The next section of the file describes the weights of the products available for orders. This section contains:
A line containing the following natural number: P the number of different product types available in warehouses ($1 \leq P \leq 10000$).

A line containing P natural numbers separated by single spaces denoting weights of subsequent product types, from product type 0 to product type P 1. For each weight, $1 \leq$ weight $\leq$ maximum load of a drone.

The next section of the file describes the warehouses and availability of individual product types at each warehouse. This section contains:

A line containing the following natural number:
W the number of warehouses ($1 \leq W \leq 10000$). Two lines for each warehouse, each two lines describing the subsequent warehouses from warehouse 0 to warehouse $W - 1$.

A line containing two natural numbers separated by a single space: the row and the column in which the warehouse is located ($0 \leq$ row $<$ number of rows; $0 \leq$ column $<$ number of columns).

A line containing P natural numbers separated by single spaces: number of items of the subsequent product types available at the warehouse, from product type 0 to product type $P - 1$. For each product type, $0 \leq$ number of items ≤ 10000 holds.

The next section of the file describes the customer orders .
This section contains:

- A line containing the following natural number.
- C the number of customer orders ($1 \leq C \leq 10000$)
- Three lines for each order, each three lines describing the subsequent orders from order 0 to C 1.
- A line containing two natural numbers separated by a single space: the row of the delivery.
- Cell and the column of the delivery cell ($0 \leq$ row $<$ number of rows; $0 \leq$ column $<$ number of columns).
- A line containing one natural number Li the number of the ordered product items ($1 \leq$ Li < 10000).
- A line containing L i integers separated by single spaces: the product types of the individual product items.
- For each of the product types, $0 \leq$ product type $<$ P holds.

Example

In the comments, “u” is an abbreviation for units of weight.

Example input file:

100 100 3 50 500	100 rows, 100 columns, 3 drones, 50 turns, max payload is 500u
3	There are 3 different product types.
100 5 450	The product types weigh: 100u, 5u, 450u.
2	There are 2 warehouses.
0 0	First warehouse is located at [0, 0].
5 1 0	It stores 5 items of product 0 and 1 of product 1.
5 5	Second warehouse is located at [5, 5].
0 10 2	It stores 10 items of product 1 and 2 items of product 2.
3	There are 3 orders.
1 1	First order to be delivered to [1, 1].
2	First order contains 2 items.
2 0	Items of product types: 2, 0.
3 3	Second order to be delivered to [3, 3].
1	Second order contains 1 item.
0	Items of product types: 0
5 6	Third order to be delivered to [5, 6]
1	Third order contains 1 item.
2	Items of product types: 2.

Data wrangling

The first task is to extract the data from the compact text file.
Returns a list of integer lists.

```
def list_lines(file_name):
    with open(file_name) as file:
        lines = file.read().splitlines()
    line_list = [[int(n) for  n in ll.split()] for ll in lines]
    return line_list


def set_params(line_list):
    top_line = line_list[0]
    params = {'DRONE_COUNT': top_line[2],
              'WT_CAP': top_line[4],
              'END_TIME': top_line[3],
              }
    return params
```

Provides the dividing line between warehouse and order sections in the line list.

```
def find_wh_lines(line_list):
    wh_count = line_list[3][0]
    wh_endline = (wh_count*2)+4
    return wh_endline


def get_weights(line_list):
    weights = np.array(line_list[2])
    return weights.astype(np.int16)
```

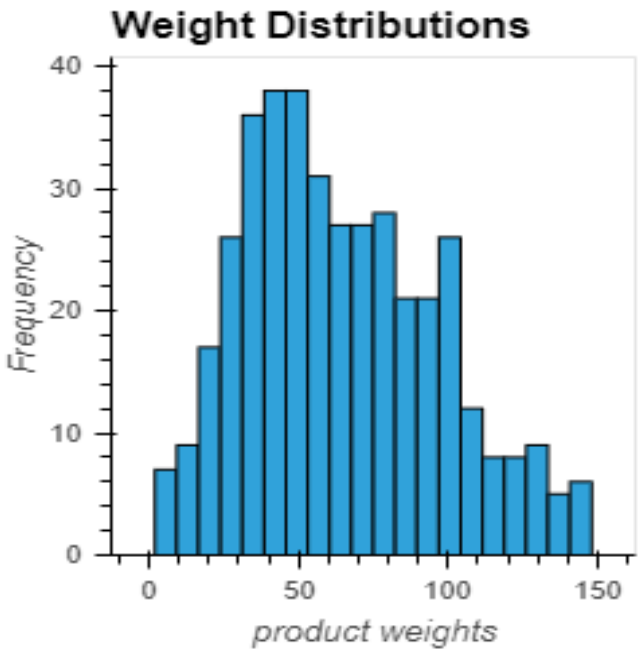
Returns a 2-d array of P products by W warehouses.

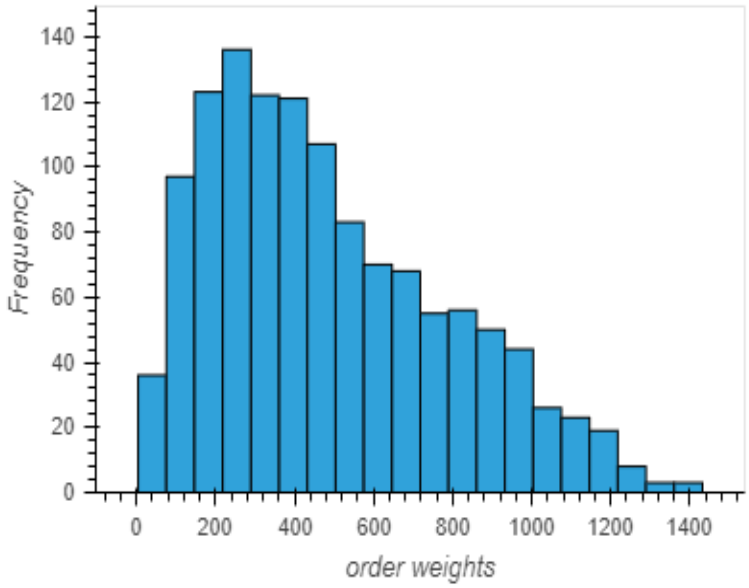
```
def get_inventories(line_list):
    wh_endline = find_wh_lines(line_list)
    invs = line_list[5:wh_endline+1:2]
    supply = np.array(invs).transpose()
    return supply.astype(np.int16)
```

Returns a 2-d array of P products by C orders.

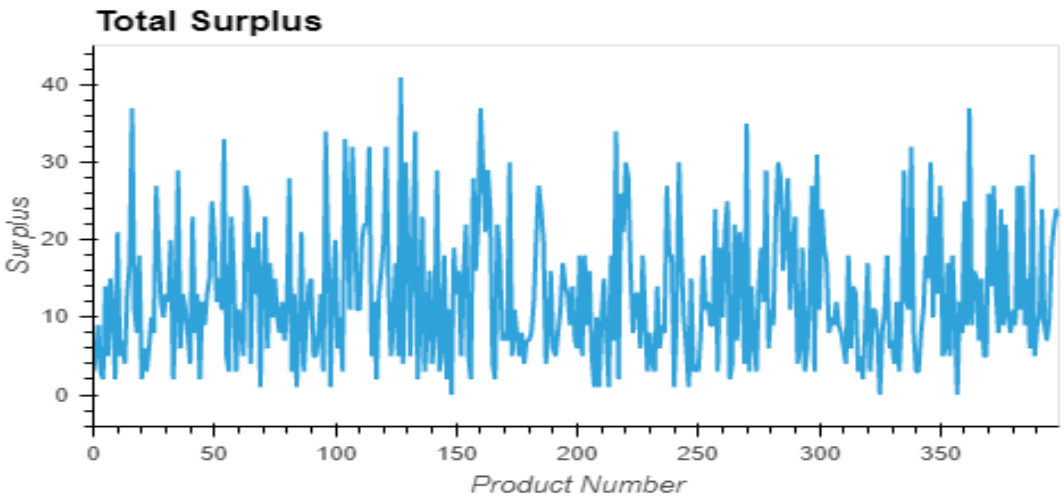
```
def get_orders(line_list):
    wh_endline = find_wh_lines(line_list)
    demand = np.zeros((line_list[1][0],
                       line_list[wh_endline][0]),
                          dtype=np.int16
)
    orders = line_list[wh_endline+3::3]
    for i,ord in enumerate(orders):
        for prod in ord:
            demand[prod, i] += 1
    return demand.astype(np.int16)


def get_locs(line_list):
    wh_endline = find_wh_lines(line_list)
    wh_locs = np.array(line_list[4:wh_endline:2])
    cust_locs = np.array(line_list[wh_endline+1::3])
    return wh_locs.astype(np.int16), cust_locs.astype(np.int16)


# main
files = ['../input/drone-delivery/busy_day.in']
line_list = list_lines(files[0])

params = set_params(line_list)
supply = get_inventories(line_list)
demand = get_orders(line_list)
wh_locs, cust_locs = get_locs(line_list)
weights = get_weights(line_list)


print(params)
for array in ['supply', 'wh_locs', 'demand', 'cust_locs', 'weights']:
    print(array, eval(array).shape)
```

