Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Datateknik
2026 | LIU-IDA/LITH-EX-A--26/002--SE

Large-scale drone delivery system in 3D space

Storskaligt leverans-system med användning av drönare i 3D rymd

Marcus Åhl

Supervisor : Amath Sow
Examiner : Fredrik Heintz

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida `http://www.ep.liu.se/`.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: `http://www.ep.liu.se/`.

© Marcus Åhl

Abstract

Drones are a promising alternative to use instead of trucks in last-mile deliveries of small packages in urban environments due to their increased mobility and lower greenhouse emissions. To explore the usage of drones to deliver small packages in a 3D environment, an any-time solver for the problem was developed. The solver uses the metaheuristic adaptive large neighbourhood search to iteratively attempt to improve an existing solution that was found using prioritised planning on a scenario problem. To find delivery routes in environments of up to a size of $350 \times 350 \times 30$ meters that is filled with randomly generated rectangular buildings, two search algorithms, Lazy-Theta* and A* with a simple drone collision avoidance algorithm were used.

While considering a drone's flight speed, current battery, carrying capacity, desired minimum distance to other drones, and delivery deadlines for each delivery, it was found that solutions in parts of fictional cities were possible to achieve and that the best performing heuristic utilised a delivery route's delay and cost, leading to the highest improvement rate (defined as the percentage of iterations that improved the current solution) of 38.46% among all scenarios. The use of multiple heuristics performed better than using single heuristics even though the greatest improvement rate of a single heuristic is 38.10% and was achieved by the Delay heuristic in a larger scenario. It was noted that seeing improvements in a solution became more common when a scenario contained more deliveries and when the number of re-planned deliveries in each iteration was roughly equal to eight. Unlike the improvement rates, the overall improvement of a solution's cost was small and often none regardless of scenario and chosen heuristics. In addition, the solver has performance issues which meant that up to 1800 iterations could be evaluated on small scenarios and environments with a budget of up to four minutes and in larger scenarios, a budget of up to 16 minutes could evaluate as few as 20 iterations. The results show that it is possible to use drones to deliver packages in a 3D environment but that more work is needed to obtain better solutions.

Acknowledgments

Thank you Amath Sow and Fredrik Heintz for your help and guidance! And a special thank you to my grandma and her partner for their support during my work!

Contents

Abstract iii

Acknowledgments iv

Contents v

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Aim . . . 1
- 1.3 Research questions . . . 2
- 1.4 Delimitations . . . 2
- 1.5 Contributions . . . 2
- 1.6 Structure of the report . . . 2

2 Related Work 3
- 2.1 Related work . . . 3
- 2.2 Vehicle Routing Problem . . . 3
- 2.3 Pickup and Delivery Problem . . . 4
- 2.4 Drones . . . 4
- 2.5 Priority Planning . . . 4
- 2.6 Adaptive Large Neighbourhood Search . . . 4
- 2.7 Search . . . 5
- 2.8 Sweep-line algorithm . . . 6
- 2.9 Collision math . . . 6

3 Method 8
- 3.1 Environment . . . 8
- 3.2 Search . . . 10
- 3.3 Main algorithm . . . 13
- 3.4 Finding an initial solution . . . 14
- 3.5 Heuristics . . . 16
- 3.6 ALNS loop . . . 17

4 Results 18

5 Discussion 25
- 5.1 Results . . . 25
- 5.2 Method . . . 26
- 5.3 The work in a wider context . . . 29

6 Conclusion **30**
6.1 Conclusion 30
6.2 Future work 31

Bibliography **32**

List of Figures

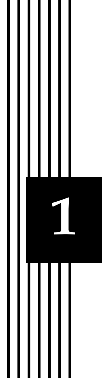
3.1 Top-down view of the four different environments. 10
3.2 A showcase of a segment between two points in 2D. The black cylinder is the radius of the traversing drone, the red cylinder is the max drone radius, and the blue rectangle is the collision point's x and y intervals. 12
3.3 A UML diagram of the program. 13
3.4 A small graph showing transition costs. 15

4.1 Environment 2, scenario 2. (a) Shows the solution cost over time for each of the four heuristics. (b) Shows the solution cost over time for each heuristic combination with at least size two. 21
4.2 Environment 3, scenario 1. (a) Shows the solution cost over time for each of the four heuristics. (b) Shows the solution cost over time for each heuristic combination with at least size two. 22
4.3 Environment 4, scenario 1. (a) Shows the solution cost over time for each of the four heuristics. (b) Shows the solution cost over time for each heuristic combination with at least size two. 23

List of Tables

3.1 The different scenarios and their respective number of agents and packages to be delivered. 9
3.2 Budget is given as the number of seconds that each scenario was run. 10

4.1 F=Final solution cost, IMP=Cost improvement, IM-R=Improvement rate. Red-marked cell shows the greatest improvement rate for each scenario. 18
4.2 F=Final solution cost, IMP=Cost improvement, IM-R=Improvement rate. Red-marked cell shows the greatest improvement rate for each scenario. 19
4.3 Improvement rates with marked cells showing the highest value for each environment-scenario combination. 19
4.4 Improvement rates with marked cells showing the highest value for each environment-scenario combination. 19
4.5 F=Final solution cost, IMP=Cost improvement, IM-R=Improvement rate. Red-marked cell shows the greatest improvement rate for each scenario. 20
4.6 Solution costs for each scenario together with what parameters that were used to achieve that result. 24



1 Introduction

1.1 Motivation

Delivery of smaller packages in both rural and urban areas are predominantly delivered by trucks that carry multiple packages and often travel a fixed and predetermined delivery route. Trucks are used because they have large carrying capacities and can travel long routes with little concern for refuelling which gives them flexibility in what routes they can travel and can reduce the number of vehicles required for a delivery service. An alternative method of delivering small packages is to use drones that have some advantages over traditional truck delivery due to a drone's high mobility afforded by its ability to freely travel in 3D space. Drones are thus not restricted by roads, traffic, or other obstacles on the ground and therefore, rough and difficult terrain are no longer the obstacles that they used to be and will not prevent package delivery by drone, allowing easier delivery to more people and in crisis (for example relief aid). In addition to the capability of travelling over rough terrain, a drone emits less greenhouse emissions for short distance (last-mile) deliveries when compared to traditional trucks [8] and can offer cheaper operating costs.

Finding optimal routes for delivery vehicles is an NP-problem that has been extensively researched on graphs and on the many common delivery problem variations that exist. However, these solutions do not scale well to larger use-cases which makes them less practical for use in real-world scenarios. Then the challenge is to find as near-optimal solutions as possible while being able to scale to larger scenarios. A method to achieve this are metaheuristics that are frequently used to solve delivery problems where the goal is to find near-optimal solutions as quickly as possible. The effectiveness of metaheuristics makes them a suitable method for real-world usage. While much work exists for the drone delivery problem in 2D, little work has been done on drone delivery in 3D space which seem necessary to bringing drone delivery closer to an everyday reality.

1.2 Aim

The aim of the project is to study the usage of drones to deliver small packages in a 3D environment while considering various constraints on both drones and deliveries. To achieve the aim, an any-time solver for the drone delivery problem in 3D was developed. The solver will be used on large-scale scenarios to gain information on how to best solve the problem

and see what affects its scalability, and test different heuristics to see how they perform in a 3D environment.

1.3 Research questions

1. Using adaptive large neighbour search, what heuristic or combination of heuristics provides the highest quality solution for the drone delivery problem in 3D?
2. What impact does the map size, number of deliveries, and number of available delivery drones have on the scalability of the drone delivery problem in 3D?
3. What time is required to obtain a solution that is close to the theoretical/known optimal solution?

1.4 Delimitations

The environments that will be used in simulations are fictional small parts of randomly generated cities. A city is comprised of rectangular buildings of varying height and it is possible for a drone to travel freely in the environment as buildings are the only obstacles. Any other obstacles such as pedestrians, vehicles, decorations, or animals are all omitted from the environments used in the work.

1.5 Contributions

The work has shown that solutions for the drone delivery problem in 3D with deadlines are possible. An understanding of the solving process has been gained where the important parts are performant algorithms for search and collision avoidance that suit the 3D in time environment that the work was carried out in. Lazy-Theta* is a well performing search algorithm but without a system to handle line-segment transitions and a good line-of-sight function, is not well suited to be used to solve the problem. Instead, A* can be used as a good option as it is also easier to discretise the resulting paths to later be processed (for example in collision resolution). It was also found that heuristics that utilise a delivery route's specific data such as cost and/or delay generally resulted in better solutions.

1.6 Structure of the report

The structure of the report is as follows, Chapter one introduces the report together with the aim and research questions. Chapter two will present related work and information on the different areas that are used in the report. Chapter three details the methodology and its motivation. Chapter four presents the results in tables and figures together with some new notation. Chapter five discusses the work and results in more detail and provides some reasoning for why the specific results were achieved. Finally, Chapter six answers the research questions and provides an alternative algorithm for line-of-sight checking.

2 Related Work

2.1 Related work

In their paper, S Deshpande and P Mani [3], use mixed integer programming to optimally schedule drones on a 2D grid with obstacles, minimising both time and distance. In addition, they optimally place recharging stations on the grid to ensure full coverage, considering a drones maximum travel distance on a single battery charge.

J Gómez-Lagos et.al [7] provide three different mixed integer linear programming models for the Pickup to Delivery Drone Routing Problem (PDDRP). In PDDRP a package is stored in a facility that a drone must first travel to to pickup a package before the drone can deliver it to a customer. The models they use to solve this problem are extensions of the multiple travelling salesman problem and the parallel machine scheduling problem, and a model directly formulated to the PDDRP problem. They also introduce new heuristics to a metaheuristic called GRASP in order to solve larger PDDRP problems.

Supervised machine learning is used by A Paul et.al [14] to solve the drone delivery service planning problem with online demand by first modelling the problem as a Markov decision process and synthesising training data to use in their machine learning model.

2.2 Vehicle Routing Problem

Given a fleet of vehicles, a depot where vehicles originate from, and a set of destination points, the vehicle routing problem (VRP) is defined as the problem of finding a route for a subset of vehicles that traverse all destination points without any two vehicles visiting the same point and both begin and end their route at the depot. The goal is often to minimise the total travel cost of all routes and to reduce the number of required vehicles. A variation of the VRP is the vehicle routing problem with time windows (VRPTW) that also specifies that each point can only be visited during some time window [6] but, should a vehicle arrive early it is allowed to wait. Arriving late is either heavily penalised or simply not accepted (i.e. any solution with a late arrival is considered invalid).

2.3 Pickup and Delivery Problem

The pickup and delivery problem (PDP) is defined by a set of packages that should be delivered by a fleet of vehicles to a set of customers [17]. Each customer should only be visited once and the total route cost of all vehicles should be minimised. In the pickup and delivery problem, a vehicle is not constrained to begin nor end its route at a depot.

2.4 Drones

A drone is an unmanned aerial vehicle (UAV) that has limited carrying capacity (both in terms of weight and size), and limited battery. However, a drone is small, can fly moderately fast, and can be cheap to operate. They are therefore well suited to deliver small packages over short distances as well as being more environmentally friendly compared to traditional truck deliveries [8]. The properties of a drone make them a good option to improve delivery of small packages in urban areas as well as in difficult or obstructed terrain. They have easy access to high-rise buildings (e.g. deliver to a window/balcony) and in disaster areas where ground vehicles may not be able to travel and speed is important (e.g. deliver first aid to an injured person). A large drawback of drones are their limited operational range from a depot due to having little power storage. It has however been shown that the battery consumption for a drone in-flight is an approximately linear function to the payload's weight [4, 20]. This enables a drone's operational range to be more realistically modelled as a function of its operation over time. The following properties of a drone are considered in this work:

- Carry capacity – the maximum package weight that the drone can carry while still maintaining flight.
- Flight speed – the constant speed that a drone will travel with during flight.
- Battery capacity – how much battery storage that is maximally available to a drone.
- Battery consumption – the amount of battery charge that is used by a drone during a delivery.
- Availability – keep track of when a drone is available to deliver a package.
- Position – a drone's location as a 3D point in an environment.
- Radius – a spherical radius that is the least distance that is considered unsafe for another drone to be within.

