April 2020
Laboratoire d'informatique
l'École Polytechnique
Alexis Roger

PARETO OPTIMAL PATHFINDING

Bachelor Thesis Report

Abstract

The goal of this project is to study how multiple robots evolve in a common environment. The main approach that we will follow is a centralized one with the goal of finding the Pareto optimal path for each robot, minimizing the global time needed for every robot to reach its goal. This will be studied for when the robots have a few fixed path possibilities each and we can only vary their speed. The solution to this coordination problem will be found primarily through the use of coordination spaces. We will therefore study these spaces in depth in order to optimize both time and space taken by finding tricks to diminish their complexities. We will be testing these methods on Crazyflie drones hence all algorithms support 3 dimensional space but this could also work fine in a more restricted space with, for instance, wheeled robots.

Supervisor: Professor Eric Goubault
Github repository: https://github.com/cosynus-lix/roger_bachelor_thesis_crazyflie

Contents

1 Introduction

Ever since we have been building robots, we have been attempting to render them more and more independent. One of the key aspects of this is when trajectory planning, to get our robots from a point A to a goal, B. For one robot this is easy enough and some very high performing algorithms have been developed but what happens when we try to add more robots in the same environment? How do we coordinate all of these robots? How do we make sure that they all reach their intended destination in the most optimal way? These are the questions we will attempt to bring solutions to in this paper.

Say we have two people: one person standing on at the point A and one standing at the point B. If we tell them to switch places, they will naturally start walking towards each other and then slightly deviate from their path to avoid bumping into each other. And then go straight to their objective. This is precisely the type of behavior we wish to incorporate into our robots. The fact that we do this so naturally on a daily basis makes it rather challenging to pinpoint our logic precisely to attempt to program it. This behavioral trait is what describes the class of robots we are working with: autonomous guided vehicles.

Autonomous guided vehicles are a type of robots used in many fields and have all kinds of different applications. For example, in numerous industries, ground based autonomous guided vehicles are the foundation of factory arrangements and optimization. For instance, Amazon, in conjunction with Kiva Systems, has been developing small wheeled robots to assist workers and work alongside them in its fulfilment centers, which serve as logistical hubs. These robots a small multi wheeled cubes that wonder around the floor, picking up shelves and bringing them to a fixed worker who then picks out the required item. Having a single robot, it can move anywhere it so desires, while avoiding fixed obstacles such as the workstations. Having two of them, again, there is a simple solution of cutting the room in two parts, each robot taking care of half the space. But what if there are one thousand robots? With many workstations and obstacles? This is the problem we are attempting to solve. But we do not only wish to have a solution, we want to have the best solution. An example of these robots in action can be seen in figure 1.

Figure 1: Amazon robots in action in a warehouse. Photo by Joel Eden Photography, Kiva Systems

1.1 Background

There are multiple different types of coordination methods that can be used with varying complexities. The choice of which to use usually boils down to the situation in which it is made to be used. All of the variations can be sorted in a couple main categories which depend on two key factors for each drone: their speed at any given time and the path they need to follow to achieve their goal. As it would be to computationally expensive to search the space of all possible paths each drone can take and to then examine all possible speed variations over each path, we usually set either the path or the speed to be constant, or at least with little variation.

The first case scenario is to have all of our robots to have a fixed speed and to adapt their path to avoid collisions. A simple example of this would be for the managing of a fleet of fixed-winged autonomously guided planes. Aircraft can not just stop air and maintain a relatively constant speed. There trajectories therefore need to be modified to avoid collisions. Hence an air traffic controller application would work with each robot having a fixed speed and will bend the most direct paths to avoid collisions while maintaining efficiency.

In the other case, we assume that the robots can freely vary there speed but need to stick to their precalculated trajectory. This scenario is more applicable to wheeled vehicles or drones, which can respectively stop and stabilize on the spot. This means that is two drones might cross paths at the same moment, resulting in a collision, one will stop and leave the other pass. However, a rigid system like this poses its own set of problems. For instance, if the path of one robot is entirely contained within the path of another robot, they can not close and so this situation is unsolvable. However, this is the most recurrent situation and the most promising. This is why we will attempt to solve this problem but we will test a few different paths for every drone in order to minimize the chances of a path being completely included in another.

There are a wide range of approaches that can be used to solve this motion planning problem and they all have their advantages and disadvantages. All of these approaches can be classified as centralized or decentralized. In the centralized model, we have a central computer calculating the trajectories and then sharing all of the robot's positions at every time to all of them. Then we have the decentralized approach, were each robot makes a decision at every point in time. From a purely optimization point of view, the centralized approach is generally preferred as it tends to lead to better results. For instance, two robots interacting in a decentralized model may easily fall victim to a Nash equilibrium, meaning that if they are both presented with the choice of one optimal solution and one less optimal one, they will both tend to take the most optimal solution for them which will result in a worst outcome overall; whereas if they had both taken the worst initial decision the general output could have been better. But at the cost of this extra precision, centralized algorithms are a lot heavier and hence require a lot more power to run and take more time. In exchange for this precision, decentralized models give the system a lot more flexibility and robustness. Therefore a trade-off must be made depending on the application.

Some companies have attempted to unite the best of both worlds. Exotec Solutions, based in France, offers logistical solutions for warehouses where a swarm of autonomous robots retrieve, move around, and deliver crates around the shelves. The have a master computer which handles most of the coordination of the many different robots and then each individual robot makes the final adjustments to the coordination. This gives the robots a high level of autonomy and also provides a lot more robustness to the entire system as every robot is not blindly depending on the master computer. This however means that every robot needs a better integrated computer, which demands more power so a bigger battery, increasing both size and weight. All companies addressing this problem are forced to make this decision on size and this often dictates the rest. Before making our choice we will first investigate the different algorithms before choosing one which suites our needs

1.2 Different algorithms

There are many types of algorithms that could come to mind for controlling swarms on autonomously guided vehicles and the first of which are completely decentralized. This is because when thinking of autonomous vehicles, they will first and foremost think of cars, such as Tesla, which are completely independent. In other words, each car figures out its path and speed individually, there is no hidden greater power controlling them. This is done through the acquisition of the surrounding area with an array of sensors, which may be radars, lidars, cameras and so one, and then make and individual assessment of the situation. The "old fashion" way of solving the motion problem in this way would be to define a set of rules and have a basic algorithm follow those rules in order to achieve the goal.

However, this technique is falling out of favor to the advantage of more modern machine learning approaches. These approaches train an artificial intelligence on the robot with the given parameters, such as sensors and range of motions, and figures out its own rules. The rules that it finds are therefore tailored to the specific application and application parameters, making them considerably more "intuitive" for the robot, resulting in a better performance. These algorithms may also turn out to be lighter in complex spaces with many sensors and data acquisition channels. On the other hand, this approach requires a prolonged access to a powerful computer and a massive amount of training data in order to train the robot to interact with the environment properly. Hence this method is easily scalable as long as the parameters don't change and that the training phase has been completed on this specific set of parameters. Nevertheless, in recent years end to end machine learning models, which take care of everything from perception to control have turned out to be not as promising as the hype around them had lead us to believe. This is why they are rather being used for perception to prediction tasks while classical algorithms still run in the background for path-planing and control.

Before moving onto centralized approaches, we can just note that some research has already been made in algorithms which are partly centralized and partly decentralized. For instance, in aeronautics, these type of algorithms are the most suited to the task as there is a global path-finding for all the planes but there is always an on-board detect-and-avoid system which turns on if ever two aircrafts come to close to each other or are about to collide. This emergency system will then make local adjustments to the paths, as small as possible, to avoid a catastrophe.

In the centralized approach, it is often assumed that the central computer has knowledge of all the surrounding environment and robots. This is because if it does not have this omniscient view there is no point in having it and a decentralized algorithm would be preferable. This environment can be either represented as a continuous space or broken down with sampling into a discrete representation of it. Down-sampling a space decreases its size and complexity and therefore considerably increases performance. On the other hand, down-sampling reduces the models precision and may block routs that existed in the original continuous space. For instance, say we have two obstacle separated by a distance L and that we sample our space with a sample size of L. If the grid aligns well we will get two obstacles separated by one free space. If the grid is offset by $\frac{L}{2}$ then we will have a solid row of obstacles. Now say we have this configuration twice, offset by $\frac{L}{2}$ relative to each other. Now at least one of the passages will be blocked, regardless of the offset of the sampling grid.

