UAV Engine-Out Problem using Fast Marching Trees (FMT)

Thesis submitted to the University of Sydney for
the Degree of Bachelor of Aeronautical Engineering (Space), Honours,
from the School of Aerospace, Mechanical & Mechatronic Engineering

Aditya Vishwanathan
SID: 311196950

October 2015

ABSTRACT

This thesis aims to solve the UAV engine-out problem, where a trajectory is planned for a UAV subject to contingency checks along each point. Each check models an engine malfunction, after which the UAV is forced into a steady glide descent and must be able to reach a pre-defined safety zone within an allotted amount of time. The inherent difficulty lies in dealing with the non-linear temporal checks in a computationally efficient manner. This is non-trivial since a high accuracy demands a fine path discretisation, which leads to extremely large number of checks being performed. The solution must also incorporate non-holonomic dynamics, wind disturbances and collision avoidance.

The importance of the problem cannot be understated as the demand for UAVs in the civilian sector is expected to rise over the next 10 years with the FAA planning to integrate UAVs into the National Airspace System. This is clearly a boon for the UAV economic industry, however a more congested airspace can directly lead to more disastrous collisions resulting from engine malfunctions if they are not handled appropriately.

This thesis proposes a solution utilising a recently developed Sampling Based planner, Fast Marching Trees (FMT). An extension of the canonical method called FMT* is used as the framework. To improve the efficiency of the algorithm, it is built upon using two novel algorithmic extensions: the “Path Reconstruction” and “Obstacle Cluster” algorithms.

MATLAB simulations are used to validate the individual components and the core algorithm under problem instances of varying difficulty. Results indicate that both algorithmic extensions induce a significant improvement in computational runtime, while not increasing the path cost of the final result. The combined algorithm is shown to yield on average a 76% runtime improvement over the Brute-Force method for a high complexity environment, and be robust to both static and varying wind fields.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor Dr. Zhe Xu. Going into this thesis, I wanted to do something in the area of path planning optimisation, but I didn't have an exact topic in mind. You were able to guide me towards a problem which was both challenging and rewarding. There was a period towards the start of the 2nd Semester where progress was slow and I was not clear in the direction I should take. Your prompt feedback via email and through meetings was to the point and pivotal in allowing me to develop my source code, thesis document, and presentation. Thanks to your patience during this time, I was able to make rapid improvement and ultimately be very proud of my final product. I wish you best of luck in the future as an academic and researcher.

Next, I would like to thank the friends that accompanied me throughout my journey as an undergrad student at USYD. Words cannot state how important your support, encouragement, and company have been over the last 5 years. The experiences we had during the all-nighters in the Mech PC lab, Aero common room, and Aero tute room will never be forgotten. I'm sure as a collective we can all agree that having a good friends circle is the most important thing an undergrad student can have.

Finally, I would like to thank my family and friends outside Uni. To my family, you have been nothing but supportive and loving during the stressful times this semester. I know its been frustrating having me stuffed up in my room all day, barely saying a word, and sometimes not even coming down to eat. Without you, I wouldn't be where I am today. Special thanks goes to my sister Aparna for proof-reading my thesis. To my friends, you have been so important in allowing me to unwind during times I just want to forget about Uni altogether. Whether it be through getting together to play football, going out on the weekend, or playing Mafia and Super Smash Bros., its allowed me to not only survive my thesis year, but my whole undergrad degree. Thanks guys!

DECLARATION

I declare that the contents of this thesis are in compliance with the University of Sydney policy on academic honesty. All opinions derived in this thesis are my own, and do not necessarily express the views of my supervisor or the University of Sydney.

The following is a list of my contributions to this thesis:

- I performed the literature review on path planning problems and current methods to solve the UAV engine-out problem.
- I developed the "Path Reconstruction" algorithmic extension.
- I developed the "Obstacle Cluster" algorithmic extension.
- I developed the MATLAB source code which integrated the standard FMT* algorithm with the two extensions, along with the UAV dynamics, obstacle avoidance and wind disturbances.
- I produced the results using the MATLAB source code.
- I analysed the results and made conclusions on my own.

The above is an accurate representation of the student's contribution.

Signed:

Aditya Vishwanathan

Dr. Zhe Xu

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

Engine failures are a large cause of concern for mission planners when dealing with the operation of Unmanned Aerial Vehicles (UAVs). This is primarily due to the lack of a skilled pilot on-board who can deal with the resulting loss of control by assessing the environment around them to perform an emergency landing. This has recently become a growing issue due to the fact UAVs are seeing a huge rise in use ranging from a plethora of civilian and military applications. With more UAVs in operation, the chances of a crash landing or collision from a loss of control increase. Therefore an autonomous system which will allow the vehicle to land safely is highly desirable. The challenge is to produce dynamically feasible routes for the unpowered UAV, which consider spatial constraints from a complex obstacle space, as well as temporal constraints for landing in pre-defined safety zones. This chapter will begin with the motivation and inherent difficulty in solving such a problem, followed by a brief description of the proposed solution.

1.1 Motivation

In recent years, UAVs have had a significant impact in applications ranging from environmental monitoring, surveillance, reconnaissance and search and rescue [12]. As evident in Figure 1.1, the demand for UAVs in this sector is expected to increase further over the next few years. This has been permitted by regulatory changes which previously forbade UAV usage due to safety concerns. In fact, between 2015 and 2025, the Federal Aviation Authority (FAA) aims to integrate mainstream civilian UAV use into the National Airspace System (NAS) [13].

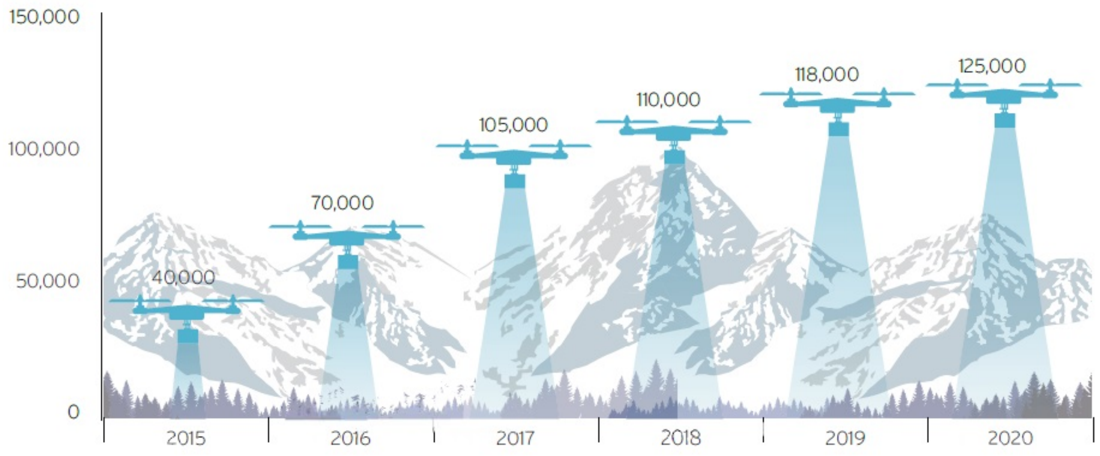


Figure 1.1: Projected annual sales of UAVs [1]

However to make such an integration possible, it must be proven beyond a reasonable doubt that particularly civilian UAV missions can be conducted with utmost safety in mind. Therefore, a top priority for mission planners is avoiding the repercussions which could stem from an emergency landing that is not handled appropriately. According to CASA [14], this means ensuring a “controlled descent” where the UAV is able to glide safely to the ground whilst avoiding terrain such as power lines, trees, and civilian infrastructure. A major accident in a populated area could be detrimental to the future of civilian UAV usage.

It is therefore necessary to develop effective methods to handle forced landings. The most commonly employed solution was to use parachutes to enable a controlled descent to the ground [15]. However, issues arose stemming from the adverse effect that disturbances, such as wind gusts, can have on the final landing position or the path taken to said position. As such, extensive research has gone into automated systems which typically consist of the three following components [16] :

1. *Candidate Site Selection System (CSSS)* - responsible for generating potential landing waypoints typically using machine vision.
2. *Multi-Criteria Decision Making System (MDMS)* - assesses the multiple landing sites based on mission objectives and costs
3. *Trajectory Generation and Guidance System (TGGS)* - generates feasible trajectories to the chosen goal within the limits of the UAV’s dynamics.

In particular, developing an effective *TGGS* is a problem which has generated much interest in the robotics community due to its inherent difficulty.

The core of the problem lies within the UAVs trajectory being subject to non-linear temporal constraints which must be satisfied for every single point along its path. This is to ensure that an engine-out malfunction can occur at any point during the flight path, and the UAV can make a controlled glide descent to a designated safety location within an allotted period of time. As well as the temporal constraints, the UAV must of course avoid obstacles along its main path, and also during its descent if an emergency scenario was to occur. Dealing with both sets of constraints in a time efficient manner is non-trivial.

Additional complexity to the *TGGS* problem lies in the dynamic constraints that are imposed on the vehicle, which ensures that the final path returned is non-holonomic. A non-holonomic system is one which is path dependent, meaning the states of the UAV will change based on how it got to a specific position. Finally, the trajectory is also subject to wind disturbances which can drastically effect the flight of an unpowered UAV.

As such, there are many layers to producing an effective *TGGS* all of which need to coherently mesh into a computationally efficient system which yields a feasible flight path for the UAV.

1.2 Proposed Solution

To solve the path generation problem this thesis proposes a strategy utilising Fast Marching Trees (FMTs), which is a recently proposed sampling based planner incorporating elements of two well established frameworks; PRMs and RRTs. The advancement in the state-of-the-art will come from investigating how FMTs scale to complex problem instances, such as the UAV engine-out problem, as well as through introducing two novel algorithmic extensions to improve runtime. A naive implementation of the algorithm is not possible since the computational expense of incorporating the non-linear temporal and spatial constraints is not feasible. Modifications to the graph searching method must be made to make the algorithm suitable. One of these will be attempting to map the non-geometric obstacle space in an effort to narrow the search space.

1.3 Thesis Contributions

- Non-holonomic path planning solution to UAV engine-out problem using recently proposed Sampling Based planner FMT*. This thesis exposes the method to an inherently difficult problem with multi-layered constraints, while incorporating wind disturbances.

- "Obstacle Cluster" algorithm: a novel extension to the FMT* framework which aims to filter invalid nodes via analysing the spatial and temporal obstacle space, and therefore save significant computational time. Previous work in the field has achieved a similar results via analysis of a purely geometric space, however this thesis extends the state-of-the-art by analysing a space which is non-geometric.
- "Path Reconstruction" algorithm: a novel extension to FMT* framework which aims to save computational runtime during the path planning process. It does this by reusing paths which were previously generated by the path planner so future temporal constraint checks become significantly shortened.

1.4 Thesis Structure

Chapter 2 - Literature Review:
Several groups of algorithms are explained and assessed for their suitability in solving this thesis problem. They are: optimisation/local reactive control algorithms, graph search algorithms, alternate route search, and sampling based planners.
Chapter 3 - Background:
This chapter provides the reader with an introduction to the canonical FMT* algorithm and its implementation to standard path planning problems.
Chapter 4 - Problem Definition:
This chapter formulates the problem scenario and states the assumptions that are carried over into the methodology.
Chapter 5 - UAV Engine-Out Solution using FMT*:
The core algorithm for solving the UAV engine-out problem utilising FMT* is thoroughly explained. Two novel extensions, namely "Path Reconstruction" and "Obstacle Cluster", are also explained in this chapter in the context of how they can improve the Brute-Force implementation.
Chapter 6 - Results and Discussion:
The core algorithm as well as the two algorithmic extensions will be assessed for their suitability in solving the UAV engine-out problem via a series of MATLAB simulations. The evaluation will be based on two factors: path cost and computational expense.
Chapter 7 - Conclusion and Future Work:
A summary of this thesis will be given and a final evaluation of the solution is provided. The individual contributions of this thesis will be listed, as well as a possible direction for future development of the algorithm.

CHAPTER 2

LITERATURE REVIEW

Path planning is a research field which has been widely explored, and as such there can exist many classes of solutions to any single problem. A general path planning problem can be defined as finding an optimal trajectory from a start to a goal position through a state space, which is usually defined in the Cartesian or Polar plane. Constraints are added to this basic framework which can make finding a feasible solution non-trivial. For the UAV engine-out problem, there exists both well defined geometric constraints in the form of obstacles as well as non-convex and non-linear temporal constraints defined by the UAVs proximity to safe landing zones along its path. Hence the chosen class of algorithm must be able to handle the above and return a solution within a tractable amount of time.

This chapter presents a literature review about the past and current methods for path planning which have the potential, or already been applied to the above path planning problem. These can be grouped into the following groups: optimisation algorithms, local reactive control algorithms, graph search algorithms, alternate route graph search, and sampling based planners. The advantages and disadvantages of each will be discussed and conclusions will be drawn regarding the suitability of each approach to the specific problem at hand.

2.1 Optimisation Algorithms

At first glance the problem definition implicates use of an optimal control algorithm to solve it since we want to minimise a cost function subject to dynamic constraints, governed by the UAVs motion, and static constraints, dictated by obstacles. The following subsection will assess both the Linear Quadratic Regulator (LQR) and Evolutionary algorithms.

2.1.1 Linear Quadratic Regulator (LQR) Control

Out of the many optimal control algorithms that exist, perhaps the most widely used in robotics is LQR [17]. The method aims to converge an initial state towards the final goal state while minimising a quadratic cost function with a set of linear constraints. Via manipulating the cost function, one can obtain a path which minimises the state deviation, control input, or a mixture of both. An additional advantage in using LQR is that it can solve both finite and infinite horizon problems, meaning that if the UAV is required to reach the goal state within a certain time, this can be enforced in the LQR model. An example of an infinite horizon LQR formula is given in the formulae below.

$$\text{minimise } J = \int_0^\infty (x^T Q x + u^T R u) dt \tag{2.1}$$

$$\text{subject to: } \dot{x} = Ax + Bu \tag{2.2}$$

$$\text{control law: } u = -Kx, K = R^{-1} B^T P \tag{2.3}$$

$$\text{algebraic riccati equation: } A^T + PA - PBR^{-1}B^T P + Q = 0 \tag{2.4}$$

Despite its clear advantages for linear systems, or systems which have well established equilibrium points to linearise about, LQR has fundamental issues which limit its usefulness for the UAV engine-out problem. These are: it cannot work with non-linear constraints or cost. Observe Eqn. 2.1 where a cost function must strictly come in a quadratic form as a function of user specified matrices Q and R. Also note the linear state space model in Eqn. 2.2. Recent studies by Hajiyev and Vural [18] have shown that by using a linearised motion model and finely tuned cost functions, LQR can yield feasible paths for UAV path planning problems. However it should be noted that this study was done assuming that the aircraft would operate within small deviations of wings-level trimmed flight, and this is not a valid assumption in our case. In addition, the LQR framework struggles to model non-convex constraints, which are the basis of the UAV engine-out problem. Hence, LQR is not feasible as a path planning scheme for the engine-out scenario.

2.1.2 Evolutionary Algorithms

Evolutionary algorithms are a unique brand of methods designed to imitate biological evolution. The underlying notion is that we begin with an arbitrary size population and expose them to environmental pressure which creates the process of natural selection. The elements which have a low fitness value are rejected, and the ones with high fitness are kept to ensure the overall fitness of the population is high. Based on this, the strong candidates are chosen and recombination and/or mutation are applied to breed the next generation [19]. This process is iterated until a candidate with a desired fitness value is found, representing the goal or solution. Evolutionary algorithms are well known for solving problems of maximisation and minimisation, but correspondingly have also found a place in path planning. Gerke [20] showed its potential in robot path planning with obstacles, and this was followed by Rathgun et al. [2] who used Evolutionary methods to successfully avoid collision between several UAVs. A flow chart demonstrating their procedure is shown below in Figure 2.1.

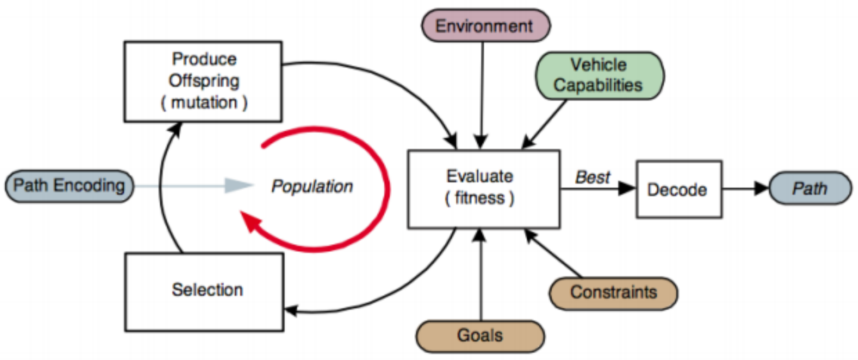


Figure 2.1: Overview of evolutionary algorithm [2].