2.5 Priority Planning

For a given problem and a set of agents where each agent has a unique assigned priority, priority planning solves part of the problem using each agent in turn, starting with the highest priority agent [5]. After a partial solution is found for an agent the partial solution is added as a constraint to the problem which ensures that the solution remains valid when solving for the next agent. Priority planning is fast but not complete and there are many different variations of priority planning that attempt at decreasing the probability of an invalid solution [11].

2.6 Adaptive Large Neighbourhood Search

Large Neighbourhood Search [19] (LNS) is a metaheuristic that uses local search to approach an optimal solution. It does this by iteratively modifying an existing solution to the problem with the help of two types of operators, namely the destroy operator and the repair operator.

The destroy operator will "destroy" or remove parts of the solution and then the repair operator will attempt to repair the solution by reintroducing the removed parts to the solution in such a way that the new solution becomes improved over the previous solution. To avoid local optimums, simulated annealing can be used to randomly accept poor solutions in order to explore more of the solution space.

Adaptive large neighbourhood search [15] (ALNS) extends LNS by allowing multiple destroy and repair operators to be chosen from a larger set of available operators. The operator of each type that is used in each iteration can be chosen randomly or by some other method. The roulette wheel policy is a simple and commonly used policy that is the same a rolling a weighted die for the destroy operator and another for the repair operator. After each iteration, the probabilities for all operators are updated given the result of the iteration and some update policy.

2.7 Search

Maps and Graphs

A graph is a pair (V, E) of V vertices and E edges that connect vertices. Two vertices that share an edge are considered neighbours. Search, i.e. finding a path as a sequence of edges to traverse in order to travel between two vertices that can represent two different locations in an environment can be done in either continuous or discrete space. Most search algorithms work on graphs that represent some discrete space whereas search in continuous space often first undergoes discretisation such that a discrete search algorithm can be used on it. When searching in discrete space, the A* search algorithm is commonly used as it returns an optimal cost path when used with an admissible heuristic.

A*

A* [9] is a search algorithm that uses an admissible heuristic to guide the search in quickly finding an optimal path. A heuristic is admissible if does not overestimate the actual cost (for example distance between two vertices) from any vertex to the goal. A* utilises two sets during search. The first is the open set that contain all vertices that should be searched next, sorted by decreasing cost using a vertex's g-value and its heuristic value h. The second set is the visited set that is maintained in order to not search duplicate vertices. An individual vertex contains the current best cost from the start vertex to itself named g and its parent. A parent is the vertex that the search moved from to reach the current vertex and is used to extract the final path from the graph after the goal has been reached. The algorithm initially puts the start vertex in the open set and then enters the main loop that:

- Retrieves the vertex s with least value from the open set.
- If s is the goal vertex, then the path has been found.
- Iterate over all neighbours s' of s. If s' has not been visited before and $c(s, s') + g(s) < g(s')$ then add s' to the open set with value $c(s, s') + h(s')$, update $g(s') \leftarrow c(s, s') + g(s)$, and set the parent of s' to be s.
- Add s to the visited set. Repeat the above steps.

Lazy Theta*

When searching vertex neighbours, Nash, Alex et al. [13] showed that a found path can be up to 13 percent longer than actual the shortest path. The Lazy-Theta* [13] search algorithm can be seen as an extension of A* that use line-of-sight checks to allow for any-angle search that enables it to find shorter paths on a discrete 3D grid.

During search, Lazy-Theta* checks two alternative paths when expanding a node s'. The first path is the same path ($s \to s'$) that A* will search, and the second path is checking the path ($s \to parent(s') \to s'$) using line-of-sight checks where $parent(s')$ gives all nodes that has s' as their parent. A difference between Lazy-Theta* and A* is that Lazy-Theta* does not have a restriction on the relation between a node and its parent and is where Lazy-Theta* gets its any-angle property from. Lazy-Theta* achieves higher run-time performance by initially not considering the first path and instead makes the assumption that the second path is valid and performs the line-of-sight check when expanding the next child vertex. If there was no obstacle blocking line-of-sight and therefore the transition, then the assumption was correct and some worked was saved. If there was no line-of-sight then the first path is the correct path and is used instead. This is different from the base algorithm Theta* [12] that performs line-of-sight checks for each visible neighbour of each expanded vertex. The lazy behaviour leads to fewer line-of-sight checks but slightly more expanded vertices.

2.8 Sweep-line algorithm

Given the set S that contain line segments, the sweep-line algorithm [18] is an efficient way of finding all intersecting segments in S. The algorithm works by "sweeping" an infinite vertical line through the entire set and reporting collisions. This can be done practically by stepping through each segment's endpoint from least x to greatest x of any segment $s \in S$ while keeping track of the segments that the vertical line is intersecting in each step to get the set L which is used to find intersections. The stepping process is often managed as an event system where a segment s is added to L when the line intersects the left endpoint of s and is removed from L when the right endpoint of s is intersected by the line.

2.9 Collision math

In Euclidean space a line segment is given by its two endpoints (which can be seen as any two points on an infinite line). Any collision check is done on a combination of line segments and points and results in a true or false statement.
In $\Re^2$, let $L_1 = (\bar{a},\ \bar{b})$, $L_2 = (\bar{c},\ \bar{d})$ be two line segments and $x_1 = (\bar{a} - \bar{c}) \times (\bar{d} - \bar{c})$, $x_2 = (\bar{b} - \bar{c}) \times (\bar{d} - \bar{c})$, $x_3 = (\bar{c} - \bar{a}) \times (\bar{b} - \bar{a})$, $x_4 = (\bar{d} - \bar{a}) \times (\bar{b} - \bar{a})$ be the cross product between the given vectors (the cross product $\times$ in $\Re^2$ is defined as the determinant), then the two line segments L_1 and L_2 intersect if

$$xor(x_1 \leqslant 0, x_2 \leqslant 0) \wedge xor(x_3 \leqslant 0, x_4 \leqslant 0) \equiv \top \tag{2.1}$$

Let $f : \Re^2, \Re^2, \Re^2 \to \Re^2$ be a function that takes a line segment $L = (\bar{a},\ \bar{b})$, a point $\bar{c}$, and returns the point $\bar{p}$ on the line segment that has the least distance to $\bar{c}$. $\bar{p}$ is given by

$$\bar{p} = f(\bar{a}, \bar{b}, \bar{c}) = \begin{cases} \bar{a} + \bar{u}t & \text{if } 0 < t < 1, \\ \bar{a} & \text{if } t \leqslant 0, \\ \bar{b} & \text{if } t \geqslant 1. \end{cases} \tag{2.2}$$

with

$$\bar{u} = \bar{b} - \bar{a}$$

and

$$t = \frac{(\bar{c} - \bar{a}) \cdot \bar{u}}{\bar{u} \cdot \bar{u}}$$

The least distance between a point and a line segment is therefore $|f(\bar{a}, \bar{b}, \bar{c})|$. f also holds in $\Re^3$.

The least distance between two line segments can be found by first checking if they intersect in which case the distance is zero. If they do not intersect the least distance from all

segment endpoints to the other segment are checked and the minimum distance among them is also the least distance between the two segments.

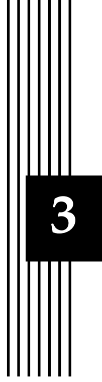
Given a drone i with radius r_i and flight speed s_i that traverses a line segment $(\bar{a}, \bar{b})$ in $\Re^3$ during time $t_i = [t_{i1},\ t_{i2}]$ and a drone j with radius r_j and flight speed s_j that traverses a line segment $(\bar{c}, \bar{d})$ in $\Re^3$ during time $t_j = [t_{j1},\ t_{j2}]$, will their collision spheres intersect and if so, for how long? To solve this:

1. Check for overlap between t_i and t_j. If there is no overlap, then there is also no collision.
2. Let $\bar{w} = \bar{c}' - \bar{a}'$, $\bar{u} = \frac{\bar{b}-\bar{a}}{|\bar{b}-\bar{a}|} * s_i$, $\bar{v} = \frac{\bar{d}-\bar{c}}{|\bar{d}-\bar{c}|} * s_j$, and $t_0 = min(t_{i2}, t_{j2}) - max(t_{i1}, t_{j1})$.
3. If $t_{i1} < t_{j1}$ then drone i begins its flight before j and the new start point $\bar{a}'$ needs to be calculated as $\bar{a}' = \bar{a} + \bar{u} * (t_{j1} - t_{i1})$. If instead $t_{j1} < t_{i1}$ then drone j begins its flight before i and $\bar{c}' = \bar{c} + \bar{v} * (t_{i1} - t_{j1})$.
4. If $\bar{v} \times \bar{u} = \bar{0}$, then there is a collision in the interval $[max(t_{i1}, t_{j1}), min(t_{i2}, t_{j2})]$ if $|\bar{w}| \leqslant max(r_i, r_j)$, else there is no collision.
5. Solve for t in $|\bar{w} + (\bar{v} - \bar{u}) * t|^2 = max(r_i, r_j)^2$ in the interval $[0, t_0]$.
6. If there are complex roots or any of the roots are outside of the interval $[0, t_0]$ then there is no collision. Else the collision interval is $[max(t_{i1}, t_{j1}) + max(0, min(root)),\ max(t_{i1}, t_{j1}) + min(t_0, max(root))]$ [10].

To solve the above problem when one of the line segments is a point (now assuming that drone j is static at the point $\bar{c}$), let $\bar{c}' = \bar{c}$ and $\bar{v} = \bar{0}$ and then follow the above steps. If both drones are static at two points $\bar{a}$ and $\bar{c}$, then the collision interval is simply

$$[max(t_{i1}, t_{j1}), min(t_{i2}, t_{j2})]$$

if $|\bar{a} - \bar{c}| \leqslant max(r_i, r_j)$ and there is overlap in time.



3 Method

3.1 Environment

The environment is a discretised 3D Cartesian grid filled with rectangular buildings with different heights which makes it easy to search in. It is also possible to change the grid resolution to increase or decrease the resolution of the environment. The implementation uses a resolution of $1m^3$ per grid cell with x, y being the ground coordinates and z being used for height above ground. The grid is centred around the point $(0, 0, 0)$.
A building is a collection of points in the grid that are stored as compressed points and is treated as an obstacle (inaccessible area). Each cell in the grid can contain a single drone and has 26 neighbours as diagonal movement is allowed. Even though a drone can take up more space in the grid due to its radius, it will still only occupy a single cell as the drone itself will fit in a single cell. The extra space is used only for collision avoidance with others drones as a safety measure. In addition, all of the empty space in the grid is implicit in order to save memory and is simply considered free space that drones can travel in.

A compressed point is a 64 bit unsigned integer that stores a limited precision 3D point with signed integers where the bits for each integer are reserved as follows: $|\ 27\ x\ |\ 27\ y\ |\ 10\ z\ |$. This allows a single compressed point to use one third less memory (compared to a full 3D point with 32 bit signed integers) and be indexed using a single value which also makes the implementation easier and faster. The valid range of values for each point coordinate is

$$x \in [-2^{26} - 1,\ 2^{26} - 1],$$

$$y \in [-2^{26} - 1,\ 2^{26} - 1],$$

$$z \in [-2^{9} - 1,\ 2^{9} - 1]$$

which when using $1m^3$ as the size for points in the grid, is able to represent $(2^{27})^2 m^2 \simeq 35.317 \times$ the surface area of the Earth.

Generating an environment

In order to generate different environments that can be used in the work, a script that takes a size in $m \times m \times m$ and a building density parameter was made. The script uses Monte Carlo

simulation to generate a set of rectangular buildings with random volume and location while trying to avoid making intersecting buildings.

