Project Plan:

Abstract

List of Tables

List of Figures

1. Introduction

1.1 Overview

1.2 Report Structure

1.3 Aims and Objectives

3. Design & Methodology

3.1 Overview

3.2 Functional Requirements

3.3 Non-Functional Requirements

3.4 Clustering

3.5 Path Finding

3.6 GUI

3.7 System Design

4. Implementation

4.1 GUI

4.2 Clustering

4.3 Weather Data

4.4 Flight Path Correction

4.5 Map Drawing

4.6 Database Management

4.7 Path-Finding

4.8 HTML Output

5. Testing

5.1 Acceptance Testing

5.2 Performance Testing

5.3 Path Finding Algorithms

5.4 User Testing

Abstract

Introduction

Drones are an emerging technology for delivery. Many companies are developing systems to utilise them, such as Amazon with their “Prime Air” platform, and DHL with their “Parcelcopter”. They hold many benefits over traditional road-based delivery methods, such as lower cost, faster delivery and lower environmental impact. However, there are some obstacles to overcome before they see widespread use, such as legal issues and low flight range.

As with any delivery method, it is important that drones do not waste time. Their routes need to be optimised in order to speed up delivery for customers and companies alike. Because of this, it is vital that a schedule is created and maintained that details where each drone will be going and when.

Aims and Objectives

The aim of this project is to create a system to take input of customers and their locations, and create an efficient schedule for these customers. The following objectives must be completed in order to succeed at the task:

- Implement a simple Graphic User Interface (GUI)

- Develop a system to generate routes between customers

- Present the results graphically

- Allow comparison of different algorithms

Structure of Report

Chapter 2 – Literature Review

Chapter 3 – Design and Methodology to be implemented

Chapter 4 – Implementation of said design

Chapter 5 – Testing

Chapter 6 – Evaluation

Chapter 7 – Conclusions and future work

Literature Review

bla bla bla bla bla

Design and Methodology

This chapter will discuss the requirements of the project. The requirements are a direct result of the aims and objectives, and are steps required to complete these.

Key pieces of code that have been implemented are then explored for details of their design and intended operation. Care has been taken to ensure the software is broken down into many distinct sections, this ensures that separation of concerns occurs, which will allow easier debugging and testing.

Finally, the methodology used throughout the development of this project is explored and justified.

Requirements

The requirements have been split into categories depending on importance using the MoSCoW system. Each requirement is classed as either “Must”, “Should”, “Could”, “Won’t”. For this project, “Won’t” has not been detailed.

“Must” requirements are those that the project cannot go without.

“Should” are important requirements, but not completely essential to the project.

“Could” are additional features that would be nice to have, but are a bonus, and their omission doesn’t affect the project in any way.

Functional Requirements

The functional requirements are functions of the implemented system. These detail tasks that the completed system will complete.

Must:

- A GUI – It is crucial to make the application usable to anyone that it has a simple GUI. This GUI has several characteristics that must be implemented in order to complete this task:

- A real map of a city. It allows the user to test out and see real-world results of their use.

- Input orders from the GUI. It is no good having a GUI if the user still has to input data using some other means

- Controllable parameters. One of the main aims of the project was to allow comparison of parameters. It will help users to see how adding or taking away drones for example will affect the results.

- Division of customers into smaller groups. As discussed in the Literature Review, it is vital that customers are broken down into manageable groups. Without this feature, the software will likely take far too long to complete path-finding, will be less likely to produce a good route, and also will not create a realistic scenario of multiple drones being able to fly at once.

- Sensible routes must be created. This is perhaps the main function of the application, and without it there is little to show. Routes must be created among customer groups that are sensible and follow some kind of rules in an attempt to optimise them.

Should:

- Multiple path finding algorithms implemented. This falls under the aim of comparing input parameters. It will be interesting for users to see and compare different algorithms, however providing there are other parameters that can be controlled, and one path-finding algorithm is included, this is not vital functionality.

- Consider weather conditions. One of the big drawbacks of drones is that they are greatly affected by the weather. In order to simulate a real-world system more accurately, this should be taken into consideration. Weather data such as wind speed, direction and precipitation may all be considered.

- Interactive user input. In a real-world system, the application would be running constantly and waiting for new orders to come in. If a new order is placed, the system shouldn’t shut down while it deals with this, it should be able to change an existing route or create a new one to accommodate the new customer.

Could:

- A display of the current state of routes as they are created. This would be an interesting feature to allow the user to see how the path-finding algorithm selected works.