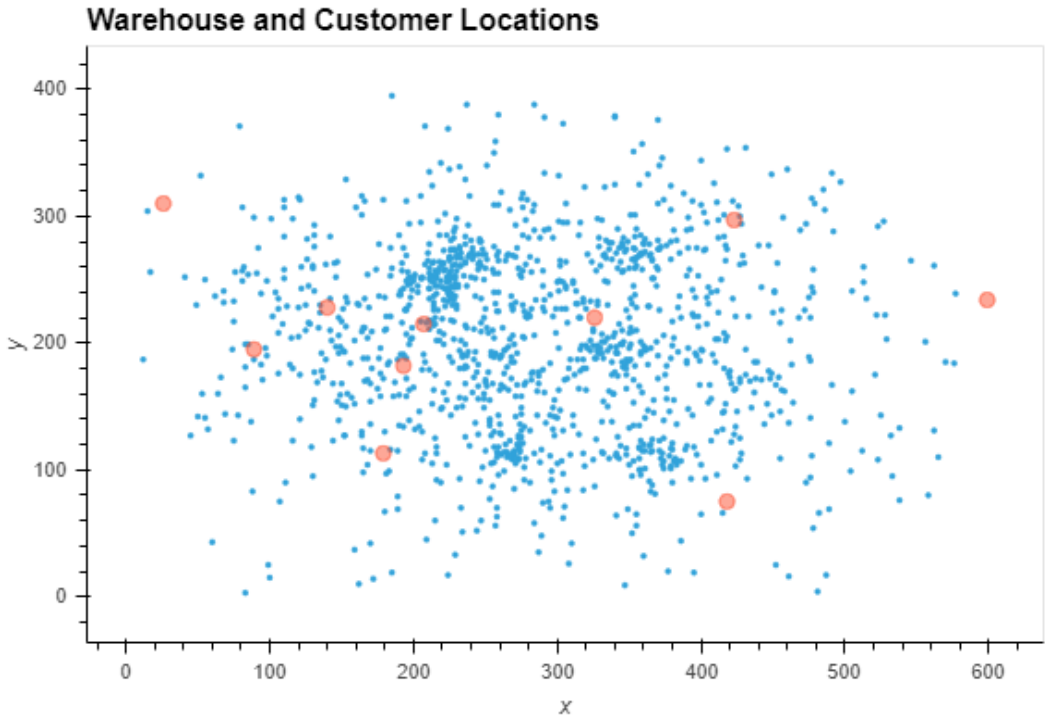
```
freqs, edges = np.histogram(weights, 20)
wt_prod = hv.Histogram((edges, freqs))
            .options(xlabel="product weights",
             width=250, title='Weight Distributions'
)

order_weights = (
      weights.reshape(weights.size, -1)* demand) \ .sum(axis=0
)

freqs, edges = np.histogram(order_weights, 20)
wt_orders = hv.Histogram((edges, freqs))
              .options(xlabel="order weights", width=400
)
```






```
surplus = hv.Curve(
              supply.sum(axis=1) - demand.sum(axis=1)
          ).options(width=500)


xlabel=('Product Number',
        ylabel='Surplus',
        title='Total Surplus'
)

customers = hv.Points(np.fliplr(cust_locs))
              .options(width=600,
                       height=400,
                       title='Warehouse and Customer Locations')

warehouses = hv.Points(np.fliplr(wh_locs))
               .options(size=8, alpha=0.5)
display(hv.Layout(wt_prod+wt_orders)
          .options(shared_axes=False),
           surplus,
           customers * warehouses)
```


Total Surplus
Surplus
Product Number
0
10
20
30
40
0
50
100
150
200
250
300
350


Warehouse and Customer Locations
y
x
0
100
200
300
400
0
100
200
300
400
500
600

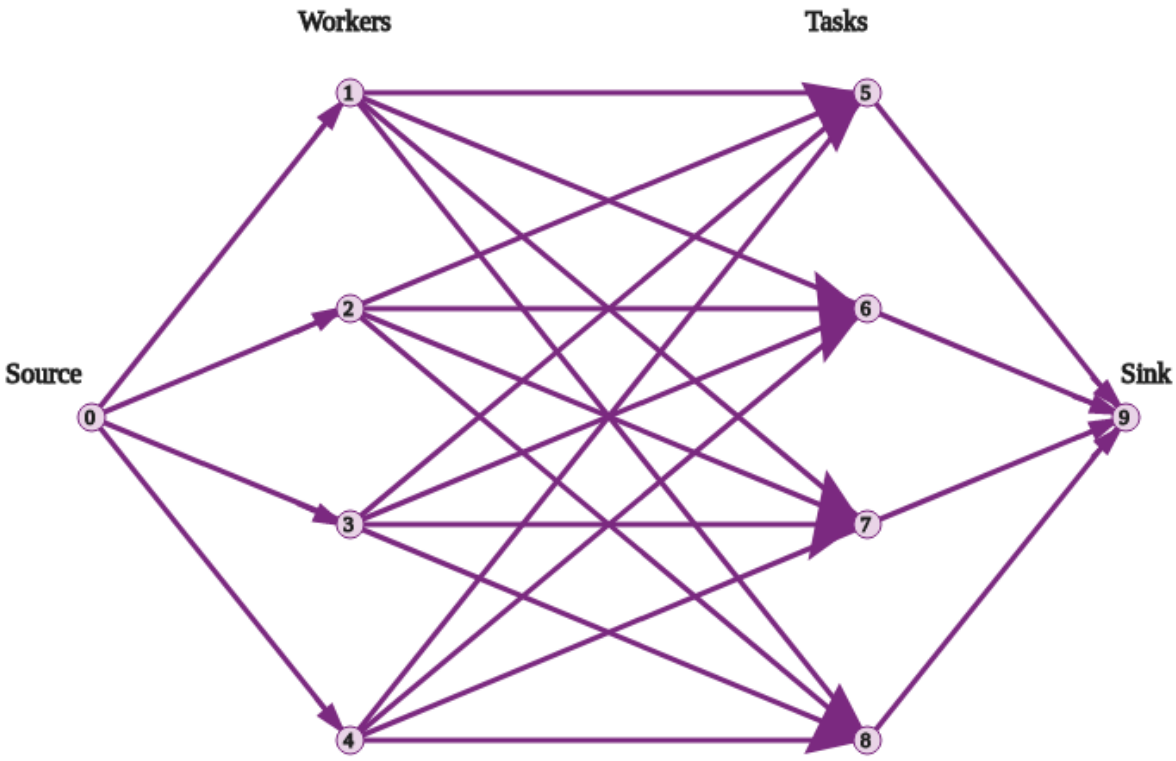
Warehouse Assignments

Warehouses are assigned to orders in a way that minimizes the total distance traveled by drones to fulfill orders. Assignments are made product by product. Of course, any time we divide a problem into parts and optimize those parts, we are almost sure to suboptimize the overall objective. Still, it's a reasonable place to start.

There are different methods to use when tackling an assignment problem. Here I use the minimum cost flow solver to assign warehouses. The basic idea is you have a network of nodes and edges. There's a notional source node, a set of worker nodes (warehouses), a set of task nodes (orders) and a notional sink node. Here's a generic view of such a network.

You can use the min cost flow solver to solve special cases of the assignment problem. In fact, min cost flow can often return a solution faster than either the MIP or CP-SAT solver. However, MIP and CP-SAT can solve a larger class of problems than min cost flow, so in most cases MIP or CP-SAT are the best choices.

The flow diagram for the problem consists of the bipartite graph for the cost matrix (see the assignment overview for a slightly different example), with a source and sink added.



The numbering of the workers and tasks is slightly different than in the section Linear Assignment Solver, because the min cost flow solver requires all nodes in the graph to be numbered distinctly.

Representation of assignment problem

How does the min cost flow problem above represent an assignment problem?

First, since the capacity of every arc is 1, the supply of 4 at the source forces each of the four arcs leading into the workers to have a flow of 1.

Next, the flow-in-equals-flow-out condition forces the flow out of each worker to be 1. If possible, the solver would direct that flow across the minimum cost arc leading out of each worker. However, the solver cannot direct the flows from two different workers to a single task. If it did, there would be a combined flow of 2 at that task, which could not be sent across the single arc with capacity 1 from the task to the sink. This means that the solver can only assign a task to a single worker, as required by the assignment problem.

Finally, the flow-in-equals-flow-out condition forces each task to have an outflow of 1, so each task is performed by some worker.

The solution consists of the arcs between workers and tasks that are assigned a flow of 1 by the solver. (Arcs connected to the source or sink are not part of the solution.) The program checks each arc to see if it has flow 1, and if so, prints the Tail (start node) and the Head (end node) of the arc, which correspond to a worker and task in the assignment.

The result is the same as that for the linear assignment solver (except for the different numbering of workers and costs). The linear assignment solver is slightly faster than min cost flow — 0.000147 seconds versus 0.000458 seconds.

Here I use the SolveMaxFlowWithMinCost variation, which doesn't require supply and demand to be equal. That allows one to do away with source and sink nodes and simplify the network. It can also find a different optimum from the balanced min-cost solver, which may be better or worse for this challenge.

Minimum Cost Flows

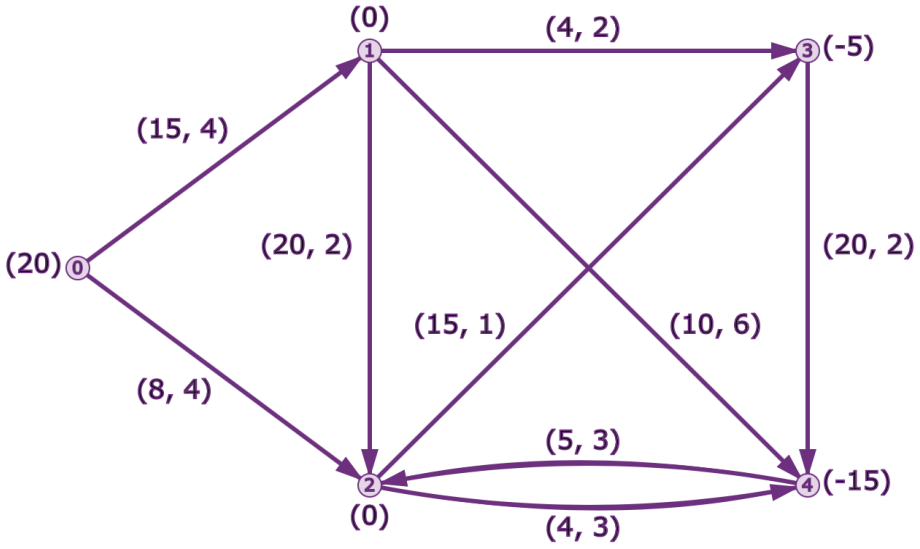
Closely related to the max flow problem is the minimum cost (min cost) flow problem, in which each arc in the graph has a unit cost for transporting material across it. The problem is to find a flow with the least total cost.

The min cost flow problem also has special nodes, called supply nodes or demand nodes, which are similar to the source and sink in the max flow problem. Material is transported from supply nodes to demand nodes.

At a supply node, a positive amount — the supply — is added to the flow. A supply could represent production at that node, for example.
At a demand node, a negative amount — the demand — is taken away from the flow. A demand could represent consumption at that node, for example.
For convenience, we'll assume that all nodes, other than supply or demand nodes, have zero supply (and demand).

For the min cost flow problem, we have the following flow conservation rule, which takes the supplies and demands into account:

The graph below shows a min cost flow problem. The arcs are labeled with pairs of numbers: the first number is the capacity and the second number is the cost. The numbers in parentheses next to the nodes represent supplies or demands. Node 0 is a supply node with supply 20, while nodes 3 and 4 are demand nodes, with demands -5 and -15, respectively.