Buildings are added to the environment one by one until the desired density has been reached. However before a building is added to the environment it is reduced in size if it overlaps with other previously added buildings to ensure a higher degree of realism as well as removing unnecessary obstacle points. The generated buildings then make up a fake city where deliveries can take place and since the environment does not impose any limits on free space, a bounding box is used to limit the environment and possible delivery locations. This also reduces the search space (and prevents the case where drones can travel freely underground).

Multiple problem scenarios can be constructed for an environment where the number of drones, packages to be delivered, percentage of packages that will have a deadline, and the earliest possible deadline are all tunable input parameters. Each drone in a generated scenario will be assigned the following random values chosen uniformly:

- A carry capacity from the range $[300, 750]$ grams with 50 gram steps.
- A flight speed from the range $[1.0, 2.5]$ meters per second with 0.1 m/s steps.
- The time when the drone is first available from the range $[0, 600]$ seconds.
- A radius from the range $[0.5, 1.5]$ meters with 0.1 meter steps.

meant to roughly replicate real world drones. Packages will be randomised as:

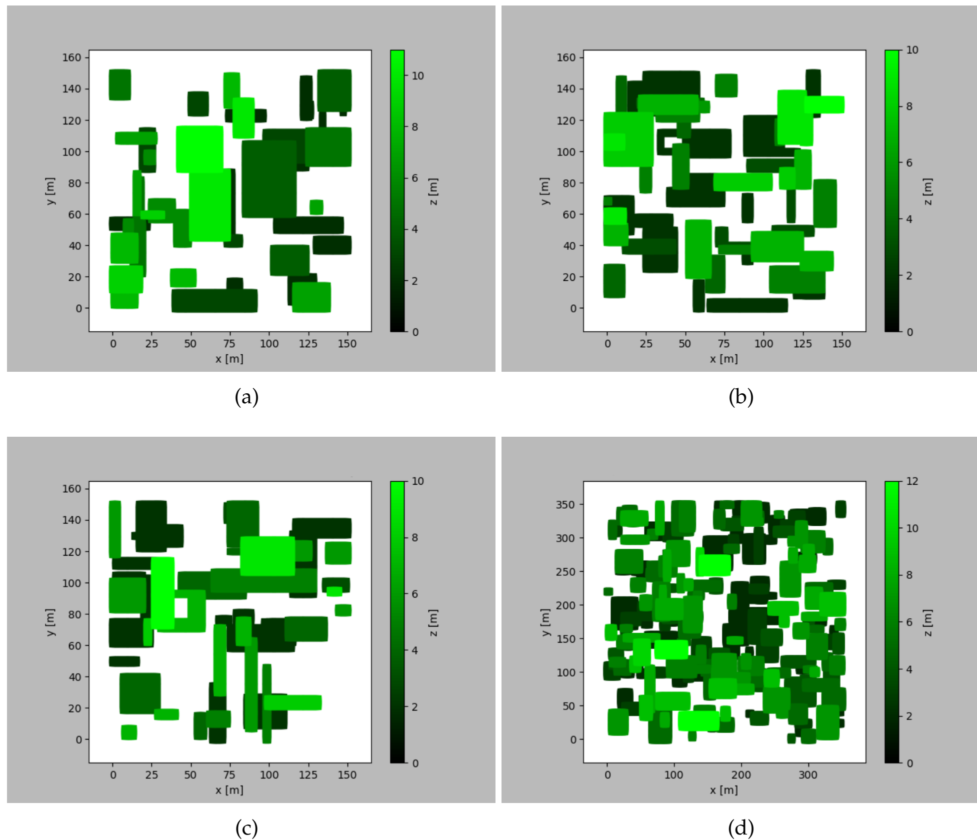
- Random delivery destination.
- A weight from the range $[300, cc]$ grams with 50 gram steps where cc is the max carry capacity of a drone which ensures that all packages can be carried by at least one drone.
- A deadline from the range $[0, 3600] + earliest_deadline$ seconds if the package should have a deadline or the value -1 to indicate that it has no deadline.

The environments were generated to be larger but still small enough that many experiments could be run on them within a limited time-frame. Figure 3.1 shows the different environments where (a) Is environment 1 with size $150 \times 150 \times 25$ meters, (b) Is environment 2 with size $150 \times 150 \times 25$ meters, (c) Is environment 3 with size $150 \times 150 \times 25$ meters, and (d) Is environment 4 with size $350 \times 350 \times 30$ meters. The environments' sizes are also the size of their bounding boxes and to not restrict movement, the height of each environment was set much higher than the buildings (since the benefit of drones are that they can fly, it seemed wrong to limit them unnecessarily).

Environment	Scenario	Number of agents	Number of packages
1	1	15	15
	2	55	55
	3	120	119
2	1	15	60
	2	55	209
	3	120	462
3	1	55	105
4	1	55	108

Table 3.1: The different scenarios and their respective number of agents and packages to be delivered.

While it is possible with a little bit of rework of the implementation to have multiple depots that both drones and packages can originate from and drones can return to, each



(a) (b)

(c) (d)

Figure 3.1: Top-down view of the four different environments.

Environment	Scenario	Single heuristic budget	Heuristic combination budget
1	1	150s	-
	2	200s	-
	3	400s	-
2	1	250s	-
	2	500s	750s
	3	1000s	-
3	1	350s	600s
4	1	500s	1000s

Table 3.2: Budget is given as the number of seconds that each scenario was run.

scenario only has one depot where all drones and packages begin. All drones must also return to the depot after delivery. The decision of having only one depot instead of multiple was made to simplify the implementation and to reduce the scope of the work.

3.2 Search

Search was implemented by two different search algorithms, A* and Lazy-Theta* (see Section 2.7). Lazy-Theta* was chosen as it provides initial paths with less cost than A* while still being fast and A* is later used to re-plan paths with the addition of collision avoidance (Lazy-Theta* did not include collision avoidance due to limitations in the collision avoidance system). Both algorithms take two points *begin* and *goal* and finds a path between those two points if one

exists. The difference between them is that Lazy-Theta* is an any-angle algorithm which uses a line-of-sight function while A* will only travel between neighbours and that a found path by Lazy-Theta* is often less jagged when travelling in a diagonal and contain fewer but longer path segments.

The search uses a rectangular bounding box to determine the valid search space (default is the full range of a compressed point), an obstacle set (a set of points that cannot be traversed), and a traversal graph. The traversal graph is initially empty as the traversable space is not stored anywhere and is continuously filled with all nodes (points in the environment) that are expanded during the search. A node in the traversal graph contains:

- The point location of the node.
- Its node parent.
- Its g-value.
- A visited flag that is set to true when this node has been visited.
- A *wait time* value that describes how long a drone was waiting at this node before leaving it that is used to allow drones to stay static at a point and wait to avoid collisions.

A neighbour to a node is any adjacent point that is not an obstacle and whose euclidean distance from the node is less than or equal to the square root of three.

Lazy-Theta*

Lazy-Theta* find paths minimising distance without a given drone and without any collision avoidance. Instead a drone is assigned to traverse a path after it has been found by Lazy-Theta*.

The line-of-sight check between two points is a ray-cast with 0.5 meters resolution that checks if any point in-between them is an obstacle. The resolution was chosen so that no obstacle would be missed (except for some corner cases) and execution speed would not be too slow. To minimise floating point errors over long distances, the ray-cast was recalibrated every 256 steps.

A*

The implementation of A* performs search using time as the cost and requires a drone before beginning the search because the drone's flight speed is required to correctly convert distance to time and to be able to perform collision checking using the drone's radius. The resulting path will have its cost expressed in distance and not time to be consistent with the paths found by Lazy-Theta* and because a solution's cost is expressed in distance.

The search is performed as follows: check for collisions at the starting point beginning at the initial start time and then wait until there is no collision given that the drone would fly its radius away from the starting point. The time to wait is calculated by taking the current time and checking for collisions. If there is a collision, wait until the end of that collision by progressing time and then repeating the collision check again. This process is repeated until no collision is detected.

The path that A* finds is constructed as a series of points that should be visited in order from start to goal. Each point in the path is called a *collision point* that contains the point, and a 4D box that contains the current point and the next point in the path stretched in x, y, z by the max radius of all drones and is used when the environment is queried at a point for possible collisions. Given a point $p = (p_x, p_y, p_z)$, the next point in the path p', and the max drone radius r_m, the 4D box is described by four intervals as:

$$\bar{u} = \frac{p' - p}{|p' - p|} * r_m,$$

$$\begin{aligned}
&\text{x: } [p_x - \bar{u}_x,\ p'_x + \bar{u}_x],\\
&\text{y: } [p_y - \bar{u}_y,\ p'_y + \bar{u}_y],\\
&\text{z: } [p_z - \bar{u}_z,\ p'_z + \bar{u}_z],\\
&\text{t: } [t_p,\ t_{p'}]
\end{aligned}$$

where t_p is the time of arrival at p and $t_{p'}$ is the time of arrival at p'. The coordinate intervals $[a,\ b]$ for x, y, z are sorted such that $a \leqslant b$.
The original idea to finding collisions was to use rectangle stabbing [1] but due to difficulties in implementing it, a simpler bucket solution was used. Finding collisions during search was done by creating buckets each with a width of five meters such that the entire environment's y-length was covered. All path segment's 4D box's y-intervals were then split into the buckets which were sorted by a y-interval's left value. An interval's indices in the he bucket was the range $[\lfloor \frac{y\ interval's\ left\ value}{bucket\ width} \rfloor, \lfloor \frac{y\ interval's\ right\ value}{bucket\ width} \rfloor]$. Each collision query then finds the relevant bucket, ignores all entries whose left y-value was less than the query point's right y-value, and then manually checks the remaining bucket for collisions.

During search, a drone travelling the segment from a node a to a node b starting at time t_0 will be checked for collisions by first finding the earliest arrival time t_1 at b that would avoid all collisions. For the drone to maintain a constant travel speed (which it must due to technical limitation) it will wait at a for $t_1 - t_0 - t_s$ seconds, where t_s is the time it takes the drone to traverse the segment while ignoring collisions. While the drone it waiting at a, additional collision checks are performed where any collision means that the segment has an unavoidable collision and should not be traversed beginning at t_0. If there is no collision during that wait, the search will try to traverse the segment starting instead at time t_1 given that the wait time is non-zero. This process continues until the segment can be traversed collision free or there is an unavoidable collision. Any wait time is stored in the search node a such that the correct path can be retrieved from the search result.

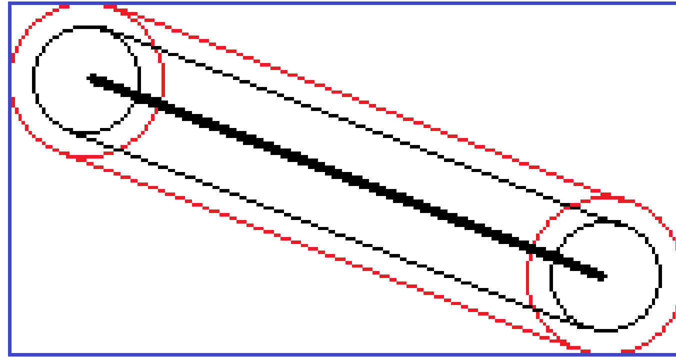


Figure 3.2: A showcase of a segment between two points in 2D. The black cylinder is the radius of the traversing drone, the red cylinder is the max drone radius, and the blue rectangle is the collision point's x and y intervals.

As a drone traverses a path with constant speed and can only travel between two points, waiting at a point was implemented as travelling from one point to the same point with different arrival times. The 4D box is constructed the same way but p' is set to be the furthest neighbour of p which gives $\bar{u} = \frac{r_m}{\sqrt{3}}(1, 1, 1)$. Collision checking differentiates between the different cases: segment-segment, segment-point, and point-point to ensure correctness as described in Section 2.9.