Once the space is sampled we can now solve our coordination problem on the discrete space. The most wide-spread method used for the coordination of multiple robots in a sample space is using coordination spaces. We will see later in detail how these spaces are constructed but the general idea is to find a path for every robot and then generate a space from all of these paths where intersections become obstacle and to find a new path to navigate this space. We then reverse engineer this coordination path to deduce which robot needs to move, and to get a list of small objectives, each associated to a specific time.

After much reflection, it was decided that we would work with Crazyflie drones, going for a centralized approach as these drones have virtually no computing power and tiny batteries. We will therefore do all the path and movement calculations beforehand on a powerful computer before deploying the drones. We will therefore do world sampling and then generates paths and a coordination space on this world. Is was also chosen to go for fixed path and variable speeds as this aligned better with the method requested by the laboratory. We will now look in details in the chosen environment.

2 The working environment

2.1 The Crazyflie

First and foremost, there first was the question of which robot would we use for testing. This was important as the robot capabilities would determine our approach and we would, in a way, tailor it to the platform it would need to run on. In the lab, we could either use Turtlebots, a wheeled rather slow-moving vehicle, or Crazyflies, tiny quadcopters. The decision was made to go with the Crazyflies as the additional degree of freedom meant that the results could be more interesting and that being quadcopters, they are omni-directional. This means that no matter where they are facing they can move with as much speed and liberty in any direction around, above, or below them. These drones measure about 3cm by 3cm and this small form factor allows us to make complicated contraptions within the confines of a single, standard sized room. This also makes them very safe for the operator and relatively resistant. As illustrated in figure 2 these drones are very simple with all the parts being interchangeable, allowing them to be easily serviceable.

2.2 The operating system

The Crazyflie drones are wonderful little creatures but one must still be able to operate them from our central workstation. We also wanted to try using a software as flexible as possible so that if complications should arise from the drones we could easily adapt and purpose the code for the available Turtlebot. The Crazyflies being open source, we could easily adapt to what we wanted. After much reflection, we decided to go with the Robot Operating System, ROS. The Robot Operating System is an industry standard software framework for robotics related software development.

ROS works on a node model. Each machine, whether it be a computer or robots, can have one or

Figure 2: Photo of a Crazyflie.
From: https://www.bitcraze.io/

more nodes. These nodes can have two main features: they can be publishers and publish data to make it available to other nodes, or they can be subscribers and listen to the publisher nodes to use their data as input. This construction allows us to have our main computer do all the calculations and publish them with one publisher node per drone. The computer publishes the drone's objective position and the subscriber node of the drone receives it and passes it to its' operating system which then deduces how to move. This is possible thanks to the drones constantly knowing there position and orientation. We will see how this is possible shortly.

Finally, ROS way be used with both Python 3 and C++. In order to have access to higher level functions and to concentrate on the motion planning algorithms, the choice to use Python 3 was made. This is also possible as being on a workstation, we were able to run python. If this where to run directly on the drones we would have had to use the C++ version.

2.3 The physical space

The room where we conducted our experiments had also been equipped to do so. Eight radio emitters where placed at the corners of the room in such a way that each drone can pick up on these signals and triangulate there position. This allows us to keep a good control of the drones as they always knew where they were in space. These beacons formed what is known as the Loco positioning system. The area in which the Crazyflies could evolve was a rectangle 3 meters wide, 6 deep and slightly under 3 high. This gave the Crazyflies plenty of space to evolve freely. The only downside of this method is that, relative to the drone size, it has a rather low accuracy of 10cm. When comparing that with a drone that is 3cm across, this is a significant uncertainty.

However, we would not test the algorithms directly on the real drones as they are still not the easiest to use and have limited battery. We would therefore run the algorithms in a simulated environment before attempting any real world experiments. We therefore created a model of our room in the software Gazebo which was designed to work the ROS system which we decided to use. We could therefore test all of our advancements and design changes rapidly anytime and anywhere. After seeing the amount of collisions and crashes suffered on the software is was also a wise choice to start there. An image of our virtual yet identical test room may be seen on figure 3. We can clearly see the eight white Loco positioning beacons in every corner and four Crazyflies in the center.

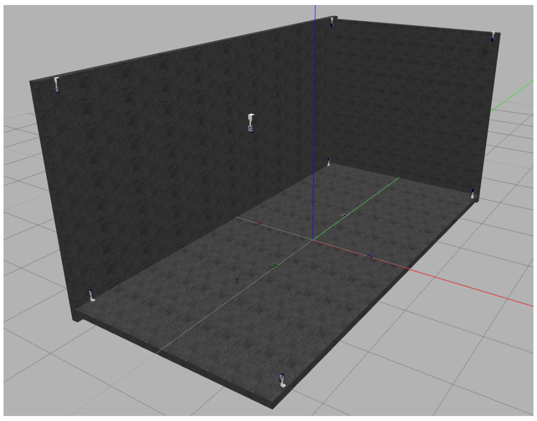


Figure 3: Simulation of the empty test room.

3 Motion planning

We will now discuss how we generate the optimal paths. But first we need to discuss which parameter we choose to observe and how we choose to maximize or minimize said parameter. This is a crucial step as changing the optimal parameter we may drastically change the result. For instance, in we wish to go from Paris to Lausanne by car, we have two options: we can get there in just under 6 hours driving

600km or we can get there in above 9 hours but only driving 500km, giving a 50% increase in travel time and a 20% reduction in distance. This is why deciding which parameter to optimize is key. In our case we choose to optimize the time it takes for our drone and so go for the quickest path. As we are working in a homogeneous environment with no slower or faster passages do to outside influence such as wind, this is analogous to taking the shortest path.

However, we do not only want one drone to achieve its' goal the fastest and then leave all other drones fend for them-self. We will therefore consider the best situation for this group of drones, which might not be the best situation for any individual drone. In order to do this we will start the time when the first drone moves and stop it when the last drone arrives at its destination. As we have control of every single one of the drones speed and path we will incorporate all of them together in order to minimize this time. This is known as the Pareto optimal solution for this problem, no drone can be made better off without making the entire group worse off.

Finding this optimal scenario is done in multiple steps. We will go over each step, reviewing each of them one at the time with the improvements as well as the optimizations that were made to it. We will first see how we handle the environment, then move onto the handling of each individual drone followed by the group coordination and finally executing the solution.

3.1 World generation and storage

The first thing to do when we want to find the movement of robots in an environment is to load said environment. The map is stored as a 3 dimensional list which goes as follows: assuming we sampled our space as 4 values for the X axis, 10 values for the Y axis and 2 values for the Z axis, we will store is as a 2 by $10 + 2$ by $4 + 2$ array. The reason for the extra 2 values along X and Y is to create a bounding box around the drones to avoid them hitting the borders of the space, meaning the walls of the room. It should be noted that the $(0, 0, 0)$ point for the drones is in the center of the room at ground level. This is why for Z there is no extra layer, it was just offset off the ground.

To easily move back and forth between different maps, while easily differentiating them from other files, a file format, .map, was created. These files encode the map in an intuitive way to make it easy to write and read. In this format, we write in Unicode text the different planes one at the time. A 0 represents a free space while a 1 is an obstacle. Each X value, in a same row, is to be separated by a comma while after each Y value, a line return is sufficient. We save all the XY planes from the bottom to the top, separating each one of them with a blank line. It is important to note that every XY plane must be surrounded by a ring of ones for the reason mentioned above. The $(0, 0, 0)$ point is therefore the third character of the second line.

The ring of ones around each XY plane is also crucial to an alternative approach as, while they are invisible, they block the drones from leaving the virtual room. This different approach will be demonstrated at the end of this paper for comparison purposes.

In order to generate a virtual environment matching these characteristics, we will load the map with the 'genMap.py' program and this script will take the map, and add all the obstacle to the 'salle_lix.world' file which is the environment descriptor file that Gazebo loads at its' launch. The program will however refrain from adding the rings of ones around the XY planes. Each obstacle will be plainly represented by a cube the size of the box centered on the coordinates. A set of scale constants are also calculated in order to rapidly go from an array position to the world coordinate system.