- When running, show the location of each drone as they move through their route. This would be a bonus feature where the system is running in real time and updates with progress of drones along their routes.

Non-Functional Requirements

Non-Functional Requirements are those that do not detail functionality of the solution. They outline things such as hardware and software environments and how the solution should be delivered.

Must:

- Use python for development. Python has been selected due to familiarity from previous projects. Python is known for being useful for implementing AI solutions, which this project aims to do. Additionally there are a vast range of libraries to aid in development.

Should:

- Completed product should ship as a .exe file. This file type will run in windows, which the majority of users around the world will be running. With a little configuration this type of file can also run on Linux and MacOS. It allows users to simple double click the file and have the program run, without worrying about installing python themselves.

- Use an appropriate API for weather data if this function is implemented. An API should be selected that allows current or at least hourly weather data to be gathered either for free or for a low cost.

- Have a fast runtime. As this software is designed to emulate a real-world system, it is vital that the time taken to create a schedule is as fast as possible. If the system takes several hours to produce a solution, it would not be viable for real world use.

Could:

- Use scikit-learn library for clustering algorithms. The library provides a range of algorithms, such as both that “kMeans” and “affinity propagation”, which are detailed in the literature review. Developing a clustering algorithm ourselves is possible, however it is not the aim of this project to do.

- Use pyeasyga for the genetic algorithm. This is another free library for python. This one implements the Genetic Algorithm. As with clustering, the task that this project is aiming to solve is not whether or not this algorithm can be implemented, thus using a library is a good time-saving method.

- Could use PyQ to produce a GUI. PyQt is a powerful tool for creating attractive GUIs. It will allow a clean and simple user interface to be developed quickly.

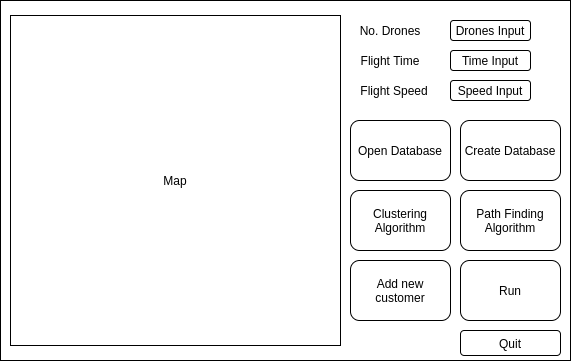
Key Components

GUI

The graphic user interface is a key part of this project. It serves to both allow input and show resulting output. A simple layout was desired so that anyone would be able to pick up and use the app without having to read instructions. Ideally, the GUI will be system independent, with it’s graphics being taken from the system defaults. This will ensure that the application is as cross-platform as possible.

Clearly labelled text inputs, buttons and drop-down boxes should be used where appropriate in order to create a clean interface.

The final GUI design is shown in diagram x.



[The GUI]

Clustering

Clustering is the second main component of the project. The clustering algorithm is responsible for breaking down the customers into manageable groups. For this task there are a range of algorithms available. The solution that is implemented should be able to group customers by their proximity to one another in a reasonable time-frame. A range of algorithms could be implemented to allow comparison between their effectiveness.

Weather Data

I would like to account for the weather with my solution. For this I will select a suitable API that will give me various pieces of data. The key two factors I wish to include are wind speed and wind direction. This will allow me to create a more realistic solution. We will also be able to use the data to improve our search algorithms to use the weather to their advantage wherever possible.

Path Finding

Path finding is the final major step in our project. Once the data has been broken into clusters, we can attempt to find a route through the customers in each cluster. Weather data is taken into account while creating routes. The effect of wind speed and bearing on flight time is used to optimise the routes and decrease the total time taken.

Time of each Leg

In order to calculate the time of each leg of the routes, we must use several calculations.

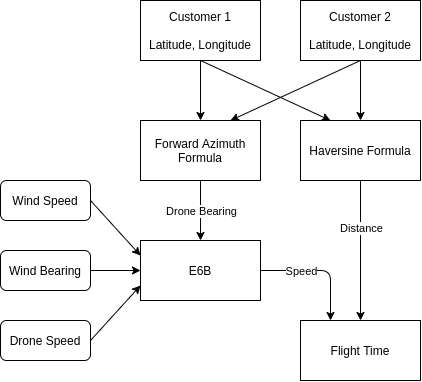
First we need to find the distance between two sets of coordinates. For this we will use the Haversine formula, which calculates the great-circle distance between two latitude longitude pairs. The great-circle is the distance accounting for the curvature of the earth. Use of this formula allows greater accuracy than doing a simple calculation using Pythagoras.