3.3 Main algorithm

There are four major parts that make up the program. They are: 1. Initialisation of the environment and scenario data such as drones and packages, 2. Finding an initial solution, 3. The ALNS loop to iteratively attempt to improve the current solution, and 4. Finding and resolving path collisions.

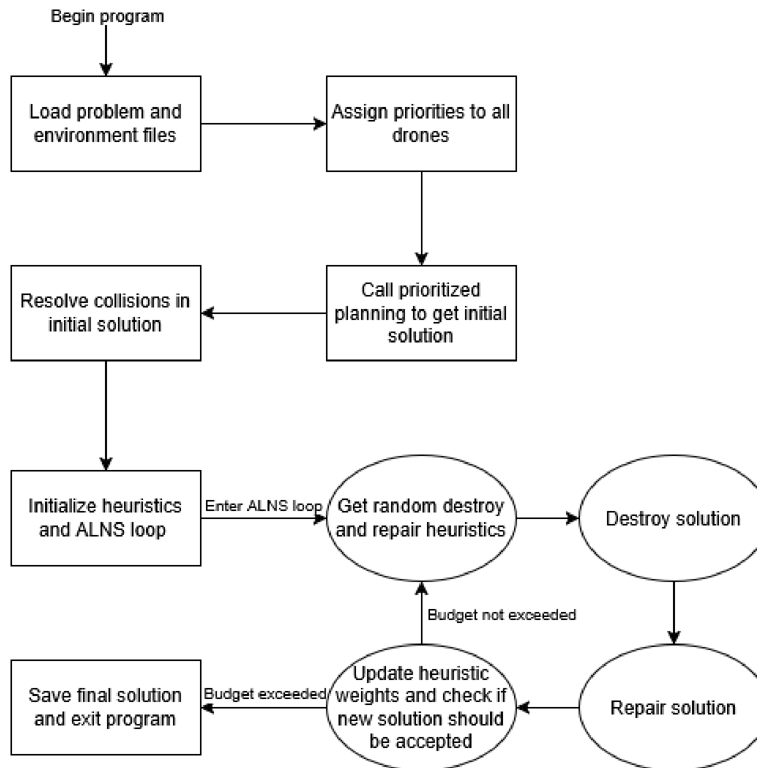


Figure 3.3: A UML diagram of the program.

Initialisation

Initialisation begins by reading the environment file and initialising the environment by filling it with obstacles as collections of impassable points and setting the bounding box's values. Then the scenario file containing information on the location of the depot, all available drones, and all packages that should be delivered is read and stored in the program.
Having separate files for the environment and a scenario was important in order to easily run multiple different scenarios in the same environment.

Deliveries and drones

A delivery is expressed as a delivery path, the package to be delivered, and a flag that determines if the delivery path should be included in collision checking during search. The package contains its origin and destination as well as its weight expressed in grams and its deadline expressed as seconds after start (which is at local time zero).

To be able to better represent a drone, it was implemented given the properties in Section 2.4 together with a schedule to keep track of a drone's availability. Its initial and final position was always the depot since there was only one depot in each of the scenarios.
To get a good estimate on a drone's battery consumption, the battery consumption rate from the paper "Drone delivery scheduling optimization considering payload-induced battery

consumption rates" [20] by M. Torabbeigi, G J. Lim, and S J. Kim was used. Their formula

$$(3.879 + 2.297x) * t$$

gives the amount of battery percentage-units that a drone carrying a package weighing x lbs consumes per minute of flight time t. The formula was converted from lbs to grams and from flight duration expressed in minutes to instead in seconds to get

$$(0.06465 + 0.0000844x) * t$$

which gives the battery percentage-units that a drone carrying a package weighing x grams consumes per second of flight time t. All drones had the same battery specification and began with full battery (100 percentage-units).

3.4 Finding an initial solution

The first step to assign random unique priorities to all drones and then sort all of the packages by earliest deadline first. Then a time variable is initialised to zero and a list of all currently available drones is made available. The list is used so that no drone is assigned to deliver a package while it is already delivering another one at any given time. The time variable t is maintained by progressing it as drones are making deliveries as well as a small amount when a drone begins a delivery to allow the current drone to leave the depot and therefore reduce the number of collisions that later need to be resolved.

The available drone list is maintained by storing all drones who are available to begin a delivery at a time before or at the current time t and sorting it by decreasing drone priority. All other unavailable drones are kept in a sorted queue with earliest return time t_r first such that when $t_{i,r} \leqslant t$, drone i is moved to the availability list, ready to begin another delivery. When a drone is retrieved from the availability list it is removed from the list and added to the queue. If the availability list is empty then t will be progressed to the return time of the first drone in the queue who is immediately after moved to the availability list.

To find an initial solution, the packages are iterated over to be delivered in turn by:

1. Finding a path from package origin to delivery destination.
2. Adding a return path to the delivery, symmetrical to the delivery path as it is the best path since no collisions are checked at this point.
3. Assigning a valid drone to perform this delivery. A valid drone is a drone that has enough battery to traverse the full path, enough carry capacity to carry the package, and a flight speed such that the delivery deadline will be met.
4. Updating the drone availability list, progressing time, and adding the delivery to the initial solution.

Sweep-line

When a drone traverses a path segment s it is not sufficient to find all intersections of s in the set of all path segments since each drone has a radius and therefore collisions occurs on a capsule and not on a line. This means that another segment can collide with s without intersecting it. Two drones collide if their collision spheres intersect given the drones' current positions and a collision is checked for as described in Section 2.9.

This problem is solved by the sweep-line algorithm which is implemented as: Iterate over all segments and for each segment s, add the add-event and the remove-event to the event queue. Because a path segment has a direction, i.e. go from a, to b and that the segment is a line, each segment's endpoints are stretched by the max radius of all drones to get two

adjusted points (see Figure 3.2) that are then ordered by least x followed by least y. The leftmost adjusted point is used in the add-event and the rightmost adjusted point is used in the remove-event to ensure that no segment collision will be missed due to a drone's radius. An add-event will add s to the work set which is the set of all segments that are intersected by an infinite vertical line at the current x-coordinate and is sorted by least y. The remove-event will instead remove s from the work set. It is important that an add-event is processed before a remove-event if both events happen at the same x-coordinate as possible intersections could otherwise be missed which is why all events are sorted by least x, and add-events before remove-events. When all events have been added to the event queue, they are iterated over and the algorithm will perform the event action (i.e. add or remove a segment from the work set). If the current event is an add-event, the added segment will be checked against the work set to find any potential collisions.

Collision resolution

The initial solution will most likely have collisions that need to be resolved. Resolving collisions is done by first running the sweep-line algorithm on all delivery path segments using the segment's x and y coordinates. The collision check is performed in two steps. The first is a simple 2D (x,y) distance check intended to avoid calling the second more expensive 4D (3D in time) check from Section 2.9 as often. The distance between the segments should be less than or equal to the max of the drones' radius before the 4D check is called and when two segments collide, the deliveries they belong to are added to a collision set.
After the sweep-line algorithm has found all collisions, the set of all colliding deliveries is reduced by ordering each pair $\{deli_i, deli_j\}$ by smaller memory address and then removing $deli_j$ (now the delivery with greater memory address) from all pairs. The remaining delivery paths are removed from the current solution before they are re-planned using A* and then re-inserted back into the solution one by one. However this will not resolve all collisions as the implementation of A* is not in 4D (because of the significant increase in search space due to time being implemented as continuous) and does therefore not include time as a state parameter which can cause it to miss more optimal paths (i.e. a collision-free path with least cost). As an example of this, a path from *start* to *goal* in Figure 3.4 is looked at with costs

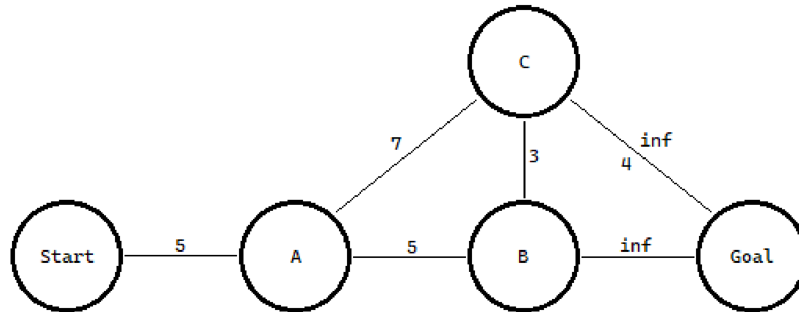


Figure 3.4: A small graph showing transition costs.

expressed in time. Beginning at *start*, A* will expand and visit node A that will expand both node B and C. Node B will be visited next and expand *goal* but report that there is a collision when travelling from B to *goal* at time 10 and since $c(B,C) + g(B) \overset{?}{<} g(C) \Rightarrow 3 + 10 \nless 12$, node C will not be updated. Lastly node C is visited which will also expand *goal* and report a collision. Since all nodes have been visited, A* will return that there is no path from *start* to *goal*. However for the sake of the example, if *goal* is visited from node C at time 13 (i.e. after having come from node B) there exists a path to *goal* with cost 17.

3.5 Heuristics

Four different heuristics were implemented and used in the work. They function by utilising: randomness, a drone's current battery level, a path's cost, or a path's delay. During ALNS, each heuristic has an initial weight which is equal to its likelihood of being chosen in each iteration. If the number of chosen heuristics is n then each heuristic's initial weight is $\frac{1}{n}$. After an iteration, the chosen heuristic k's weight $w(h_k)$ is updated depending on whether the new solution was accepted or rejected. This ensures that well performing heuristics will be chosen more often. However a poor performing heuristic can still be chosen on occasion.
To update the weights, the variable v is assigned the value 1.1 if the solution was accepted and the value 0.95 if the solution was rejected and influences how fast a heuristic's weight will change. The weights are updated as:

$$p = \frac{w(h_k)(1-v)}{n-1}$$
$$\forall i \neq k, w(h_i) \leftarrow w(h_i) + p$$
$$w(h_k) \leftarrow w(h_k) * v$$

which is simply increasing or decreasing the weight of k and then evenly distributing the difference to the other heuristics in order for the total weight sum to equal one and therefore still be a valid probability.

The neighbourhood size is denoted by q and determines how many deliveries that are destroyed and repaired in each iteration and a valid drone is any drone that could have delivered a destroyed delivery.

Random heuristic

The Random heuristic does not consider any drone nor delivery information. It is simply pure random to avoid potential bias and move uniformly in the search space. The heuristic is implemented as:
Pick q unique deliveries with uniform probability and then destroy them. Repair by randomly assigning new valid drones to the destroyed deliveries and then re-planning the delivery paths.

Battery heuristic

The Battery heuristic will attempt to distribute the workload by choosing drones with high remaining battery, believing that they have not been scheduled to deliver many packages. This might free up highly utilised drones that can then be assigned to other deliveries. The heuristic is implemented as:
Pick q unique deliveries with uniform probability and then destroy them. For each destroyed delivery, repair it by assigning a new random valid drone with 35% probability or else the valid drone that has the most available battery. Then re-plan the delivery path.

Cost heuristic

The Cost heuristic will choose deliveries with high cost (the delivery cost is the total distance cost of a delivery path). By destroying and re-planning the high cost deliveries, cheaper paths can hopefully be found. The heuristic is implemented as:
Begin with an empty list of size q sorted by increasing cost. Then iterate over all deliveries and for each delivery i, begin at the first element in the list and replace that element with i if i's cost is greater or with a 20% probability if the cost is equal or smaller. When an element in the list is replaced it is moved one step to the left (removed if it was the first element). Then keep iterating over the list until an element is not replaced or the end of the list is reached.

The final list will likely contain the most costly deliveries as well as some cheaper deliveries and will not be guaranteed to be sorted.
All q deliveries in the list are destroyed and then repaired by randomly assigning a new valid drone with 65% probability or else reusing the old drone to the delivery and then re-planning the delivery path.