For demonstration purposes, assume we have the following *test.map* file on the left. This is what will be loaded as the map but only then simplified version on the right will be loaded for the display.

```
1  1  1  1  1
1  1  0  0  1          []  .   .
1  1  0  1  1          []  .   []
1  0  0  1  1          .   .   []
1  1  1  1  1
```

3.2 Individual drones

Now that we have a map in which the drones can evolve, we need to find the path that each drone can take. As we are going with the fixed path and variable speed method, we need to have a way to find a path from the drone's starting position to it's' goal. This is a basic path finding problem. To solve this path finding problem we implement an A-star search algorithm. This algorithm was chosen as it is a wide spread standard path finding algorithm which, if the correct functions are chose, can provide the optimal path every time.

A-star, often referred to simply as A*, is a widely used path finding algorithm. It is an extension to Dijkstra's original path finding algorithm. Both of these algorithms where made to work over graphs. Our three dimensional grid can also be interpreted as a graph where we take the movement from any square to the neighboring one as a movement of cost 1. This allows us to apply the previous algorithms.

Let us begin by analyzing Dijkstra's algorithm. The general idea of this path finding algorithm is to find the shortest path from our original point to the goal. To do that we explore all the points around the source point and move away from it in circular motions. This means that we will really cover the entirety of the space with this method. Hence, if there where multiple possible objective points it would be of great value. However, in our case we just have a single goal point with known location. Therefore exploring all possible points around the original point is a waste of time. Dijkstra's algorithm always explores the path that is the closest to the origin, in other words is minimizes $f(n) = g(n)$ where n is the next node, f is the total weight of a node and g is its' distance from the start node.

In order to avoid so much useless exploration, we implemented an A-star algorithm. It works in exactly the same way except the cost function where an extra term is added: $f(n) = g(n) + h(n)$. This h term is a heuristic which estimates the distance to the end node. This change means that the algorithm will priorities nodes that are in the general direction of the end node. This results in a much more direct and targeted path search as we do not waste time looking away from the objective. In the worst case, if to find a path we need to first go back before going towards the objective, or the heuristic function was not chosen properly, it will take as much time as Dijkstra's algorithm to find a solution. This is why choosing the correct heuristic is crucial.

As we are working on a grid, the first heuristic that comes to mind is the Euclidean distance. However, as we are constrained by a grid, we cannot go to a square 30 degrees lower. On top of this, for the drone path finding, they are constrained to moving in a single one of these directions: left, right, forwards, backwards, up or finally down. The diagonals where removed as by taking a diagonal they would impede on an obstacle's space, crashing into it. The most accurate heuristic was realized to be the L_1 norm. This norm is also fittingly called the Manhattan distance as it is as if we are walking around city blocks in New York. Its' exact definition in an n dimensional grid between A and B is as follows:

$$L_1 = \sum_{i=1}^{n} |A_i - B_i|$$

Sometimes, we wanted to be able to explore multiple paths to the goal to have different possibilities later on in the coordination step. If we continues running the algorithm in the same loop we would get a tree exploration of possibilities near the end. This is because before exploring a node it checks that it did not already reach it in a better way. So if we wanted the algorithm to explore different paths from the start or to be able to give more than $2 * n$ paths, as we can achieve the goal from either side in each of the n dimensions of the grid, we needed to find a better way of inciting the program to explore multiple paths. To solve this issues we introduced penalties. Each time the algorithm finds a path from the start to the goal, it clears everything, the node queue, and the nodes costs and already explored nodes. It then proceeds to rerun the A-star algorithm except that the scoring function is slightly modified:

$$f(n) = g(n) + h(n) + b(n) * p$$

The new term b is a bonus term which counts the amount of times the node n has already been used and multiplies that by the penalty factor. This is the final cost function which is used. Modifying the value of this term will have very interesting consequences. For instance, is the penalty term is of 1 and the first path followed a diagonal going first right then above, it will now go first above then right. If the penalty is at 2, it will offset its diagonal by 1, so the first move will be to go above and then repeat the previous path. This is assuming there are no obstacles blocking this behavior. In the following image (figure 4) we can see an illustration of this. In green we have the start and in red the end. The 2 different paths are marked with different shades of blue and when they overlap it is marked in grey. From left to right we have $p = 0$, $p = 1$ and $p = 2$.

But depending on the weight of the penalty and the size of the obstacles this can incite the drone to take a completely different path around them. This diversity is exactly what we want as it will minimize

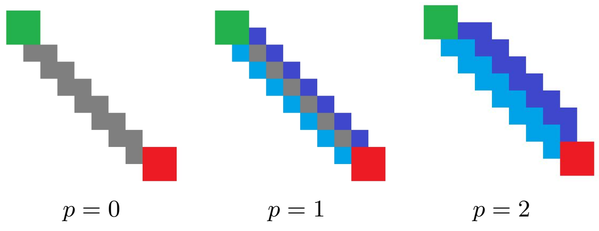


Figure 4: Illustration of the impact of p on the alternative path.

the chances of not being able to solve the configuration problem later on. It is import to note that this method does not allow a drone to deviate from its path for less than 3 blocks. This is because he will still get a penalty when returning to the original path and so making a side step for 1 or 2 blocks would give a total path cost of 2 extra compared to the path from which it side-stepped.

From a coding point of view, a *Drone* class was implemented in *drone3D.py* and each drone we wish to control is just an instance of said class. The A-star was implemented in such a way to be n dimensional compatible and can be found in *nDimAstar.py*. The n dimensional A-star algorithm is very important as it allows us to reuse it in coordination spaces of any size. For drones we run the algorithm with $checkDiag = False$ as they are not allowed to move diagonally and $projections = False$ as we use a complete grid representation of the environment in an n dimensional array.

3.3 The coordination of 2 drones

Let us first assume that we only have two drones and that we found a single possible path for each drone. Each drone will also only move on an XY plane for now. We will use this to illustrate the basic functioning of the algorithms and then increase their complexity by relaxing these conditions and eventually removing them all together.

First and foremost, let us introduce a setting which will simplify visualization and explanation. Here under is a map of an area. S_i represents the start position of drone i and G_i its' goal. When marking the drones we will refer to them as D_i. In figure 5 we can see both drones with the path that they intent on following in their respective color. Let P_i the path that drone i will take. Each path is just the list of points the drone needs to go through. As we view these as arrays, $(0,0)$ is in the top left corner.

$$map = \begin{matrix} S_1 & 1 & 0 & S_2 \\ 0 & 0 & 0 & G_1 \\ G_2 & 0 & 0 & 1 \end{matrix}$$

$$\begin{aligned} P_1 &= \{(0,0),(0,1),(1,1),(2,1),(3,1)\} \\ P_2 &= \{(3,0),(2,0),(2,1),(1,1),(1,2),(0,2)\} \end{aligned}$$

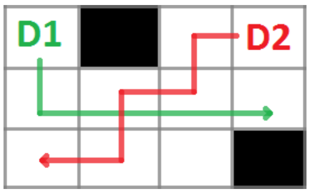


Figure 5: Two drones with their respective paths.

We will now build a coordination space from these paths. But first we shall explain what a coordination space is. A coordination space is a space created by the cross product the paths of each drone, excluding some obstacles. These obstacles in question are points were at least two paths have the same point in space. This is because two drones cannot superpose each other on the same tile, it would result in a crash. Formally, the coordination space C is defined as followed, with O the set of obstacles:

$$\begin{aligned} X &= P_1 \times P_2 \times ... \times P_n \\ O &= \{p \in X \\ &\qquad \text{s.t. } \exists i,j \in p \text{ and } \exists a,b \in [\![1,n]\!] \\ &\qquad \text{s.t. } a \neq b \text{ and } P_a[i] = P_b[j]\} \\ C &= X - O \end{aligned}$$

In other words, it can be seen as just aligning the paths to the axis of a graph and crossing out where they meet. An example of this can be seen on figure 6.

As both drones are allowed to move simultaneously, we are allowed to move diagonally within this coordination space. The goal is to start at the 0 coordinate, all drones at their starting position, and reach the maximum in every dimension, the top right corner in our 2 dimension example. A few different methods have been implemented to navigate this space. As we have previously implemented an A-star, we can first start by reusing this. This gives us a baseline for other solutions. We then implemented an exact left

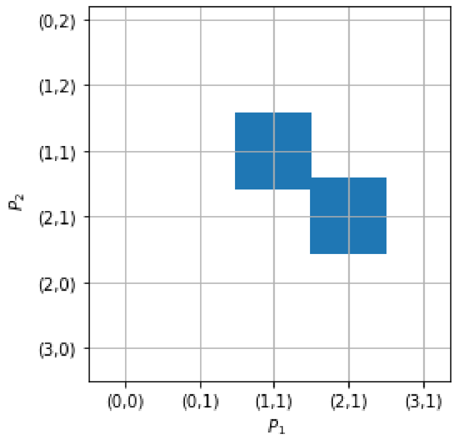


Figure 6: Coordination space of P_1 and P_2

greedy monotone algorithm based on Ghrist, O'Kane, and LaValle their paper on "Computing Pareto Optimal Coordinations on Roadmaps" [1].