The next piece of data needed is the bearing between the two locations. There is a formula which gives the us the bearing known as the forward azimuth. As we travel along a great-circle arc, the angle we are travelling at changes slightly. The forward azimuth is the bearing required when we account for the great-circle arc.

We can use the data gathered to conduct two further calculations. The calculations are the same that are performed by pilots using device called an E6B. This device is used to calculate the required flight bearing and corrected speed accounting for wind.

Using the drone speed and desired bearing and the wind speed and bearing we can calculate the corrected bearing. Once we have this angle we can use it in the next stage to calculate the actual ground speed of the drone when accounting for the wind.

Finally we will use the great-circle length and E6B corrected flight speed to calculate the time of each leg of the journey. This series of calculations is crucial to optimising the path-finding algorithms in the next stage.

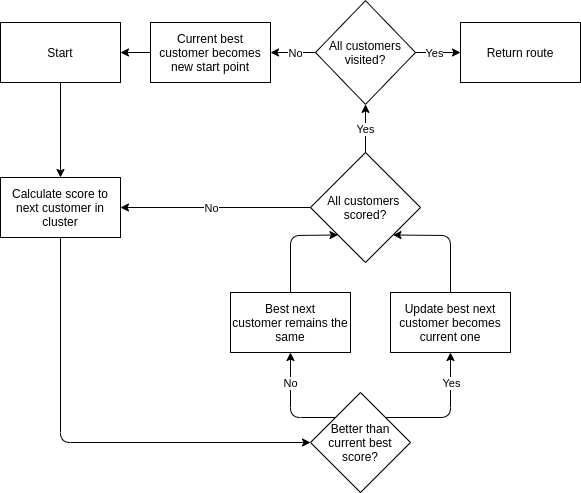


[Calculate flight time per leg]

Greedy Best First

The basic flow of the algorithm is to calculate the time taken to get from the current location to each other location that is unvisited and pick the one with the smallest corresponding time. The process repeats until there are no more unvisited locations, at which point it returns back to the depot.

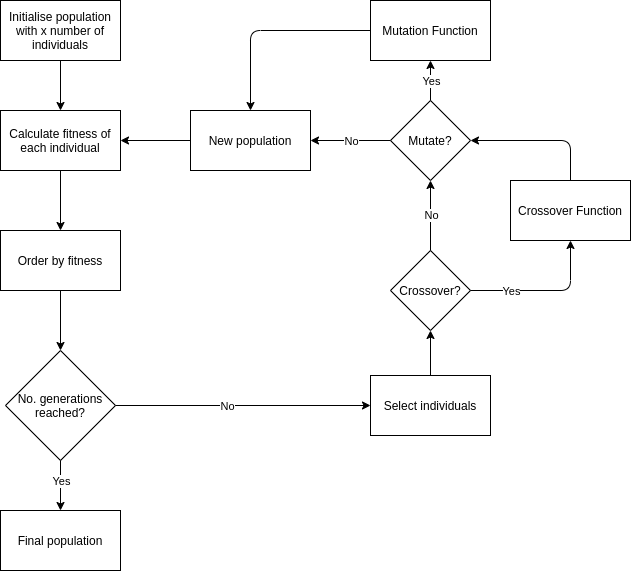
This should be implemented and available as an option through the GUI.



Genetic Algorithm

The second algorithm to be implemented is the genetic algorithm. It is more complex than greedy best first and a third party library may be required to speed along development.

As with greedy best first, this should be available as an option in the GUI.



Database

The database should be simple. One table will be required which will outline customers alongside their coordinates and details of their order, such as item name and order time.

For the purposes of this application, there is no need to store in-depth information about users, nor have a separate table for users as I will not be implementing a login in system of any kind. The scope of the project is just to simulate a real system, thus more data than outlined is not necessary. A sample table layout is shown in figure x.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Latitude** | **Longitude** | **Item\_name** | **Timestamp** |
| 1 | 1.234 | 5.678 | Mobile Phone | 05/05/20 - 15:28 |

[Table representing the data structure in the database]

Output

The final key component of the program is the output. I intend to have two forms of output:

1 – The map shown on the GUI in diagram x. This will be a real map and once the program is run, there should be clearly labelled customers, clusters and routes marked on.