The work done by Rathgun et al. showed that despite the algorithm possessing a sense of randomness, particularly during the process of mutation and recombination, the final path produced was generally feasible and close to optimal. One significant drawback however with Evolutionary algorithms is the computational expense of randomising variables repeatedly. Further, the closer the final path comes to the optimal solution, the more iterations are needed, which can result in an extremely long computational runtime if a high quality path is desired. Hence there exists a trade off between how accurate a solution is and the time taken to achieve that result. Recent developments in this field have made significant improvements to the runtime by using parallel computing and Graphical Processing Units (GPUs) [21, 22]. However in general for UAVs of smaller size and weight, computational expense remains a huge barrier in effectively utilising Evolutionary algorithms.

2.2 Local Reactive Control Algorithms

Local reactive control algorithms are unique when compared to standard path planning algorithms. Instead of looking at a problem as an entire state space to solve over, these methods aim to produce feasible paths in local regions around the vehicle's current location. The following subsection discusses both Artificial Potential Field and Receding Horizon Control applied with the Mixed Integer Linear Programming (MILP) framework.

2.2.1 Artificial Potential Field

Artificial Potential Field is a method which models collision avoidance by creating force fields around vehicles and obstacles. Even though it pre-dates many of the other path planning techniques discussed in this Chapter, it has been used extensively in robot path planning and is still very relevant today. Originally introduced by Andrews [23] and Khatib [3], the algorithm works by creating an imaginary force around obstacles which repels the vehicle, while the goal will attract the vehicle. This creates a potential field which is visualised in Figure 2.2.

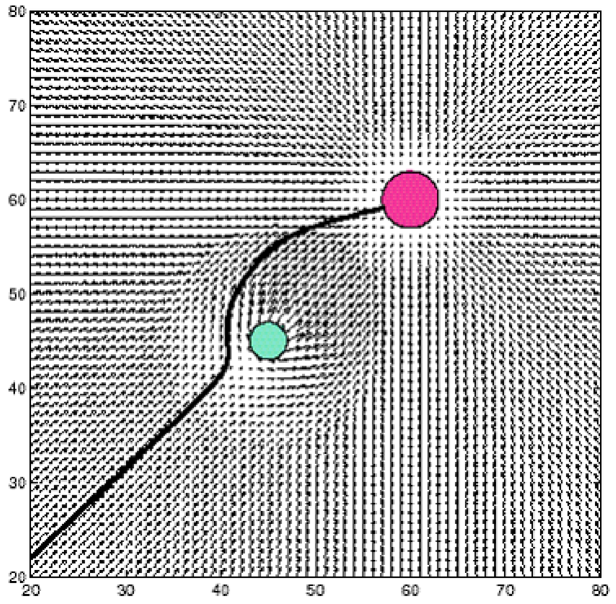


Figure 2.2: Khatib's potential field simulation [3], where the gradient field represents forces on the particle.

The result is a feasible path from start to goal which is collision free, however not guaranteed to be optimal. Despite being commonly used in robot path planning and sometimes UAV planning [24], this method suffers from limitations which restrict its use in more complex problems. For one, the method assumes a convex obstacle field, which is not a valid assumption in our UAV engine-out problem. Further, the canonical potential field method was also shown to get stuck in local minima, as well as perform poorly in narrow regions [25], which is a likely scenario for a UAV in a cluttered environment. This is because obstacles lying very close to each other can impart oscillatory forces on the vehicle. Despite these issues, variants of the algorithm have been created and also applied to UAV path planning. One such example was the Total Field Collision Avoidance proposed by How and Sigurd [24], which effectively dealt with the problems of local minima and oscillatory motion. However it still assumed convexity for obstacles, which is ultimately the deterrent in applying Artificial Potential Field to this thesis problem.

2.2.2 Receding Horizon Control with MILP framework

The UAV path planning problem consists of continuous dynamics and discrete decisions relating to obstacle avoidance. It was shown by Bernporad, Morari [26] and Williams, Brailsford [27] that the MILP framework can be effective in such scenarios. MILP works by minimising a linear cost function with a set of convex constraints by using a mixture of binary and integer variables. This poses an issue since non-linearities within the UAV dynamic model will need to be handled appropriately. In addition obstacles need to be convex. This covers regular polygons and conic sections such as circles and ellipses, however does not include more complex obstacle fields. Despite the limitations, Richard and How [4] designed a linearised model of aircraft dynamics and successfully utilised MILP to solve a path planning problem. An example path is seen below in Figure 2.3.

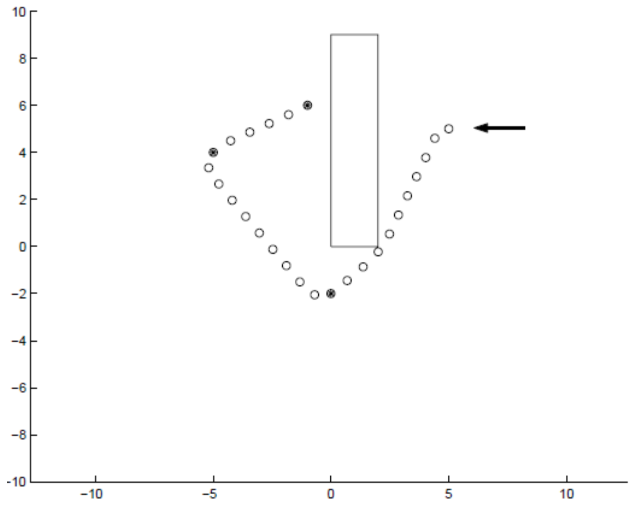


Figure 2.3: Richards and How [4] simulation of vehicle travelling from start (arrow) to goal (last black circle) while avoiding obstacles.

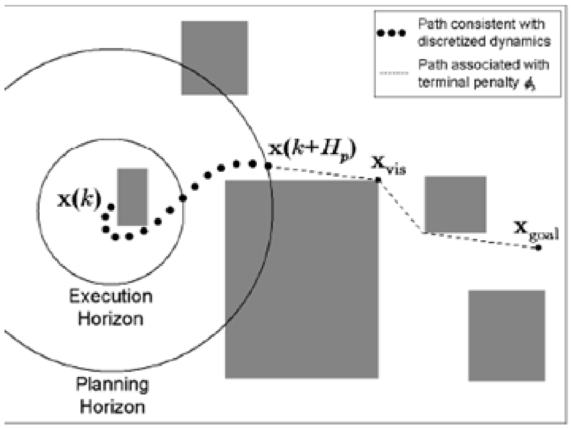


Figure 2.4: Resolution levels or "horizons" found within Receding Horizon Control [5].

One of the well documented issues with MILP is the computational cost, as it is an NP-hard problem. This means it has an exponential runtime with respect to the number of binary variables and waypoints. For a complex obstacle instance, such as the ones found in our *TGGS* problem, this would result in paths being generated in intractable amounts of time. Research to alleviate the high computational expense has led to methods such as Receding Horizon Control [5]. This method works by splitting the path optimisation into several executions called "horizons", seen in Figure 2.4. Since planning is now done in these smaller horizons, only obstacles relatively close to the vehicle at that time need to be accounted for in the optimisation. This not only reduces computational expense but also models a real world UAV scenario where certain parts of the environment may not be visible at the current state and only as the vehicle progresses does the visibility graph expand. Despite these improvements, implementation of a Receding Horizon Control/MILP planner still assumes convexity of obstacles, which is not a valid assumption for our problem.

2.3 Graph Search

Graph search algorithms are some of the most fundamental and underlying approaches to vehicle and robot path planning. As the name implies, it relies on a graph data structure being formed by discretising the continuous environment. A graph in its most simplest form contains nodes which have edges to other nodes. Due to the structure and typical implementation of these algorithms, it can be quite computationally expensive. This is summarised well by Elbanhaw and Simic [28], that to attain a high grid resolution involves undesirable discretisation which degrades performance significantly. Despite this, the usual approach is to utilise one of the many variants of the canonical graph search algorithms to improve runtime. The rest of this section will focus on two underlying graph search algorithms: Dijkstra's algorithm and the A^* algorithm.

2.3.1 Dijkstra's Algorithm

Dijkstra's algorithm [29] is arguably the most widely known graph search algorithm. The algorithm aims to find the shortest path from a start node to a goal node using label correction techniques. These labels will contain what is commonly known as a Value function in dynamic programming, which represents the optimal cost to reach that node from the starting point. In doing so, it ensures that once a node is visited, it does not need to be visited again. The algorithm completes once the goal node has been visited, and thus the shortest path is obtained. Many limitations prevent the canonical algorithm from being implemented in real systems. One of these is the inability to deal with negative costs. Another is that it is uninformed about the location of the goal during the search, which leads to significantly higher computational run-times [30]. As a result, it is more common to see variants of Dijkstra's algorithm in most path planning problems.

2.3.2 A* Algorithm

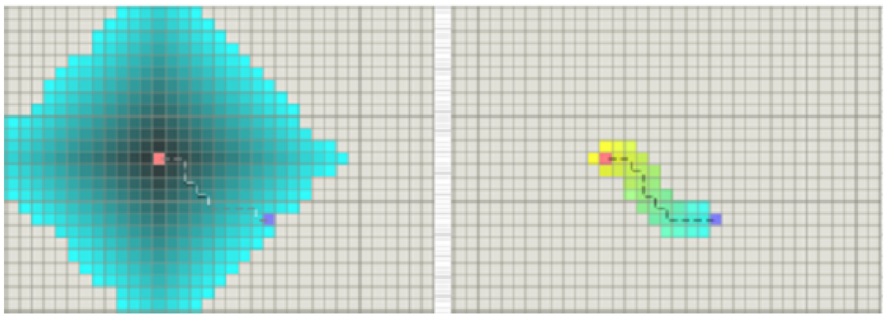


Figure 2.5: A^* algorithm shown on the right and Dijkstra's algorithm shown on the left [6].

Figure 2.5 visualises one of the most well known variants of Dijkstra's which is the A* algorithm. The algorithm combines elements of Dijkstra's algorithm with a Best-First search [31], which will favour nodes close to the goal. It does this by introducing a heuristic, which in simple trajectory planning is usually the straight line distance between each node and the goal. The effect of this heuristic is shown clearly in Figure 2.5 which shows that Dijkstra's algorithm will expand the visited graph in all directions, while the A* algorithm will move towards the goal. This makes the A* algorithm more computationally efficient in most cases, and therefore it has seen extensive use in robot and UAV path planning [32, 33].

2.4 Alternate Route Graph Search

The graph search algorithms mentioned above, while underlying methods in path planning, are rarely used in isolation. The reason for this is that the optimal path for a specified cost will often be infeasible when cross checked with the problem constraints. An intuitive example demonstrating this is thinking about a person who wishes to drive their car from a point A to B. Let us assume that the driver wishes to minimise the time taken during the journey. However, there is an additional constraint on the amount of fuel they can expend during the trip. Consider one route from A to B which takes a very short amount of time, but consists of several up hill slopes which burn a considerable amount of fuel. Now consider a second route which in contrast takes slightly longer, but only consists of a very slight incline. It is clear that an optimal search algorithm will return the first route, even though it might not be feasible with respect to the fuel constraint.

While the above example is relatively simplistic, it shares similarities with the UAV engine-out problem in that any route found by the path planner must satisfy the temporal and spatial constraints. This motivates the use of alternate route algorithms for our problem since they are designed to return paths higher in cost which can satisfy the constraints. A few examples of well established alternate route algorithms are provided below and are assessed for their potential effectiveness for this thesis problem.

2.4.1 k shortest path algorithm

One method for returning alternate routes is Eppstein's k shortest path algorithm [34]. As the name suggests, the algorithm returns the optimal path and $k-1$ other paths in increasing order of cost. It does this in an efficient computational runtime of $O(m + nlogn + kn)$ by utilising heap-ordered trees to store the costs for each path.

It should be noted that the algorithm is more suitable for sparse graphs and for a dense state space there would be no guarantee of a feasible path returning even for extremely high values of k. In addition, the algorithm doesn't provide a means to customise the quality of alternate routes provided from the optimal route, which is highly desirable for the UAV engine-out problem. For example, if we know the optimal route is located very far from a marked safety zone, we know that a feasible route would have to be considerably different from the optimal in order for the UAV to stay close to this safety region. However there is no way to specify this in the algorithm and hence the only way to obtain such an alternate route is to input a high enough value of k, which could be a computationally expensive procedure.

2.4.2 Multi-criteria Objective Functions

Another intuitive approach is to alter the optimal graph search algorithms to optimise multiple objective functions or a single multi-criteria objective function. We can refer to our original example of a car travelling from A to B and see that by using an objective which is a combination of both the time taken and the fuel burned, a feasible route can be returned.

Work done by Geisberger et al. [35] showed that this can be a reasonable approach to take for a linear combination of two costs. This has a benefit over the k shortest path algorithm in that it is more customisable as the user can define how important one objective is relative to another. The obvious drawback however is that it can only be applied for a linear combination of costs. The UAV engine-out problem consists of highly non-linear temporal constraints governing the path, as well as spatial constraints from the obstacle avoidance. A linear combination of these two costs will not necessarily yield a feasible final route. Therefore, this option is not suitable.

2.4.3 Alternate Routes in Road Networks

A more recent method proposed by Abraham et al. [36] in 2010 was motivated by the need for alternate routes in road networks. Due to modern day advancements in GPS technology, more emphasis is being placed on tailoring routes to a user's specific preferences. Therefore, the optimal route will be subject to several factors such as tolls, time spent on highways, off-road segments etc. This paper proposes a specific definition for what a "good" alternate route entails, which will now be described to give some background to their algorithm.

1. Limited Sharing - $l(Opt \cap P) \leq \gamma.l(Opt)$ for $0 \leq \gamma \leq 1$
2. Local Optimality - P is T-locally optimal for $T = \alpha.l(Opt)$ for $0 \leq \alpha \leq 1$
3. Uniformly Bounded Stretch - P is $(1 + \epsilon)$ for $\epsilon \geq 0$

Limited sharing is the idea that a desirable alternate route should have a limited amount of similarity when compared to the optimal. For the case of a discretised graph, this means we can limit the amount of common edges or vertices shared between the two routes or compare overall cost. Local optimality enforces that each decision made by the path planner must be locally optimal, i.e. no unnecessary detours are taken by the vehicle. Finally, uniformly bounded stretch deals with a special case where a path can be significantly shortened by connecting two points along a sub-path.

Overall, the approach yields an extremely versatile and robust method for producing feasible alternate routes. A significant advantage it has over other similar algorithms is its customisability, since the quality of an alternate route can be significantly altered by changing the constants γ, α and ϵ. Consequently, it has been applied along with the popular Sampling Based Planner RRT by Choudhury et al. [8] to produce alternate routes for a helicopter emergency scenario, a problem which is extremely similar to this thesis problem. Therefore, the road network algorithm is a suitable candidate.

2.5 Sampling-Based Planning

The most appropriate class of solutions to this problem are Sampling-Based Planners (SBP). An SBP algorithm finds paths by sampling random points in the environment and then connecting them, which results in much faster solutions to difficult optimisation problems at the expense of completeness and optimality. Instead, SBP algorithms are deemed probabilistic complete, in that given enough runtime, the probability of finding a feasible path is extremely likely [28]. The two most popular SBP methods are Rapidly-Random Trees (RRTs) and Probabalistic Roadmaps (PRMs), out of which the former has been explored in much greater detail for this class of problem. Recently a new method known as Fast Marching Trees (FMTs) has also been proposed, and this will also be explored in the following section.

2.5.1 RRT

RRTs [7] are an extremely well established method when it comes to robot, and specifically UAV path planning [37, 38]. The benefit of RRTs compared to most other path planning algorithms is the ease of its implementation and relatively low computation time. It is also biased in such a way that the state space moves towards unexplored regions. An RRT begins by sampling nodes one-by-one in the search space. Each new node that is generated is connected to its nearest neighbour. The algorithm will then choose the next nodes in the direction of the newly generated point. As a result, the coverage of the method is high and the majority of the search space can be explored quickly, which can be seen in Figure 2.6.