Delay heuristic

The Delay heuristic uses a delivery's delay which is the total time that the assigned drone spent waiting during the delivery. A high delay means that a drone is waiting more and that there are likely many drones travelling in the same area at the same time. By re-planning in these areas, more efficient paths might be found.
The Delay heuristic uses the same method as the Cost heuristic but instead of sorting the list on increasing cost, it is instead sorted on increasing delay.

3.6 ALNS loop

ALNS was chosen as it fits the problem of iteratively improving solutions and allows the use of different heuristics which provides flexibility and an opportunity to affect the iterative process. Before running the program, any combination of heuristics can be chosen from the heuristics in Section 3.5 and prior to entering the ALNS loop, the chosen heuristics' weights are initialised with uniform probability and a copy of the initial solution is made. The copy is used as the working solution (i.e. the solution that is being modified in all iterations) and whenever a new solution is accepted, the working solution is overwritten. In case that the new solution is globally better, it will also replace the current best solution which is saved separately. Storing two solutions avoids the case where the best solution is overwritten by a locally better solution that can occur when a worse solution had previously been accepted by simulated annealing and then been improved upon.
Simulated annealing is used to accept some worse solutions in order to decrease the risk of getting stuck in local minima as well as allowing greater exploration of the problem space. The simulated annealing temperature T_0 is initialised to 1.3 and is based on relative solution cost so that a more expensive solution than the working solution will be less likely to be accepted. A solution that is less expensive will however always be accepted.
The cooling rate α was chosen to be 0.985 because it was known beforehand that the main algorithm was a bit slow and in order to avoid accepting many worse solutions later in the runtime, the temperature should decrease faster. At the iteration number i, the current temperature T_i is equal to $T_0 * \alpha^i$ and given a solution with cost c_1 and the working solution with cost c_2, the probability of the new solution being accepted by simulated annealing is $e^{-c1(c2*T_i)^{-1}}$.

In each iteration a destroy-repair heuristic pair will be chosen by using the roulette wheel policy and the heuristics' weights. The respective heuristics will then be used to first destroy and then repair the solution. A delivery is destroyed by excluding its delivery path from the collision checking which makes the path invisible to the search algorithm and will thus effectively be removed from the solution which is important to ensure that the destroyed deliveries do not interfere with each other when they are later repaired. The drone assigned to a destroyed delivery will be unassigned and marked as available which allows it to be picked to deliver other paths when repairing. The drone will also be given back the battery charge that it spent on travelling a delivery route.

A delivery is repaired by first assigning a new drone to it using the repair heuristic and then re-planning the delivery. The repaired solution will be accepted as the new working solution if it has a lower cost than the previous working solution or if it was accepted by simulated annealing. The iterating will continue for the given budget amount of seconds and then the solution with the least cost will be presented as the final solution.

4 Results

Notation: q is the neighbourhood size as a percentage of the number of agents in a scenario (see Table 3.1 for the different scenarios). h is short for heuristic with r=random, b=battery, c=cost, and d=delay (see Section 3.5 for heuristic descriptions). *Improvement rate* is defined as the number of iterations that improved the working solution given as a percentage of the total number of iterations.

Environment 1										
q	h	Scenario 1			Scenario 2			Scenario 3		
		F	IMP	IM-R	F	IMP	IM-R	F	IMP	IM-R
5%	r	3059.05	0.00	0.00%	11616.00	0.00	7.08%	26089.20	0.00	26.32%
	b	3059.05	0.00	0.00%	11616.00	0.00	7.69%	26088.60	0.60	18.75%
	c	3059.05	0.00	0.00%	11616.00	0.00	7.23%	26089.20	0.00	5.00%
	d	3059.05	0.00	0.09%	11615.90	0.10	6.03%	26084.70	4.50	25.00%
15%	r	3059.05	0.00	0.53%	11616.00	0.00	14.00%	26085.50	3.70	12.50%
	b	3059.05	0.00	1.60%	11616.00	0.00	1.96%	26089.20	0.00	0.00%
	c	3059.05	0.00	0.00%	11615.40	0.60	**14.81%**	26089.20	0.00	12.50%
	d	3059.05	0.00	0.00%	11614.70	1.30	3.70%	26078.90	10.30	**27.78%**
33%	r	3059.05	0.00	0.83%	11616.00	0.00	3.45%	26089.20	0.00	22.22%
	b	3059.05	0.00	**2.52%**	11616.00	0.00	3.70%	26089.20	0.00	0.00%
	c	3059.05	0.00	1.96%	11616.00	0.00	0.00%	26089.20	0.00	0.00%
	d	3059.05	0.00	2.15%	11616.00	0.00	1.64%	26089.20	0.00	0.00%

Table 4.1: F=Final solution cost, IMP=Cost improvement, IM-R=Improvement rate. Red-marked cell shows the greatest improvement rate for each scenario.

Before each scenario was run, an initial base solution was found and to ensure consistency, each scenario that was run with different parameters began the ALNS loop with the base solution for that scenario.

Table 4.1 shows the results for the three different scenarios in the first environment together with the parameter values used in each different experiment. Some rows show zero cost improvement which means that the initial solution was not improved upon and that the final solution cost is equal to the initial solution cost. Table 4.2 shows the experiments run on the second environment. To better read the results, the neighbourhood size and heuristic

combination for each scenario that achieved the highest improvement rate is highlighted in red. In addition, because many experiments did not show any cost improvement, improvement rate will be looked at almost exclusively to determine how well a parameter combination performed.

Environment 2										
q	h	Scenario 1			Scenario 2			Scenario 3		
		F	IMP	IM-R	F	IMP	IM-R	F	IMP	IM-R
5%	r	8394.27	0.00	0.23%	37612.70	3.00	16.24%	80545.7	3.70	25.00%
	b	8394.27	0.00	0.32%	37613.20	2.50	13.93%	80549.4	0.00	25.00%
	c	8394.27	0.00	0.00%	37615.70	0.00	3.64%	80547.9	1.50	22.22%
	d	8394.27	0.00	0.00%	37613.60	2.10	17.56%	80537.5	11.90	33.90%
15%	r	8394.27	0.00	1.64%	37613.60	2.10	19.57%	80546.7	2.70	4.17%
	b	8394.27	0.00	0.74%	37614.60	1.10	15.91%	80549.4	0.00	4.17%
	c	8394.27	0.00	0.83%	37615.70	0.60	11.11%	80549.4	0.00	0.00%
	d	8394.27	0.00	0.40%	37608.50	7.20	**33.33%**	80529.9	19.50	**38.10%**
33%	r	8394.27	0.00	1.16%	37614.30	1.40	4.17%	80549.4	0.00	0.00%
	b	8394.27	0.00	2.63%	37615.70	0.00	0.00%	80549.4	0.00	10.00%
	c	8394.27	0.00	**6.09%**	37613.00	2.70	30.00%	80549.4	0.00	0.00%
	d	8394.27	0.00	0.60%	37612.30	3.40	31.25%	80549.4	0.00	30.77%

Table 4.2: F=Final solution cost, IMP=Cost improvement, IM-R=Improvement rate. Red-marked cell shows the greatest improvement rate for each scenario.

Environment	q	Arithmetic mean			Median		
		Scenario 1	Scenario 2	Scenario 3	Scenario 1	Scenario 2	Scenario 3
1	5%	0.02%	7.01%	**18.77%**	0.00%	7.16%	**21.88%**
2		0.14%	12.84%	**26.53%**	0.11%	15.08%	**25.00%**
1	15%	0.53%	**8.62%**	13.20%	0.27%	**8.85%**	12.50%
2		0.90%	**19.98%**	11.61%	0.78%	**17.74%**	4.17%
1	33%	**1.87%**	2.20%	5.56%	**2.05%**	2.55%	0.00%
2		**2.62%**	16.36%	10.19%	**1.90%**	17.09%	5.00%

Table 4.3: Improvement rates with marked cells showing the highest value for each environment-scenario combination.

Environment	h	Arithmetic mean			Median		
		Scenario 1	Scenario 2	Scenario 3	Scenario 1	Scenario 2	Scenario 3
1	r	0.45%	**8.18%**	**20.35%**	0.53%	7.08%	22.22%
2		1.01%	13.33%	9.72%	**1.16%**	16.24%	4.17%
1	b	**1.37%**	4.45%	6.25%	**1.60%**	3.70%	0.00%
2		1.23%	9.95%	13.06%	0.74%	13.93%	10.00%
1	c	0.65%	7.35%	5.83%	0.00%	**7.23%**	5.00%
2		**2.31%**	14.92%	7.41%	0.83%	11.11%	0.00%
1	d	0.75%	3.79%	17.59%	0.09%	3.70%	**25.00%**
2		0.33%	**27.38%**	**34.26%**	0.40%	**31.25%**	**33.90%**

Table 4.4: Improvement rates with marked cells showing the highest value for each environment-scenario combination.

Table 4.3 shows the arithmetic mean and median for the improvement rates in both environment one and two for their three scenarios. The highlighted cells shows what neighbourhood size that achieved the greatest values for each environment and scenario pair. In Table

4.4 the arithmetic mean and median for the improvement rates in environment one and two is instead shown based on the different heuristics.

h	Environment 2 – Scenario 2			Environment 3 – Scenario 1			Environment 4 – Scenario 1		
	F	IMP	IM-R	F	IMP	IM-R	F	IMP	IM-R
r	37613.6	2.10	19.57%	16655.5	0.10	15.48%	38148.0	0.00	5.26%
b	37614.6	1.10	15.91%	16655.6	0.00	3.70%	38148.0	0.00	0.00%
c	37615.7	0.00	11.11%	16655.0	0.60	17.86%	38146.6	1.40	**20.00%**
d	37608.5	7.20	33.33%	16652.0	3.60	9.91%	38148.0	0.00	15.00%
rb	37612.8	2.90	10.14%	16655.6	0.00	12.67%	38148.0	0.00	5.41%
rc	37613.1	2.60	14.71%	16654.5	1.10	15.25%	38148.0	0.00	10.00%
rd	37611.5	4.20	18.97%	16655.6	0.00	17.32%	38148.0	0.00	5.41%
bc	37631.8	1.40	20.00%	16655.0	0.60	19.05%	38148.0	0.00	3.70%
bd	37610.3	5.40	22.03%	16653.3	2.30	13.89%	38147.5	0.50	15.91%
cd	37616.9	2.50	**38.46%**	16652.3	3.30	**22.64%**	38147.2	0.80	12.90%
rbc	37615.7	0.00	6.54%	16655.6	0.00	10.59%	38146.6	1.40	17.39%
rbd	37615.7	0.00	21.15%	16655.5	0.10	7.41%	38148.0	0.00	11.76%
rcd	37614.7	1.00	17.65%	16651.9	3.70	15.12%	38146.6	1.40	12.50%
bcd	37613.0	2.70	26.67%	16655.6	0.00	13.70%	38148.0	0.00	19.04%
rbcd	37615.7	0.00	30.23%	16655.6	0.00	12.22%	38148.0	0.00	11.11%

Table 4.5: F=Final solution cost, IMP=Cost improvement, IM-R=Improvement rate. Red-marked cell shows the greatest improvement rate for each scenario.