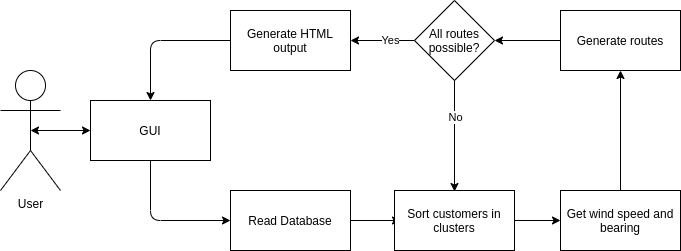
2 – A separate file outlining some of the key parameters used in the program such as inputs, search algorithm and clustering algorithm. Here there should also be the actual schedule that outlines which drone will go to which customer.

Architecture

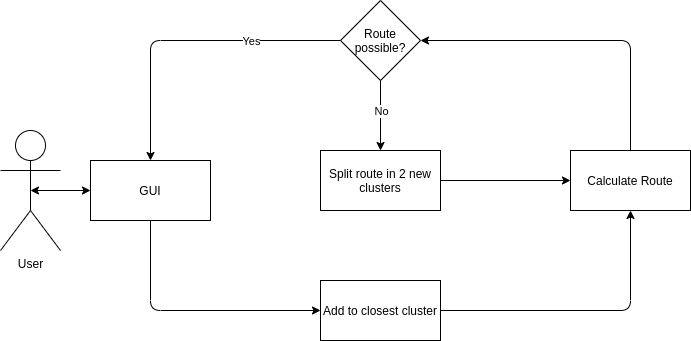
These components are brought together to create a simple program which is able to take user input of customer locations, parameters of number of drones and flight time/speed, as well as which path finding algorithm they would like. Once the program is run, it follows a simple workflow to solve the problem and display the resulting clusters and routes to the user.

The workflow of the main program functionality is shown in figure x.

The workflow of adding a new customer after the program has run is shown in diagram x.



[Diagram of general program flow when run]



[Diagram of program flow when new customer is added after execution]

Development Methodology

The project development has followed the agile methodology. When followed, agile allows the software to be iteratively designed, tested and evaluated. The requirements are created in order to produce the requirements specification. A design is created based on these requirements and an implementation is created based on the design. This implementation is then tested and evaluated against the requirements. If there are bugs or improvements to be made, the process begins again.

Throughout development, unit tests are written to ensure that every component works correctly on it’s own. An isolated test is written before the component is integrated with others. Once complete, these are commented out so as to not use unnecessary resources when developing other parts of the project, however they can always be revisited later on for further testing. Once unit testing is complete, the agile process begins back at the design stage for any modifications or further functions.

Implementation

A number of freely available software libraries were used. Libraries were used to aid development speed and to ensure certain algorithms were correctly implemented.

Care has been taken to follow the design outlined in Section x. Some changes were made throughout the implementation stage due to the agile methodology I used, however the vast majority of features remained as specified there.

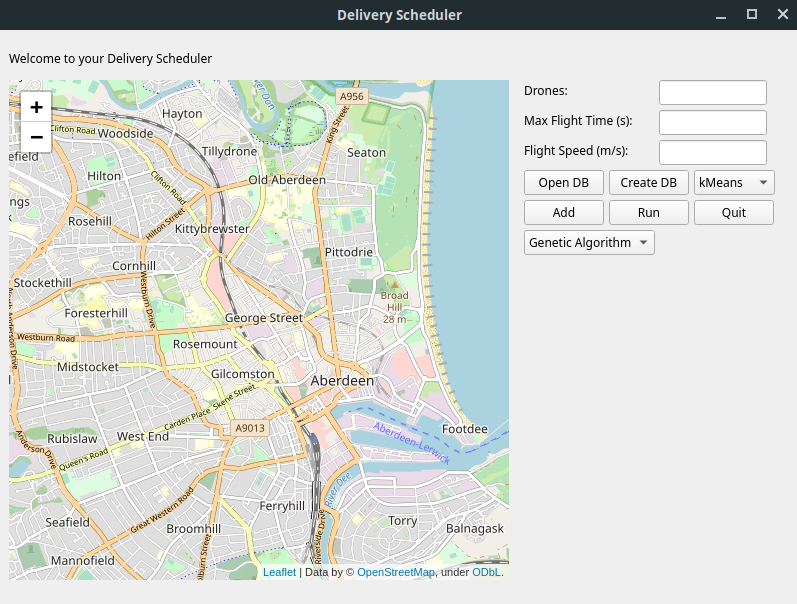
GUI

The GUI was implemented using the PyQt5 library created by Riverbank Computing Limited: <https://pypi.org/project/PyQt5/>

The is library was selected due to it’s ease of use and extensive documentation. It allows complete control over the layout of the GUI and has many ‘widgets’ available with just a few lines of code, such as buttons, drop-down boxes and embedded HTML.