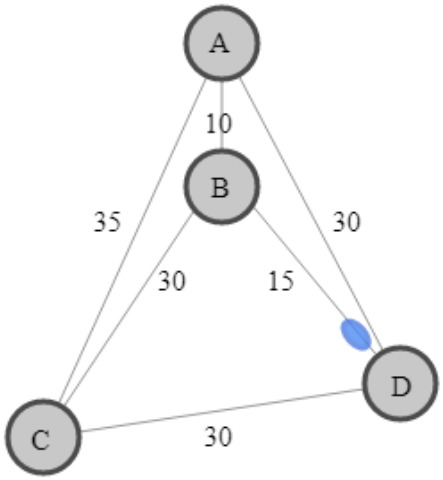


Vehicle Routing

One of the most important applications of optimization is vehicle routing, in which the goal is to find the best routes for a fleet of vehicles visiting a set of locations. Usually, "best" means routes with the least total distance or cost. Here are a few examples of routing problems:

- A package delivery company wants to assign routes for drivers to make deliveries.
- A cable TV company wants to assign routes for technicians to make residential service calls.
- A ride-sharing company wants to assign routes for drivers to pick up and drop off passengers.

The most famous routing problem is the Traveling Salesperson Problem (TSP): find the shortest route for a salesperson who needs to visit customers at different locations and return to the starting point. A TSP can be represented by a graph, in which the nodes correspond to the locations, and the edges (or arcs) denote direct travel between locations. For example, the graph below shows a TSP with just four locations, labeled A, B, C, and D. The distance between any two locations is given by the number next to the edge joining them.



By calculating the distances of all possible routes, you can see that the shortest route is ACDBA, for which the total distance is 35 + 30 + 15 + 10 = 90.

The problem gets harder when there are more locations. In the example above, there are just six routes. But if there are ten locations (not counting the starting point), the number of routes is 362880. For 20 locations, the number jumps to 2432902008176640000. An exhaustive search of all possible routes would be guaranteed to find the shortest, but this is computationally intractable for all but small sets of locations. For larger problems, optimization techniques are needed to intelligently search the solution space and find an optimal (or near-optimal) solution.

A more general version of the TSP is the vehicle routing problem (VRP), in which there are multiple vehicles. In most cases, VRPs have constraints: for example, vehicles might have capacities for the maximum weight or volume of items they can carry, or drivers might be required to visit locations during specified time windows requested by customers. OR-Tools can solve many types of VRPs, including the following:

- Traveling Salesperson Problem, the classic routing problem in which there is just one vehicle:
  The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one (e.g. the hometown) and returning to the same city. The problem is that the traveling salesman wants to minimize the total length of the trip.

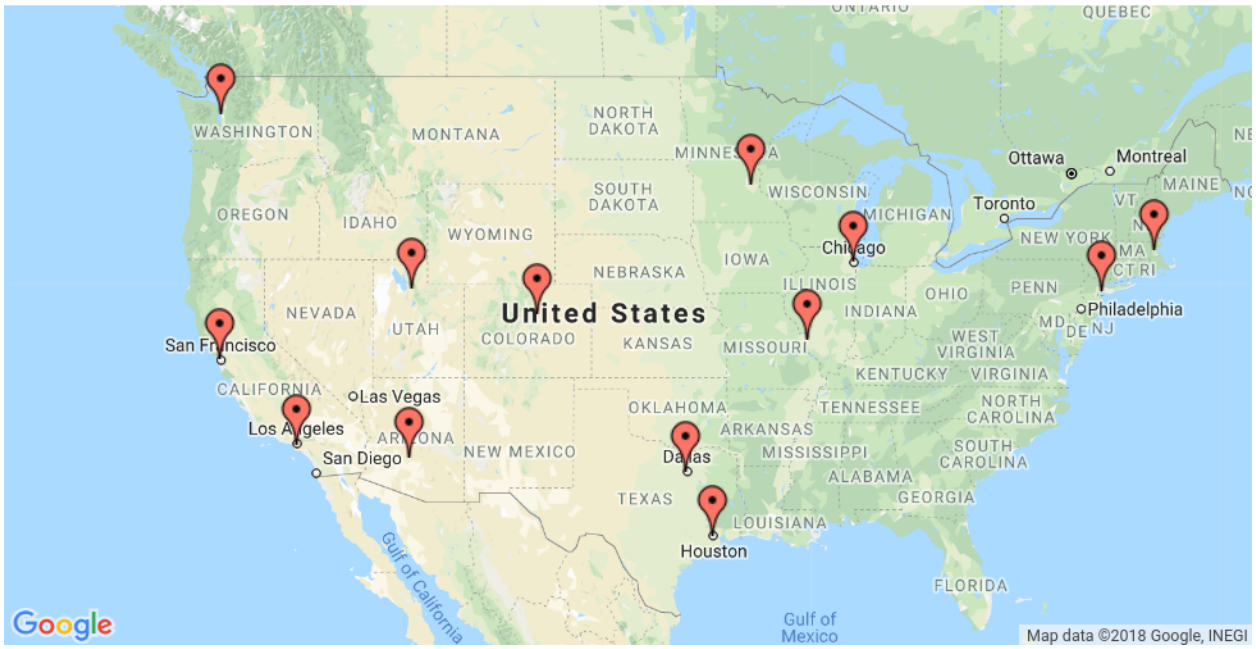
- Vehicle routing problem, a generalisation of the TSP with multiple vehicles:
  The vehicle routing problem (VRP) is a combinatorial optimization and integer programming problem which asks "What is the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of customers?". It generalises the well-known travelling salesman problem (TSP).

- VRP with capacity constraints, in which vehicles have maximum capacities for the items they can carry:
  The capacitated vehicle routing problem (CVRP) is a VRP in which vehicles with limited carrying capacity need to pick up or deliver items at various locations. The items have a quantity, such as weight or volume, and the vehicles have a maximum capacity that they can carry.The problem is to pick up or deliver the items for the least cost, while never exceeding the capacity of the vehicles.

- VRP with time windows, where the vehicles must visit the locations in specified time intervals:
  Many vehicle routing problems involve scheduling visits to customers who are only available during specific time windows. These problems are known as vehicle routing problems with time windows (VRPTWs).

- VRP with resource constraints, such as space or personnel to load and unload vehicles at the depot (the starting point for the routes):
  A VRPTW that also has constraints at the depot: all vehicles need to be loaded before departing the depot and unloaded upon return. Since there are only two available loading docks, at most two vehicles can be loaded or unloaded at the same time. As a result, some vehicles must wait for others to be loaded, delaying their departure from the depot. The problem is to find optimal vehicle routes for the VRPTW that also meet the loading and unloading constraints at the depot.

- VRP with dropped visits, where the vehicles aren't required to visit all locations, but must pay a penalty for each visit that is dropped:
  A VRP with capacity constraints in which the total demand at all locations exceeds the total capacity of the vehicles, no solution is possible. In such cases, the vehicles must drop visits to some locations. The problem is how to decide which visits to drop.



Limitation Concerns

Vehicle routing problems are inherently intractable: the length of time it takes to solve them grows exponentially with the size of the problem. For sufficiently large problems, it could take OR-Tools (or any other routing software) years to find the optimal solution. As a result, OR-Tools sometimes returns solutions that are good, but not optimal. To find a better solution, change the search options for the solver. See Changing the search strategy for an example.

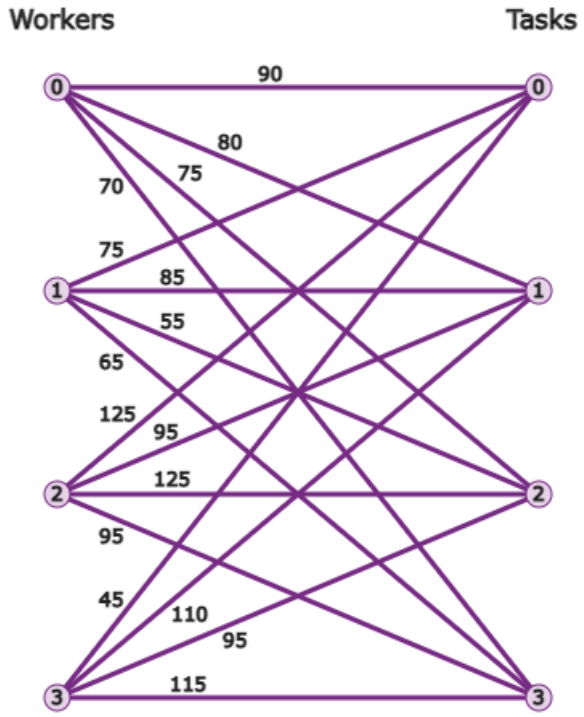
The routing solver does not always return the optimal solution to a TSP, because routing problems are computationally intractable. For instance, the solution returned in the previous example is not the optimal route. To find a better solution, you can use a more advanced search strategy, called guided local search, which enables the solver to escape a local minimum—a solution that is shorter than all nearby routes, but which is not the global minimum. After moving away from the local minimum, the solver continues the search.

Note: We should add that there are other solvers, such as Concorde, dedicated to solving very large TSPs to optimality, which surpass OR-Tools in that area. However, OR-Tools provides a better platform for solving more general routing problems that contain constraints beyond those of a pure TSP.

Assignment problem

This section describes the linear assignment solver, a specialized solver for the simple assignment problem, which can be faster than either the MIP or CP-SAT solver. However, the MIP and CP-SAT solvers can handle a much wider array of problems, so in most cases they are the best option.

The assignment problem is a fundamental combinatorial optimization problem. In its most general form, the problem is as follows:



The problem instance has a number of agents and a number of tasks. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. It is required to perform as many tasks as possible by assigning at most one agent to each task and at most one task to each agent, in such a way that the total cost of the assignment is minimized. Alternatively, describing the problem using graph theory.

The assignment problem consists of finding, in a weighted bipartite graph, a matching of a given size, in which the sum of weights of the edges is a minimum.

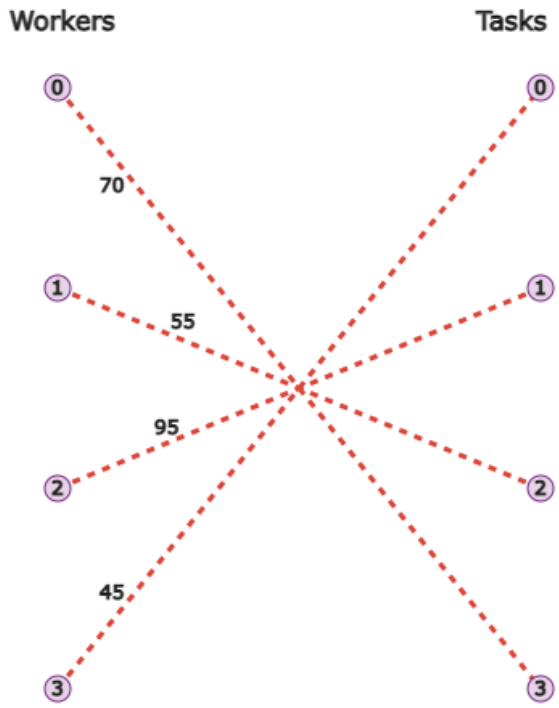
If the numbers of agents and tasks are equal, then the problem is called balanced assignment. Otherwise, it is called unbalanced assignment. If the total cost of the assignment for all tasks is equal to the sum of the costs for each agent (or the sum of the costs for each task, which is the same thing in this case), then the problem is called linear assignment. Commonly, when speaking of the assignment problem without any additional qualification, then the linear balanced assignment problem is meant.