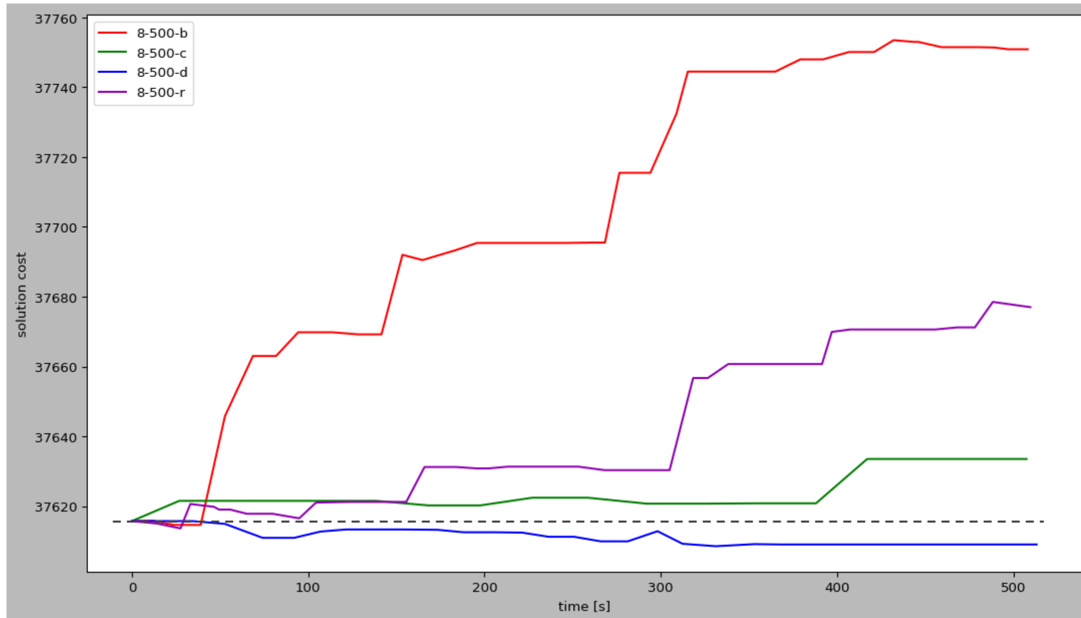
Similarly to Tables 4.1 and 4.2, Table 4.5 shows the results for three different scenarios. The environments are however different and the neighbourhood size was set to a constant 15% in all the scenarios. The table shows how each heuristic combination performed across three different environments with similar scenarios (for a reminder of the scenarios, see Table 3.1). Again, the greatest improvement rate is highlighted in red in the row corresponding to the heuristic combination used to achieve that result.

The figures 4.1, 4.2, and 4.3 show how the working solution's cost changed over time. Each figure is divided into two sub-figures where the above figure shows cost over time for the single heuristics and the below figure shows the cost over time for the remaining heuristic combinations. The scenarios shown in the figures are the same as in Table 4.5.

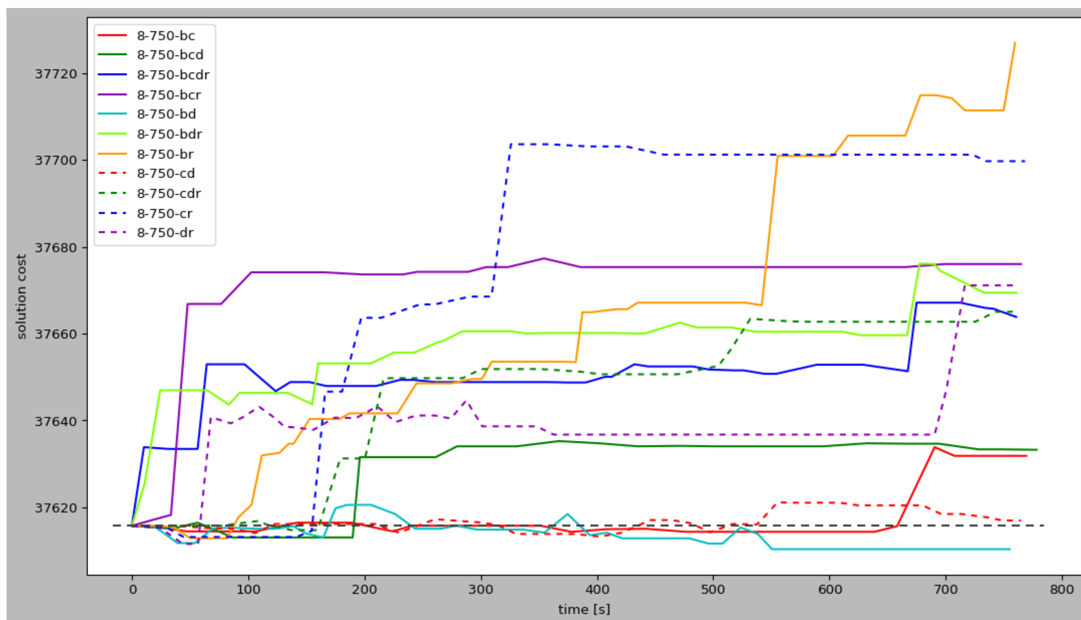
In the top-left corner of each figure, there is text that contain information on each of the experiments and the colour that corresponds to which experiment. The first number in the text is the neighbourhood size and the second number is the budget in number of seconds while the remaining text shows what heuristic combination that was used.

The grey dashed line in each figure shows the base solution's cost for that scenario and is used to easily tell if a solution's cost was less than the base solution's cost. In the figures it can also be seen that sometimes the budget is exceeded which is because the remaining budget was checked before each iteration began and an iteration was not aborted if it exceeded the budget after it had already begun.

Table 4.6 shows the cost of the best solutions for each scenario and what parameters that were used to achieve them. In two cases, the best solution was also the initial solution (i.e. no improvements were made). The optimal solution cost is not guaranteed to be the actual optimal solution and instead it shows the solution cost when the environment is obstacle-free and all collisions were ignored during path finding. The optimal solution costs are used to greedily estimate an optimal solution.

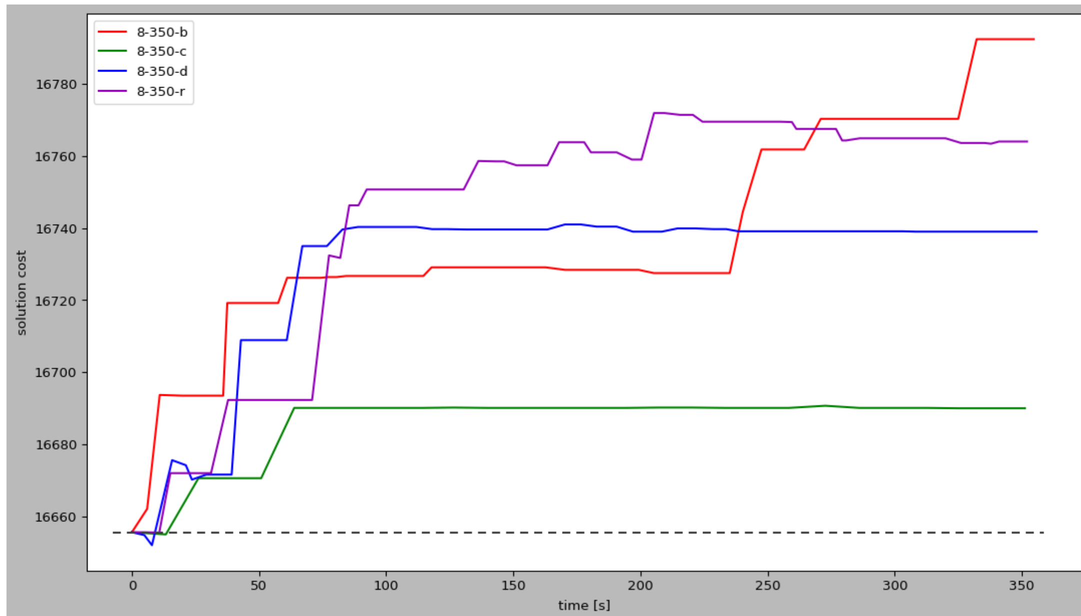


(a)

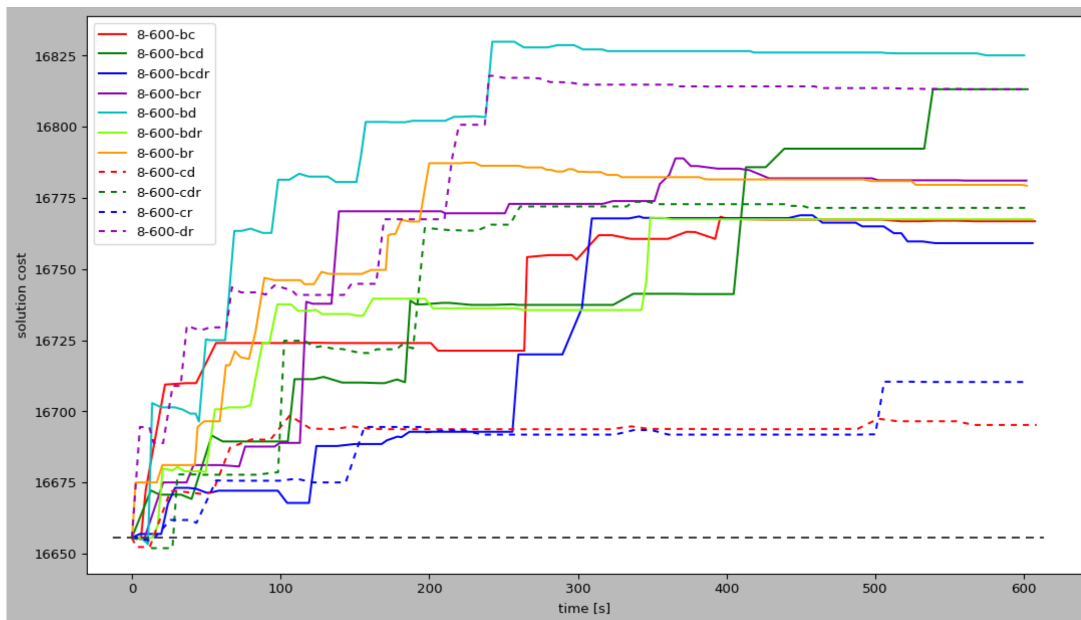


(b)

Figure 4.1: Environment 2, scenario 2. (a) Shows the solution cost over time for each of the four heuristics. (b) Shows the solution cost over time for each heuristic combination with at least size two.

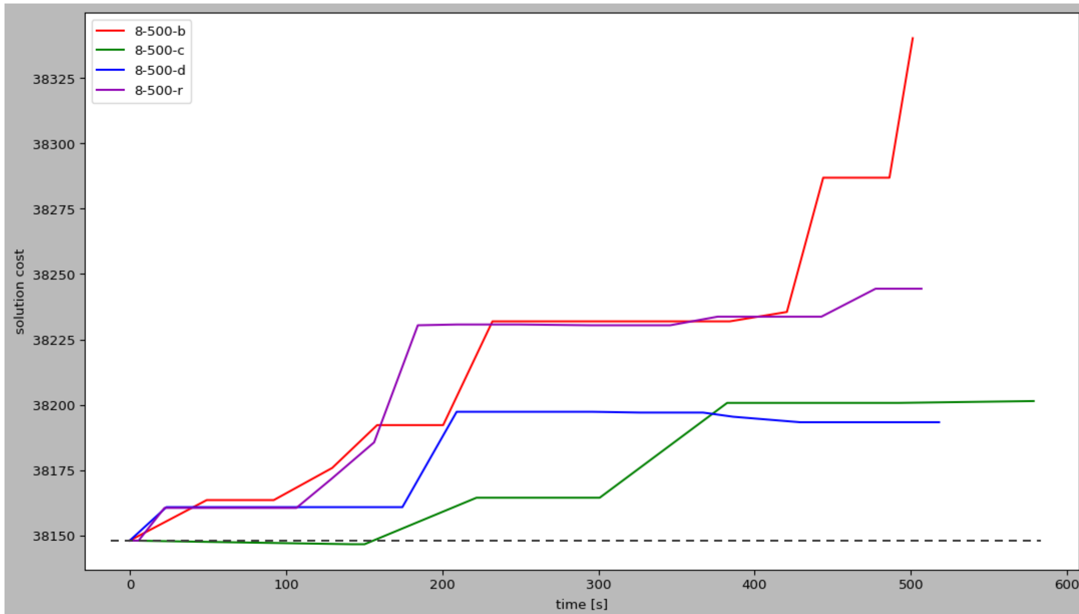


(a)