Alternative libraries were explored, however they were universally found not to provide the same level of control over the design that PyQt5 offers.

Figure x shows the implemented GUI. It follows the intended design, with the only changes being to the size of buttons on the right hand side of the interface.



[Implemented GUI]

Clustering

Two clustering algorithms have been implemented, and are available to select through the GUI from a drop-down box.

Affinity Propagation

Early on during the design and implementation of the project I believed that this algorithm would be the most appropriate for clustering. It is different from other algorithms in that it does not take a user input of the desired number of clusters. The algorithm creates however many clusters are necessary depending on some other parameters such as how similar they should be to one another. During testing it was realised that this algorithm wasn’t appropriate for the project, and as the option to select it has since been removed.

The algorithm was implemented with help from the scikit-learn library: <https://scikit-learn.org/stable/about.html>

This library contains a range of algorithms for various machine learning tasks and is freely available.

kMeans

kMeans was the second clustering algorithm implemented. It has been discovered to be more practical for this project, as users are likely to know how many drones they have/wish to have. The number of drones is taken as an input from the GUI. This number of clusters are then created, with customers being grouped by proximity to one another.

The kMeans algorithm is used in the first stage of the programs main functionality. It also is used when adding a new customer after the program has been run.

As with affinity propagation, the algorithm was implemented with help from the scikit-learn library.

Wind Data

In order to account for wind I needed an API that would give both wind speed and direction. For this I have used the darksky API, which offers up to 1000 free API calls per 24 hours:

<https://darksky.net/dev>

There is a wealth of data returned by the API, and as an input it only needs a latitude and longitude pair.

I am only using the current wind speed and bearing at the time of the call. However in future expansion of the project we could easily use more, such as precipitation levels and predicted weather for the coming hours and days. This would allow us to predict ‘no-fly’ times and adjust delivery times for customers accordingly.

Generating Routes

Haversine + Azimuth + E6B

As outlined in the Design section, this is a multi-section process. Diagram x outlines how the data should be gathered and manipulated in order to complete the task.

The Haversine formula was implemented using a python library. As an input all we need to give is the latitude and longitude of the start and end point, alongside the desired unit we want the answer returned in. The library is available here:

<https://pypi.org/project/haversine/>

For calculating the forward azimuth, I was unable to find any suitable libraries, so implemented this myself. The built in maths module for Python gave us the functionality required. Care was taken during testing to ensure that the results given by the solution implemented are accurate.

The E6B was also implemented entirely by myself, with assistance from the maths module. As with the forward azimuth calculation, extensive testing was conducted to ensure the algorithms had been coded correctly.

Genetic Algorithm

For the genetic algorithm, the pyeasyga library was used to aid development:

<https://pyeasyga.readthedocs.io/en/latest/readme.html>

The library provides a class that defines the flow of the algorithm. From this I was able to quickly create a working algorithm by defining how the code should be manipulated at each stage, without spending additional time testing that everything was done in the correct order. See diagram for flow chart of the algorithm.

- Creation of individuals is done by taking the latitude and longitude pairs of customers in each cluster and shuffling them to produce a random route.

- Crossover is done by randomly selecting a position and swapping data from either side of two individuals in order to create two new children.

- Mutation randomly selects two customers in a route and swaps their position in the route. For example if the parent route goes 1 – 2 – 3 – 4 – 5 and customer 2 and 4 are selected, the resulting child route would be 1 – 4 – 3 – 2 – 5.

- Selection is done by taking a random sample of 10% of the population. The individuals in the sample are ranked according to the fitness function and the best individual is selected.

- The fitness function calculates the total length of the route and from there the total time taken for a drone to complete the route. The time is the score of each route, rather than distance. This is the case since wind is taken into consideration. Drones have a set amount of time they can fly, however the distance is in part dictated by external factors such as the wind speed and bearing relative to the drone.