A left greedy algorithm is a simple and intuitive procedure in which the algorithm will always take the best option available to achieve its' goal. In this case, a left greedy algorithm will try to maximize the amount of drones that move at every iteration. It will therefor move as diagonally as possible. A big problem with this method is that it will tend to get stuck very easily. This is why the left greedy algorithm described by Ghrist and Lavale, works to optimize a monotones path. When we have any monotonous path navigating a space, we can use it to bound the obstacles in order to guide the left greedy algorithm around them in the proper way. Any basic and sub optimal algorithm could work for this step as long as the result is monotone. But as we had an A-star available we would just use it and make sure the path it gave was monotone after. Once we have this path, we need to add was is called a "critical event". In the same way as a black hole event horizons, the left greedy algorithm will be allowed to get as close as it wants to the critical event without never passing it. These critical events guide the algorithm so that it stays in a monotonous space. Therefore is A-star went above the first obstacle and below the second one, so will the left greedy algorithm.

Adding these critical events poses bit of a challenge initially to optimize the time it takes. In order to do that, we transverse the space in reverse order. We are here going to explain it in 2 dimensions as we will see later that this is all that we do, adding critical events to 2 dimensional grids. We start in the corner which we are trying to reach, so the maximum X and maximum Y. We then iterate backwards on every row one at the time, adding the critical events as we go. Every obstacle which is below the initial path given by the A-star has its' left most boundary extended all the way to the bottom, and in the same way any obstacle to the left has its' bottom boundary extended to the left most side of the graph. These boundaries ensure that the left greedy passes on the correct side to not get stuck. When we iterate over a line, if we are before the reference path and we encounter an obstacle we add a wall under and if we are after the path we add a wall to the left. As we iterate top to bottom and right to left we will add an entire line extending to the bounder of the grid. An example of this process can be seen on the figure 7 with the red blocks being added by the critical event generation done here. Once the critical events are set a path can be generated, noted LG on the figure 7, based on the $A*$ path in black.

This left greedy was later adapted into a more powerful and robust algorithm. This left greedy 2.0 algorithm is a more powerful version which does not rely on these critical events. This algorithm is still greedy but if it gets stuck in some place it will add an obstacle on the last place it was and will back-step by one. This sequence of execution is then repeated until the objective is achieved. This is a lot more robust as it is able to navigate around obstacles and solve a non-monotonous problem. It has the notable advantage of avoiding a corner case that would occur with non-monotonous original paths: critical events would end up blocking the path that the drone needed to follow. This left greedy algorithm is what we will use from now on to create paths.

If we look on at the figure 6 we realize a big issue of most of the graphs: it is sometimes possible to squeeze between two obstacles diagonally. Because even if they appear blocked on the graph the algorithm can still pass between them. This would be analogous to the two drones exchanging places, teleporting through one another. In real life, this would result in the two drones colliding in each-other. To avoid this we must do an inferior convexification of the space. Any diagonal configuration of obstacles with a downwards slop must be blocked with an extra layer of blocks bellow it to block any diagonal path passing through two obstacles. In other words, if the square (i, j) is free and the squares $(i + 1, j)$ and $(i, j + 1)$ are blocked by obstacles then an obstacle must be added on the square (i, j). An illustration of this can be seen with the green square in figure 7.

However, it was possible for there to be a wall formed from the top of the grid to almost the bottom of it, creating strange boundary conditions when the left greedy algorithm was ran. It would trap itself in the top part and would not be able to proceed. To solve this issue, an artificial wall of obstacles is added to the top and right most side. So conceptually it is the same as if we had an extra obstacle at every $max(X)+1$ and at every $max(Y)+1$. This also blocks out considerable portions of the map if this case was to occur, making the path-finding algorithms a lot faster. A lot of this downward convexification was done with the help of a paper detailing precisely this, how to avoid deadlocks in multi robot environments, reference [2]. All of this gives a space which has local non positive curvature, which is necessary for the left greedy algorithm.

The following graphic, figure 7, shows a summary of all that is done for these 2 drones interacting drones and their final configuration space. We then progress along the selected optimal path, and moving each drone to the corresponding coordinate every time. So when we go from the second to the third point of the left greedy path, from $(1, 1)$ to $(1, 2)$, the first drone will stay at $(0, 1)$ and the second one will move to $(2, 1)$.

It is also important to note that the shortest path in the coordination space is the Pareto optimal path for the system. As we can also see on figure 7 that it is possible for there to be multiple shortest paths. All of these paths are said to be Pareto equivalent. Any paths that are Pareto equivalent are of the same lengths and are therefore minimal paths to traverse the space with the constraints given above. The fact that the left greedy path across the grid is both unique and a Pareto optimal path is given in the article from Ghrist and La Valle [1].

One of the most complicated procedures to get right was to have drones side step to leave another one pass. This caused many troubles and only after having implemented all of these upgrades was it able to solve it. We can see an example of this in the images of figure 8. On the first image we have both drones in their starting position (image 1). They then move towards each other (image 2) before having the red drone move aside in order for the blue to pass

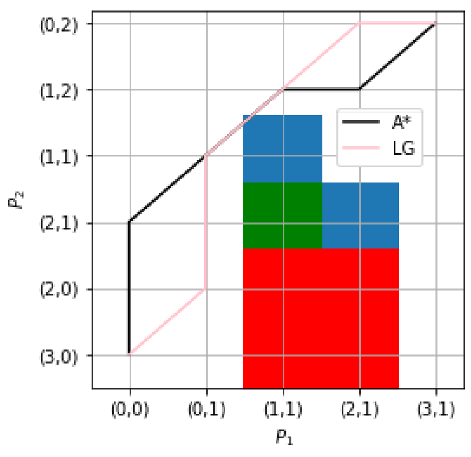


Figure 7: Coordination space of P_1 and P_2
Blue: obstacles
Green: obstacles added upon convixification
Red: obstacles added to represent critical events

(image 3). What is not seen here is that the red does not stop, it continues advancing towards the objective on a parallel route. They then finish their trajectories (image 4), having exchanged places. As the ted had to shift out of the way it had to make 2 extra moves to arrive at its goal and so the blue waited. This is the cost of moving aside which is necessary in this case. Therefore this is indeed the Pareto optimal solution.

Finally, let us bring our 2 dimensional robots into 3 dimensions. This changes absolutely nothing. The extra coordinate just needs to be added to the start and end points. The A-star algorithm for the initial path finding will adapt automatically. The coordination spaces work in the exact same way but as the paths are more complex the configuration space will be bigger. This also grants us more liberty is designing an interesting environment for the drones to evolve in.

3.4 The coordination of n drones

Now that our drones can evolve in 3 dimensions we wish to be able to deploy as many drones as we wish. We will now assume that we are deploying n drones in our space with n greater than 2. We will now see how this impacted the coordination space and how we decided to handle it.