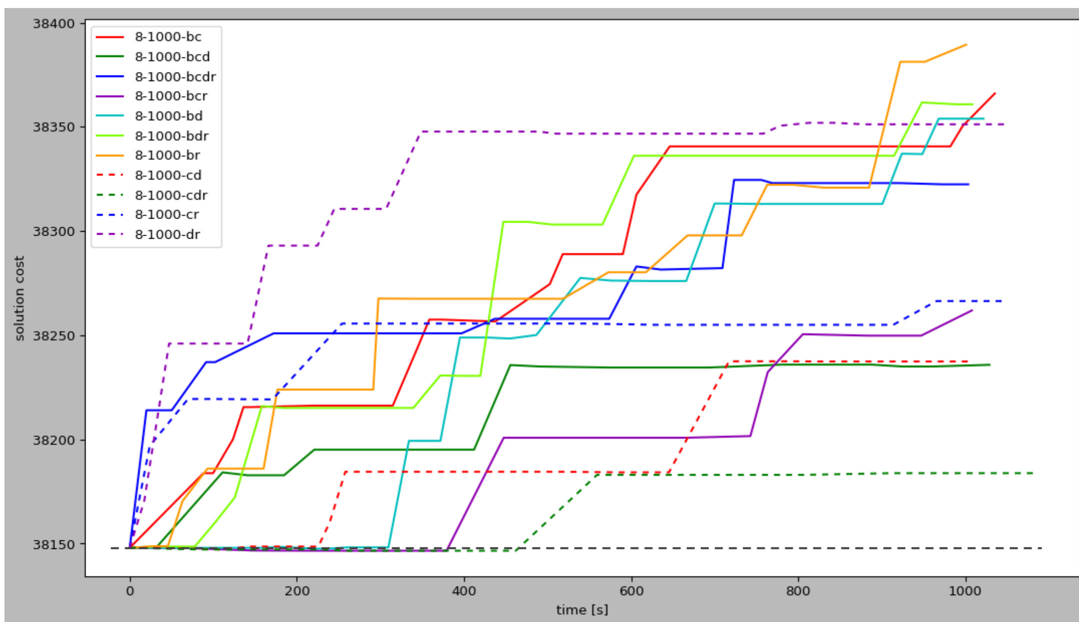


(b)

Figure 4.2: Environment 3, scenario 1. (a) Shows the solution cost over time for each of the four heuristics. (b) Shows the solution cost over time for each heuristic combination with at least size two.



(a)



(b)

Figure 4.3: Environment 4, scenario 1. (a) Shows the solution cost over time for each of the four heuristics. (b) Shows the solution cost over time for each heuristic combination with at least size two.

<table>
<tr><th>Environment</th><th>Scenario</th><th>Optimal* solution cost</th><th>Best solution cost</th><th>Parameters used</th></tr>
<tr><td rowspan="3">1</td><td>1</td><td>2514.35</td><td>3059.05</td><td>initial</td></tr>
<tr><td>2</td><td>10585.00</td><td>11614.70</td><td>8d</td></tr>
<tr><td>3</td><td>23449.70</td><td>26078.90</td><td>18d</td></tr>
<tr><td rowspan="3">2</td><td>1</td><td>6288.18</td><td>8394.27</td><td>initial</td></tr>
<tr><td>2</td><td>32746.70</td><td>37608.50</td><td>8d</td></tr>
<tr><td>3</td><td>68306.70</td><td>80529.90</td><td>18d</td></tr>
<tr><td>3</td><td>1</td><td>14338.80</td><td>16651.90</td><td>8rcd</td></tr>
<tr><td>4</td><td>1</td><td>33121.30</td><td>38146.60</td><td>8 × {c, rbc, rcd}</td></tr>
</table>

Table 4.6: Solution costs for each scenario together with what parameters that were used to achieve that result.

5 Discussion

5.1 Results

From the Figures 4.1, 4.2, and 4.3 it should be clear that most iterations do not improve the solution regardless of the chosen combination of heuristics, and that no solution was improved significantly (nor noticeably). Three possible reasons for this are that the initial solution ignores collisions during path finding and uses Lazy-Theta* which can find up to 13% shorter paths compared to A* (see Section 2.7) and thus the solution before collision resolution using A* is much better in terms of cost. Since not all paths will be re-planned during collision resolution it is likely that many paths are still the original Lazy-Theta* found paths which makes it difficult to improve them when using a repair heuristic with A*.
The second reason is that few iterations were computed (low-end is 20 iterations whereas high-end is 1800 iterations) and therefore there was little (compared to the size of all possible delivery combinations) exploration of the neighbourhood and overall problem space. The reason that few iterations were computed was likely not due to a small budget (ranging between 250 and 1000 seconds, see Table 3.2) and instead due to the program being slow. The most computationally heavy parts of the main algorithm is the search and the collision avoidance during search. While the search algorithm is fast, the search space is still large and collisions will have to be avoided which slow things down. Avoiding collisions requires that collision avoidance is performed in each node visited during search and since querying collisions have linear time complexity, the search is understandably slow.
The third reason is the possibility that the scenarios and environments used were too simple. There can have been sufficient space for drones to easily avoid each other and therefore be able to travel a more optimal path or the deliveries could have been very spread out, also making travel easy. In either case it is hard to make any definitive claim on the difficulty of the scenarios and environments without more analysis.

Looking at the Table 4.3 it can be seen that the highest improvement rates were all when the neighbourhood size had a value $q \simeq 8$ and that increasing the number of deliveries almost always increased the improvement rate in all scenarios. This is not too surprising since more deliveries mean that there are also more different ways of destroying and repairing a solution and a higher likelihood of a given re-planning to be better. A smaller neighbourhood size will likewise increase the likelihood of improving the solution since fewer re-planned paths need to be improved overall for it to be accepted. However a very small neighbourhood size was

not effective as it only allows very small steps in the neighbourhood which makes it hard to escape local minima. Therefore it seems necessary to find a good neighbourhood size that is as small as possible while still allowing sufficient exploration of the problem space in each iteration to efficiently find improvements.
It should however be noted that most improvements were on solutions that had previously been accepted due to simulated annealing and therefore had higher cost than the best solution (i.e. the improvements were often not globally better and came only after a solution had been made costlier). This should not affect the results negatively as it still shows what neighbourhood size and heuristics that found improvements more often (even though those improvements might have been easier to find).
Table 4.4 shows that the Delay heuristic had high improvement rates for both scenario two and three in the second environment and looking at Tables 4.1 and 4.2 it can also been seen that the Delay heuristic almost always led to the greatest cost improvements. The Random heuristic generally performed well compared to the other heuristics but was not the best performer except for scenario two and three in the first environment. Given the results it is difficult to determine if any heuristic is truly better than the others as the results differ decently between the different scenarios. However, the Delay heuristic is generally good.
It should be noted that due to the difference in the number of iterations between the small and large scenarios, the large scenarios can have inflated improvement rates. This is because the number of improvements given a large number of iterations and a small improvement rate can be greater than the number of improvements given the opposite. Looking at figure 4.3 it can be seen that it shows little motion due to having few iterations and making small improvements, suggesting that the improvements were less impactful compared to the smaller scenarios but easier to find. Regardless of the potential inflation, the Delay heuristic seem to be good in scenarios with many packages but it is unclear why it performs better. Further more, the probable reason for why scenario one has small improvement rates is that it is the smallest scenario, making it difficult to find improvements even when having been run for the most amount of iterations.
Table 4.5 shows that combining heuristics did not yield any significant improvements over single heuristic runs. The best combination seem to be combining the Cost and Delay heuristics but also that combining any two or more heuristics lead to slightly higher improvement rates.

All of the environments (see Figure 3.1) had buildings as tall as 10 meters with bounding boxes supporting travel up to 25 meters. There was no restriction nor incentive for a drone to travel at any particular height which meant that most path segments were close to the ground in order to minimise distance. That is however not very realistic considering that the ground is "reserved" for pedestrians and ground vehicles. It is possible that different results would have been achieved if the drones were forced to fly above the ground, especially above the buildings where there are no obstacles unless they need to descend in order to pickup or deliver a package.

5.2 Method

During search, possible collisions between two points needs to be checked so that they can be avoided. An attempt was made to find potential collisions at a point by using two 4D domination queries given a point and all other path segments' 4D boxes (see Section 3.2) as an alternative to 4D rectangle stabbing [1]. In the attempt the 4D box had an additional bit that was used as a flag to note if it was part of the result in the first 4D domination query. In the second 4D domination query, only the boxes with their bit set to one were included as the final query result and after each query, all bits were reset to zero. The 4D domination query was actually implemented as a 3D paper stabbing (as Theorem 9 from the paper "Linear Space Data Structures for Two Types of Range Search" by B Chazelle and H Edelsbrunner [2]) query

followed by a linear check on the fourth coordinate. A 2D domination query is effectively checking if a value q is in an interval $[a, b]$ by taking q as a point $(q, -q)$ with $a \leqslant q$ and $-b \leqslant -q$. Given a query point (x, y, z, t), the 8D domination query was implemented as the intersection of a 4D domination query on a new point (x, x, y, y) and another 4D domination query on the point (z, z, t, t).
This attempt failed and was therefore both incorrect and slower than the later used bucket solution (see Section 3.2). As the problem of finding drone collisions is to find all cases where two or more drones' collision spheres contain another drone at some point in time for each expanded node in the search, it is important that this process is fast which neither attempt in the implementation is. Future work is to perform 4D rectangle stabbing to find collisions or alternatively find another method that might be more suitable for the problem.

A downside with querying a single point when checking for collisions is that the approach can only find collisions at a given point and not on a segment (i.e. anywhere between two points). This means that some collisions might not be found when travelling between two points even if both endpoints are checked. An alternative is to check points on the segment by discretising it but that does not eliminate the problem, instead it only makes it less noticeable by reducing the distance between any two checked point. Reducing the distance between two points in a path was the greatest motivation for the use of A* when re-planning paths instead of using Lazy-Theta* (see Section 2.7). As a path found by A* will have all points be immediate neighbours, the distance between two points is bounded by the max distance between any two neighbours which is $\sqrt{3}$ in the work as opposed to Lazy-Theta* where the max distance is only bounded by the size of the environment. Even though A* was used and the distance between two neighbouring points is less than or equal to $\sqrt{3}$, many collisions were still left unresolved but it did reduce the total cumulative collision time as well as the size of each collision. The size of a collision can be seen as the least distance between the drones during a collision and the collision span.
If collision checking during search can take a segment as input then Lazy-Theta* can be used instead of A* to find better delivery paths. Lazy-Theta* would have had the additional benefit of producing paths with fewer path segments which is good since the collision checking scales on the total number of path segments. Therefore it is beneficial to reduce the number of path segments to save memory and speed up the process. It can be seen from the simple example that follows that Lazy-Theta* will produce paths with fewer path segments:
Traversing from the point $(0, 0)$ to the point $(x, 0)$ will generate x path segments when using A* and one path segment when using Lazy-Theta* (because of its any-angle property). If we assume that there are obstacles that block line-of-sight then Lazy-Theta* will in the worst case produce an equal number of path segments as A*.

Resolving collisions as a step before the ALNS loop worked fairly well consider the problems with the A* search (see the example in Section 3.4) and finding collisions (see above 5.2). It was however not without problems. When a delivery path was re-planned it was moved to begin at some time when the newly assigned drone was available to deliver it and its deadline would not be missed (using the previous path as an estimate). This worked most of the time but could cause cases where a drone would end a delivery after it should have begun another delivery. This often small overlap in time was allowed because it was much easier to allow than to solve. There are two ways of solving it, either re-plan the path starting at some other time or by assigning a new drone to fly it, or delay the other path which would require it to be re-planned as well with no guarantee that the problem would not repeat itself. The third option is to assign drones to deliveries in the initial solution before search instead of after which would also eliminate the collision resolution step entirely. The difficulty with this option is that it would be more time consuming as little information is available to determine if a drone assignment is good before a path is found. There is then risk that many delivery paths would need to be searched for until one is found where the drone assignment is also valid.