Greedy Best First

I searched for a library to aid with implementing the greedy best first search algorithm, however was not able to find one. As the algorithm is reasonably simple, I opted to implement it myself from scratch. This was a reasonably quick process, and additional testing was done to ensure that the algorithm is working correctly.

The algorithm uses a heuristic in order to decide which is the next closest customer the drone can visit. As with the genetic algorithm, the heuristic is based on flight time of the drone, which takes into account the wind speed and bearing.

Database

SQLite was selected as the database used to store details of customer locations and item names. I opted to use SQLite over a noSQL database type due to the simple schema the data needed to adhere to. One table within a database is all that was required, and is unlikely to change in future versions. SQLite unlike most other SQL databases does not have a seperate server process, it reads and writes directly to ordinary files. It is also largely cross-platform and completely free to use.

Python ships with library for reading and modifying sqlite databases, this was used for the implementation of this project.

Output

Folium

The drawing of the map is done with Folium. Folium allows data to be drawn on a real map. In our case we can have latitude longitude pairs with associated item names. This data can be used to place markers on the map at those exact locations. Once routes have been calculated we are able to draw lines on the map to represent the path of each drone. The map is saved as a regular HTML file.

BeautifulSoup

BeautifulSoup is used to edit HTML files. With BeautifulSoup we are able to read the HTML file generated by Folium and modify the body of the file to include data from elsewhere in the application, such as the number of drones and the number of routes required.

Challenges

Impossible Clusters

There are instances where the path finding algorithms are unable to find a possible route through the customers in a cluster. This can be caused by the cluster simply being too large, or by strong winds slowing the progress of each drone.

In order to solve the problem, I opted to use kMeans to split any impossible clusters in two. Once the two smaller clusters are created, the selected path finding algorithm again attempts to create routes. This is an iterative process which will continue until the route is direct from the depot to the customer. If it is still not possible the process ends and the user is informed via the HTML output file.

There is still room for improvement with this function, as the smaller clusters are all maintained. If for example just one customer is too far away for delivery, and they were a part of a cluster of 15 others originally, the original cluster would be broken down into up to 6 smaller clusters. There is no attempt to combine these smaller clusters again, even if it is just one customer who is causing the route to be too long. This could lead to wasted time as routes are generated where drones serve just one or two customers and have plenty of battery life left.

Weather API

Originally I was using the Openweathermap API for collecting weather data (<https://openweathermap.org/api>). Unfortunately there were occasions where the API would not return either the wind speed or bearing. After troubleshooting I could not find a solution and so decided to switch API to Darksky.

This swap required some modifications to the code due to a the data returned being structured differently. It also will lead to problems down the line, as Darksky have recently announced that they have joined Apple. The result of this is that their API will likely cease to exist after 2021 (<https://blog.darksky.net/>). This does give some time to find a suitable alternative, but changes will have to be made to the code in future.

Limitations of Folium

When I began the implementation I had intentions of allowing users to easily add customers by clicking a location on the map, and then an add button from the pop-up which would bring them to the add dialog box and have the latitude and longitude details already filled in.

Unfortunately, Folium does not include such functionality out of the box. In future iterations I will work to add this feature, however it would require custom Javascript to be written and added to the HTML file. As I am relatively new to Folium and it’s parent library leaflet.js, I did not want to spend time familiarising myself with their code and how I would implement what I wanted. Because of this, I opted not to implement this feature, as it isn’t completely essential.

Implementation Methodology

As described in the design methodology section, the agile methodology has been used for this project. Using the requirements specification, an initial design was created for the application. This bare-bones version was implemented to cover just the “must have” section of the requirement specification.

The process then iterated to include more of the “should have” functionality, as well as refining and optimising the “must have” functions. As this process iterated, the design was modified depending on obstacles discovered during implementation.

Throughout development the application has been tested. Each new function added to the program has been tested extensively to ensure it’s correctness.

The application was managed using a git repository. Each commit to the repository adds or improves functionality of the application and is given a description of the change along with a timestamp. See appendix x for the full git project log.

Testing

Various testing methods have been used to ensure correctness of algorithms, as well as allowing comparison of different combinations of drone parameters and path finding algorithm. This section will detail the testing that has been performed.

Acceptance Testing

Acceptance testing is a crucial part of the project. This testing pertains to the requirements specification. It is a simple check to ensure that crucial functionality has been completed.

Must Have

All of the “must have” functions have been completed. These are crucial to the application and it could not function without them. Because of this, I set out to complete these early on in development.