The assignment problem can be stated in the form of n x n cost matrix C real members as given in the following table:

		Jobs				
		1	*2*	*3*	*j*	*n*
Persons	1	C_{11}	C_{12}	C_{13}	C_{ij}	C_{1n}
	2	C_{21}	C_{22}	C_{23}	C_{2j}	C_{2n}
	3	C_{31}	C_{32}	C_{33}	C_{3j}	C_{3n}
	i	C_{11}	C_{12}	C_{13}	C_{Cj}	C_{in}
	n	C_{n1}	C_{n2}	C_{n3}	C_{nj}	C_{nn}

One of the most well-known combinatorial optimization problems is the assignment problem. Here's an example: suppose a group of workers needs to perform a set of tasks, and for each worker and task, there is a cost for assigning the worker to the task. The problem is to assign each worker to at most one task, with no two workers performing the same task, while minimizing the total cost.

An assignment corresponds to a subset of the edges, in which each worker has at most one edge leading out, and no two workers have edges leading to the same task. One possible assignment is shown below The total cost of the assignment is 70 + 55 + 95 + 45 = 265..



Path Construction

There are a variety of solutions for TSP, which include several kinds of construction methods. The construction method is starting from a city, adding the unvisited cities into the tour constantly, and the process will be ended until all the cities have been inserted.

Insertion method is a typical kind of construction method. Firstly, it selects a city randomly from all the cities and finds the nearest city. Furthermore, it inserts the unvisited city into the constructed sequence according to different rules. Finally, the method stops until all the cities have been inserted into the sequence. Insertion methods can be divided into these four categories by selected different rules:

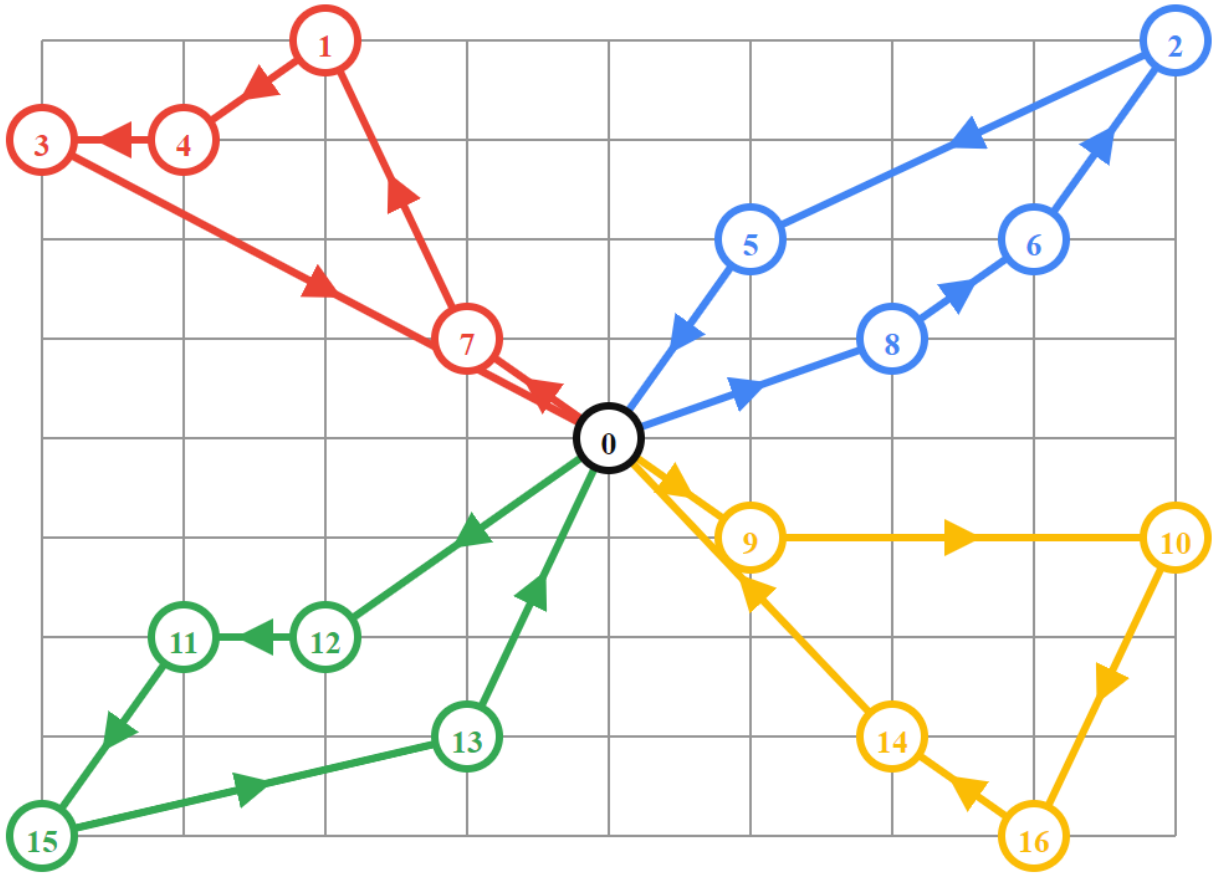
- Nearest Insertion (NI)
- Cheapest Insertion (CI)
- Farthest Insertion (FI)
- Random Insertion (RI)

For the NI, it selects the nearest city for the next city to be inserted; for the CI, it chooses the next city with the shortest added distance after inserting; for the FI, it selects the farthest distance city from all nearest cities as the next one; for the RI, it chooses the inserted city randomly.

Among these insertion methods, the performances of FI and RI are better than others, which averagely exceed about 15% compared to the HK bound, while the performances of NI and CI are a little worse with around 20%.

Space Filling Curve [50] is another heuristics construction method for TSP. It first sorts the nodes in the order of their appearance along the space filling curve and then visits these nodes in the plane to make the short tour. The time complexity of this method is O(N Log N), and the space complexity is O(N).

Once we fix warehouse assignments, the next step is to determine which order customers should be served. OR-Tools also has a routing solver to find an optimal sequence. Given a starting location and a set of customers to serve, it produces a routing sequence that minimizes the longest distance traveled by any one vehicle.



Here We use the simplest version of the routing solver as a quick way to find customers close together for drones delivering a load to two or more customers. The solver can deal with multiple vehicles, capacity constraints, and more.

Simplest version of the routing solver is like:

Common Routing Tasks

The following sections explain how to do some common tasks related to solving vehicle routing problems.

Search limits

Vehicle routing problems with many locations can take a long time to solve. For such problems, it is a good idea to set a search limit, which terminates the search after a specified length of time or number of solutions returned.

Setting initial routes for a search

For some problems, you might want to specify a set of initial routes for a VRP, rather than letting the solver find an initial solution—for example, if you have already found a good solution to a problem, and want to use it as a starting point to solve a modified problem.

To create the initial routes do the following steps:

- Define an array containing the initial routes.
- Create the initial solution using the method ReadAssignmentFromRoutes.

Setting start and end locations for routes

So far, we have assumed that all vehicles start and end at a single location, the depot. You can also set possibly different start and end locations for each vehicle in the problem. To do so, pass two vectors, containing the indices of the start and end locations, as inputs to the RoutingModel method in the main function.

Allowing arbitrary start and end locations

In other versions of the vehicle routing problem, vehicles are allowed to start and end at arbitrary locations.
To set up the problem this way, simply modify the distance matrix so that distance from the depot to any other location is 0, by setting the first row and column of the matrix to have all zeros.

This turns the depot into a dummy location that has no effect on the optimal routes.

In an earlier version I allocated drones based on the workload at each warehouse and put orders in straight numeric sequence. We'll keep the code here even though it's not needed except for consistency of variable names.

This is the adjustment using Vehicle Routing. Drones are still assigned to a single warehouse and take turns for delivery according to a straight rotation - again, opportunities for improvement.

Implementation

Warehouse assignments

Warehouses are assigned to orders in a way that minimizes the total distance traveled by drones to fulfill orders. Assignments are made product by product. Of course, any time we divide a problem into parts and optimize those parts, we are almost sure to suboptimize the overall objective. Still, it's a reasonable place to start.

There are different methods to use when tackling an assignment problem. Here We use the minimum cost flow solver to assign warehouses. The basic idea is you have a network of nodes and edges. There's a notional source node, a set of worker nodes (warehouses), a set of task nodes (orders) and a notional sink node. Here's a generic view of such a network.

Here We use the SolveMaxFlowWithMinCost variation, which doesn't require supply and demand to be equal. That allows one to do away with source and sink nodes and simplify the network. It can also find a different optimum from the balanced min-cost solver.