First and foremost, we generate n drones sequen-

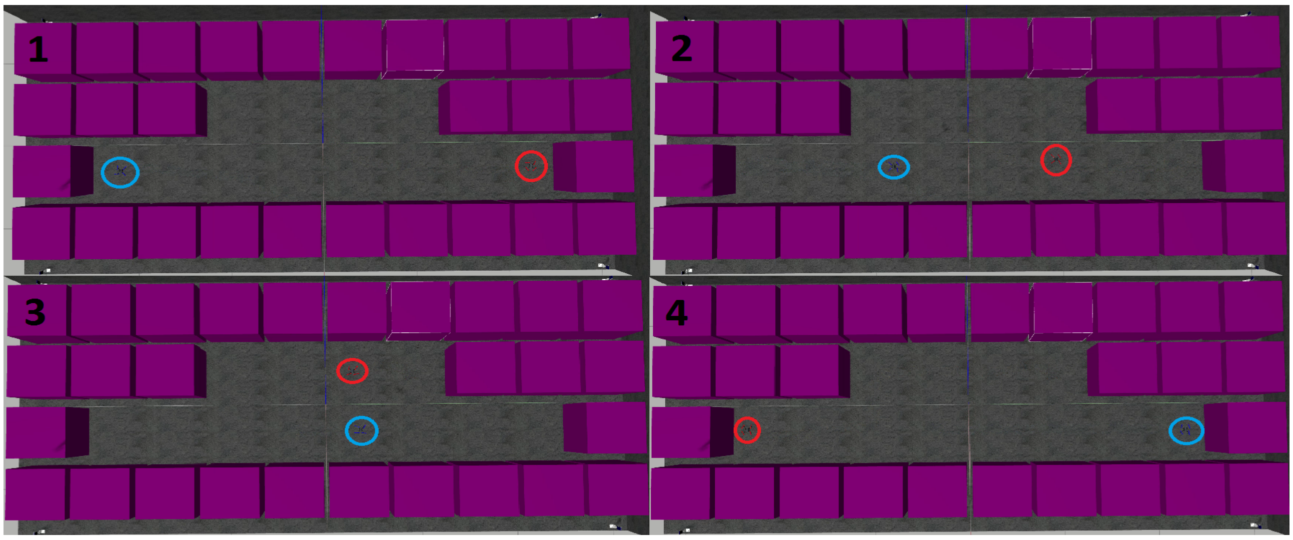


Figure 8: Coordination of 2 drones to cross each other.

tially. Upon generation, each drone will establish a path from its starting point to its end point. Defining the coordination space as before, it would be an n dimensional grid as represented here under. We define $|.|$ as our norm. For a path, this norm would be its length, which is equivalent to the amount of steps the drone will make plus one for the starting position. Each position is stored in the form of a tuple of the shape (x, y, z). This will therefore make up one unit in the following measure as they will all be of same length. As we work in space that are relatively small, we do not need to worry about overflowing x, y or z as they are unsigned integers less than $2^6 4$. In Python, it is not specified if the integers are signed or not but regardless our tables are small enough. In other words, the sampling rate can still be increased by a substantial amount before this causes a problem. We will call the norm of the path $|P_i| = p_i$ to make notations simpler. For the following calculations we will use the average path length in order to be able to make more elaborate comparisons. This slight lack of precision is compensated by the fact that it is negligible with regards to the objects we will be manipulating and that normally the path lengths should not have to great of a variance.

$$\bar{p} = \frac{1}{n}\sum_{i=1}^{n}|P_i| = \frac{1}{n}\sum_{i=1}^{n}p_i$$

The norm $|.|$ can also be extended to be used for the coordination spaces. For spaces, we will count the amount of spaces is has, hence the amount of elements. This measure will ignore addition space needed by python to store a list but will allow us to get a solid grasp on the theoretical size needed. When having this extra term could mater, we will add a ϵ to our calculations. However, the exact size of this term cannot be approximated as Python overallocates memory resources to lists to not have to constantly extend them. The total size of the space is therefore obtained as followed. As the coordination space is the product of the paths, makes sense that from the definition of the norm, that it is the area if $n = 2$ and the volume if $n = 3$ of the rectangle with sides of length p_i. Extending this realization, the norm of the coordination space is therefore the product of the norm of the individual paths.

$$X = P_1 \times P_2 \times ... \times P_n$$

$$|X| = p_1 \dot{p}_2 \dot{:} .. \dot{p}_n = \Pi_{i=1}^{n} p_i = \bar{p}^n$$

From this calculation, we realize that the size of the space grows exponentially with the amount of drones that we have in the system. This is not acceptable as this array needs to be stored in memory and exponential memory usage is extremely bad practice. However, implementing an n dimensional list is the fastest implementation and the simplest, but just as fast as it was to implement, when the paths were a bit complex, hence a rather long, and that we would add just a couple drones, it would crash immediately. Assuming the average path length does not change, just going from two to four drones results in a squaring of the size of the space, leading the program crashing.

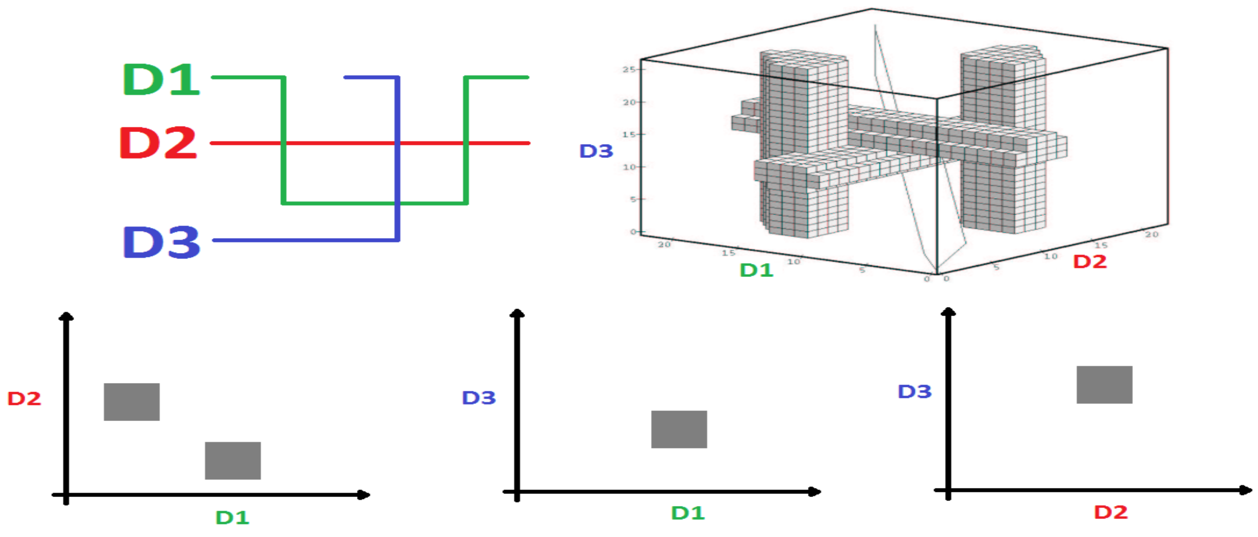
Even when it was possible to calculate the coordination space as it was small enough, it would take an outstanding amount of time. This is because, in order to build this space we needed to go through every point of the space and at every space compare every position of where is every drone to every other position of the other drones. From the moment two of these positions would match an obstacle needed to be added at that point. Initially this would mean that we needed to check each of the n drone positions to the $n - 1$ other positions in order to determine if there was an obstacle. By realizing that comparing the i-th point of the path of drone A to the j-th point of drone B was identical to comparing the j-th point of drone B to the i-th point of the path of drone A by reflectively of the equality, we can avoid half of the checks and hence go from $n(n-1)$ checks to $\frac{1}{2}n(n-1)$. However, even with this improvement when the coordination space could be calculated it would take extremely long.

After seeing the problems brought along by the exponential growth, another solution needed to be found. Diving back into research articles, rereading an Article from LaValle on optimal motion planning for individual robots, reference [3], helped me realize that the n dimensional coordination space was actually constructed by a set of projections of 2 dimensional planes in the n-th dimension. This is due to the fact that if the i-th point of the path of drone A to the j-th point of drone B are identical, then for all the possible positions of the other drones they will remain identical and hence obstacles. This meant that the obstacles we generated where cylindrical by construction. We could therefore abuse this construction method to reduce considerably the amount of information that needed to be stored and rebuild the n dimensional coordination space locally when necessary. The hope of this method being to avoid an exponential growth of the coordination space with respect to the amount of drones or path complexity.

Let us begin with a practical example explaining how this works. It is easiest, as it is the only one were we can fully represent the complete coordination space, to first view this technique in three dimensions. If we look at figure 9, we have the drone paths in real space in a plane in the top left corner. This results in three dimensional coordination space which is located in the top right corner as there are three drones. In order to build this coordination space the old way, by brute force, it would take up a space of approximately $\bar{p}^3$ space in memory and would require doing $\bar{p}^3\frac{1}{2}3(3-1)$ comparisons, which simplifies as $3\bar{p}^3$. As we realized in the paragraph above, when constructing the coordination space, if two drones share a common position at some point they share it all through the other dimensions of the space. We can therefore create a list of pairwise interactions and project these through the other dimensions of the coordination space. In figure 9 we have detailed the separate pairwise interactions between all the possible drones. Again we use the fact the the interactions between drone A and Drone B are the same as if we considered the drones in reverse order to avoid building unnecessary graphs. In the case of 3 drones, we therefore have the interactions between three different pairs of drones: drones 1 and 2, drones 1 and 3 and finally between drones 2 and 3. We then take all of these planes and combine them into a complete coordination space with one drone on every axis. With one drone on every axis we know have a shell of the coordination space we need to generate. We then project all of these planes over the remaining axis to generate a complete space. It is important to note here that the pairwise interactions are sufficient and again it is sufficient to project them through the other dimensions to generate the entire coordination space.