Should Have

Of the “should have” requirements, four of six have been completed fully, and one of the two incomplete requirements is partially complete. These are important features but non-essential.

The partially complete requirement is the interactive user input. I wanted to have the application running and allow users to be able to add, remove and edit an order while the system was running. It has partially been completed as although the program doesn’t have a state where it is running and updating constantly, when clusters and routes have been found and you add a new customer, everything doesn’t start again. The program is able to append the new customer to the closest cluster and find a route for just the updated cluster.

The completely incomplete requirement is that the software is not delivered as a .exe file. I wanted to be able to distribute the solution as a .exe file so that users wouldn’t have to worry about installing python and maintaining an architecture. As it stand, there are 11 python files which are all required to stay together in the same directory in order to use the application. Additionally, end users would currently have to install the third party libraries themselves.

This requirement was not met as the third-party library I have attempted to use to create the .exe file does not appear to create a valid file on my machine. I suspect this is because I am using Linux as my operating system, which does not natively support the .exe file type. With future development I will look to solve this issue.

Could Have

Both of these requirements pertain to providing a more interesting graphical output for the user. Ideally I wanted to show how the routes changed as they were generated, as I felt this would be interesting for the user to see. Additionally I wanted to show where drones were along their routes, which would rely on the partially-met ‘should have’ requirement being met.

Because of time restraints and the non-essential nature of these requirements, they have not been met.

Functional Testing

Functional testing has been conducted to confirm the results shown in the Acceptance Testing section.

Optimising Genetic Algorithm

Tests were conducted of the effect changing parameters of the genetic algorithm on performance of the algorithm. I wanted to find a set of parameters that give good results for the problem.

For this I ran a set of tests with varied population size and number of generations to find a good balance of run time and results. This testing was performed on the same data set split into various amounts of clusters. The number of generations and population sizes tested are every combination of 50, 150 and 300 of each, and the number of clusters tested on were 3, 5 and 7.

The results can be found in appendix x.

Comparing Genetic Algorithm vs Greedy Best First

In this section I will detail the tests conducted to allow comparison of the genetic algorithm and the greedy best first search algorithm. For these tests the genetic algorithm had a population size of 300 and ran for 250 generations. This was found to give good results.

Route length & time

Genetic Algorithm vs Greedy Best First

- Length of Routes

- Time taken to calculate

- Various sizes of clusters

- Sometimes GA > GBF by route length by < by time

-Genetic algorithm parameters pop size and generations

-Genetic algorithm multiple runs

-Genetic algorithm different parameters

-Genetic algorithm different parameters different cluster size

Unit Testing

- Overview of class tests

FUTURE WORKS

- Add convergence to GA

TABLES

3 Clusters:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Population Size | No. Generations | Route 1 | | Route 2 | | Route 3 | | Run time (s) |
|  |  | Distance(km) | Time  (s) | Distance(km) | Time  (s) | Distance(km) | Time  (s) |  |
| 50 | 50 | 6.88 | 474.72 | 10.03 | 671.52 | 9.94 | 631.5 | 1.16 |
| 150 | 50 | 6.88 | 474.72 | 10.03 | 671.52 | 9.49 | 585.18 | 3.72 |
| 300 | 50 | 6.88 | 474.72 | 10.03 | 671.52 | 9.49 | 585.18 | 7.9 |
| 50 | 150 | 6.88 | 499.48 | 10.4 | 699.65 | 9.49 | 585.18 | 3.52 |
| 150 | 150 | 6.88 | 499.48 | 10.03 | 671.52 | 11.2 | 688.61 | 11.11 |
| 300 | 150 | 6.88 | 499.48 | 11.05 | 717.1 | 10.25 | 700.34 | 23.82 |
| 50 | 300 | 6.88 | 499.48 | 10.03 | 671.52 | 9.69 | 594.61 | 6.99 |
| 150 | 300 | 6.88 | 474.72 | 10.03 | 671.52 | 10.25 | 700.34 | 22.31 |
| 300 | 300 | 6.88 | 474.72 | 11.05 | 717.1 | 11.5 | 711.21 | 47.76 |