OR-tools function to assign warehouses to orders using a max-flow min-cost solver. Numbering scheme is as follows:

- warehouses = 1250 to 1259
- customers/orders = 0 to 1249

Supply and demand do not have to be equal.

```
def assign_whs(supply, wh_locs, demand, cust_locs):

    assignments = []
    count = 0
    distances = distance_matrix(cust_locs, wh_locs)

# iterate over products
    for i in range(400):
        item_count = 0
```

```
# Network description

        start_nodes = np.repeat(np.arange(1250,1260), 1250).tolist()
        end_nodes = np.tile(np.arange(0,1250), 10).tolist()
        capacities = np.tile(demand[i], 10).tolist()
        costs = np.transpose(distances).ravel().astype(int).tolist()
        supplies = np.negative(demand[i]).tolist() + supply[i].tolist()

# nodes in numerical order
# Build solver

        min_cost_flow = pywrapgraph.SimpleMinCostFlow()

        for s in range(len(start_nodes)):
            min_cost_flow.AddArcWithCapacityAndUnitCost(
                start_nodes[s], end_nodes[s], capacities[s], costs[s]
                )
        for s in range(len(supplies)):
            min_cost_flow.SetNodeSupply(s, supplies[s])


# Solve

        if min_cost_flow.SolveMaxFlowWithMinCost() ==
            min_cost_flow.OPTIMAL:
            for arc in range(min_cost_flow.NumArcs()):
                if min_cost_flow.Flow(arc) > 0:
                    warehouse = min_cost_flow.Tail(arc) - 1250
                    customer = min_cost_flow.Head(arc)
                    product = i
                    quant = min_cost_flow.Flow(arc)
                    cost = min_cost_flow.UnitCost(arc)
                    assign = [warehouse, customer, product, quant, cost]
                    assignments.append(assign)
                    item_count += quant
        count += item_count

    print(f"Products available: {supply.sum()} \n"
            f"Products ordered: {demand.sum()} \n"
            f"Products delivered: {count}")

    return np.array(assignments)
```

```
# main

assignments = assign_whs(supply, wh_locs, demand, cust_locs)
assign_df = pd.DataFrame(assignments,
                         columns=['wh', 'cust', 'prod_', 'quant', 'dist']
                        )
assign_df
```

- **Products available: 14576**
- **Products ordered: 9368**
- **Products delivered: 9368**

	wh	cust	prod_	quant	dist
0	3	187	0	1	36
1	4	777	0	1	176
2	5	99	1	1	159
3	5	148	1	1	88
4	5	295	1	1	89
...	...	...	...	...	...
9231	6	1169	399	1	175
9232	6	1180	399	1	211
9233	6	1181	399	1	269
9234	6	1207	399	1	85
9235	6	1217	399	1	74

Customer Sequence

Since we fixed the warehouse assignments, the next step is to determine which order customers should be served. OR-Tools also has a routing solver to find an optimal sequence. Given a starting location and a set of customers to serve, it produces a routing sequence that minimizes the longest distance traveled by any one vehicle.

Here I use the simplest version of the routing solver as a quick way to find customers close together for drones delivering a load to two or more customers. The solver can deal with multiple vehicles, capacity constraints, and more.

```
def order_orders(df):

    customers = df.cust.unique()
    demand = df.groupby('cust')['quant'].sum()

    locs = np.vstack((cust_locs[customers], wh_locs[0]))

    distances = np.ceil(distance_matrix(locs, locs)).astype(int)

    customer_map = dict(zip(customers, range(len(customers))))

    data = {}
    data['dists'] = distances.tolist()
    data['drone_count'] = 1
    data['warehouse'] = len(locs) - 1

#Create the routing index manager

    manager = pywrapcp.RoutingIndexManager(
              len(data['dists']),
              data['drone_count'],
              Data['warehouse']
    )

    routing = pywrapcp.RoutingModel(manager)

# Create and register a transit callback

    def distance_callback(from_index, to_index):
        """Returns the distance between the two nodes."""

# Convert from routing variable Index to distance matrix NodeIndex.

        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        return data['dists'][from_node][to_node]
```

```
transit_callback_index =
    routing.RegisterTransitCallback(distance_callback)

# Define cost of each arc

    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)


# Setting first solution heuristic

    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
            routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)


# Solve the problem

    solution = routing.SolveWithParameters(search_parameters)


# Get vehicle routes

    routes = []
    for route_nbr in range(routing.vehicles()):
        index = routing.Start(route_nbr)
        route = [manager.IndexToNode(index)]
    while not routing.IsEnd(index):
      index = solution.Value(routing.NextVar(index))
      route.append(manager.IndexToNode(index))
    routes.append(route[1:-1])


# Single vehicle approximation

    route = routes[0]

    reverse_dict = {v: k for k,v in customer_map.items()}
    cust_ids = [reverse_dict[r] for r in route]

    df['cust_sort'] = pd.Categorical(df.cust, cust_ids)
    df = df.sort_values('cust_sort')
    return df
```

In an earlier version of the notebook I allocated drones based on the workload at each warehouse and put orders in straight numeric sequence. I'll keep the code here even though it's not needed except for consistency of variable names.

```
def load_drones(df):
    test_wt = 0
    load_wts = []
    df = df.sort_values('cust')

    for i,tup in enumerate(df.itertuples()):
        test_wt += tup.weight

        if test_wt <= params['WT_CAP']:
            load_wt = test_wt
        else:
            load_wt = tup.weight
            test_wt = tup.weight
        load_wts.append(load_wt)

    df['load_weight'] = load_wts
    df['load_tag'] = df.load_weight.eq(df.weight).cumsum()-1
    return df

def set_loads(assignments):
    assign_df = pd.DataFrame(
               assignments,
               columns=['wh', 'cust', 'prod_','quant', 'dist']
               )

# Monster method chain to deal with quantities > 1 and define loads

    assign_df = assign_df
         .reindex(assign_df.index.repeat(assign_df.quant)) \
         .reset_index(drop=True) \
         .assign(quant=1,
                 weight = lambda x: weights[x.prod_.to_numpy()],
                 work = lambda x: x.dist * x.weight) \
         .groupby('wh', as_index=False).apply(load_drones) \
         .sort_values(['wh', 'cust', 'load_tag']) \
         .reset_index(drop=True)
   return assign_df
```

```
def assign_drones(assign_df):

    wh_work = assign_df.groupby('wh')['work'].sum()
    drones_per_wh = (wh_work/ wh_work.sum() * params['DRONE_COUNT'])
    drone_counts = drones_per_wh.round(0).astype(int)

    if drone_counts.sum() != params['DRONE_COUNT']:
       drone_counts=np.ediff1d(
                    drones_per_wh.cumsum()
                    .round(0)
                    .astype(int), to_begin=drone_counts[0]
       )
    drone_whs = np.repeat(np.arange(len(wh_locs)), drone_counts)
    drone_dict = dict(zip(range(params['DRONE_COUNT']), drone_whs))

    drone_assigns = {}
    for k, v in drone_dict.items():
        drone_assigns[v] = drone_assigns.get(v, []) + [k]

    df_list = []
    for grp, df in assign_df.groupby('wh'):
        drone_ids = drone_assigns[df.wh.iloc[0]]
        df['drone_id'] =
        df.load_tag % len(drone_ids) + min(drone_ids)
        df_list.append(df)

    df_end = pd.concat(df_list)

    return df_end
# main

   assign_df = set_loads(assignments)
   df_end = assign_drones(assign_df)
   df_end = df_end.groupby(
            ['wh', 'cust', 'load_tag', 'drone_id', 'prod_'],
            as_index=False
            )['quant'].sum()
```

	wh	cust	load_tag	drone_id	prod_	quant	cust_sort	weight	load_weight
0	0	777	0	0	225	1	777	108	108
1	0	777	1	1	150	1	777	110	110
2	0	777	2	2	277	1	777	100	100
3	0	777	2	2	62	1	777	2	102
4	0	193	2	2	15	1	193	93	195
...	...	...	...	...	...	...	...	...	...
9303	9	160	436	28	128	1	160	21	67
9304	9	1116	436	28	386	1	1116	31	98
9305	9	1078	436	28	369	1	1078	46	144
9306	9	1248	436	28	265	1	1248	28	172
9307	9	198	437	29	256	1	198	87	87

This is the adjustment using Vehicle Routing. Drones are still assigned to a single warehouse and take turns for delivery according to a straight rotation - again, opportunities for improvement.

```
def load_drones_improved(df):
    test_wt = 0
    load_wts = []
    for i,tup in enumerate(df.itertuples()):
        test_wt += tup.weight
        if test_wt <= params['WT_CAP']:
            load_wt = test_wt
        else:
            load_wt = tup.weight
            test_wt = tup.weight
        load_wts.append(load_wt)

    df['load_weight'] = load_wts
    df['load_tag'] = df.load_weight.eq(df.weight).cumsum()-1

return df
```

```
def reset_loads(df_end_reordered):
    df_end_reordered = df_end_reordered
                        .reset_index(drop=True) \
                        .assign(
                                  weight = lambda x:
                                  weights[x.prod_.to_numpy()] * x.quant) \
                        .groupby('wh',as_index=False)
                        .apply(load_drones_improved)
    return df_end_reordered

def assign_drones(df_end_reordered):
    df_list = []
    for grp, df in df_end_reordered.groupby('wh'):
        drone_ids = df.wh.iloc[0]
        df['drone_id'] =
            df.load_tag % df.drone_id.nunique() + min(df.drone_id)
        df_list.append(df)
    df_end = pd.concat(df_list)

return df_end

df_end_reordered = df_end.groupby('wh').apply(order_orders) \
                          .pipe(reset_loads) \
                          .pipe(assign_drones)

df_end_reordered
```

	wh	load_tag	prod_	drone_id	quant	weight_sum
0	0	0	225	0	1	108
1	0	1	150	1	1	110
2	0	2	15	2	1	93
3	0	2	62	2	1	2
4	0	2	277	2	1	100
...	...	...	...	...	...	...
9088	9	436	128	28	1	21
9089	9	436	265	28	1	28
9090	9	436	369	28	2	92
9091	9	436	386	28	1	31
9092	9	437	256	29	1	87

Visualization

Drone

First we had to create a Drone class for ease of use in visualization. Every Drone object will have the Drone ID and the drone capacity.

Each Drone object will have an:

- ID
- X, Y: Coordinates
- Capacity
- Number of packages at exact moment
- Assigned client's id
- X_Client, Y_Client: Coordinates of assigned client

```
class Drone:
    def __init__(self, id, capacity):
        self.id = id
        self.capacity = capacity
        self.num_of_packages = 0
        self.temp_client_id = None
        self.x, self.y = 0, 0
        self.x_client, self.y_client = None, None
        self.x_prev_client, self.y_prev_client = None, None

    def __repr__(self):
        return '(Drone {}, Capacity: {})'.format(self.id, self.capacity)

    def change_position(self, x, y):
        """Change drone position"""
        self.x, self.y = x, y

    def create_log(self, log):
        """Print log and save to file"""
        print(log)
        with open("logs.txt", "a") as file_object:
            file_object.write(log + "\n")

    def get_distance_from_client(self):
        """Euclidian distance from client"""

    return np.sqrt((self.x_client-self.x)**2 + (self.y_client-self.y)**2)
```

Deliver package: update number of packages and set appropriate client attributes.

```
    def deliver_package(self, elapsed_time):
        self.num_of_packages -= 1
        self.x_prev_client = self.x_client
        self.y_prev_client = self.y_client
        self.create_log(f"Time: {elapsed_time} min
        (Drone: {self.id} | Packages: {self.num_of_packages}) -
         Package delivered to client with id {self.temp_client_id}")
        if self.num_of_packages == 0:
            self.x_client = 0
            self.y_client = 0
        else:
            self.x_client = None
            self.y_client = None
        self.temp_client_id = None
```

Load packages equal to capacity.

```
    def load_packages(self):
        self.num_of_packages = self.capacity
        self.temp_client_id = None
```