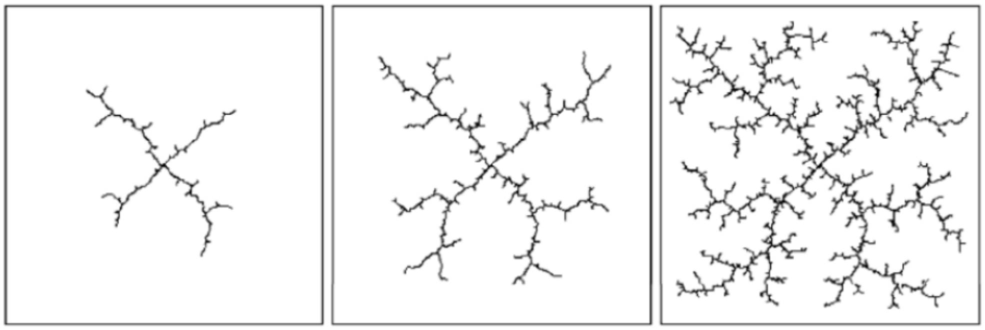


Figure 2.6: RRT expanding over the free search space [7].

Many variants of the canonical RRT algorithm have been produced in an effort to further optimise it in terms of runtime, and also produce more optimal paths. The latter was solved by the introduction of RRT* [39]. This was an important landmark because it paved the way for research in applying this optimised RRT with path planning for the helicopter engine-out problem, which possesses several similarities to our UAV problem. Yomchinda et al. [40] made a significant contribution by developing a feasible solution incorporating all necessary vehicle dynamics in an emergency landing scenario. This was extended upon by Choudhury et al. [8], who introduced their RRT*-AR algorithm, which included collision avoidance and expanded upon the canonical RRT and RRT* to provide more possible alternate routes and with lower computational expense. These alternate routes were defined using the work done by Abraham et al. [36], which was explained in Section 2.4.3. A visual example of their work is seen below in Figure 2.7

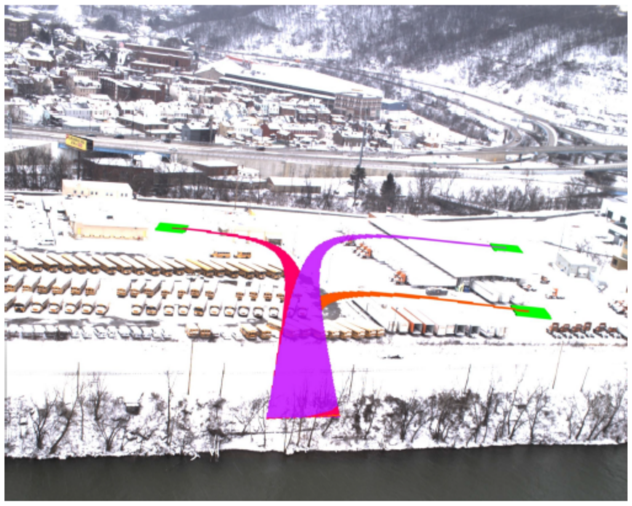


Figure 2.7: RRT*-AR algorithm in action with a simulation of a Bell 206 helicopter undergoing an emergency landing scenario [8].

Overall, RRTs are an established and well explored algorithm for aerial vehicles. This is because they provide an excellent basis for non-linear systems which are highly-dimensional and involve differential constraints. As such, utilising RRTs to solve the UAV engine-out problem would be a suitable approach.

However there are disadvantages within the RRT framework which suggest looking into alternative methods. RRTs are formed using an inherently greedy search since each branch of the tree is expanded towards its next node via finding its respective local optimum. This can result in a sub-optimal final path being generated. RRT* successfully alleviated this issue by introducing a node "re-wiring" step which ensured that vertices are reached via a minimum cost path [39]. However, this step does introduce extra computational processing which is avoidable if we utilise a more reliable approach to find the next best expansion node through dynamic programming (as will be discussed with FMTs).

An additional concern with using RRTs arises when we consider a disturbance which acts upon the UAV causing it to be significantly perturbed from its initial path. Assuming an emergency case, the guidance and control mechanisms might not be able to align the UAV with its RRT generated path. Consequently, this would require the formation of a new tree to be built from scratch since the original RRT is only valid for mapping a path from the initial starting position to the goal. This is undesirable since our goals were to develop fast paths and be robust to external influences. Despite this thesis not dealing with these extreme disturbances, a framework which can be expanded on to incorporate them in future work is desirable. This conveniently leads onto PRMs which allow us to cope with disturbances to a much better extent.

2.5.2 PRM

PRMs [41] are an alternative SBP algorithm to RRTs which are usually applied to multi-query path planning problems [28]. This is because it has the advantage of only needing to create the roadmap once, assuming a static environment. Let us revisit the problem mentioned at the end of the RRT section. We can see in Figure 2.8 that since a fully described graph has been generated from start to end, if a disturbance knocks the vehicle to another node, we can still effectively re-plan a route to the goal using the same roadmap.

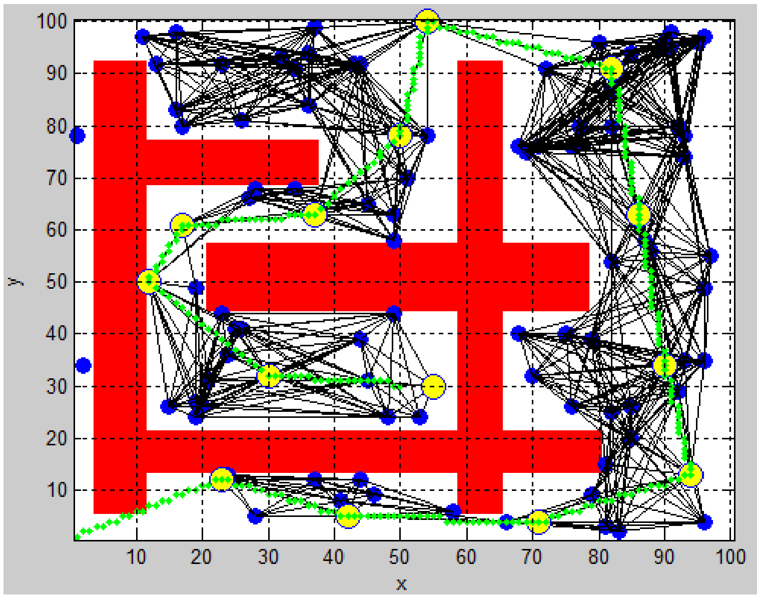


Figure 2.8: PRM roadmap and corresponding shortest path from bottom left hand corner to most inner yellow circle. (This graph was generated using the freely available MATLAB robotics toolbox).

A PRM samples points quite differently to RRTs. Instead of one at a time, a set number of random points are randomly generated and tested to ensure they do not lie in the obstacle space. Depending on a set distance around each point, edges are formed by connecting neighbours. These edges must also be checked to ensure they do not intersect with the obstacle space. The overall result is a graph or roadmap which can then be processed using a graph search algorithm such a Dijkstra's or A* to compute the optimal solution.

PRMs have been used extensively in robot and UAV path planning. An example is seen in the work done by Yan et al.[42] which presents a slightly modified PRM algorithm by using a tailored sampling distribution to navigate a UAV through a series of 3D obstacles. However PRMs are notorious for being less efficient than RRTs in single-query problems and mapping narrow spaces [28]. The former is largely due to the time required to generate the roadmap. It is extremely important as per the description of our *TGGS* problem that computational expense is minimised, since we require a feasible path to be returned in real-time. The latter is due to the inherent nature of random sampling which cannot guarantee providing ample samples within the narrow regions. This can sometimes lead to sub-optimal or no feasible paths being returned, particularly when traversing the narrow space is the only way to get from the starting position to the goal.

Recent developments in the research field however have made large strides in making PRMs suitable for this class of problem. The runtime limitation was addressed by Salzman et al. [43] by implementing sparsification of the roadmaps using edge contraction. This in effect improves the efficiency of the query part of the motion planning as the size of the data structure is greatly reduced, without a significant loss in path quality.

Additionally, Denny et al. [44] recently introduced a "Lazy Toggle PRM" method, which tackles both limitations by effectively combining Lazy collision detection and Toggle PRM to make it efficient when dealing with single-query problems with high complexity. Lazy collision detection [45] works by delaying the validation of the roadmap until query time, which means that edges are lazily added without checking their feasibility against the obstacle space. Infeasible edges are then culled while searching for the optimal path.

Toggle PRM [9] deserves special mention since it maps both the free space and the obstacle space to force the planner to produce paths through narrow passages. This can be best explained to the reader via analysing a visual example in Figure 2.9 below.

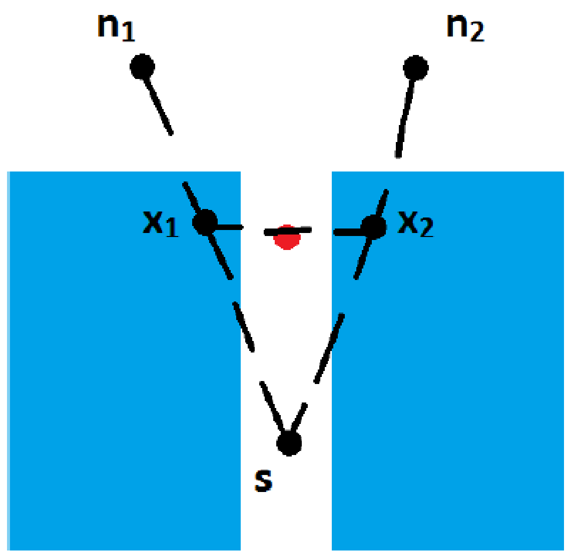


Figure 2.9: Visualization of Toggle PRM [9]. Nodes s, n_1 and n_2 belong to the free space, and x_1 and x_2 are witnesses of the failed connections in the obstacle space.

The concept is that when we find failed connections within the free space (C_{free}), we leave behind witness nodes within the obstacle space (C_{obst}). The witness nodes x_1 and x_2 are examples of failed connections between s and n_1 and s and n_2 respectively. If we then toggle the planner to make connections in C_{obst}, we see that x_1 and x_2 result in a failed connection, since the line passes through C_{free}. We then similarly leave behind a witness in C_{free}. The result of this is that we now have manufactured a sample within the narrow passage, seen by the red dot in Figure 2.9. Therefore, by performing this coordinated mapping of C_{free} and C_{obst}, we are able to produce sample points within problematic narrow regions in the state space and thereby improve the optimality of the final path.

It should be noted that Toggle PRM is not designed for non-geometric cases, but it does serve as a good foundation and motivation to explore such an algorithm for a more complicated problem instance with multi-layered constraints. This will be an area of research which this thesis will explore, and an advancement to the state-of-the-art.

Ultimately it should be noted that the PRM framework is difficult to use for this problem due to its difficulty in modelling UAV motion constraints. This problem arises because unlike RRT, PRM will not search the state space and form the optimal path concurrently. A static roadmap is formed first where there is no knowledge of the vehicles state, therefore making it extremely inconvenient for mapping differential constraints. However, methods such as Toggle PRM motivate a unique approach of mapping the obstacle space in an effort to narrow the search space, which I believe can be utilised for this thesis problem to achieve a similar result.

2.5.3 FMT

FMT*, a variant of the canonical FMT, was introduced by Janson et al. [10] and has had relatively little exposure to path planning problems when compared to the two aforementioned SBP algorithms. It is unique from PRM and RRT in that it combines elements of single-query planners and multi-query planners, thereby effectively giving the user advantages from both. What makes FMT* an extremely promising method for minimising computation time is the way it "lazily" expands its tree of paths.

FMT* begins by statically sampling the search space, similar to PRM. The number of samples drawn is chosen by the user. Points are them connected using a tailored disk structure, where each node has a fixed radius around it locate neighbour nodes. FMT* allows for asymptotic optimality of the final path by setting the radius around each node as a function of the number of samples drawn. From here, FMT* starts a dynamic programming recursion to obtain the most feasible path relative to the constraints of the problem and the specific cost function chosen. A visualisation of this is shown in Figure 2.10 below.

It is important to note that the dynamic recursion would be where the significant computational expense lies. It gets around this issue by evaluating the cost of each edge "lazily". That is, by relaxing the constraints of the problem to make the cost evaluation of the edge relatively time efficient. In the short term, this does impact optimality of the paths. However Janson et al. [10] shows that the final path converges towards the optimal solution thanks to the asymptotic optimality of FMT*.

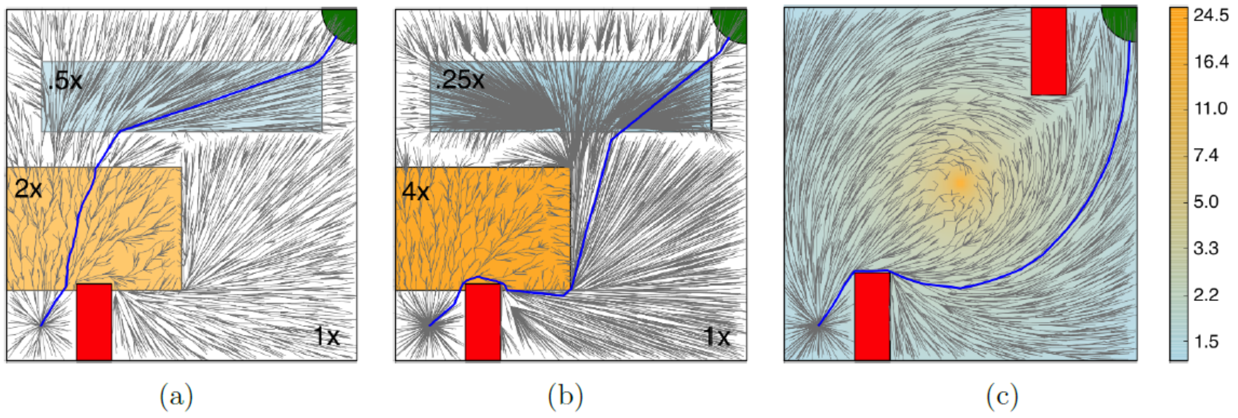


Figure 2.10: Paths returned by FMT* in 3 different scenarios. Each scenario varies in its cost environment, depicted by the quadrilaterals of different cost multipliers. The last figure shows a gradient based cost [10].

It is here that we shall explore the concept of asymptotic optimality in more detail. The notion was first introduced by Frazzoli et al. [46] as a method to ensure algorithms such as RRT would converge to the optimal path as the number of samples n approaches infinity, which became the basis for the RRT* algorithm. While this definition was apt for algorithms which were sequential in n, Janson et al. [10] argued that a slightly different definition was required for algorithms which processed all samples at once (i.e. FMT and PRM). They introduced the concept of probabilistic convergence, which was identical in a practical sense for such algorithms that used static sampling of points. In short, the probability of sub-optimal connections formed in the state space would approach zero as n approached infinity. This also allowed them to define a strict upper bound for the rate of convergence in terms of runtime: $O(n^{-1/d+\rho})$. For implementation purposes, it meant that the radius around each point for which to locate neighbours would now be a function of n, and would decrease logarithmically as n would increase. The precise relationship is shown below:

$$r = \gamma(\frac{log(n)}{n})^{1/d} \tag{2.5}$$

$$\text{where: } \gamma > 2(1/d)^{1/d}(\mu(X_{open})/\zeta_d) \tag{2.6}$$

Note that γ in Eqn 3.2 above is a tuning parameter with a strict lower bound. d is the dimension of the free space X_{open}, $\mu(X_{open})$ denotes the volume of the obstacle free space, and ζ_d is the volume of the unit ball in d-dimensional space.

Overall, FMT* has numerous advantages and is very suitable for this style of problem. Namely, its relatively low computational expense makes it ideal for handling non-linear constraints. It has the advantage of being able to add UAV motion constraints, which makes it ideal for solving a high degree-of-freedom problem. It is also asymptotically optimal, meaning we do not sacrifice path quality in search for efficiency. Finally its inherent nature of forming a dynamically growing cost map is beneficial since each path can be bound by a maximum value, and the algorithm can be terminated if this maximum cost is exceeded. This is contrasting to an algorithm such as RRT where each branch of the tree might have a different cost associated with it, meaning some branches may reach the cost limit before others.

However, one point not addressed is how the algorithm deals with additional complexity in the form of non-geometric constraints. We can ensure that sub-optimal connections with respect to the geometric obstacles are kept to a minimum, however adding additional constraints which cannot be modelled in the form of a heuristic could lead to a significant increase in computational time, and drastically reduce the convergence rate of the canonical algorithm. Therefore a point of interest becomes finding a method through which we can narrow down the search space and be able to effectively filter sub-optimal points without having to waste time on expensive collision checks.