Using this projection method results in a substantial decrease in the amount of comparisons needed to generate a complete coordination space. As we simply do a single bi-dimensional comparison for every point in each of the two dimensional coordination spaces between two drones, we get a total of $\bar{p}^2$ comparisons. We know times this by the amount of projections we have, with as we saw above is 3, and we get a total of $3\bar{p}^2$ calculations to populate our entire coordination space as opposed to the $3\bar{p}^3$ we had before.

This may not seem like that much of an increase but the real difference comes when we increase the amount of drones and hence the dimension of the coordination space. The general formula for the amount of projections needed to fully describe the coordination space is the amount of two dimensional coordination spaces between two drones which can be deduced by basic combinatorics. We start with n drones which



3 dimensional graph from paper [3]

Figure 9: Coordination of 3 drones to cross each other.

can all interact with $n-1$ drones as they cannot interact with themselves, giving a total of $n(n-1)$ interactions. Ignoring symmetry we finally get $\frac{1}{2}n(n-1)$ possible interactions.

Therefore, if we wish to populate a n dimensional coordination space through brute force, it would take:

$$\bar{p}^n\frac{1}{2}n(n-1) = \frac{n(n-1)\bar{p}^n}{2}$$

However, calculating the projections and using those to populate the coordination space requires the calculation of:

$$\frac{1}{2}n(n-1)\bar{p}^2 = \frac{n(n-1)\bar{p}^2}{2}$$

We Can see that indeed in areas with a low amount of drones the difference is minimal but that as their amount, n, increases, the amount of calculations increases exponentially with the brute force white increasing quadratically with the projections method. In the same way, an increase of the average path length in the brute force case leads to an exponential increase of the amount of comparisons while this increase is only quadratic with the projection method. This method therefore proves its worth as a valuable tool to scale the size of the coordination space.

Using two dimensional projections also means that we do most of the preprocessing rapidly and simply reusing the algorithms in the previous subsection when we were experimenting with coordination spaces of two drones. This initial work involves waking the projections lower convex and if we are using the left greedy path finding algorithm from LaValle, described in reference [1], as we have a monotonous reference path we can add the critical events. Ass above, it is important to ensure that the reference path is monotone in all of the n dimensions otherwise it is possible that this technique will not work.

Nevertheless, we may now compute our coordination spaces rapidly but we still did not find a way to store them without crushing our memory with the huge space. We realized just above that the projections were sufficient to store all of the necessary data to reconstruct the space. Therefore, if we modify our algorithms to go lookup the information they need with these projections we would just need to store them. Assuming we make a list of projections, each projection would be of a size of p^2 and there would be a total amount of $\frac{1}{2}n(n-1)$ of them as shown above.

Therefore the amount of memory needed to store all the necessary information would be of:

$$\bar{p}^n$$

for a complete coordination space and of:

$$\frac{\bar{p}^2}{2}n(n-1)$$

for a list of the projections. The amount of lists created is not taken into account here but is much greater when we generate a complete coordination space so just goes to reinforce the following point. Again, we see that the complete space is exponential in both the amount of drones and the path the average path length, while the projections method is quadratic.

We can therefore deduce that the using projections solves all of our issues, in both storing the coordination space and populating it. However, all of our algorithms where designed in such a way that they use n dimensional maps. We therefore need a function to access the data stored at a point x, an n dimensional point. This is mainly to know if there is an obstacle there or not, and hence if we can move there.

From an implementation point of view, we store all the projections in the following ordered list: we store the interactions of the path of drone A, on the X axis, with the path of drone B, on the Y axis, with A<B. This condition on A and B is of the utmost importance to be able to find the interactions with between two paths. Say we wish to know if there is an obstacle blocking the coordinate x. We therefore need to check all of the projections to see if some are in conflict. Thanks to the ordering we can check this in the following way. This example is a Python function using basic string comprehension as it is just a whole lot clearer than attempting to explain it. The idea is that it iterates over the different projections and if it finds just one where there is an obstacle it will return *True* but otherwise return *False*.

```
def checkWall(configSpaces, point):
    n = len(point)
    for i, p1 in enumerate(point):
        for j, p2 in enumerate(point[i+1:]):
            tmp = n*i - (i*i+i)//2 + j
            current = configSpaces[tmp][(p1,p2)]
            if current == 1:
                return True
    return False
```

The only technical part of this code is the formula for *tmp*. This formula's goal is to deduce the position of the projection representing the interactions between drone i and drone $j+i+1$. This formula was established with $i < j+i+1$ so it is absolutely necessary that it remains that way. This is obvious in this loop by definition but can be more ambiguous in other uses. Here *configSpaces* represents the list of projections and the formula used to access the correct one, the *tmp* variable was obtained thanks to some combinatorics work. This was done by realizing that for $i = 0$, we had $n-1$ possibilities, for $i = 1$ we have $n-2$ possibilities and so on until $i = n-1$ where we have a single possibility. The j variable counts along which of these possibilities we are. Here is a more explicit view of the formula:

$$tmp = \left(n \cdot i - \frac{i(i+1)}{2}\right) + j$$

There is however a final wrinkle with this method which needs to be ironed out. We have made sure to lower convexify each of the independent projections. Nevertheless, as we learned the hard way when testing, this does not result in the n dimensional space being lower convex. With just three projections it is possible to trap the monotone left greedy path-finding algorithm even if there is only one obstacle on each projection. An example of one of these formations may be seen in the right half of the three dimensional graph of figure 9. The formation of the three obstacles just right of the center of the horizontal axis could indeed form a spot in which a monotone left greedy algorithm may get stuck only if the tree obstacles share a common point. This will not trap the A-star algorithm. Furthermore, if the A-star gives a monotone path through the n dimensions, this formation will normally be blocked by the adding of critical events. This problem was therefore mainly a problem for the independent monotone left greedy path-finder. By the time this problem was discovered, the projection method had proven its worth as a viable method of storing the coordination space which was both light and rapid.

To solve our problem of the independent monotone left greedy path-finder getting stuck in pockets we either needed to lower convexify the entire n dimensional space, trying to find a way to make that fit in our projections or with relatively little extra storage of complexify our path-finding algorithm for it to no longer be exclusively monotone. It turns out that the solution we found is a pretty ideal solution as we found a way to make use of these pockets properties to store them inside of our projections without having to store extra data or complexify our path-finding algorithm. By definition of a pocket that can trap our independent monotone left greedy path-finding algorithm, is a point x where there is an obstacle on the next block. In other words, $x[i]$ is free but $x[i]+1$ is not. Therefore any point which fulfilled these conditions was a pocket to block. It was first thought that we could keep a separate list of these virtual obstacles as they could not simply be added to a projection. This was an inefficient way of stirring these as we were not able to fully exploit all the additional data that the projections could store. We then realized that this needed to be an obstacle that should appear in every single one of the projections.

A useful side note is that in the projections, every point is stores an integer. If this integer is 0, the space is free, 1 means that there is an obstacle and 2 is used as a marker for the A-star path when adding the critical events. We therefore chose the marker 3 to denote the presence of these pockets. The obstacle detection algorithm was reviewed in such a way that if it would find a single 3 it would ignore it but if it found one or just a few of them but would block the path-finder completely if it found one in every projection, so $\frac{1}{2}n(n-1)$ total.

The problem of how to find these pockets remained. It would be extremely computationally expensive to iterate over every point in the n dimensional space, checking every projection for obstacles around it as we saw previously. It was then considered to use a projections from a single drone interacting with all the other drones, for example the first $n-1$ projections of the list, and to attempt to restrict the search to the neighborhood of obstacles. However, research down this road did not lead to any result and was not explored to far as it was still a very expensive method. Nevertheless, when studying this method, we realized that we did not need to lower convexify the entire space. We could restrict ourselves to the pockets the path-finding algorithm will get trapped in. In order to see where the algorithm will get trapped, what better way to run the algorithm?