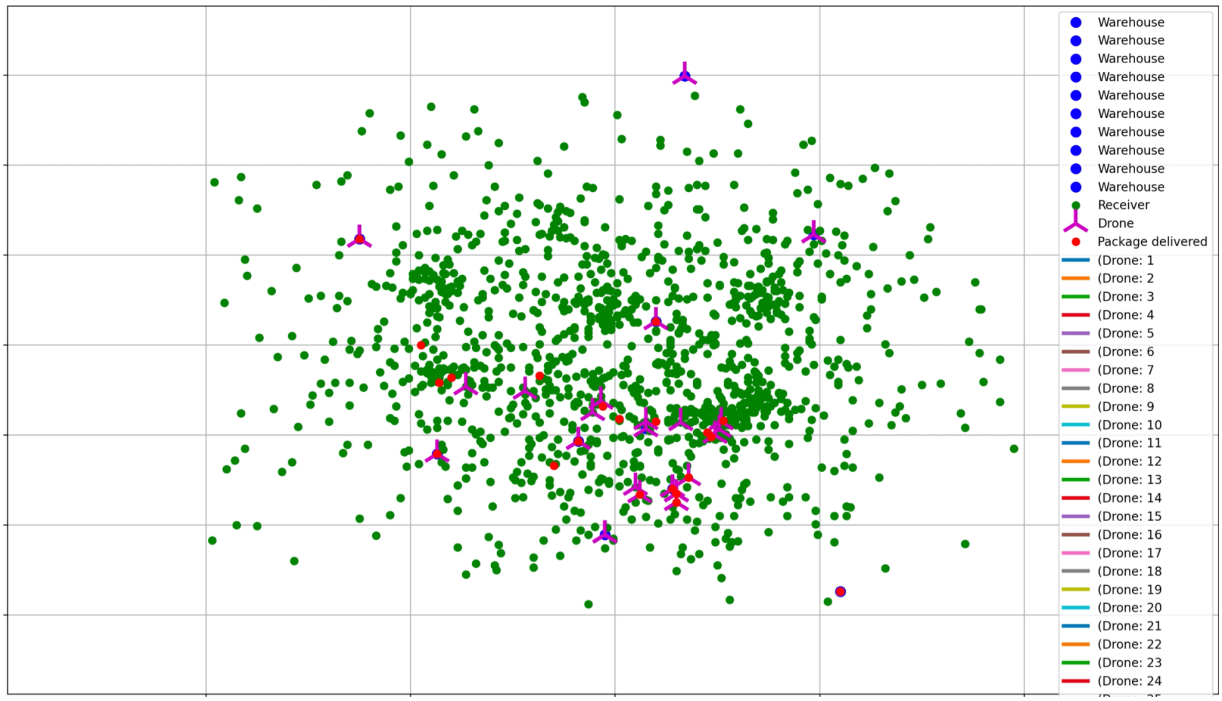
Assign passed client to drone.

```
    def specify_client(self, client):
        if self.temp_client_id == None:
            self.temp_client_id = client.id
            self.x_client = client.x
            self.y_client = client.y
```

Drone travel each call change position by 1 in direction to assigned client, if distance <= 1 deliver package.

```
    def travel(self, elapsed_time):
        if self.x_client is not None and self.y_client is not None:
            distance = self.get_distance_from_client()
            if distance <= 1:
```

```
        if self.x_client == 0 and self.y_client == 0:
            self.load_packages()
            self.create_log(f"Time: {elapsed_time} min(
                Drone:   {self.id} |
                Packages: {self.num_of_packages}) - In base"
            )
            self.change_position(0, 0)

        else:
            self.deliver_package(elapsed_time)
            self.x = self.x_prev_client
            self.y = self.y_prev_client

    else:
        v_x = self.x_client - self.x
        v_y = self.y_client - self.y
        new_x = self.x + 1/distance * v_x
        new_y = self.y + 1/distance * v_y
        self.change_position(new_x, new_y)
```



Client

Second thing we will do is to create a Client class to wrap the customers and warehouses in. Each Client object will have:

- ID
- X, Y: Coordinates

```
class Client:
    def __init__(self, id, x, y):
        self.id = id
        self.x, self.y = x, y

    def __repr__(self):
        return f'Client {self.id} ({self.x}, {self.y})'

    def __eq__(self, other):
        return self.id == other.id
```

Euclidian distance from a given point.

```
    def get_distance_from(self, point):
        """Euclidian distance from given point"""
        return np.sqrt((point.x-self.x)**2 + (point.y-self.y)**2)
```

Pyplot Visualization

We will be using pyplot to visualize our solution. To do so, we will need to first of all create a new class to encapsulate and tie up our code more.

The WithVisualization class will have information about:

- Drones: All available drones
- Clients: Not visited clients
- X_Visited, Y_Visited: Visited clients coords
- Total time: Total time passed since running visualization

```
from matplotlib import pyplot as plt

class WithVisualization:
    def __init__(self, obj):
        self.drones = obj.drones
        self.clients = obj.clients
        self.solution = obj.processed_solution
        self.warehouses = obj.warehouses
        self.total_time = 0
        self.threshold = 2000

        self.x_clients, self.y_clients =
        self._initialize_client_positions(obj.clients)
        self.x_visited, self.y_visited = [], []
        self.x_drones, self.y_drones = [], []

    @staticmethod

    def _initialize_client_positions(clients):
        x_clients, y_clients = [], []
        for c in clients:
            x_clients.append(c.x)
            y_clients.append(c.y)
        return x_clients, y_clients
```

```
def update_drone_positions(self):
        """Drone path update"""
        for drone in self.drones:
            self.x_drones.append(drone.x)
            self.y_drones.append(drone.y)

    def update_visited_clients(self, x, y):
        """Updating visited clients"""
        self.x_visited.append(x)
        self.y_visited.append(y)

    def assign_clients(self):
        """Assign all clients to drones"""
        for drone in self.drones:
            if self.solution[drone]:
                drone.specify_client(self.solution[drone].pop(0))


    def assign_client(self, drone_id):
        """Assign one client to specified drone with passed ID"""
        for drone in self.drones:
            if drone.id == drone_id:
                if self.solution[drone]:
                    drone.specify_client(self.solution[drone].pop(0))
                    return True
                else:
                    return False

    def plot_figure(self, sizes=(12, 12)):
        fig = plt.figure(figsize=sizes)

    def plot_warehouses(self, ax):
        for warehouse in self.warehouses:
            ax.plot(warehouse.x, warehouse.y,
                    'bo', markersize=8, label="Warehouse")

    def visualize_solution(self):
        """Run real time visualization"""
        fig, ax = plt.subplots(figsize=(20, 20))
        k, elapsed_time, D = 1, 0, len(self.drones)
        delivered = False
```

```
while not delivered:
    delivered = True
    k = 1 if k == 4 else k + 1
    for drone in self.drones:
        if drone.temp_client_id ==
            None: is_assigned = self.assign_client(drone.id)
            if is_assigned or
               drone.x != 0 or
               drone.y != 0: delivered = False
            drone.x, drone.y = drone.x_client, drone.y_client
            Self.update_visited_clients(
            drone.x_prev_client, drone.y_prev_client
            )
        else:
            drone.travel(elapsed_time)
            delivered = False
    elapsed_time += 1
    self.update_drone_positions()
    self.plot_warehouses(ax)
    ax.plot(self.x_clients, self.y_clients,
              'go', markersize=6, label="Receiver")
    ax.plot(self.x_drones[-D:], self.y_drones[-D:],
               'm{}'.format(k), markersize=24,
                markeredgewidth=3, label="Drone")
    ax.plot(self.x_visited, self.y_visited,
                'ro', markersize=6, label="Package delivered")
    for s in range(D):
        ax.plot(self.x_drones[s::D], self.y_drones[s::D],
                    linewidth=3,label=f"(Drone: {s + 1}")

    ax.set_ylim(
        min(self.y_clients) - 100, max(self.y_clients) + 100
    )
    ax.set_xlim(
        min(self.x_clients) - 100, max(self.x_clients) + 100
    )
    ax.legend()
    ax.grid()
    fig.canvas.draw()
    renderer = fig.canvas.renderer
    ax.draw(renderer)
    plt.pause(0.001)
    ax.cla()
```

```
        self.plot_warehouses(ax)
        ax.plot(self.x_clients, self.y_clients,
                    'go', markersize=6, label="Receiver")
        self.update_drone_positions()
        for s in range(D):
            ax.plot(self.x_drones[s::D], self.y_drones[s::D],
                        linewidth=3, label=f"(Drone: {s + 1},
                        Packages: {self.drones[s].num_of_packages})")


        ax.set_ylim(min(self.y_clients) - 100, max(self.y_clients) + 100)
        ax.set_xlim(min(self.x_clients) - 100, max(self.x_clients) + 100)
        ax.legend()
        ax.grid()
        plt.show()


    def plot_solution(self):
        """Plot final solution"""
        for d in self.drones:
            if d in self.solution:
                x_s, y_s = zip(*[(point.x, point.y) for
                                 point in self.solution[d]])
                plt.plot(x_s, y_s, label=f'{d}')
        plt.plot(self.x_clients, self.y_clients,
                     'ro', markersize=6, label="Receiver")
        for warehouse in self.warehouses:
            plt.plot(warehouse.x, warehouse.y,
                         'bo', markersize=8, label="Warehouse")
        plt.grid()
        plt.legend()
        plt.ylim(min(self.y_clients) - 100, max(self.y_clients) + 100)
        plt.xlim(min(self.x_clients) - 100, max(self.x_clients) + 100)
        plt.show()
```

The Solution class' purpose is to group all of the drones, clients, warehouses, and the final drone path.

```
class Solution:
    def __init__(self, drones, clients, solution, warehouses):
        self.drones = drones
        self.clients = clients
        self.processed_solution = solution
        self.warehouses = warehouses
```

Pyplot and Python QT

To use Pyplot's real time animation and visualization we will integrate it with Python QT. We will use python's magic functions.

```
%matplotlib qt
```

Extract Drones and Clients positions

Now we will need to create a drones array, warehouses array, and customers array.

```
drones = []
drone_ids = set(df_end_loading['drone_id'].tolist())
for drone_id in drone_ids:
  drones.append(Drone(drone_id, float('inf')))

warehouses_clients = []
warehouses_ids = set(df_end_loading['wh'].tolist())
for warehouses_id in warehouses_ids:
                            warehouses_clients.append(Client(warehouses_id,
wh_locs[warehouses_id][0], wh_locs[warehouses_id][1]))

custs = []
custs_ids = set(df_end['cust'].tolist())
for custs_id in custs_ids:
        custs.append(Client(custs_id,
                            cust_locs[custs_id][0],
                            cust_locs[custs_id][1])
                     )
```