5 Clusters:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Population Size | No. Generations | Route 1 | | Route 2 | | Route 3 | | Route 4 | | Route 5 | | Run time |
|  |  | Dist (km) | Time  (s) | Dist (km) | Time  (s) | Dist (km) | Time  (s) | Dist (km) | Time  (s) | Dist (km) | Time  (s) | Time  (s) |
| 50 | 50 | 6.7 | 404.03 | 6.26 | 408.22 | 4.31 | 287.36 | 6.37 | 408.3 | 8.72 | 544.23 | 1.34 |
| 150 | 50 | 5.54 | 348.73 | 6.26 | 408.22 | 4.31 | 287.36 | 6.37 | 408.3 | 9.11 | 559.83 | 4.32 |
| 300 | 50 | 5.54 | 348.73 | 6.26 | 408.22 | 4.31 | 287.36 | 6.37 | 408.3 | 8.68 | 541.22 | 9.49 |
| 50 | 150 | 5.54 | 348.73 | 6.26 | 408.22 | 4.31 | 287.36 | 6.37 | 408.3 | 8.68 | 541.22 | 4 |
| 150 | 150 | 5.54 | 348.73 | 6.26 | 408.22 | 4.31 | 287.36 | 6.37 | 408.3 | 8.95 | 556.4 | 13.06 |
| 300 | 150 | 5.54 | 348.73 | 6.26 | 408.22 | 4.31 | 287.36 | 6.37 | 408.3 | 8.72 | 544.23 | 28.63 |
| 50 | 300 | 5.84 | 383.1 | 6.26 | 408.22 | 4.31 | 287.36 | 6.37 | 408.3 | 8.72 | 544.23 | 8.08 |
| 150 | 300 | 5.54 | 348.73 | 6.26 | 408.22 | 4.31 | 287.36 | 6.37 | 408.3 | 8.72 | 544.23 | 26.07 |
| 300 | 300 | 5.54 | 348.73 | 6.26 | 408.22 | 4.31 | 287.36 | 6.37 | 408.3 | 8.68 | 541.22 | 57.73 |

7 clusters:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Pop Size | No. Gens | Route 1 | | Route 2 | | Route 3 | | Route 4 | | Route 5 | | Route 6 | | Route 7 | | Run time |
|  |  | Dist (km) | Time  (s) | Dist (km) | Time  (s) | Dist (km) | Time  (s) | Dist (km) | Time  (s) | Dist (km) | Time  (s) | Dist (km) | Time  (s) | Dist (km) | Time  (s) |  |
| 50 | 50 | 6.7 | 406.54 | 5.71 | 373.8 | 5.34 | 307.6 | 4.31 | 282.3 | 6.14 | 381.6 | 5.0 | 343.3 | 4.82 | 306.2 | 1.51 |
| 150 | 50 | 6.7 | 406.54 | 5.71 | 373.8 | 5.34 | 307.6 | 4.31 | 282.3 | 6.14 | 381.6 | 5.0 | 343.3 | 4.82 | 306.2 | 5.06 |
| 300 | 50 | 6.27 | 374.02 | 5.71 | 373.8 | 4.52 | 306.5 | 4.31 | 282.3 | 6.14 | 381.6 | 5.0 | 343.3 | 4.82 | 306.2 | 11.31 |
| 50 | 150 | 6.7 | 406.54 | 6.0 | 373.3 | 4.52 | 306.5 | 4.31 | 282.3 | 6.14 | 381.6 | 5.0 | 343.3 | 4.82 | 306.2 | 4.6 |
| 150 | 150 | 6.27 | 374.02 | 6.0 | 373.3 | 4.52 | 306.5 | 4.31 | 282.3 | 6.14 | 381.6 | 5.0 | 343.3 | 4.82 | 306.2 | 15.17 |
| 300 | 150 | 6.7 | 406.54 | 6.0 | 373.3 | 4.52 | 306.5 | 4.31 | 282.3 | 6.14 | 381.6 | 5.0 | 343.3 | 4.82 | 306.2 | 34.21 |
| 50 | 300 | 6.7 | 406.54 | 5.71 | 373.8 | 4.52 | 306.5 | 4.31 | 282.3 | 6.14 | 381.6 | 5.0 | 343.3 | 4.82 | 306.2 | 9.3 |
| 150 | 300 | 6.27 | 374.02 | 5.71 | 373.8 | 5.34 | 307.6 | 4.31 | 282.3 | 6.14 | 381.6 | 5.0 | 343.3 | 4.82 | 306.2 | 30.8 |
| 300 | 300 | 6.27 | 374.02 | 6.0 | 373.3 | 4.52 | 306.5 | 4.31 | 282.3 | 6.14 | 381.6 | 5.0 | 343.3 | 4.82 | 306.2 | 68.14 |