2.6 Summary and Justification

This chapter presented a review on several classes of path planning algorithms that have potential to be used in the UAV engine-out problem.

As the problem definition aims to minimise a cost function subject to a set of constraints, the initial thought was to explore optimisation algorithms for their suitability. LQR was deemed not suitable due to its ability to only solve costs which are a quadratic function of the state and control, which is not true for this thesis problem. Evolutionary algorithms have been successful in robot path planning and collision avoidance, however computational runtime remains a hurdle in applying it to the UAV engine-out problem.

The next class of algorithms discussed were Local Reactive Control algorithms. Artificial Potential Field has been used to solve path planning problems with obstacle avoidance, however it assumed a convex obstacle field and also had the issue of getting stuck in local minima. Receding Horizon Control with the MILP framework, while having made strides in improving the runtime of a standard MILP optimisation, also requires similar assumptions to be made about the obstacle space.

Graph search algorithms were introduced as fundamental trajectory planning methods which guarentee optimality and completeness, however are not feasible when applied by themselves. As an extension, alternate route graph search was analysed as a means to use standard search algorithms but continually change the quality of the paths to eventually yield a feasible solution to the constrained problem. Out of the three analysed, the alternate routes for road networks algorithm was deemed suitable, and had been used previously for a very similar problem for helicopters.

Finally, Sampling Based Planners were assessed as the most suitable methods due to their vast exposure in the path planning field. While not guaranteed to be optimal, they are probabilistically complete and provide an efficient means to deal with non-convex constraints, which is the inherent difficulty in the UAV engine-out problem. RRT and its variants have already been heavily explored for UAV motion and very similar applications to this thesis. PRMs on the other hand have not been utilised due to difficulties in integrating UAV motion constraints and relatively high comptuation time. FMTs are an unexplored method for this style of problem and have numerous advantages over PRMs and RRTs.

Ultimately, I see FMTs as the most suitable approach for the UAV engine-out problem. Since it is such a recent method, exposing it to a much more complex problem instance to see how it scales with non-convex obstacles will be explored. The canonical FMT* method cannot be implemented alone since the resulting computation time of the dynamic program would be too high. Therefore modifications need to be made to the search method to make it more computationally efficient. One of the ways I wish to approach this is to map the obstacle space in an effort to narrow the search space. This was an idea motivated by Toggle PRM, which had only attempted such a task with a geometric obstacle space. Therefore the advancement in state-of-the-art comes through exposing this extension to a much more complex non-geometric obstacle environment.

CHAPTER 3 BACKGROUND

The literature review introduced some of the potential methods that could be used to solve this thesis problem. The conclusion drawn was that FMT, and more specifically a variant called FMT*, could be used to produce a computationally efficient algorithm for returning a feasible trajectory for the UAV. This chapter is designed to give the reader sufficient background knowledge about the standard FMT* methods and its general applications to path planning.

3.1 FMT* Algorithm

The canonical FMT* algorithm can be used to solve the standard path planning problem described in Chapter 2, which is finding a path from a start to goal position while satisfying specific problem constraints. A high level description of the canonical form consists of three key features [10]:

- Static sampling of the search space with neighbours connected using disk-connected graphs.
- Graph construction and graph search occurring simultaneously.
- Using dynamic programming recursion with "lazy" cost determination.

These features share similarities but also differ significantly from the more standard SBP algorithms PRM and RRT. Similar to PRM, FMT* samples the state space all at once, with the number of points specified by the user. Another similarity is that neighbours are determined using a fixed region around each node. This radius can be specified, or can be determined as a function of the sample size as in the case of FMT*.

While making use of static sampling, the algorithm closely resembles RRT by searching and constructing the path simultaneously to form a dynamic tree structure. This means that graphically, we expect the state space of the RRT and FMT* to look similar.

Finally, a distinguishing feature of FMT* is that it uses dynamic programming to assess all nodes that have been explored by the planner, contrasting to RRT which uses a fundamentally greedy approach to construct its tree. FMT* recursively locates nodes and stores them in a separate "fringe" data structure, which was also done in Dijkstra's algorithm. By sorting this fringe based on a "lazily" evaluated cost, it finds the next best node to expand towards in a relatively efficient manner. A high level pseudo code description is given in Algorithms 1 and 2.

Algorithm 1 FMT*

```
1: Input:
2: $C$ ←state space
3: $C_{obst}$ ←obstacle space
4: $x_{init}$ ←initial state
5: $x_{goal}$ ←goal state
6: Output:
7: $V$ ← visited nodes from start to goal
8:
9: $V$ ←addNode($x_{init}$)
10: $X_{open}$ ←sample($C$)
11: $curr \leftarrow x_{init}$
12: while $curr \neq x_{goal}$ || $isEmpty(X_{fringe})$ do
13:     $v$ ←findNeighbours($curr, X_{open}$)
14:     for $x \in v$ do
15:         if collisionCheck($x, C_{obst}$) then
16:             $C$ ←addEdge($curr, x$)
17:             $X_{fringe}$ ←addNode($x$)
18:         else
19:             $X_{open}$ ←remove($x$)
20:         end if
21:     end for
22:     $curr$ ←removeMin($X_{fringe}$)
23:     $V$ ←addNode($curr$)
24: end while
25: return $V$
```

Algorithm 2 Trace Path

```
1: **Input:**
2: $V \leftarrow$ visited nodes
3: **Output:**
4: $P \leftarrow$ final path from start to goal
5:
6: $P \leftarrow$addNode($V(end)$)
7: $curr \leftarrow V(end)$
8: while $curr \neq V(start)$ do
9:     $curr \leftarrow$getPrev($curr$)
10:    $P \leftarrow$addNode($curr$)
11: end while
12: return $P$
```

We see in Algorithm 1 that we have three special categories of nodes which are continually maintained. They are as follows:

- X_{open}: consists of the sampled points from the search space. Every point in the search space except for the start and goal positions will start in X_{open}. These points do not necessarily have to lie outside C_{obst} and any invalid points will be removed as we progress through the algorithm.

- V: a list of visited nodes which is updated each iteration of the *while* loop. Note that this list will track both the node and its predecessor, i.e. the node which initially formed an edge to it.

- X_{fringe}: a sorted list of nodes that have been seen by the visited nodes, but have not been visited yet. This means they have edges that link to them already in the state space. The list is sorted from lowest to highest cost, which will depend on the relevant cost function used.

For clarity, a visualisation of the three categories is shown in Figure 3.1 below.

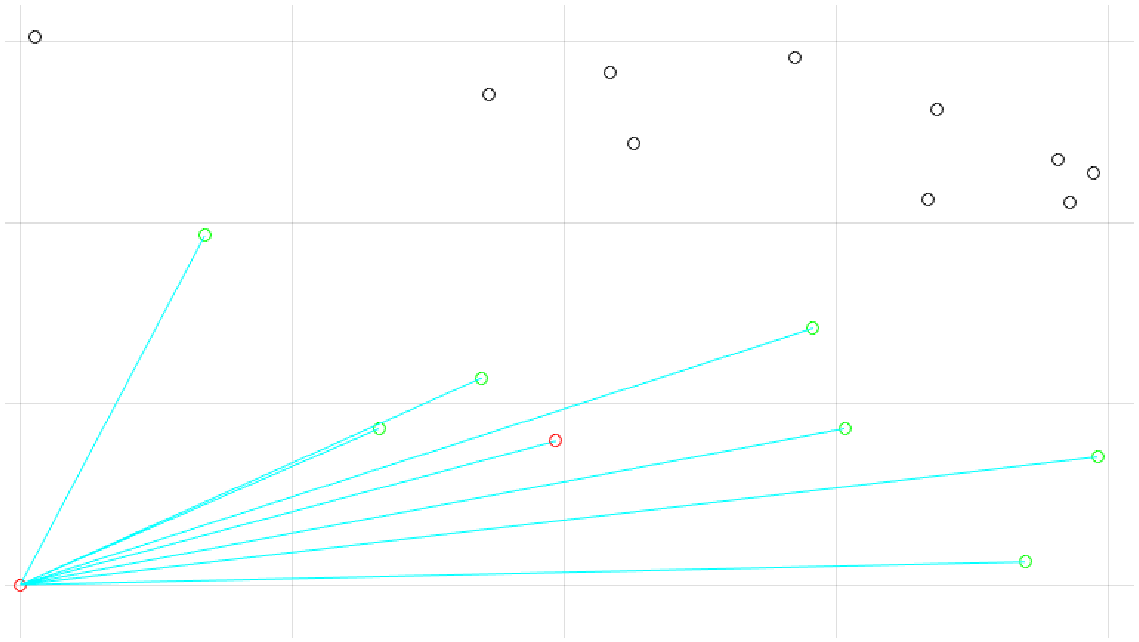


Figure 3.1: FMT* algorithm showing the different categories of nodes. X_{open} is shown in black, V is shown in red, X_{fringe} is shown in green.

Algorithm 1 contains the body of the whole algorithm and will search the entire state space until it has either reached the goal or until a valid path to the goal cannot be formed. We begin by statically sampling the state space and place all points into X_{open}. A traversal variable *curr* is set to track the latest node added to V. After this, we begin the main loop of the algorithm which will find and connect valid neighbours to the current node, and add them to X_{fringe}, which we keep in sorted order. We then extract the first node from the fringe, which by definition has the lowest cost, and visit this node. Note that the algorithm will terminate if X_{fringe} is empty, meaning that we have no potential nodes to visit next and therefore no valid path exists to the goal.

Algorithm 2 is required because simply returning a list of visited nodes V from Algorithm 1 does not provide a path from start to goal. We must trace back using the predecessor nodes we stored in V to extract said path. This is done simply by iterating from the last visited node, which is the goal node assuming that Algorithm 1 terminated successfully, and continually tracking back through predecessors until the start node is reached.

There are several important primitives particularly in Algorithm 1 which deserve special mention due to specific implementation procedures. They will now be explained in detail.

sample: There are numerous ways in which the state space can be sampled. Ultimately the goal is to achieve adequate coverage such that when we develop a graph and search through it, we can ensure that there is at least one valid path from start to goal. Usually for complex problems sampling strategies can be biased towards certain parts of the state space. However for this canonical case using a uniform distribution is both effective and simple to implement, since it has an inbuilt method in MATLAB.

findNeighbours: There are generally two approaches to finding connection points around a sampled node. Firstly, we can specify a number of desired neighbours and then perform a *knn* search, which involves finding the closest k nodes, typically measured in terms of Euclidean distance. The second option is a range search where we specify a horizon distance d around which the vehicle can see. As such, it can only expand towards a node within this horizon. This second approach is utilised since it provides a more realistic scenario for a real life vehicle, and it also provides a means through which we can optimise the sampling, which was discussed in Section 2.5.3.

collisionCheck: A straight line edge between two nodes is checked for its validity in C_{free}. If any part of the edge crosses C_{obst}, then the edge is not feasible and not added to the graph. It should be noted that the difficulty of performing this check varies greatly depending on what type of obstacles or constraints are placed on the problem. For convex geometric obstacles such as regular polygons, a mathematical definition of the closest distance between a line and a curve can usually be used, and this makes the check quite efficient. However for non-convex obstacles the edge usually has to be discretised and each point must be checked individually. This is what makes obstacle avoidance a computationally expensive procedure since every potential edge at worst has to be cross-checked with every constraint.

removeMin and addNode for X_{fringe}: The removeMin procedure only involves extracting the first element from X_{fringe}, which will by definition be the node with the lowest cost since the nodes are kept in sorted order. However it is worth mentioning that when nodes are added to the fringe, the way their cost is determined is in a “lazy” manner. This means that when a node in X_{open} is assessed as a valid X_{fringe} node, the cost stored for that node is one which relaxes the constraints from C_{obst}. A simple example is choosing a cost function which minimises Euclidean distance of the path, which allows for a straight line lazy cost to be determined. A visual representation of this is seen in Figure 3.2. Note that if a node is found which already lies inside the fringe, it will be updated in the fringe if the new cost is lower than the current cost.

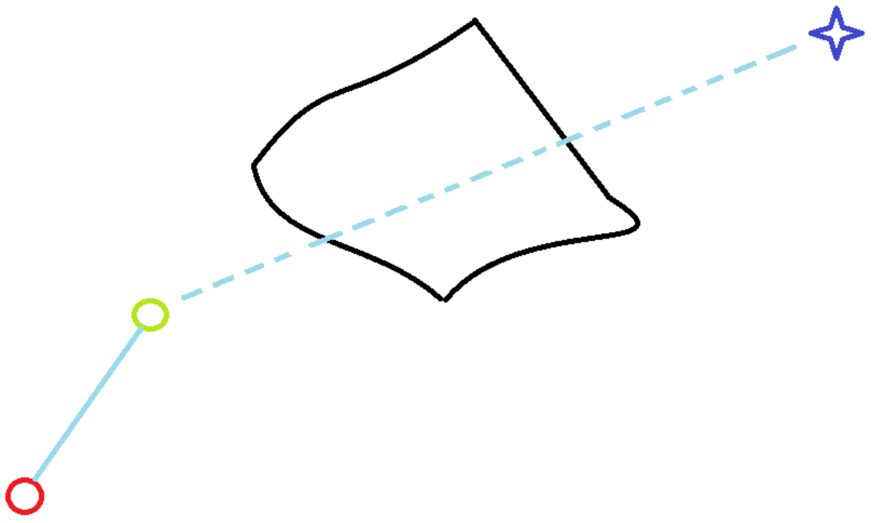


Figure 3.2: Current node (in red), detects a fringe node (in green), and the cost of that fringe node assumes a “lazy” path moving towards the goal (blue star) which relaxes the constraints from the obstacle space (in black).

3.2 FMT* Implementation

Using these methods, two examples of standard 2D path planning problems are simulated in Figures 3.3 and 3.4. A grid of 1000 by 1000 was filled using a clustered obstacle space consisting of several circles with varying radii (with a maximum and minimum bound of 50 and 100 units respectively). A total of 1000 points were sampled statically in the state space. The cost we are trying to minimise is the Euclidean distance of the overall path. The range set for the *findNeighbours* method was 100 units. Note that Figure 3.4 has a much more cluttered obstacle space and is therefore representative of a more computationally complex problem.

We can see the tree structure in Figure 3.3 not expanding too far or being diverted significantly from the optimal path, shown in red. Hence, this is an example of a case where using the lazy straight line heuristic proved effective and did not result in many sub-optimal connections. Contrastingly, we see in Figure 3.4 that the FMT* tree has spread across the majority of the grid in search of the optimal path. Therefore, the heuristic is less effective in this scenario.

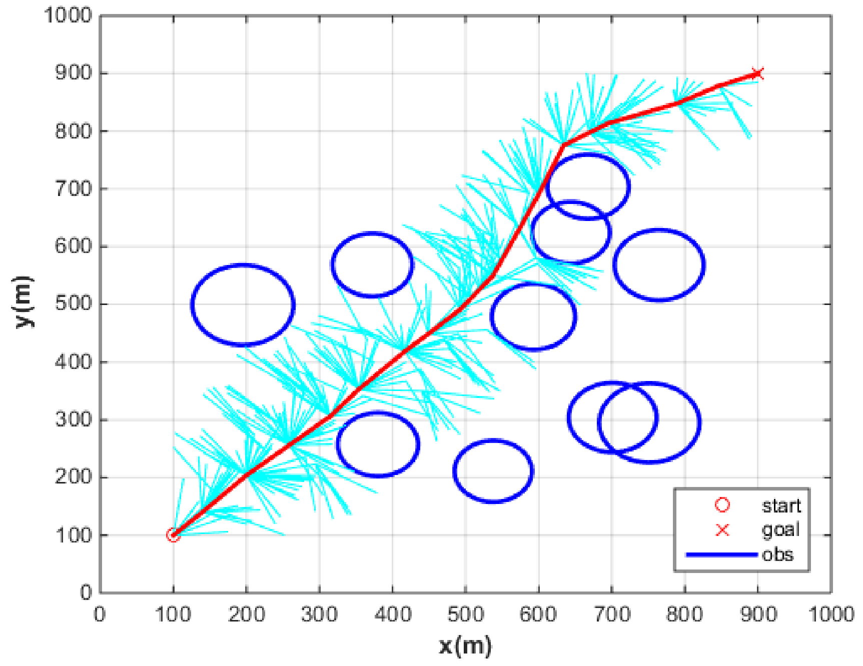


Figure 3.3: Search space after FMT is implemented with the final path in red. The vehicle's initial position is the red circle at the bottom left corner, the goal position is the red cross at the top right.