Generate Instructions From Output

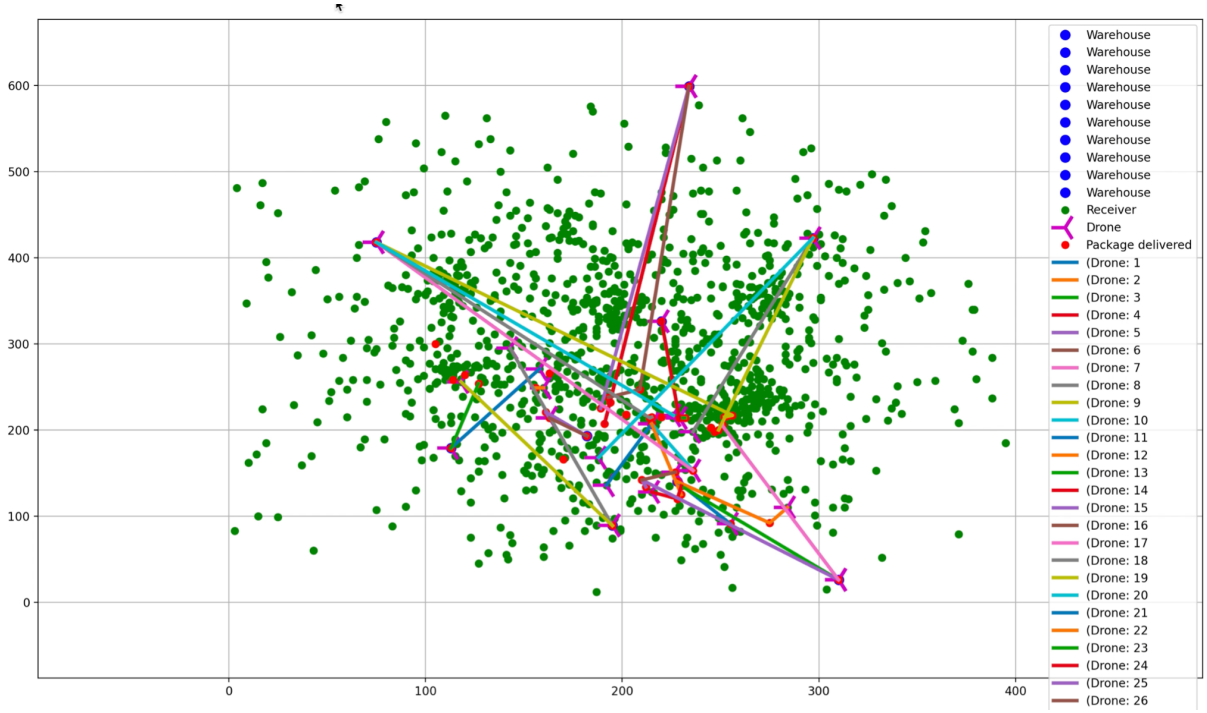
Now we need to get each drone path from the final output. That will mean create a dictionary with a drone as a key and a list as a path.

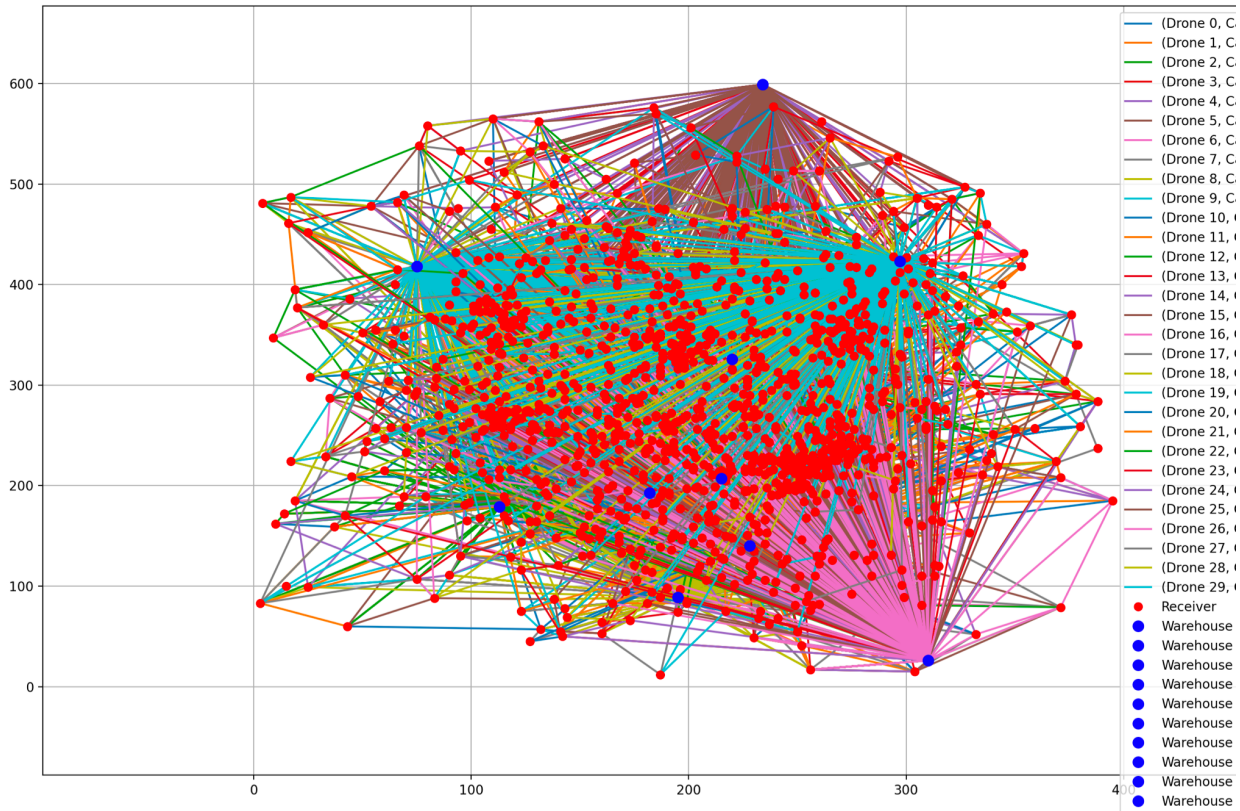
```
from collections import defaultdict

def generate_solution(insts):
    res = defaultdict(list)
    for inst in insts:
        if inst[1] == 'L':
            res[drones[inst[0]]].append(warehouses_clients[inst[2]])
        elif inst[1] == 'D':
            res[drones[inst[0]]].append(custs[inst[2]])

    return dict(res)

solution = generate_solution(instructions)
```

Visualize

Finally, let's create a Solution object that will be constructed from the previously created drones, customers, warehouses arrays, and the drone path.

```
final_sol = Solution(drones, custs, solution, warehouses_clients)

vis = WithVisualization(final_sol)
vis.visualize_solution()
```


Warehouse
Receiver
Drone
Package delivered
(Drone: 1
(Drone: 2
(Drone: 3
(Drone: 4
(Drone: 5
(Drone: 6
(Drone: 7
(Drone: 8
(Drone: 9
(Drone: 10
(Drone: 11
(Drone: 12
(Drone: 13
(Drone: 14
(Drone: 15
(Drone: 16
(Drone: 17
(Drone: 18
(Drone: 19
(Drone: 20
(Drone: 21
(Drone: 22
(Drone: 23
(Drone: 24
(Drone: 25
(Drone: 26
0
100
200
300
400
500
600


Receiver
Warehouse
0
100
200
300
400
500
600

Accuracy and Performance

So you've built a model and trained it on some data... Now what? In this chapter, we'll discuss how to evaluate your model, and practical advice for improving the model based on what we learn evaluating it.

In this chapter we will discuss these main ideas:

- Confusion Matrix for Binary Classification
- Regression metrics
- Mean squared error

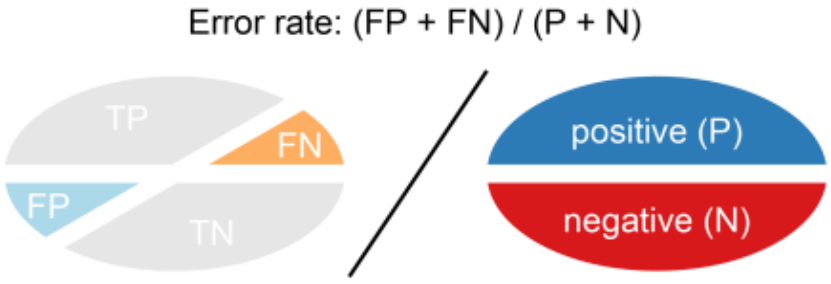
Confusion Matrix for Binary Classification

A confusion matrix of binary classification is a two by two table formed by counting of the number of the four outcomes of a binary classifier. We usually denote them as TP, FP, TN, and FN instead of “the number of true positives”, and so on.

		Predicted	
		Positive	**Negative**
Observed	**Positive**	TP (# of TPs)	FN (# of FNs)
	Negative	FP (# of FPs)	TN (# of TNs)

Error rate

Error rate (ERR) is calculated as the number of all incorrect predictions divided by the total number of the dataset. The best error rate is 0.0, whereas the worst is 1.0.



Error rate is calculated as the total number of two incorrect predictions (FN + FP) divided by the total number of a dataset (P + N).

- $\mathrm{ERR} = \frac{\mathrm{FP} + \mathrm{FN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FN} + \mathrm{FP}} = \frac{\mathrm{FP} + \mathrm{FN}}{\mathrm{P} + \mathrm{N}}$

Regression Predictive Modeling

Predictive modeling is the problem of developing a model using historical data to make a prediction on new data where we do not have the answer.

Predictive modeling can be described as the mathematical problem of approximating a mapping function (f) from input variables (X) to output variables (y). This is called the problem of function approximation.

The job of the modeling algorithm is to find the best mapping function we can given the time and resources available.

Mean Squared Error Definition

The mean squared error (MSE) tells you how close a regression line is to a set of points. It does this by taking the distances from the points to the regression line (these distances are the "errors") and squaring them. The squaring is necessary to remove any negative signs. It also gives more weight to larger differences. It's called the mean squared error as you're finding the average of a set of errors. The lower the MSE, the better the forecast.

MSE formula = (1/n) * Σ(actual – forecast)2
Where:

n = number of items,
Σ = summation notation,
Actual = original or observed y-value,
Forecast = y-value from regression.
General steps to calculate the MSE from a set of X and Y values:

Find the regression line:

1. Insert your X values into the linear regression equation to find the new Y values (Y').
2. Subtract the new Y value from the original to get the error.
3. Square the errors.
4. Add up the errors (the Σ in the formula is summation notation).
5. Find the mean.