It then dawned on us that we could shoot two birds with one stone. On one hand we could convexify the space just enough for the path-finding algorithm to find a solution and on the other we could make this same path-finding algorithm work in environments where monotone paths did not exist. We would run the path-finding algorithm until it got stuck in a pocket and whenever that happened we would block the point where it is trapped as a virtual point-like obstacle with a 3 in every projection and backtrack the algorithm to its previous step. We would then resume the path-finding operation in this modified space. This technique worked brilliantly. It can sometimes take a multiple successive backtracks as this remains a greedy algorithm so it will not take the best path to exit the pocket in which it is stuck but will instead retreat little by little. This retreating effect has the added side effect of allowing it to continue generating a Pareto efficient path. This is opposed to implementing algorithms to exit the pocket which will keep movements that will be undone at execution time which is slightly less optimal as we count the total path length.

3.5 Exploring m paths

A final touch to the coordination space is to generate multiple paths for each drone and to test all of the paths combinations. Now that we had spaces that could be stored relatively easily and computed rapidly, we could afford to test a few combinations. Speaking of speed, it is worth noting that the independent left greedy algorithm will be the fastest algorithm if there is a monotone path through the space but otherwise A-star will usually compute a path faster. Returning to having multiple paths for each drone, if we try all the possible combinations of path we would have the following amount by combinatorics as we count the amount of possible arrangements. Let us assume that we calculate m possible paths for each of the n drones. There would therefore be a total of m^n combinations.

However optimized we make our coordination spaces, this exponential growth of the amount of path combinations to check makes it unrealized to have either a large m or a large n. We therefore needed to find a way to try minimizing the amount of path configurations for which we needed to create a coordination space. Two separate methods have been implemented to do just that. When we have a list of paths generated by the A-star algorithm, it may return multiple times the same path for a drone to navigate the space due to a highly constricted space or a low penalty for reusing the same path segments. Our first step in minimizing the amount of coordination spaces we will need to generate and check is to remove these duplicates. For each drone we will need to check each path against every other, ignoring symmetry as before we have:

$$\begin{array}{lc} \frac{1}{2}m(m-1) & \text{per drone} \\ \frac{n}{2}m(m-1) & \text{for all the drones} \end{array}$$

This series of checks is linear in n and quadratic in m so is a very reasonable price to pay to try and reduce the amount of coordination space we need to check.

The path comparison was also optimized in the best way possible where it takes $\bar{p}$ point comparisons only in the worst case scenario, if both paths are identical. If the path lengths are different it moves on to the next path combination directly and the moment it finds one movement which is different it also moves on.

Returning to the original problem, we are looking for the shortest path to cross the coordination space. We are therefore attempting to minimize the amount of movements we do in this coordination space, finding the coordination space where the shortest path is needed to transverse it. This allows us to make another massive optimization. A way to give a lower-bound to the path length needed to transverse a coordination space is to calculate the Euclidean distance from the start point, 0, to the objective point, the end of all the drone paths. As the path-finding algorithms favor the diagonal as well, in the case of a cube with no obstacles at all this path's length and the Euclidean distance will be identical but otherwise the path length will always be bigger. Therefore, when we have already calculated a path through a coordination space we have an upper-bound for the minimal path. Hence, before calculating a coordination space we will calculate the Euclidean distance from the start, 0, to the objective, lengths of the individual drone paths, and we will only calculate the coordination space if this distance is lower than the minimal path distance found until now. This provides a significant reduction in the amount of coordination spaces we need to generate and navigate. It is however hard to exactly quantify exactly how much computations this method spares us.

Empirically, we have noted that the coordination spaces of the first few drone path combinations will be checked but it will then check less and less path. Solving the problems with three drones, each with four different paths, we would have a theoretical maximum of 64 coordination spaces would need to be checked. The space in which we tested was very compact so after removing the duplicate paths we were left with about 20 paths to check and in those we finally only checked about 6 paths. These optimizations therefore allowed us to avoid generating and navigating over 90% of the theoretical amount we would need to check. This just proves how crucial and useful these optimizations are.

4 Executing the solution

Finally, ounce we have the path that we choose to follow in our coordination space we back propagate this to have a sequence of move that every drone needs to follow. We that open a ROS publisher node for each drone to connect to and make our drones take off, whether it is in a simulation or in reality. We then iterate over every point of the paths, waiting each time for all the drones to have reached their location before moving onto their next objective. This method has proven to be very successful as after adjusting the distance a drone could be from a way-point and still being counted as being there, they evolve fluidly through the space and we do not see any notable jerks in the movements of the individual drones.

4.1 Simulations

Many simulations were ran and in this part we will present them along with their interesting points. They give us a peak into how different parameters affect our algorithms. All of the videos where numbered in order of use to ease following the explanations.

To begin, say that we have two drones and that we wish for them to exchange cases, this would give a coordination graph as the one seen in figure 6 with two obstacles barely touching at the tip. Now if this is left as is we get the result seen in simulation [1] where the two drones try to exchange places but end up colliding with each other and crashing. We therefore lower convexify the space as seen in figure 7. But if we only do this the algorithm will resort to telling us that no possible path was found without the drones crashing. Modifying the A-star path-finding algorithm to find multiple paths and adding penalties when it reuses the same blocks allows it to find a different solutions for each drone and a combination of movements which allows the drones to cross without crashing. This can be witnessed in the video of the simulation [2]. A useful note is that in dark grey we have the boundaries of the room and in magenta we have the obstacles to avoid. The bottom and left barrier are missing to be able to see the drones evolve in the space.

However, this was only a 2 dimensional movement with only two drones. To increase complexity, we simulated a doughnut through which all of the drones must pass. In other words they all start on the same side of a room which is divided by a wall in the middle which has an opening which only allows for a single drone to go through at the time. This simulation is video [3], and we then increased the sampling rate, decreasing the size of the grid, in order to have the smaller hole seen in video [4].

Using this much more precise sampling rate we can load more complex terrains with floating islands and other obstacle and try to see how the drowns evolve in this new and more complex environment. An example of the drones in this new environment is on video [5], which attempts to follow the drones, and video [6] which is a stable top view of the world. We realize that the drones hug the boundaries a lot and will favor few long straight lines. This is the result we would get using the L_1 norm, an exact heuristic, for the A-star algorithm at the path-finding step. If we change this norm to the Euclidean distance, in video [7] we see that the drone prefers making diagonal zigzags a lot more. This shift of behavior is amusing to see and depending on the context either one or the other could be favored. More complex movement algorithms, which allow diagonals through space only is there is no obstacle on either side would take greater advantage of this behavior of the heuristic function but would create problems when avoiding the other drones. The slight swaying motion of the drones, while slightly decreasing their battery life, has the advantage of keeping the drones steady in harsher conditions.

The final simulation, video [8], is an illustration of the ”force” method which will be seen in the following section. However, we can see that is unreliable from the red and blue drone bumping into each other at the top.

4.2 Real life flights

Real life testing was planned with the Crazyflie drones using the Loco positioning system in the Lix laboratory and using DJI Trello drones using an Optitrack positioning system in an ENSTA laboratory. However, both of these establishments have been closed the day before testing was scheduled to start in a recent attempt by the French government to mitigate the spread of the COVID-19 pandemic. No real life test have therefore been conducted. These algorithms should nevertheless work as well in real life as they do in simulations.

5 A decentralised attempt

For now we have only studied the case of having a centralized machine doing all of the calculations as that is what best applied to the case of the Crazyflies. We are now going to try moving away from this technique to give a little more diversity.

The technique which we are now going to implement is in fact very close to a completely decentralized method in continuous space. This is rather a behavioral method were at each instance the drone needs to choose its’ action. This technique has been inspired from the realm of physics and we will therefore use a few analogies to better explain this method.

The initial goal of creating this method was to attempt to have a counterbalance to the method implemented in the previous part. The previous method was a discrete centralized model and this is why we contrasted it by putting together a decentralized model which could ideally work in continuous space. Ones expectations with this method however need to be sync with reality. We are not working with a truly continuous space as we have to sample it for the algorithms to actually accept it but also we are doing this as decentralized as the Crazyflies will allow it.

To begin, when we claim that this method works in continuous space, we still need to sample obstacles as the Crazyflie drones do not have any on-board sensor for environment feedback. We therefore sample the obstacles in a discrete space to do any preliminary calculations but will then allow the drones to move freely through the space and hence to not be constrained by this discrete map and to evolve continuously in the environment. This initial sampling of the space is done in exactly the same way as before. Again, we need to make the choice of computational efficiency or environment fidelity and we make it in the same fashion.