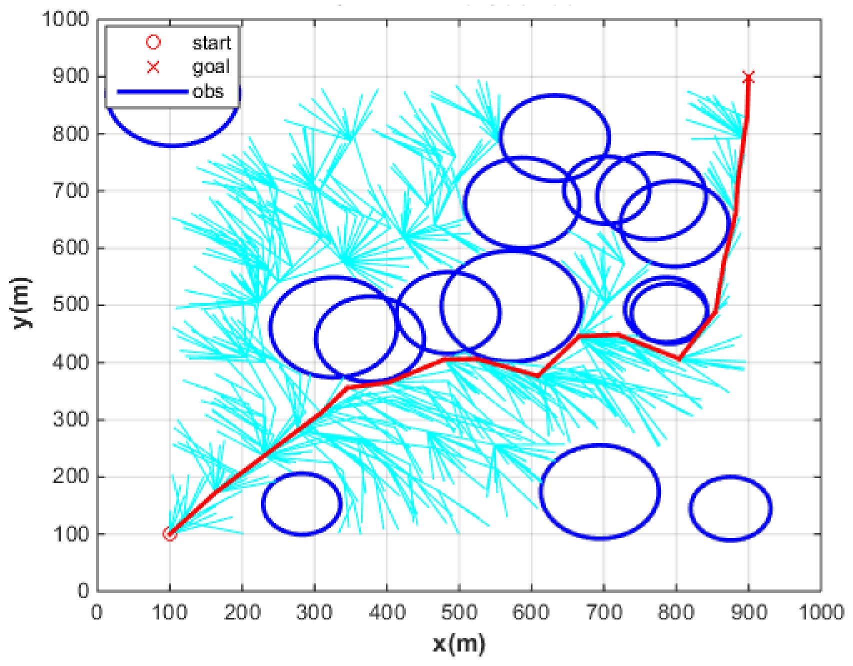


Figure 3.4: Search space after FMT is implemented with the final path in red. We can see that in contrast to Figure 3.3, the tree structure spreads across the majority of the grid in search of the optimal path.

3.3 Summary

This chapter gave the reader an insight into the FMT* algorithm, and its applications to simple 2D path planning problems. We saw the benefits of using its dynamic programming framework combined with a lazy heuristic to form a growing cost-to-reach tree within the search space. Certain differences between this algorithm and PRM and RRT were also noted in this chapter. The application to two varying problems in complexity showed how the FMT* tree is directly impacted by the straight line heuristic, which in some cases can cause a performance drop.

CHAPTER 4 PROBLEM FORMULATION

This chapter will introduce the state space to the reader to familiarise themselves with a visual representation of the problem. Next, the main objective and constraints will be discussed, including both the spatial and temporal components. Finally, a section will be devoted to the UAV model, including motion constraints, the effects of wind on the vehicle, and assumptions being made while the aircraft is in flight.

4.1 State Space

The problem is represented in a 3D cartesian plane as can be seen in Figure 4.1. The state space contains 5 degrees of freedom: $x(m), y(m), z(m), \omega(^{\circ})$ and $t(s)$. The first three are translational components of the vehicle, the fourth is the heading angle of the vehicle, and the last is time. Note that the size of the grid is completely arbitrary, but should be kept in scale with the velocity parameters of the UAV, which will be discussed within the UAV model sub-section. The $z = 0m$ plane will act as our sea-level reference, and hence when the vehicle reaches this plane, it's motion has ended. Note also that wind will be accounted for in the x and y axes, and once again this will be discussed more in the UAV model sub-section.

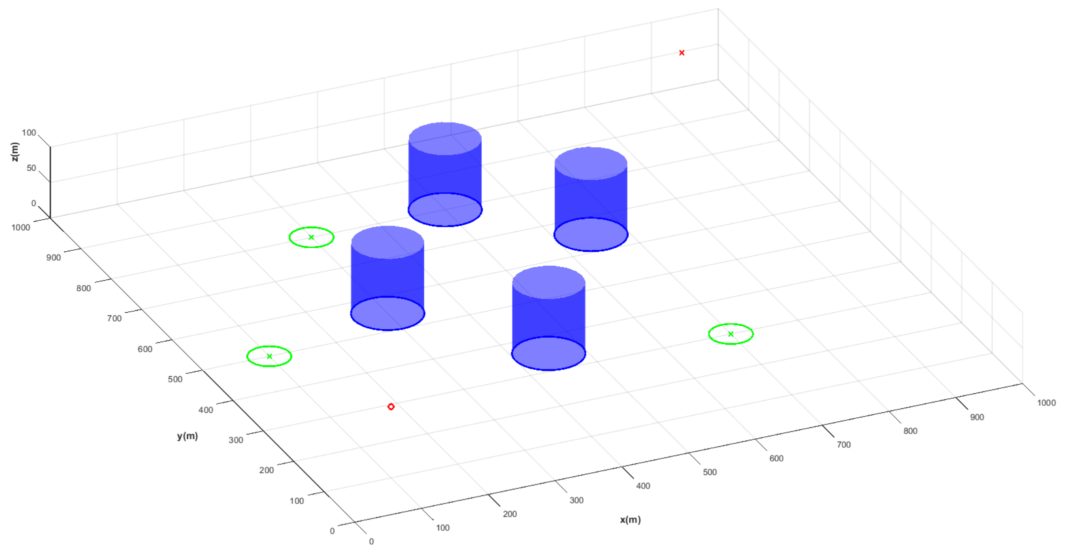


Figure 4.1: 3D representation of state space

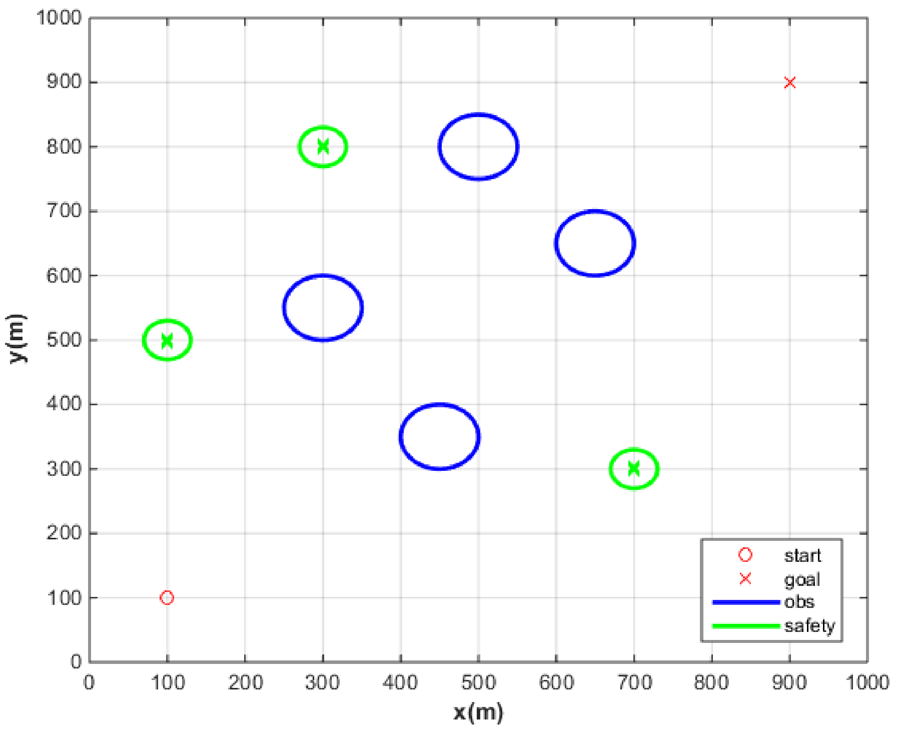


Figure 4.2: 2D representation of state space

4.2 Objective and Constraints

$$\text{minimise } J = \sum_{E \in P} t_{trans} + t_{rotate} \tag{4.1}$$

$$\text{subject to: } E \notin C_{temp} \cup C_{obs} \text{ for all } E \in P \tag{4.2}$$

The main objective of this path planning problem is minimise the total time spent while travelling from a start node to a goal node, subject to a set of constraints. This time includes both the translational (t_{trans}) and rotational (t_{rotate}) components, and is the sum along all the edges E belonging to the final path P. There are two classes of constraints associated with this thesis problem. We have spatial constraints (C_{obs}) in the form of obstacles as well as temporal constraints (C_{temp}) related to the UAVs proximity to safety zones located in the state space.

4.2.1 Spatial Constraints

The geometric obstacles used in this thesis problem will consist of cylinders randomly generated in the state space, which are given varying radii. These cylinders represent an inaccessible area for the UAV which can represent impassable terrain, or even areas of extreme turbulence which are not safe for the UAV to enter. Also note that these cylinders effectively extend across the entire z axis, and therefore for the sake of collision checks, they can be considered circles in the 2D space as seen in Figure 4.2

4.2.2 Temporal Constraints

The green circles at sea-level in Figures 4.1 and 4.2 represent safety zones. These are predefined random areas where the UAV is safe to land. The parameters and number of these zones is completely arbitrary, however to make it realistic, instead of a standalone point in the state space, the UAV is allowed to get within a radius of the centre.

At any point along the UAVs path, the worst case scenario is that an engine failure occurs and therefore a contingency check is required to make sure it can reach a safety zone. Note that the UAV must be able to reach at least one safety zone from every point along the path for it to be deemed valid. For this thesis problem, the contingency measure is done in terms of time. The UAV will be given a velocity during its descent, and once an emergency path to a safety zone has been determined, the time taken to reach said safety zone can be calculated and cross checked against the time to land.

Finding an emergency path from a point to a given zone can be done in several ways. The most straightforward is to assume a straight line trajectory, however the presence of obstacles makes this invalid. Instead, we can utilise the path generated by FMT* to find the shortest path to each zone.

4.3 UAV Model

The real motion of a UAV is subject to extremely complex non-linear flight dynamics which are subject to lift, drag, effects of control surfaces, and many more factors. In addition, we would typically find that all degrees of freedom of the vehicle are coupled, meaning a change in one would result in a change in the other. However this is usually controlled using modern day auto-pilot systems which would manipulate control surfaces on the UAV to minimise these coupling effects. Therefore we make the assumption in this problem that the translational and rotational degrees of freedom (x, y, z, ω) are decoupled. Note that the problem also has time(t) as an additional degree of freedom separate to the UAVs motion. Finally, the UAV shall be considered as a point mass travelling through the state space.

4.3.1 Modes of Motion

For the sake of this thesis there shall be two modes of motion considered:

1. Cruise condition
2. Glide descent

Firstly for the cruise case, we assume that the UAV is travelling at a constant velocity and constant altitude. This will model the UAVs normal, fully functional motion from its starting point to its goal. Note that this motion is influenced by wind in the x and y axes. The heading angle ω will be unaffected by the wind.

The next mode is the glide descent, which models the emergency engine-out case for the UAV. Since the vehicle has effectively lost power, it shall be put into a descent at a constant velocity. Note that it can still manoeuvre in the x, y and ω axes, however the z axis motion is uncontrollable. This velocity will match a realistic glide ratio for a UAV, scaled to the dimensions of the state space. This motion is also effected by wind in the x and y axes. Once the UAV has been put into this mode, it can never return to its cruise state, and after a period of time will reach sea-level ($z = 0m$). At this point, the trajectory of the UAV has officially reached its end point.

4.3.2 Wind Disturbances

Consider the UAVs motion to translate along any arbitrary vector $\vec{x}$ in 3D space. Consider a wind velocity vector of $\vec{w}$. Via vector projection [47], we can calculate the effective velocity of the wind travelling along $\vec{x}$ as such:

$$\vec{w}_{proj} = \frac{\vec{w}.\vec{x}}{|\vec{x}|} \tag{4.3}$$

This velocity can either have a propelling or retarding effect on the true velocity v_{net} which will be a function of the projected wind velocity and the unaffected UAV velocity v :

$$v_{net} = v + \vec{w}_{proj} \tag{4.4}$$

The above is sufficient to model the UAV motion in a constant wind field. However, a more interesting simulation would be to include a varying wind field where the wind velocity is a function of the specific location of the UAV. In this case, we are not able to utilise one single projection for the entire direction vector $\vec{x}$, but rather we will have a projection for each individual point along the vector. For simplicity, an average net velocity will be determined by splitting the vector up into n segments and averaging the individual net velocities:

$$v_{net} = \frac{\sum\limits_{i=0}^{n} v + \vec{w(i)}_{proj}}{n} \tag{4.5}$$

4.3.3 Motion Constraints

The motion constraints can be split into two primary classes: translational and rotational. Each has a set of characteristics equations which can be utilised to determine the overall cost of each motion.

Translational:
Consider translational motion along a vector $\vec{x}$ which experiences a wind vector v_{wind}. Equa-

tions 4.3 to 4.6 summarise the cost determination of the UAV along this vector.

$$|v_{xy}| = k_1 \tag{4.6}$$

$$v_z = k_2 \tag{4.7}$$

$$v_{net} = v_{xyz} + v_{wind} \tag{4.8}$$

$$\therefore t_{trans} = \frac{|\vec{x}|}{v_{net}} \tag{4.9}$$

Note that k_1 and k_2 represent a positive and negative constant for the x, y planar and z glide velocities respectively. They should be set relative to a realistic UAV glide ratio.

Rotational:
Consider rotation from an initial vector $\vec{x_1}$ to a vector $\vec{x_2}$ where the angular displacement between the two is $\Delta\omega$. Equations 4.7 and 4.8 describe the cost determination using these parameters.

$$|\dot{\omega}| \leq \dot{\omega}_{max} \tag{4.10}$$

$$\therefore t_{rotate} = \frac{\Delta\omega}{|\dot{\omega}|} \tag{4.11}$$

Equation 4.7 is an example of a differential non-holonomic constraint on the path. The term non-holonomic means that the UAV system is path dependant, meaning a path can be planned to a goal which returns the same values, but not necessarily the same states. For this thesis, we consider a finite turning rate $\dot{\omega}$ and bound it by a maximum value.

An assumption being made here is that the UAV will perform the turn on the spot before performing any translational motion. This is what allows the cost terms t_{trans} and t_{rotate} to be added to determine the overall cost.

4.4 Summary

This chapter fully formulated the thesis problem which was introduced in Chapter 1. The first component of this was familiarising the reader with the state space, both in 2D and 3D, and the main objective. Inside the state space were the spatial constraints, in the form of impassable cylindrical terrain. Also included were the safety zones, which are the central components to forming the temporal constraints of the thesis problem. Finally, the UAV motion constraints, assumptions, and modes of motion were explained, including how wind disturbances would impact the flight path.

CHAPTER 5

UAV ENGINE-OUT SOLUTION USING FMT*

This chapter will introduce the three main algorithmic approaches used to solve the UAV engine-out problem. Firstly, the underlying framework for the solution is provided by examining the Brute Force FMT* algorithm. Following this, two novel extensions are proposed in order to address the limitations of the Brute Force approach in terms of computational runtime. These are: the "Path Reconstruction" algorithm, which is introduced to improve the efficiency of temporal constraint checking, and the "Obstacle Cluster" algorithm, which maps the complex obstacle space and uses it to filter out nodes which are highly likely to fail the temporal constraint based on their proximity to other invalid nodes.

5.1 Brute Force FMT*

FMT* will involve forming edges between nodes and verifying that these edges are feasible with respect to certain constraints. For this thesis problem, the Brute Force FMT* implementation involves two main procedures to check each edge in the state space:

- Collision Checking
- Contingency Checking

Each verification step will now be examined in more detail.

5.1.1 Collision Checking

Any edge in the 3D Cartesian space must not intersect with any of the cylinders which lie in the obstacle space. However, as outlined in the Problem Formulation, the cylinders extend across the entire z axis. This means that the edge can be projected onto the x, y axes and we can treat the problem as a line segment intersecting with a circle in 2D space. This problem has a well defined mathematical solution and is provided for the reader in Appendix A. It is important to note that each edge must be cross-checked with every single obstacle (considering the worst case scenario), and therefore the obstacle checks contribute to a significant portion of computational runtime. Our goal while optimising the Brute Force algorithm will be to minimise these checks.

5.1.2 Contingency Checking

A contingency check for the UAV is when an engine malfunction is simulated, and consequently it is forced into an unpowered descent towards the ground and must reach one of the pre-defined safety zones within a certain period of time. For the Brute Force FMT* implementation, this involves performing the following steps for each edge in the state space:

1. Discretise the edge into equal length segments.
2. Find the closest safety zone.
3. Perform a collision check for the lazy path towards the zone.
4. Formulate a full FMT* search towards the zone.