At this point we make a score to calculate how far our solution from the optimal solution, we do this by some formula then calculate the accuracy on this score, so how the score work

Scoring

Each completed order will earn between 1 and 100 points, depending on the turn in which it is completed.
The order is completed in the first turn at the end of which all items in the order are delivered.
For an order completed in turn t and a simulation taking
T turns in total, the score for the order is calculated
as $(T - t) / T \times 100$, rounded up to the next integer.

For example, if the simulation takes 160 turns (T = 160), and an order consists of three items, delivered at turns 5, 15 and 15, then the order is considered completed at t = 15, and the score is calculated as $(160 - 1 5)/160 = 0.90625$, multiplied by 100 and rounded up to 91 points.
For an order completed in the last turn of the simulation (t = 159, T = 160), the score would be $(160 - 159)/160 = 0.00625$, multiplied by 100 and rounded up to 1 point

Calculation Of The Accuracy

```
def calculate_score(file):
    submission = pd.read_csv(file)
    allcommands = submission[submission.columns[0]].values

    delivery_times = orders_df.copy()
    missing_items = orders_df.copy()

    inventory_ops = pd.DataFrame(
                    columns=['action', 'wh', 'item', 'count', 'turn'])

    for ddd in tqdm(range(DRONES)):
        dronecommands =
            [iii for iii in allcommands if iii.split()[0 == str(ddd)]
        currentloc = warehouse_df.loc[0].values
        currenttime = 0
        currentweight = 0

        for cmd in dronecommands:
            _, action, locidx, prod, count = cmd.split(' ')

        # add time steps required to reach new location and
          perform loading / unloading / delivery

            if action == 'L' or action == 'U':
                newloc = warehouse_df.loc[int(locidx)].values
            elif action == 'D':
                newloc = orders_df.loc[int(locidx), ['row', 'col']].values
            elif action == 'W':
                # no further action needed in case of "wait" commands
                currenttime += locidx
                continue

            dist = int(np.ceil(np.sqrt(np.sum((currentloc - newloc) ** 2))))
            currenttime += dist

            # add one step for loading / unloading / delivery itself
            currenttime += 1

            # check if end of simulation is reached
            if currenttime > TURNS:
                raise Exception('Maximum simulation time exceeded')
```

```
# update current location
currentloc = np.copy(newloc)

# update drone weight
if action == 'L':
    currentweight +=
        int(count) * products_df.loc[int(prod), 'weight']

elif action == 'D' or action == 'U':
    currentweight -=
        int(count) * products_df.loc[int(prod), 'weight']

if currentweight > MAXLOAD:
    raise Exception('Maximum drone load exceeded')

# note latest delivery time of each item for each order
# and note how many items were delivered

if action == 'D':
    if missing_items
            .at[int(locidx), 'p_{}'.format(prod)]:
       Delivery_times
            .at[int(locidx), 'p_{}'.format(prod)]
       = currenttime

       Missing_items.
         at[int(locidx), 'p_{}'.format(prod)] -= int(count)

    else:
        raise Exception('Too many items delivered')


# save list of loading / unloading operations for
  checking warehouse inventory

if action == 'L' or action == 'U':
    inventory_ops = inventory_ops.append({
        'action': action,
        'wh': int(locidx),
        'item': int(prod),
        'count': int(count),
        'turn': currenttime
    }, ignore_index=True)
```

```
for wh in range(len(warehouse_df)):
    for item in tqdm(range(len(products_df))):

        # all inventory operations at this warehouse involving this product
        tmp = inventory_ops[
            (inventory_ops['wh'] == wh) &
            (inventory_ops['item'] == item)
            ]
        if not len(tmp):
            continue
        tmp = tmp.sort_values(by='turn')

        # get initial stock
        inv = products_df.loc[item, f'wh{wh}_inv']

        # if overall fewer or just as many products
          are removed as are stored
        # in the warehouse, all good

        if len(tmp[tmp['action'] == 'L']) <= inv:
            continue

        # otherwise, "simulate" loading and unloading to
         see whether inventory goes negative

        for iii in tmp.index:
            if tmp.loc[iii, 'action'] == 'L':
                inv -= 1
            else:
                inv += 1


            # check inventory after each step
            if inv < 0:
                raise Exception('Removal of unstocked product attempted')

completion_times =
    np.max(delivery_times.iloc[:, 3:-2].values, axis=1)
completed =
    np.max(missing_items.iloc[:, 3:-2].values > 0, axis=1) <= 0
tmp = np.where(completed)[0]
order_scores =
    np.ceil(100 * (TURNS - completion_times[tmp]) / TURNS)
score = int(np.sum(order_scores))
return score
```

```
def calculate_accuracy(output_file, optimal_solution_file):
   output_score =
       calculate_score(output_file)
   optimal_solution_score =
       calculate_score(optimal_solution_file)
   accuracy = output_score / optimal_solution_score
   return accuracy

print(calculate_accuracy('output.csv', 'optimal_solution.csv'))
```

Finally

We achieve **83%** accuracy from the optimal solutions, and our code runs on Google co-lab notebooks for about **180 seconds.**

Conclusion

Consumers want their purchases now. The promises of quicker delivery have become even greater, and with high capability comes great expectation. With many e-commerce businesses currently offering one-day deliveries, fewer people are likely to choose one-day shipping or more, even if shipping is free.

Drones are set to become the future of logistics with their reduced cost, higher convenience and delivery times of less than 30 minutes. Both online retailers and brick and mortar stores will adopt it to smoothen their last mile delivery process. Stores like Walmart can experience increased efficiency and convenience with their local presence while online retailers like Amazon can offer quick deliveries at reduced costs. The early adopters will be the winners as they will be able to provide their services at cheaper and quicker rates leading to brand promotion.

Advantages

Lead Time

A drone delivery system that can deliver goods in 30 minutes or less will reduce lead time. Traffic, stop signs, and other challenges on the road all have a considerable impact on delivery times. Thankfully, due to drones travelling through air, deliveries will avoid all traffic-related influences.

Reduced Shipping & Operational Costs

According to ARK Investment Management, it's possible that companies could charge just £1 to deliver drone packages. Amazon has stated that they plan to start the process with small packages, weighing less than 5 lbs. Considering that most orders shipped by Amazon weigh less than that, the reduced shipping and operational costs will benefit a high percentage of customers.

Speed Delivery

Gone are the days where an item is picked in a warehouse, put on a carrier and delivered through the usual steps to a customer. Instead, at the click of a mouse, the system will arrange for the drone to pick the order and deliver directly to the person who ordered it, eliminating route restrictions and many other logistic obstacles.

Reduce Return Issues

In the age of progressive technology, speed is everything. Not only do consumers want instant gratification through speedy delivery times, if they are unhappy with the product they will also expect a refund quickly so they can either order a replacement or simply get their money back.

Drones not only reduce return time cycles but also the bother of having to go to the post office to return a parcel. Heavy duty drones will pick up damaged goods and return to the sender. Returns will even process while the drone is still in transit. The quicker turnaround will result in faster returns without the stress of waiting for confirmation.

Subsequent work

Many different adaptations, tests, and experiments have been left for the future due to the huge scope of the project (i.e. the experiments with real data are usually very time consuming, requiring even several hours to finish a single run).

Future work concerns deeper analysis of particular mechanisms, new proposals to try different methods, or simply curiosity.

Future Research Directions in HVRP :

Utilizing HVRP can help companies or organizations reduce their total travel cost by determining the optimal route that serves more consumers in less dependence on fossil fuels. Future developments in this field could address the definition extensions of this problem in which traffic limited zones, in which it is forbidden to use traditional fuel propulsion are considered. Other developments could address the possibility of partial battery recharging and the usage of the so-called regenerative braking which allows EVs to recover a percentage of their battery charge while traveling on downhill roads. Moreover, multiple recharging technologies, with different recharging times, costs, and availability could be considered.

We still need to study this approach to gain more knowledge of its pros and cons and to determine it's constraints and accuracy.

Planning as soon as possible to excel in this field to reach a notch in the market of delivery and remote shopping, to be the first in the EMEA who apply the Sky Express delivery by the fleet of hero drones started from KafrElSheikh University with a team of 7 talented engineers.

References

1. Ha, Q.M., Deville, Y., Pham, Q.D., Ha, M.H.: Heuristic Methods for the TravelingSalesman Problem with Drone. Technical Report, CTEAM/INGI/EPL, pp. 1–13(2015)

2. Jiang, X., Zhou, Q.,Ye, Y.: Method of Task Assignment for UAV Based on ParticleSwarm Optimization in logistics. Proceedings of the 2017 International Conferenceon Intelligent Systems, Metaheuristics & Swarm Intelligence - ISMSI ’17, pp. 113–117, ACM Press (2017)

3. Mourelo, F.S., Harbison, T., Weber, T., Sturges, R., Rich, R.: Optimization of aTruck-Drone in Tandem Delivery Network using K-means and Genetic Algorithm.Journal of Industrial Engineering and Management, Vol. 9 (2), pp. 374–388 (2016)

4. Solomon, M.M.: Algorithms for the vehicle routing and scheduling problems with time window constraints. Oper. Res. 35(2), 254–265 (1987)

5. Mathew, Neil & Smith, Stephen & Waslander, Steven. (2015). Planning Paths for Package Delivery in Heterogeneous Multirobot Teams. Automation Science and Engineering, IEEE Transactions on. 12. 1298-1308. 10.1109/TASE.2015.2461213.

6. Jalel Euchi, Abdeljawed Sadok, Hybrid genetic-sweep algorithm to solve the vehicle routing problem with drones, Physical Communication,Vol 44, 2021, 101236, ISSN 1874-4907

7. Schermer D., Moeini M., Wendt O. (2018) Algorithms for Solving the Vehicle Routing Problem with Drones. In: Nguyen N., Hoang D., Hong TP., Pham H., Trawiński B. (eds) Intelligent Information and Database Systems. ACIIDS 2018. Lecture Notes in Computer Science, vol 10751. Springer, Cham.

8. Wang, X., Poikonen, S., Golden, B.: The Vehicle Routing Problem with Drones:Several Worst-Case Results. Optimization Letters, Vol. 11 (4), pp. 679–697 (2016)

9. https://developers.google.com/optimization/reference For Optimization of Network Flow and Graph & Routing.

10. https://numpy.org/doc/stable/ For Data Analysis.

11. https://scikit-learn.org/stable/user_guide.html For Machine Learning Modelling.