Once we have the space sampled we need to find a way for each drone to reach its’ destination point. Again we will reuse our A-star algorithm to do this. Until this point the algorithm works in exactly the same way as before. Reading the space and finding a path for every drone to follow. However, this is where we take a radical turn from the method using coordination spaces as now, instead of pursuing our calculations we launch the drones.

After the previous method where we made many complex calculations, costly in both time and memory, launching the drones at this stage seems completely absurd. Nevertheless, this is where the advantages of a decentralized method become apparent. We skip all of the big calculations in preparation and go for many small calculations along the way. We will therefor launch our drones and at every time step there will calculate a speed vector which they need to follow. The calculation of this speed vector is individual to each drone and is the decentralized part. However, having only worked in simulations, it was one machine doing all of these calculations but in theory they can be managed by the drones.

The only flaw to this decentralization attempt still relies on a global map to which every drone has access and where every drone puts its last updated position. This is because again, the Crazyflies cannot sense their environment so it must be given to them. They could in theory broadcast their position continuously to all that are listening around but to avoid saturation of the communication channel and since they connect to a central computer to have the map it seemed to be a fairly reasonable choice to also have their positions updated in this map.

Now, at every time frame the drones will calculate an inertial vector which they will follow and the entire nuance of this technique is to calculate this vector correctly. First, this vector represents the drone's movement direction. We therefore need to scan the surroundings of the drone to deduce where it shall move. To make this realistic if the drone had sensors and for it to not be to computationally expensive, we choose a radius of a couple times the drone in which we will explore the environment. From this section of the space we will deduce four intermediary movement vectors which we will then sum to get the final movement.

First of all, we need a sub-vector which will guide us along the path to the obstacle. We can see this as if water is flowing from the start position of the drone all the way to the end position we are just leaving the drone being guided by the current. This vector is the same vector as the one going from the closest point of the path to the next point along said path, just translated to the drone's coordinates.

Second, we have a sub-vector which pulls us towards the path to follow. This is to incite the drone to stick as close as possible to the path while retaining a bit of elasticity to deviate from the path if need be. Say for instance two drones need to cross each-other on the same path, this leeway allows both of them to slightly deviate from the path to cross and then return to it. To continue the analogy of the flowing water, pick a corridor through which the drone's path passes with the water flowing left to right, a corridor having obstacles both bellow and under. Now take the plane normal to the path, so the vertical plane. For now we had just assumed we had a flat ground on which the water flowed with a wall on either side, the obstacles. Now what we are doing is changing the bottom to a V shape with the lowest point being the ideal path. Therefore we will be attracted to this lowest point but may still deviate from it if the force is strong enough.

We can give a nicer visualization of both of these forces using a vector field. We will go back to the example which we saw at the start of the previous section. We have the two drones with their trajectories shown on figure 10. We choose to focus more on drone 1 and decide to map its trajectory through the space. We then sample points where we will calculate the forces which come into play. In figure 11 we have detailed the two forces seen above. In blue we have the first force we saw, being the direction of the path it will follow and in red we have the path proximity vector which will pull the drone back towards the path. The resulting force applied to the drone is the sum of both. In order to properly seen the vectors, a small sample of them was detailed but as this works in continuous space this can be calculated anywhere, on infinitely many points.

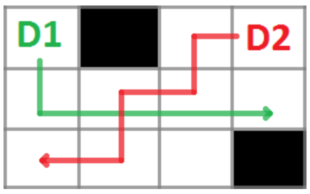


Figure 10: Two drones with their respective paths.

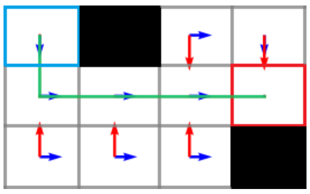


Figure 11: The forces which will apply to drone 1 for it to follow its path.

Third, we implement a vector which is used to avoid obstacles. When looking for obstacles to avoid, the drone considers these three distinct regions: anything behind the drone is ignored, anything on the side partially affect the trajectory and anything in front of the drone strongly alters the trajectory. Any obstacle in front of the drone will apply a repulsion force very much like magnets. In fact all of the following interactions are inspired by magnets. The amount of repulsion a drone will receive from a wall is inversely correlated to the distance between the drone and the obstacle in order to have the drone stop dead in its tracks if it is going to collide with the obstacle head on. This can be seen as the drone and the wall both being magnets of the same polarity. For obstacle on the side it work in exactly the same way except that the only component which will affect the drone's trajectory is the repulsion vector's projection onto the normal of the trajectory. This allows the drone to stabilize between two rows of obstacles without being blocked from moving forwards in this corridor. In order to avoid the drones escaping, this is why we added an invisible ring of obstacles around every XY plane of the maps of the environment.

Forth is the impact other drones have on the trajectory of the drone. Other drones are treated in a very similar ways to obstacles but the front and side actions are inverted. Therefore, if two drones are heading towards each other they will both offset each other to the side in order to cross without collision. Again, this can be seen as two magnets repulsing each other.

This different approach had only mitigated success rates and although interesting in the simulator is was to unreliable to attempt to deploy this in real life as the risk of damaging equipment, mainly Crazyflies, was to great.

6 Conclusion

To conclude this report, our decentralized technique had quite unreliable results whereas our technique using coordination spaces, even though slower to compute, was a lot more powerful and optimised. We could envision a system incorporating both techniques into one. This is assuming we have more complex robots, able to sens their environments, giving them a bit of decision making capabilities from their neighbourhood would give them a lot more flexibility and would allow it to adapt to an unstable environment, where the main layout is known be there can always be a slight delay or unexpected obstacle that the flexible distributed system could solve without putting the entire system on hold in order to recalculate the coordination spaces.
I had the extreme chance of speaking with an employee at Exotec Solutions which provide logistical solutions for warehouses with robot swarms and he indeed told me that they work with this hybrid approach. A entirely top down command structure is to slow to react and adapt while leaving a bit of the decision making to the actual robot allows it to bounce back from a bad situation and adapt to a changing environment a lot faster.

7 Acknowledgements

I would like to give a special thanks to my bachelor thesis supervisor, Professor Eric Goubault from the Lix, computer science laboratory of Ecole Polytechnique. His guidance and help was extremely precious to assist me in completing this paper. I would also like to thank the entire Cosynus team of Lix for their rapid integration and amazing support and feedback. I would also like to thank Emmanuel Haucourt and Pierre-Yves Strub for being my reference instructors.
I would like to thank Exotec Solutions for taking the time to answer questions of mine at the end of a presentation they gave at Ecole Polytechnique in February 2020. As well as the authors of all of the papers [1], [2], [3], [4], [5], [6], [7], who's papers where of great help in helping me grasp the subject and build a solid foundation on which I could devise and build my own algorithms.

References

[1] O'Kane Ghrist and LaValle. "Computing Pareto Optimal Coordinations on Roadmaps" (2005).

[2] Martin Raußen Lisbeth Fajstrup Eric Goubault. "Detecting Deadlocks in Concurrent Systems" (1998).

[3] Steven M. LaValle and Seth A. Hutchinson. "Optimal Motion Planning for Multiple Robots Having Independent Goals" (1998).

[4] Jean-Paul Laumond Thierry Simeon Stéphane Leroy. "Path coordination for multiple mobile robots: a resolution-complete algorithm" (2019).

[5] Yi Guo and Lynne E. Parker. "A Distributed and Optimal Motion Planing Approach for Multiple Mobile Robots" (2002).

[6] Daniele Bellan Charles E. Wartnaby. "Decentralised Cooperative Collision Avoidance with Reference-Free Model Predictive Control and Desired Versus Planned Trajectories" (2011).

[7] Jerry Ding Ryo Takei Haomiao Huang and Claire J. Tomlin. "Time-optimal multi-stage motion planning with guaranteed collision avoidance via an open-loop game formulation" (2012).

Statement of Academic Integrity Regarding Plagiarism

I, undersigned Roger Alexis, hereby certify on my honor that:

1. The results presented in this report are the product of my own work.

2. I am the original creator of this report.

3. I have not used sources or results from third parties without clearly stating thus and referencing them according to the recommended rules for providing bibliographic information.

Declaration to be copied below:

I hereby declare that this work contains no plagiarized material.

Date 10/04/2020, Palaiseau

Signature [illegible]