The first step is to discretise the edge into several equal length segments. This is an important step since the algorithms accuracy is heavily dependent on how fine the discretisation factor is. For the general case it was concluded that a linear spacing between points was most suitable and convenient. Note that there are more complex discretisation strategies available which might yield more suitable results for specific problem instances.

For each node in the line segment, the next step is to find the closest safety zone in terms of travel time. For all zones, a lazy cost to reach it is determined by adding the time to rotate towards the zone and the translational time to travel there in a straight line. This will give an overall cost for each zone, out of which the minimum is chosen. A visual representation of this is shown in Figure 5.1.

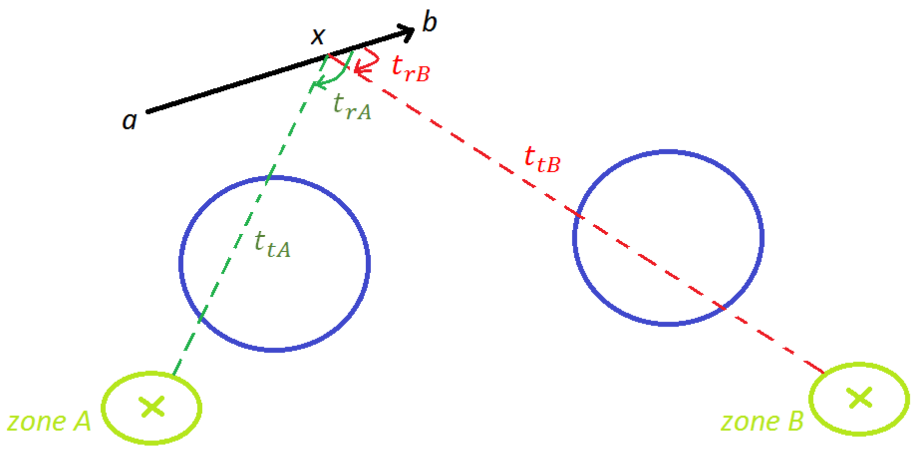


Figure 5.1: Visualisation of determining a lazy path where we choose the best zone via calculating the total time required to travel to it, assuming a straight line trajectory and ignoring obstacles (shown in blue). This includes time to rotate towards the zone (t_r) and the translational time (t_t). The zone with the lowest time value is returned.

Determining the validity of each lazy path involves recalling the glide descent mode of motion described in Section 4.3.1. The UAV will have a threshold time to reach its zone, subject to how long it takes to fall to the ground. By definition, the lazy straight line path is the shortest path the UAV can take towards that zone. Hence, if the UAV is unable to reach its closest zone along this path, the node shall be deemed invalid as it cannot satisfy the temporal constraint. If the straight line path comes within the time limit, we then perform a collision checking procedure to validate the edge against the obstacle space. If the collision check returns positive, then the node is valid, and we can continue by repeating the process for the next node along the line segment.

If however the straight line path collides with an obstacle, we must then perform a full FMT* search towards the relevant zone. Once complete, the optimal path is returned and its cost is cross checked with the same threshold value used to evaluate the lazy path. Once again, if it meets the threshold, the node is valid and we can continue to the next node in the line segment. If not, then the node is invalid and the entire edge will be removed from the state space. This FMT* search will follow the pseudo-code given in Algorithms 1 and 2.

Since the obstacle space may be cluttered, it is very likely that the full FMT* algorithm will have to be used for the majority of the nodes along each line segment, as the straight lazy line paths will rarely be viable. When we consider that this must be repeated for each and every edge in the search space, it becomes extremely computationally expensive. Therefore this is the main area where the “Path Reconstruction” and “Obstacle Cluster” algorithms will aim to reduce the computational runtime.

Algorithm 3 below summarises the four steps involved in each contingency check. In Line 9 the edge is discretised, and then for each node the closest safety zone is found in Line 11. If this lazy path is valid against the time threshold, it is then tested with a collision check. Finally, if the lazy path crosses any obstacle, we obtain the optimal path P from the FMT* search and validate this cost value. An edge can only be marked as valid in Line 22 if every node along it is also valid.

Algorithm 3 Contingency Check

```
 1: Input:
 2: $C_{obst}$ ←spatial obstacle space
 3: $C_{zone}$ ←safety zones
 4: $a$ ←start of edge
 5: $b$ ←end of edge
 6: Output:
 7: $validity$ ← boolean for whether passed or failed constraint
 8:
 9: for $x \in$ linearly discretised edge $\vec{ab}$ do
10:     $z$ ←find closest safety zone (lazy path)
11:     if lazy cost to $z >$ threshold then
12:         return $validity = false$
13:     end if
14:     if collision check of lazy path is valid then
15:         continue
16:     end if
17:     $P$ ←path from FMT* search towards $z$
18:     if cost of $P >$ threshold then
19:         return $validity = false$
20:     end if
21: end for
22: return $validity = true$
```

5.1.3 Overall Algorithm

The obstacle and contingency checking are now incorporated into the final Brute Force FMT* algorithm shown in Algorithm 4.

Algorithm 4 Brute Force FMT*

```
 1: Input:
 2: $C_{obst}$ ←spatial obstacle space
 3: $C_{zone}$ ←safety zones
 4: $x_{init}$ ←initial state
 5: $x_{goal}$ ←goal state
 6: $V$ ←visited nodes (initially empty)
 7: Output:
 8: $P$ ← visited nodes from start to goal
 9:
10: $X_{open}$ ←sample($C$)
11: $V$ ←addNode($x_{init}$)
12: $curr \leftarrow x_{init}$
13: while $curr \neq x_{goal}$ || $isEmpty(X_{fringe})$ do
14:     $v$ ←findNeighbours($curr, X_{open}$)
15:     for $x \in v$ do
16:         if collisionCheck($curr, x, C_{obst}$) && contingencyCheck($curr, x, C_{zone}, C_{obst}$) then
17:             $C$ ←addEdge($curr, x$)
18:             $X_{fringe}$ ←addNode($x$)
19:         else
20:             $X_{open}$ ←remove($x$)
21:         end if
22:     end for
23:     $curr$ ←removeMin($X_{fringe}$)
24:     $V$ ←addNode($curr$)
25: end while
26: $P$ ←tracePath($V$)
27: return $P$
```

The only discernible difference between Algorithm 4 and 1 is Line 16 where we must perform the contingency check in addition to the collision check. This is a significant addition since this is where the majority of the computational time for the entire program will be spent. As was noted in the previous subsection, the major issue arises when C_{obst} is cluttered around the safety zones C_{zone}. This makes a finding a valid lazy path quite rare, and forces the planner to perform full FMT* searches towards the zones for each node along every edge. This is an appropriate time to introduce the first novel extension proposed in this thesis which is “Path Reconstruction”.

5.2 Path Reconstruction Algorithm

Consider the contingency check described in Algorithm 3. Evidently, it is possible for every single node along an edge to have to build a full path using FMT* towards the same zone. An intuitive approach to make this procedure more efficient is to use the results from previous paths to plan new ones. For example, if we have a node a which has already formed a full path towards a zone z, and we wish to plan a new path from a node x along the same edge, we can simply attach the currently forming FMT* from x onto the pre-built path from a. This concept is visualised below in Figure 5.2.

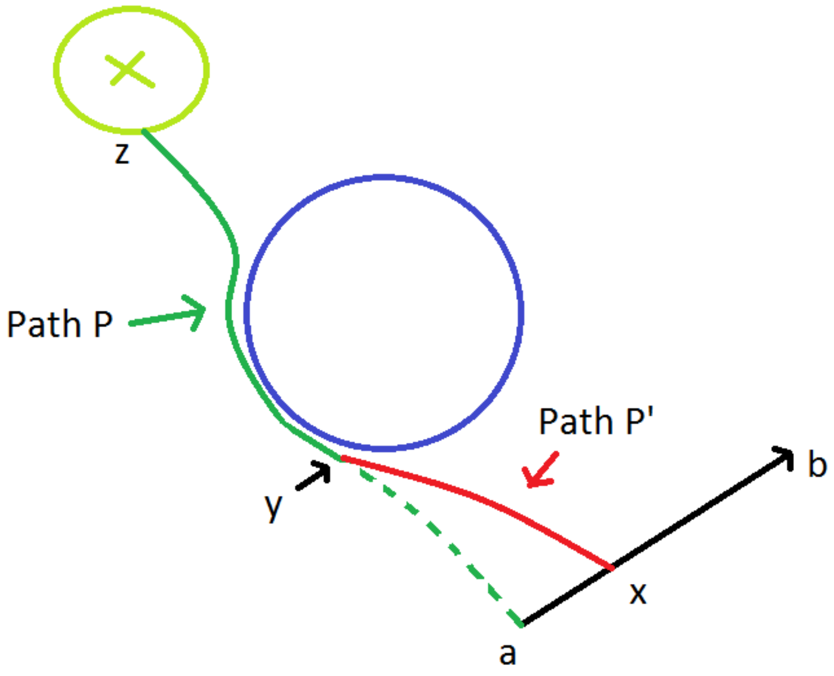


Figure 5.2: "Path Reconstruction" begins to form an FMT from any aribtrary node x towards a goal point z. If a node y which existed along a previously formed path P can be connected to, then instead of completing the FMT from x, we can simply concatenate P' (x to y) and P (y to z).

The benefit with such an approach is that the computational expense of determining the path is reduced significantly since we do not have to iterate the path planner until we reach the final goal. Instead, the path is returned as soon as we have a connection to a pre-existing path and the concatenation of the two paths result in a valid path cost.

One point which should be stressed is that we do not want this concatenation to result in a sub-optimal path. This can result in a node incorrectly being labelled as invalid, and therefore we might ignore an edge which could be a part of the optimal trajectory for the UAV. We can ensure this does not happen by allowing the FMT* search to progress organically and if the expansion leads towards a path, then it will attempt to attach itself to it. The importance of not trivially joining the starting node with a point along P is demonstrated with a counter example, as seen in Figure 5.3 below. The red path P'' is formed by the FMT* since it is the most optimal route, while the green path PP' is clearly a sub-optimal route towards the zone.

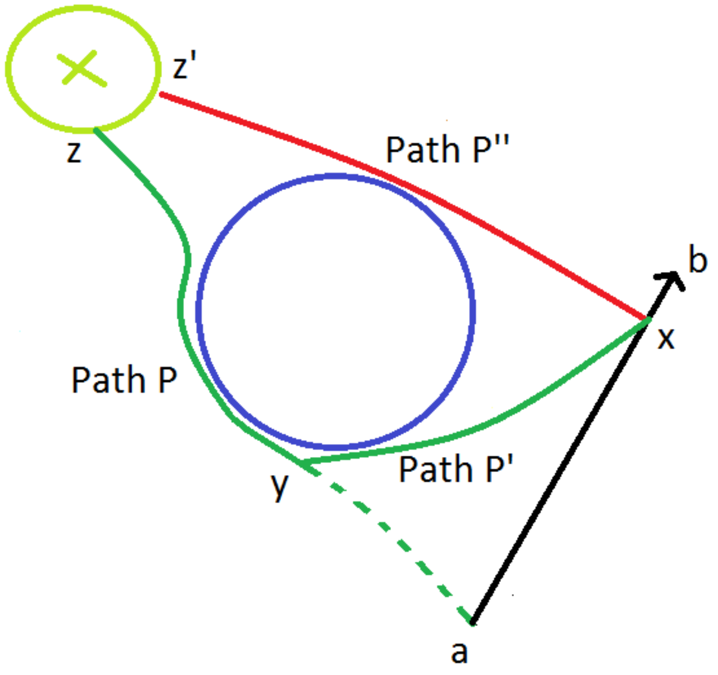


Figure 5.3: "Path Reconstruction" will form the FMT naturally and follow path P'' (in red) since it is the most optimal path towards the zone. The concatenation of P' and P (in green) is a sub-optimal path resulting from a naive connection between the node x and a node y.

The pseudo code representation of "Path Reconstruction" is shown in Algorithm 5 below.

Algorithm 5 Path Reconstruction

```
1: **Input:**
2: $P$ ←path which already exists to the same zone
3: $X_{open}$ ←sampled points from the main algorithm
4: $C_{obs}$ ←spatial obstacle space
5: $x_{init}$ ←node along the edge being checked
6: $x_{zone}$ ←closest zone node
7: **Output:**
8: $validity$ ← boolean for whether passed or failed constraint
```

```
 9: $curr \leftarrow x_{init}$
10: while $curr \neq x_{zone}$ || $isEmpty(X_{fringe})$ do
11:     $P' \leftarrow$path so far up until $curr$ node
12:     $v \leftarrow$findNeighbours($curr, X_{open}$)
13:     for each valid connection $n$ in intersection of $v$ and $P$ do
14:         $P'' \leftarrow$concatenatePaths($P', P$)
15:         $cost \leftarrow$combined cost of $P''$
16:         if $cost < threshold$ then
17:             return $validity = true$
18:         end if
19:     end for
20:     for $x \in v$ do
21:         if collisionCheck($x, C_{obst}$) then
22:             $C \leftarrow$addEdge($curr, x$)
23:             $X_{fringe} \leftarrow$addNode($x$)
24:         else
25:             $X_{open} \leftarrow$remove($x$)
26:         end if
27:     end for
28:     $curr \leftarrow$removeMin($X_{fringe}$)
29:     $V \leftarrow$addNode($curr$)
30: end while
31: $P \leftarrow$tracePath(V)
32: if $cost$ of $P < threshold$ then
33:     return $validity = true$
34: end if
35: return $validity = false$
```

The overall structure of Algorithm 5 resembles the FMT* algorithm, however the key difference lies in Lines 13 to 19. For the set of neighbours found by the *findNeighbours* method, we determine the ones which also reside in the input path P. Note that P must be a path which leads to the same zone as the current node x. If no previous path exists to the same zone, then the full path using the Brute Force FMT* must be formed. For each node n which is common between v and P, we then compute the cost of the concatenated path between the initial position, the intersection point, and the final zone position (Lines 15 and 16). If this cost is lower than the threshold dictated by the glide descent, then we have a valid connection and the node passes the temporal constraint (Line 35).

In summary, "Path Reconstruction" is an algorithm which aims to reduce the computational runtime of the temporal constraint checks by utilising paths that have been formed previously to form new paths. This way, we can ensure that a full FMT structure is not required for each node along an edge. The algorithm still maintains the optimality of the canonical FMT* by allowing the FMT to grow organically, and not bias the search towards pre-existing paths. This way, we get the best of both worlds, as we maintain the quality of paths, while reducing the number of collision checks significantly. This will be clearly demonstrated in Section 6.

5.3 Obstacle Cluster Algorithm

The "Obstacle Cluster" algorithm is the second novel extension proposed in this thesis to improve the computational runtime of the main program. The goal is to be able to filter invalid nodes based on their vicinity to other invalid sets of nodes within the non-geometric temporal space.

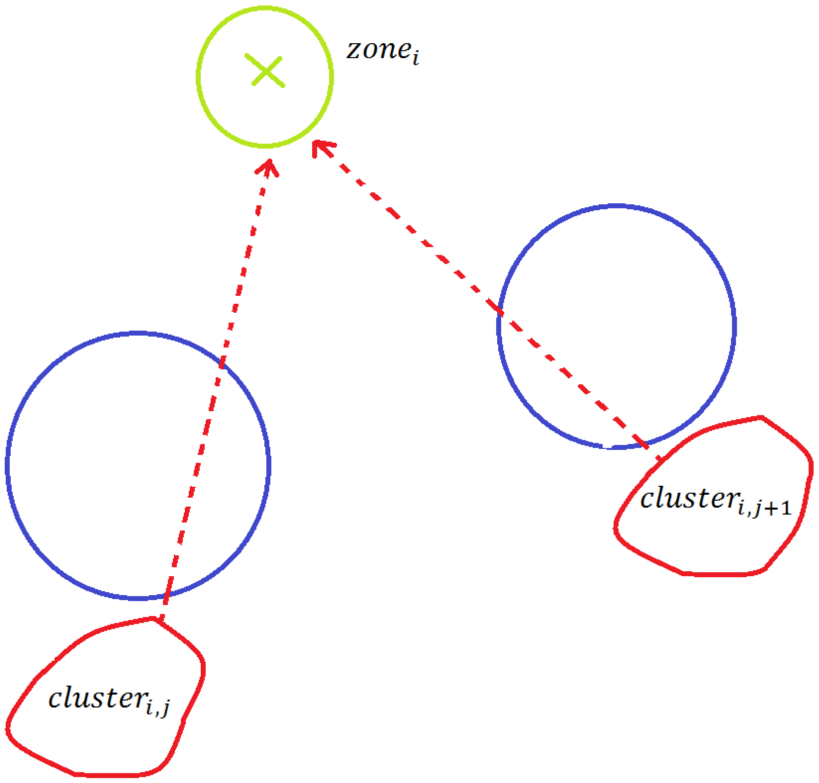


Figure 5.4: Visualisation of the obstacle space as a "cluster" of points. Each cluster has an associated zone which it targets, but cannot reach due to failing the temporal constraint requirement.