The battery consumption rate (see Section 3.3) is a percentage-unit drain instead of kilojoules which means that the battery consumption is very specific to the drones used in this work (which are the same drones as in [20] regarding their battery consumption). It was assumed that the linear relationship found in [20] would hold as heavier packages were used in this work (max 750 grams) compared to the paper (max 400 grams). There was also no time included for the drop-off (i.e. the actual delivery of the package) and therefore the battery consumption rate per delivery is lower than it should be given the extra delivery time required for the drop-off. To get more accurate results, it is good to use more of a drone's physical specifications and plan the delivery more thoroughly.
Something that was not done was having more than one depot that packages and drones can originate from and that drones can return to. Having multiple depots would of course add more complexity but it would also add more flexibility and the possibility of having scenarios in very large environments (for example an entire city). In larger environments it would be interesting to determine what benefits recharging stations either in the environment or in the depots can have for the drone delivery problem. Work on placing recharging stations have already been done by S Deshpande and P Mani [3] and could be applied to a top-down view of the environment (similar to Figure 3.1). It should be noted that a depot in the work is a single point in empty space and is not attached to any building. If the take-off point for deliveries is not outside with free space around it, a building (representing the warehouse) should be placed next to the depot to get correct paths.

Constraint based search [16] (CBS) is an alternative method that can be used to find an initial solution instead of prioritised planning (see Section 2.5). CBS is complete and can handle other constraints that might be placed on a solution. For example, only use a specific drone for some deliveries or give priority to certain deliveries. The fact that prioritised planning is incomplete did not seem to affect the produced initial solutions which might be because each scenario was easier to solve due to them having many drones compared to the number of deliveries (see Table 3.1). It is also possible that the environments were too small or that the drones had too much available battery for the scenarios to be considered difficult to solve.

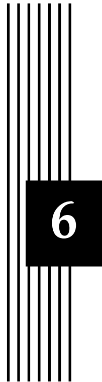
The repair heuristic (see Section 3.5) did only decide what drone to pick for a new path and was not related to the searching for a new path as it might otherwise be. As what drone to assign to a delivery is by itself a combinatorial assignment problem, more work can be done on finding near-optimal solutions. This is however not a problem if a uniform fleet of drones are employed which was not the case in this work.
The destroy Cost and Delay heuristics (see Section 3.5) used in the work might not have enough randomness to be effective at exploring a neighbourhood. This is because they are dependent on the order that deliveries are traversed in and therefore it can lead to very consistent outputs when the most costly/delayed deliveries appear last in the unsorted list of deliveries. To introduce more randomness into the destroy part, either the initial weights in the ALNS loop can be altered to give the Random heuristic a larger weight, make the traversal less order dependent, or introduce simulated annealing to the randomness in the heuristics. For example, the Cost heuristic could begin with a 40% probability (instead of the current constant 20%) to replace a delivery with another less costly one and then decrease this probability over time.
The destroy and repair heuristics in each iteration are chosen in pairs but are weighted individually which means that information on the performance of the pair is lost. It might then be better to store weights per each destroy-repair heuristic pair to avoid that problem. Although there might not be any noticeable difference since the repair heuristics are mostly very similar except for the Battery heuristic.

The sources used in the work are mostly from proceedings with higher reputation. Literature was chosen based on perceived relevance and quality from what was available (when the work was carried out). Many sources are older simply because they describe the fundamentals of older methods that are still relevant and used today. Some newer sources were also chosen as they contain the current improvements on topics used in the work.

5.3 The work in a wider context

Successfully establishing a system that can perform last-mile deliveries using drones have the potential to help the environment by reducing greenhouse emissions [8] in addition of being used as an option of delivering necessary items (for example medicine) to obstructed or hard-to-reach areas. It could also be used to deliver smaller items to people that have difficulties going out to buy them themselves (for example, if someone is ill, elderly, or disabled).

Since each drone can only carry small packages, many drones will have to be used for high density cities with a lot of delivery requests. It can therefore led to increased noise pollution, or risk collision with birds or having insects being caught in spinning blades. A more positive effect is that there will possibly be less ground traffic, leading to more area that can be repurposed.



6 Conclusion

6.1 Conclusion

The purpose of the work was to study the drone delivery problem in 3D. In order to study the problem, a system that can near-optimally assign delivery routes to a fleet of drones to minimise total route cost was implemented. The system had to consider various constraints, including limited battery capacity, delivery deadlines, and weight restrictions for pickups and drop-offs.

The research questions and their respective answers:

1. Using adaptive large neighbourhood search, what heuristic or combination of heuristics provides the highest quality solution for the drone delivery problem in 3D?
   – The Delay heuristic was overall the best heuristic regardless of whether it was used by itself or in combination with others as can be seen in Table 4.6 and when looking at the improvement rates. The Cost and Random heuristics were also good while the Battery heuristic struggled to produce good results. The best heuristic combination was Cost and Delay. However it is difficult to say what heuristic combination was truly best as no scenario showed real improvements.

2. What impact does the map size, number of deliveries, and number of available delivery drones have on the scalability of the drone delivery problem in 3D?
   – The map size and the number of deliveries have great impact on the scalability as they both contribute to the size of the search space. Map size affects the maximum delivery distance which means potentially longer paths and the number of deliveries will affect how many drones that are out performing deliveries at a given time which influences the time complexity of collision avoidance. The total number of available drones had very little impact on scalability.

3. What time is required to obtain a solution that is close to the theoretical/known optimal solution?
   – It is difficult to say anything as no real improvements were had. The program was slow but despite that, it can be seen from the figures in Chapter 4 that most attempts plateaued after roughly half of their budget had elapsed which suggests that the given

budgets were sufficient. Therefore the budgets will probably function as good guidelines for any work that tries something similar.

The work has shown that finding collision free solutions to the drone delivery problem in 3D with time is difficult but that it is possible to find solutions with reduced collision time and where no deadlines are missed. More work is clearly required before a good solution to the problem can be found. The biggest hurdle is to quickly find collision free paths for many drones in order to be able to explore the problem space efficiently.

The work has also shown that a smaller neighbourhood size (around a value of eight) led to better results and that combining heuristics achieved slightly better results compared to single heuristics which suggests that more work is required to find better heuristic combinations and when a specific combination may perform better.

6.2 Future work

Lazy-Theta* (see Section 2.7) uses a line-of-sight function which in the work is implemented as a ray-cast between two points A and B. A ray-cast is slow as it needs to perform checks in discrete steps to detect obstacles and it can even miss obstacles due to the step length. A possible faster solution (depending on the implementation) but that costs more memory given the environment used in the implementation is:

1. Fill all empty space in the environment with 3D rectangles such that the number of rectangles are minimised.

2. For each of a rectangle's six faces, the rectangle will store the faces' neighbours in six 2D rectangle stabbing data structures that are used to find transitions between two rectangles. Two rectangles are neighbours if they have partial or complete overlap between two of their faces.

3. Construct a 3D rectangle stabbing data structure to be able to find the rectangle that contains a query point.

4. Line-of-sight will be checked between two points A and B by:

   i. Find the rectangle R containing A.
   ii. Check if B is in R. If yes, then there is line-of-sight, if not then continue.
   iii. Get the vector from A to B and use it to find the intersection point on one of R's faces.
   iv. If a neighbour exists, update R to be the neighbour and then goto ii., if no neighbour exists then there is no line-of-sight.

5. Corners and diagonal neighbours will also have to be checked as special cases.

The algorithm will step trough a sequence of 3D rectangles until the point B is contained in one of them or an obstacle is hit (because no neighbour exists). As the time complexity is dependent on the number of empty rectangles and their size, a highly dense environment with small empty spaces is not suitable for the algorithm.

Bibliography

[1] Peyman Afshani, Lars Arge, and Kasper Green Larsen. “Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model”. In: *Proceedings of the twenty-eighth annual Symposium on Computational Geometry*. 2012, pp. 323–332.

[2] Bernard Chazelle and Herbert Edelsbrunner. “Linear space data structures for two types of range search”. In: *Discrete & Computational Geometry* 2.2 (1987), pp. 113–126.

[3] Saayuj Deshpande and Purushotham Mani. *Drone Delivery Optimization*. 2023. arXiv: 2311.17375 [math.OC]. URL: https://arxiv.org/abs/2311.17375.

[4] Kevin Dorling, Jordan Heinrichs, Geoffrey G. Messier, and Sebastian Magierowski. “Vehicle Routing Problems for Drone Delivery”. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 47.1 (2017), pp. 70–85. DOI: 10.1109/TSMC.2016.2582745.

[5] Michael Erdmann and Tomas Lozano-Perez. “On multiple moving objects”. In: *Algorithmica* 2 (1987), pp. 477–521.

[6] Michel Gendreau and Christos D Tarantilis. *Solving large-scale vehicle routing problems with time windows: The state-of-the-art*. Cirrelt Montreal, QC, Canada, 2010.

[7] Javier Gómez-Lagos, Benjamín Rojas-Espinoza, and Alfredo Candia-Véjar. “On a pickup to delivery drone routing problem: Models and algorithms”. In: *Computers & Industrial Engineering* 172 (2022), p. 108632.

[8] Anne Goodchild and Jordan Toy. “Delivery by drone: An evaluation of unmanned aerial vehicle technology in reducing CO2 emissions in the delivery service industry”. In: *Transportation Research Part D: Transport and Environment* 61 (2018). Innovative Approaches to Improve the Environmental Performance of Supply Chains and Freight Transportation Systems, pp. 58–67. ISSN: 1361-9209. DOI: https://doi.org/10.1016/j.trd.2017.02.017. URL: https://www.sciencedirect.com/science/article/pii/S136192091630133X.

[9] Peter E Hart, Nils J Nilsson, and Bertram Raphael. “A formal basis for the heuristic determination of minimum cost paths”. In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[10] Florence Ho, Artur Goncalves, Ana Salta, Marc Cavazza, Ruben Geraldes, and Helmut Prendinger. “Multi-agent path finding for UAV traffic management”. In: *International Conference on Autonomous Agents and Multiagent Systems* (2019).

[11] Hang Ma, Daniel Harabor, Peter J Stuckey, Jiaoyang Li, and Sven Koenig. "Searching with consistent prioritization for multi-agent path finding". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 7643–7650.

[12] Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Felner. "Theta^*: Any-angle path planning on grids". In: *Aaai*. Vol. 7. 2007, pp. 1177–1183.

[13] Alex Nash, Sven Koenig, and Craig Tovey. "Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 24.1 (July 2010), pp. 147–154. DOI: 10.1609/aaai.v24i1.7566. URL: https://ojs.aaai.org/index.php/AAAI/article/view/7566.

[14] Aditya Paul, Michael W Levin, S Travis Waller, and David Rey. "Data-driven optimization for drone delivery service planning with online demand". In: *Transportation Research Part E: Logistics and Transportation Review* 198 (2025), p. 104095.

[15] Stefan Ropke and David Pisinger. "An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows". In: *Transportation science* 40.4 (2006), pp. 455–472.

[16] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[17] Martin WP Savelsbergh and Marc Sol. "The general pickup and delivery problem". In: *Transportation science* 29.1 (1995), pp. 17–29.

[18] Michael Ian Shamos and Dan Hoey. "Geometric intersection problems". In: *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE. 1976, pp. 208–215.

[19] Paul Shaw. "Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems". In: *Principles and Practice of Constraint Programming — CP98*. Ed. by Michael Maher and Jean-Francois Puget. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 417–431. ISBN: 978-3-540-49481-2.

[20] Maryam Torabbeigi, Gino J Lim, and Seon Jin Kim. "Drone delivery scheduling optimization considering payload-induced battery consumption rates". In: *Journal of Intelligent & Robotic Systems* 97.3 (2020), pp. 471–487.