The inherent difficulty in mapping a non-geometric space is providing a clear definition of the requirements a node must satisfy in order to be placed in said space. For a geometric space, this is relatively simple, as we have well defined obstacles that are known before runtime.

Therefore this kind of space will be static, as opposed to a non-geometric space which will dynamically form as we progress through the FMT. Let us now analyse what the obstacle space in the UAV engine-out problem will look like.

Figure 5.4 above visualises the obstacle space as a set of clusters. Each cluster will be comprised of nodes which target the same zone, however have failed the temporal constraint and are therefore invalid. Each zone can have multiple clusters, as seen in Figure 5.4.

Recall that there were two ways identified as per the contingency check (Algorithm 3) for a node to fail the temporal constraint:

1. Its lazy cost by forming a straight line towards the zone exceeds the threshold.
2. Its true cost by forming a full path or using "Path Reconstruction" exceeds the threshold.

We shall only consider nodes of the latter case to belong to these clusters, even though technically nodes which fail the lazy cost test still belong to the overall obstacle space. There are two main reasons for this approach. Firstly, we should consider the worst case cost of processing nodes into these clusters, which adds additional overhead and runtime to the core algorithm. Nodes which fail the first case can occur in very large numbers, and as will be explained later in this section, when nodes are identified as being candidates for a cluster, they must perform a linear search for the nearest cluster. If we were to include the first class of nodes, then not only will the size of each cluster increase, but the number of clusters will also, leading to an increase in computational time. Secondly, filtering nodes based on purely on their lazy cost is essentially a geometric test for their proximity to the closest zone. Our goal here is to map the non-geometric space, which is defined by the second class of nodes.

Now that we have a clear definition of what nodes will lie in these clusters, we can now explore how each cluster is dynamically formed. As well as linking clusters to specific zones, each cluster will also be identified by a "local minimum" node and a radius $r_{cluster}$. The "local minimum" is a node whose lazy cost value to reach the associated zone is the lowest of all the nodes in the cluster. The radius $r_{cluster}$ defines a local search zone around the "local minimum" for which new nodes can be detected and added to the cluster. This is visualised in Figure 5.5 below.

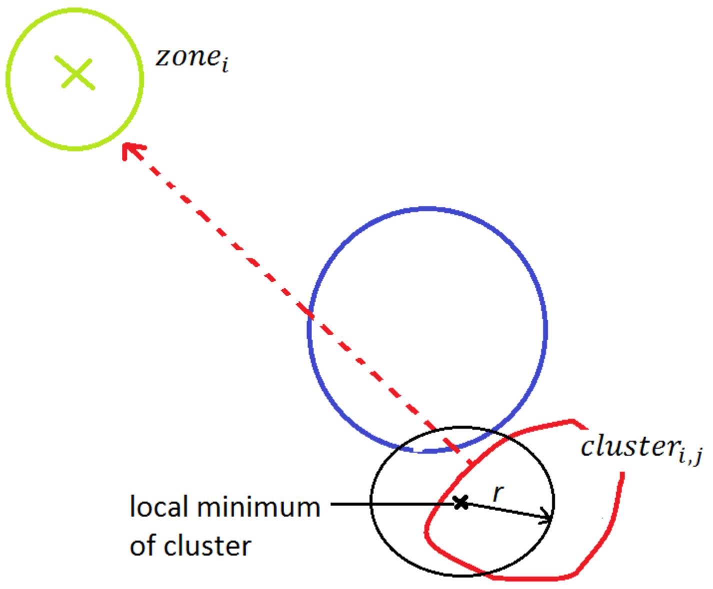
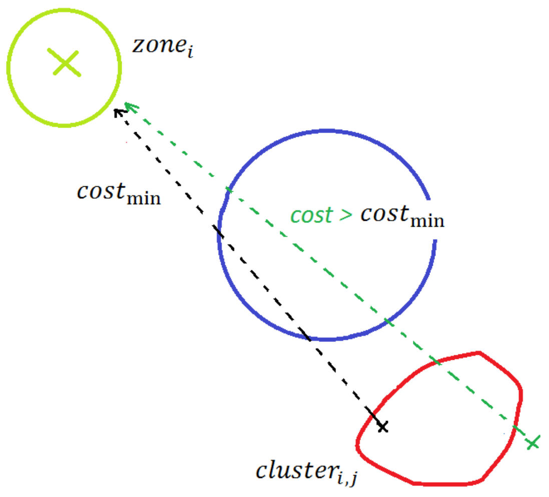


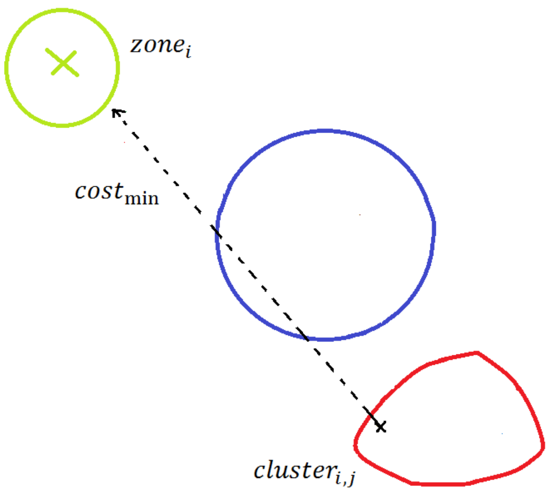
Figure 5.5: A cluster (shown in red) is defined by a "local minimum" node (black cross) and a radius r. The circle shown in black is the search zone via which new nodes can be considered to join the cluster.

Now we shall consider how nodes are added to the cluster, and how the cluster will dynamically grow over time. This process has been visualised in Figures 5.6 and 5.7. As in Figure 5.6, if a new node is detected which has a higher lazy cost to the zone than the current minimum, it will be lazily added to the cluster. We therefore achieve an improvement in runtime since the planner will mark the edge as invalid without having to perform the contingency check.

In Figure 5.7, we see that a node with a lower lazy cost is detected. In this case, we do need to verify that the node is in fact invalid. To do this, we perform the full FMT* search or use the "Path Reconstruction" algorithm to retrieve a path, and cross-check the path cost against the threshold. If the path cost is greater than the threshold, we can add it to the cluster as in Figure 5.7(b). If not, then this node passes the temporal constraint and therefore does not belong in the obstacle space.

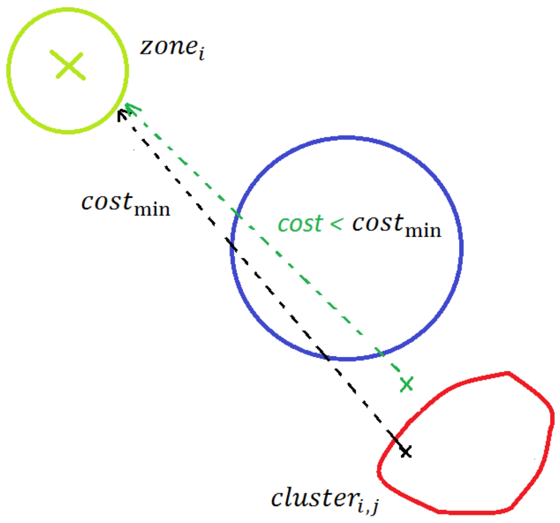


(a) New node detected

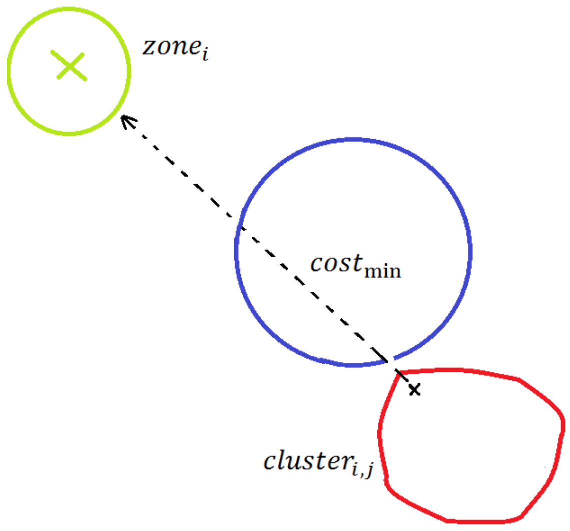


(b) New node added lazily

Figure 5.6: Cluster finds a new node (in green) which has a higher cost than the current local minimum (black). The cluster than adds this node and keeps its current local minimum.



(a) New node detected



(b) New node added as local minimum

Figure 5.7: Cluster finds a new node (green) which has a lower cost than the current local minimum (black). The cluster than adds this node and replaces its local minimum.

The one case not captured by the Figures above is if a node is discovered which does not lie within the search zone of another cluster. In this case, we must go through the full contingency check and verify that it is invalid. If it is, then we form a new cluster with this node as the “local minimum”.

The overall “Obstacle Cluster” algorithm can be simplified into two methods: “Check Cluster” and “Add Cluster”. A pseudo-code description of both algorithms has been provided in Algorithms 6 and 7 below.

Note that two separate algorithms are used in this way because information must be passed between the different components of the contingency check. In "Check Cluster" we return a series of flags to fully define if and how a node should be added to the existing clusters. This includes $flag$ (Line 8 of Algorithm 6) which will decide whether an FMT* search is required to verify the nodes validity. These outputs are then passed on to "Add Cluster" which will add the invalid nodes to their associated clusters. "Add Cluster" is also charged with making new clusters (Line 11 of Algorithm 7) in case a node was found not in the vicinity of any existing clusters.

Algorithm 6 Check Cluster

```
1: Input:
2: $x \leftarrow$node being checked
3: $cost_x \leftarrow$lazy cost of the node to the zone
4: $zone_x \leftarrow$closest zone to $x$
5: $C_{cluster} \leftarrow$clusters
6: $r_{cluster} \leftarrow$radius of cluster
7: Output:
8: $flag \leftarrow$ flag for whether path verification is needed
9: $C_{add} \leftarrow$ cluster to which the node should be added
10: $flag_{min} \leftarrow$ flag for whether node should replace current min
11:
12: for each cluster $C$ in $C_{cluster}$ belonging to $zone_x$ do
13:     if $x$ lies within $r_{cluster}$ of $C$ then
14:         if $cost_x < cost_C$ then
15:             return $flag = true, C_{add} = C, flag_{min} = true$
16:         else
17:             return $flag = false, C_{add} = C, flag_{min} = false$
18:         end if
19:     end if
20: end for
21: return $flag = true, C_{add} = empty, flag_{min} = true$
```

Algorithm 7 Add Cluster

```
1: Input:
2: $C_{cluster} \leftarrow$clusters
3: $x \leftarrow$node to be added
4: $cost_x \leftarrow$lazy cost of the node to the zone
5: $C_{add} \leftarrow$cluster to which $x$ should be added
6: $flag_{min} \leftarrow$flag for whether node should replace current min
7: Output:
8: $C_{cluster} \leftarrow$new clusters
```

```
9: if $C_{add}$ is empty then
10:     add new cluster to $C_{cluster}$ with $cost_x$ as minimum
11: else
12:     if $flag_{min} = true$ then
13:         add $x$ to $C_{add}$ and replace current minimum with $cost_x$
14:     else
15:         add $x$ to $C_{add}$
16:     end if
17: end if
18: return $C_{cluster}$
```

5.3.1 Choice of Cluster Radius

One variable which is not set to a specific value during the algorithm is the cluster radius $r_{cluster}$. This is because it can be varied based on the goals of the user. Consider a cluster with a relatively large radius, as in Figure 5.8. We can see that because it will accept the node in green which is located on the other side of the obstacle, the node might have a longer lazy cost than the current minimum, but still have a valid true path to the zone. This is an example of when the algorithm falters, since a valid node is ignored and incorrectly added to the cluster instead. It will however result in a significant improvement in computational runtime.

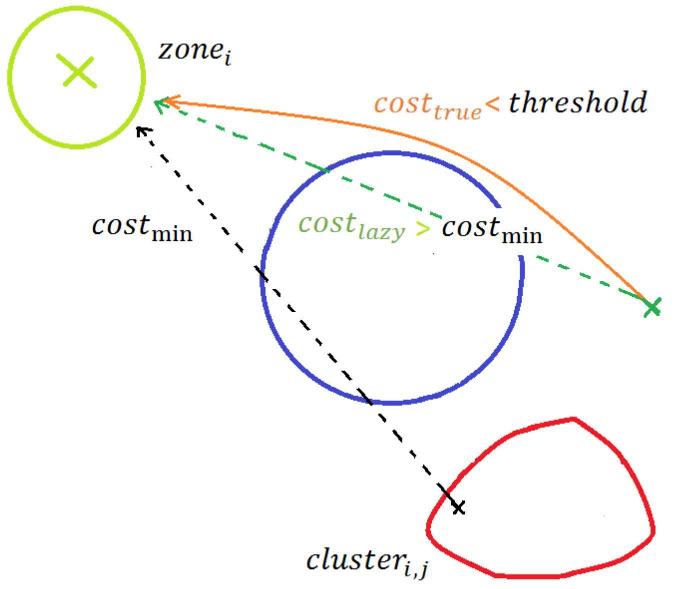


Figure 5.8: The cluster radius finds a node (in green) which has a higher lazy cost to the goal, but its true cost makes it valid. Therefore, it will be added to the cluster even though it does not lie in the obstacle space.

On the other hand, having too small a radius will result in very little computational gain, since very few nodes will be considered for joining the clusters, and as a result we will be performing almost as many full constraint checks as the canonical algorithm. Therefore choosing $r_{cluster}$ is a balancing act based on the desire for correctness, or speed.

The complexity of the obstacle space is also an important factor in deciding $r_{cluster}$. The more complex an obstacle space, the more unreliable a lazy cost heuristic is, and therefore it might be safer to choose a smaller radius. In contrast, when we have a sparse obstacle space, using a relatively large radius might not result in a significant loss in path quality.

One heuristic which can be used to decide $r_{cluster}$ is motivated by Figure 5.8. We can see that when most nodes in a cluster take roughly the same path around an obstacle, the lazy cost heuristic is more accurate. It is only when $r_{cluster}$ stretches across the diameter of the obstacle when we typically see problems arising. Therefore we can restrict $r_{cluster}$ based on the average diameter of the obstacles D_{avg} as such:

$$r_{cluster} < D_{avg} \tag{5.1}$$

Note that this is only an approximate starting value, and $r_{cluster}$ may need to be tweaked further based on the specific problem instance.

5.4 Summary

This chapter thoroughly described the core algorithm used to solve the UAV engine-out problem. The Brute-Force implementation of FMT* was first presented, after which two novel extensions, “Path Reconstruction” and “Obstacle Cluster”, were introduced to the reader. The former was a method to reduce the amount of full FMT* searches required for each edge by attaching currently forming paths with previous ones. The latter was a unique approach at utilising the non-geometric obstacle “clusters” to filter out points which were likely to be invalid.

CHAPTER 6 RESULTS AND DISCUSSION

This chapter will present the results obtained from applying the algorithms in Chapter 5, which were developed to solve the UAV engine-out problem. The main focus of the MATLAB simulations will be to show the performance difference between implementing the Brute-Force implementation alone, compared to using combinations of both the "Path Reconstruction" and "Obstacle Cluster" algorithms. Two simulations are used for this analysis, with the second being much more complex than the first in terms of the density of the obstacle space. This will provide an idea as to how the core algorithm, including the algorithmic extensions, scale with the complexity of the problem instance. Finally, the effect of adding wind disturbances on the path planning problem will be analysed, with the focus being on assessing the robustness of the overall solution. Note that all simulations were processed using MATLAB R2014b (64-bit) on a 3.50GHz Quad-Core Intel i5-4690k processor.

6.1 Simulation A (Low Complexity)

This simulation is designed to the give the reader an analysis of a relatively simple path planning problem so they can get acquainted with what a standard solution using the core algorithm looks like. It will also give the reader a clear visualisation of how the algorithmic extensions impact the final path. The simplicity of the simulation is mainly reflected through the sparseness of the obstacle space, which makes the spatial and temporal constraint checks relatively time efficient when compared to Simulation B. The simulation parameters are provided below in Table 6.1.

Parameter	Value
No. of Samples	1000
Neighbour Radius	100 m
No. of Obstacles	8
Avg. Radius of Obstacles	55 m
v_{xy}	10 m/s
v_z	1 m/s
$\dot{\omega}_{max}$	90 °/s

Table 6.1: Simulation A Parameters

The implementation results will now be provided for four separate categories. They are: the Brute-Force implementation, "Path Reconstruction", "Obstacle Cluster", and finally the full algorithm including both extensions.

6.1.1 Brute-Force Implementation

Figures 6.1 and 6.2 show the Brute-Force solution for Simulation A in 2D and 3D respectively.

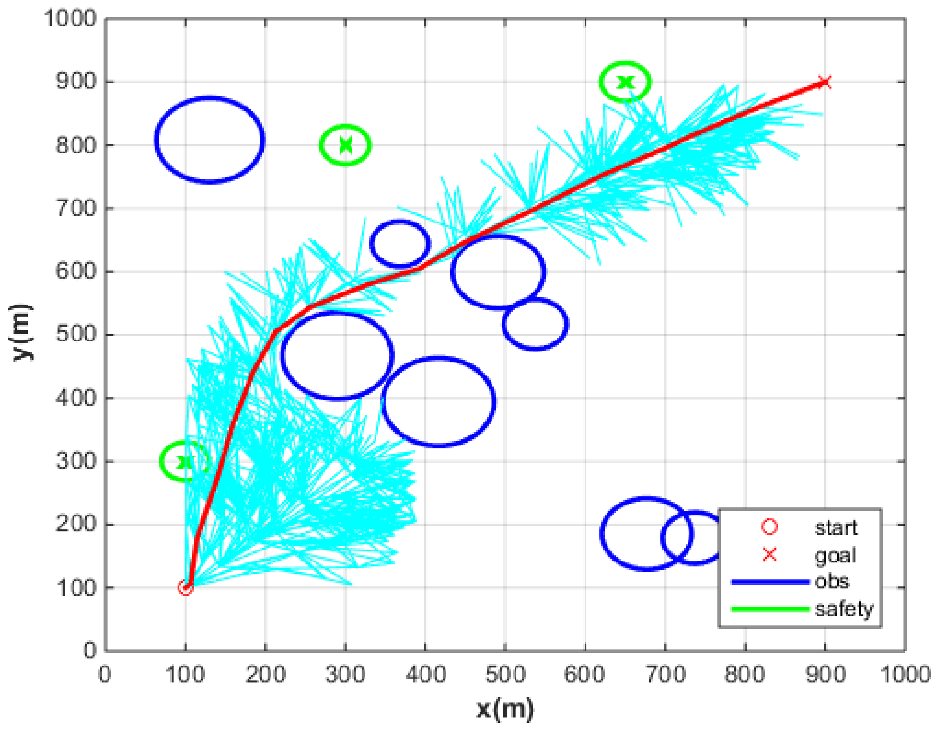


Figure 6.1: Simulation A: 2D visualisation of FMT* structure (light blue) along with final path (red) for Brute-Force implementation.

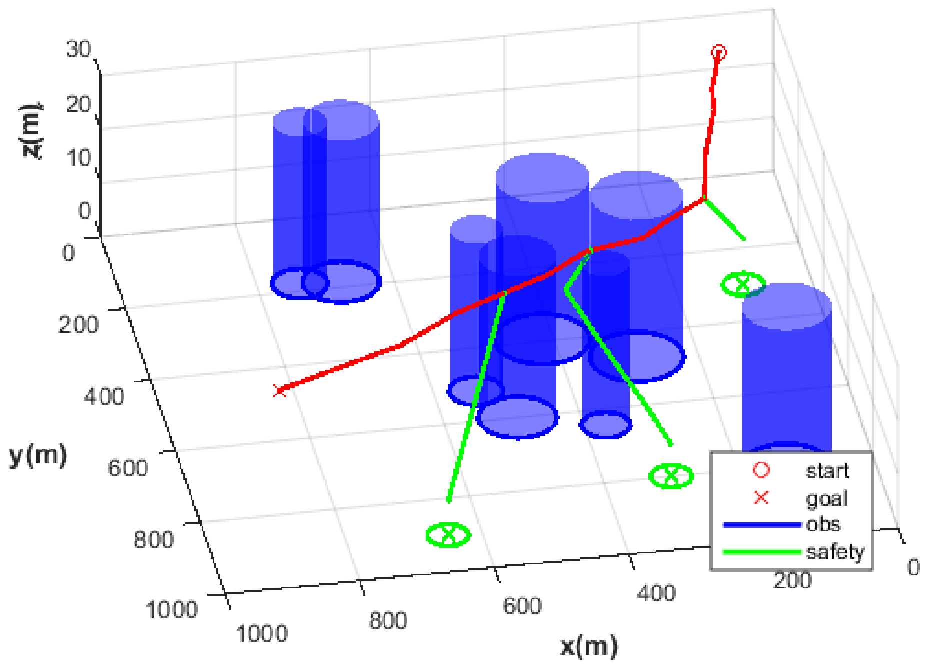


Figure 6.2: Simulation A: 3D visualisation of final path (red) returned for Brute-Force implementation. Also shows examples of emergency paths (green) taken at three different locations along the path to each of the safety zones.

Firstly, note in the 2D visualisation the clear effect of the straight line lazy heuristic for cost determination. We can see that the FMT initially tries to expand directly from the start node towards the goal but this proves unsuccessful due to the presence of the obstacle space. This causes noticeable cluttering towards the bottom left corner of the grid. However, we see a change once the UAV reaches roughly halfway up the y axis, where the straight line path towards the goal becomes viable. From here onwards, the FMT structure is significantly less dense.

The 3D visualisation is indicative of the effect of the temporal constraints on the final path. The green paths are shown as examples of emergency-out scenarios where the UAV would have to divert from its main path (in red) and descend towards the nearest safety zone. Note that the furthest left and right green paths are examples of when the lazy path to the zone was successful, and therefore a simple straight line trajectory could be taken by the UAV. The middle path on the other hand was formed by a full FMT* search towards the zone due to the presence of an obstacle.

Table 6.2 below shows the performance of the Brute-Force algorithm in terms of path quality and efficiency.

Result	**Value**
Final path cost	125.51 seconds
Translational time	122.90 seconds
Rotational time	2.61 seconds
Collision checks	167915
Computational runtime	57.50 seconds

Table 6.2: Simulation A: Brute-Force algorithm results

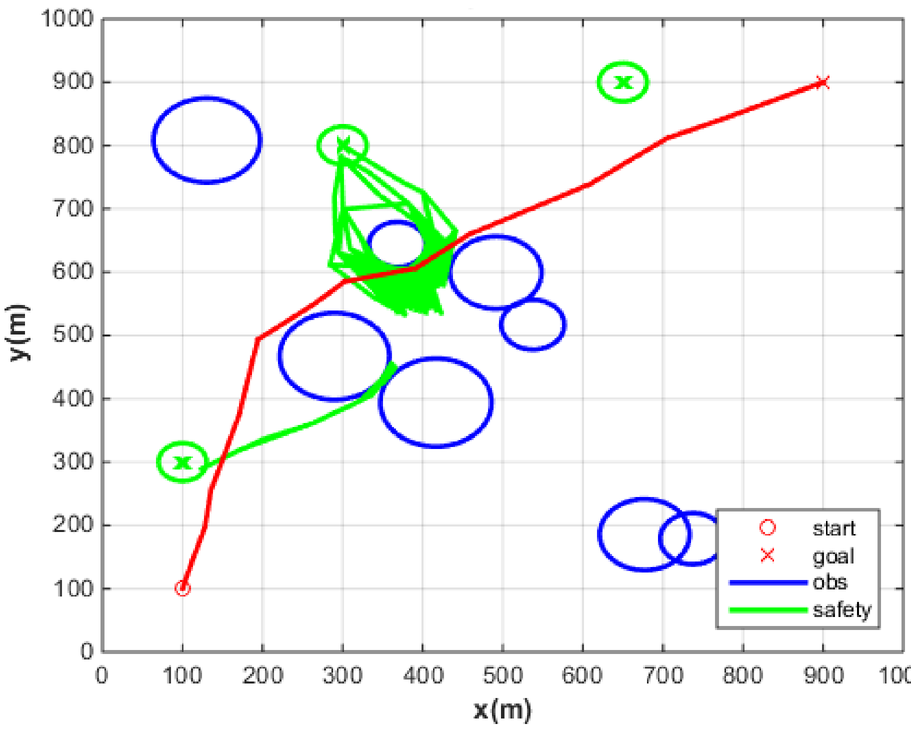
The final path cost characteristics described in Table 6.2 are indicative of the optimal path using this specific sampling of nodes, since the Brute-Force implementation inherently assures this. Note that computational efficiency has been measured via a collision check count and the computational runtime. Collision checks in particular are indicative of not only how many obstacle checks existed for forming the main FMT, but also for the paths that needed to be formed for the temporal constraints of the problem. Therefore, they are an extremely effective means of comparison between implementations, and slightly more reliable than computational runtime which can be effected by external factors.

6.1.2 Path Reconstruction Implementation

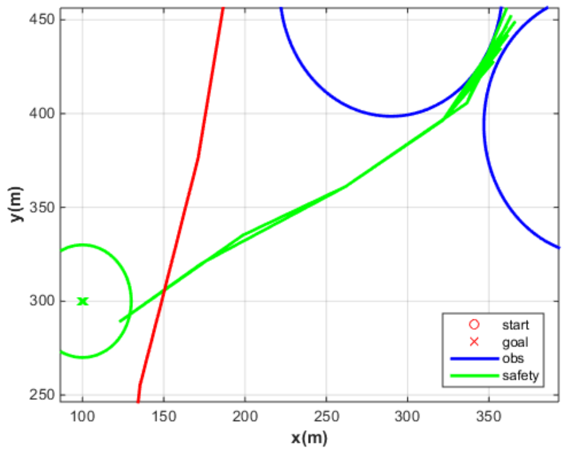
Result	**Value**
Final path cost	125.51 seconds
Translational time	122.90 seconds
Rotational time	2.61 seconds
Collision checks	83510
Computational runtime	25.49 seconds

Table 6.3: Simulation A: "Path Reconstruction" algorithm results

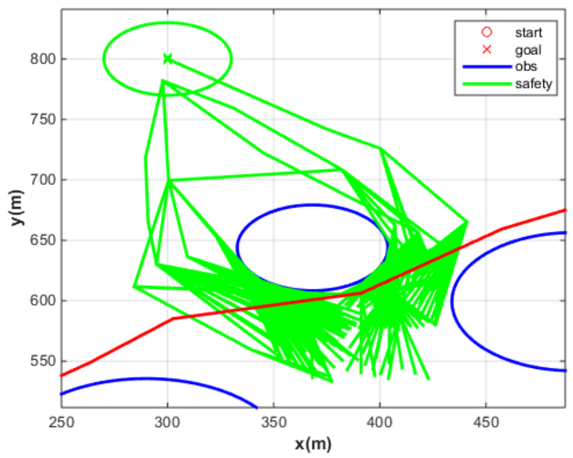
The final results for implementing "Path Reconstruction" in addition to the Brute-Force implementation are shown above in Table 6.3. We can see that the optimality of the path is still maintained, indicated by the exact same final path characteristics as in Table 6.2. However, where we see a significant difference from Table 6.2 is the computational efficiency. Note the 50.27% reduction in number of collision checks, as well as the 55.67% improvement in runtime. This was predicted in Chapter 5 when describing the methodology behind the algorithm since we are not changing the inherent correctness of FMT*. We are merely searching for nearby paths along the way for which to connect to, and thereby significantly shorten the amount of time spent in the secondary FMT formation. Figure 6.3 shows both the final path returned by the algorithm, as well as the secondary FMTs formed along the way. Note the heavy inter-connection which exists between the paths, which is the main reason for the significant improvement in computational efficiency.



(a)



(b)



(c)

Figure 6.3: Visualisation of secondary FMTs formed by "Path Reconstruction". Figure 6.3a shows the overall path in red, which is the exact same as the one formed by brute-force in Figure 6.1. Figures 6.3b and 6.3c show close ups of how the FMTs intertwine as per the algorithm.

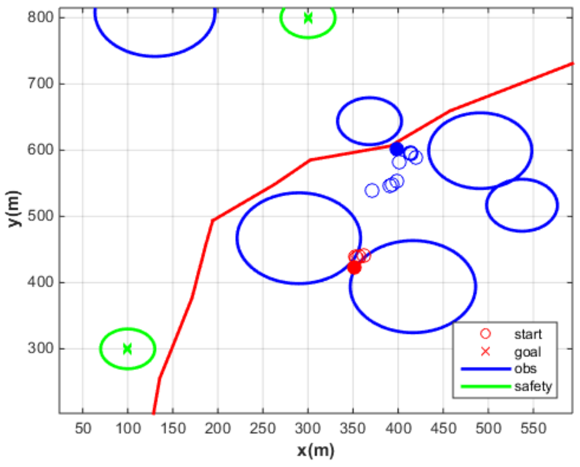
It should be noted that if the secondary FMTs were to be analysed from the Brute-Force implementation, they will intertwine somewhat similarly to that seen above, however each path would be computed individually. This contrasts to the sharing of results from previously formed paths which "Path Reconstruction" allows for.

6.1.3 Obstacle Cluster Implementation

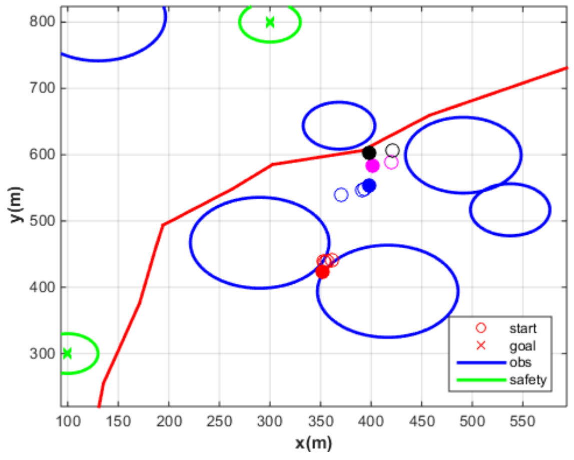
Result	**Value** ($r_{cluster} = 100m$)	**Value** ($r_{cluster} = 25m$)
Final path cost	125.51 seconds	125.51 seconds
Translational time	122.90 seconds	122.90 seconds
Rotational time	2.61 seconds	2.61 seconds
Collision checks	159017	161943
Computational runtime	43.73 seconds	48.72 seconds

Table 6.4: Simulation A: "Obstacle Cluster" algorithm results

The first point evident from Table 6.4 is that there are varied results depending on which cluster radius $r_{cluster}$ is chosen. The heuristic described in Section 5.4.1 means we should choose $r_{cluster} < D_{avg}$, where D_{avg} is the average diameter of the obstacle space. For Simulation A, $D_{avg} = 110m$, therefore we shall set a buffer around this value and consider a maximum $r_{cluster} = 100m$. By using this value, Table 6.4 shows that path cost is not sacrificed in order to achieve a performance improvement. The exact same path is returned as the Brute-Force solution, and the number of collision checks and runtime have reduced by 5.30% and 23.95% respectively. Figure 6.4a below shows a visualisation of the clusters



(a) $r_{cluster} = 100m$



(b) $r_{cluster} = 25m$

Figure 6.4: Visualisation of "Obstacle Cluster" algorithm shown with varying $r_{cluster}$. Note each cluster is shown in a different colour, and the filled circles represent the local minimum of each cluster.

Evidently due to the sparse obstacle space, the relative number of nodes which have been filtered into these clusters is low. This is reflected in the much smaller performance improvement as compared to the "Path Reconstruction" algorithm. We expect that for a more complex obstacle space (such as the one seen in Simulation B), the "Obstacle Cluster" algorithm will have a more prominent impact.

For $r_{cluster} = 25m$ we similarly see no reduction in path quality, since once again the same path is returned. However, we observe a smaller performance improvement compared to the larger cluster radius. For this case, only a 3.56% and 15.27% reduction in collision checks and runtime are observed respectively. This is expected and was predicted in Chapter 5 since a small cluster radius means less points considered to join the obstacle space. The net effect is more points will need to go through the full constraint check as opposed to being efficiently filtered out into clusters. Note also in Figure 6.4b the clear overlapping of the purple, black and blue clusters, which is the effect of setting such a small $r_{cluster}$.

6.1.4 Complete Algorithm Implementation

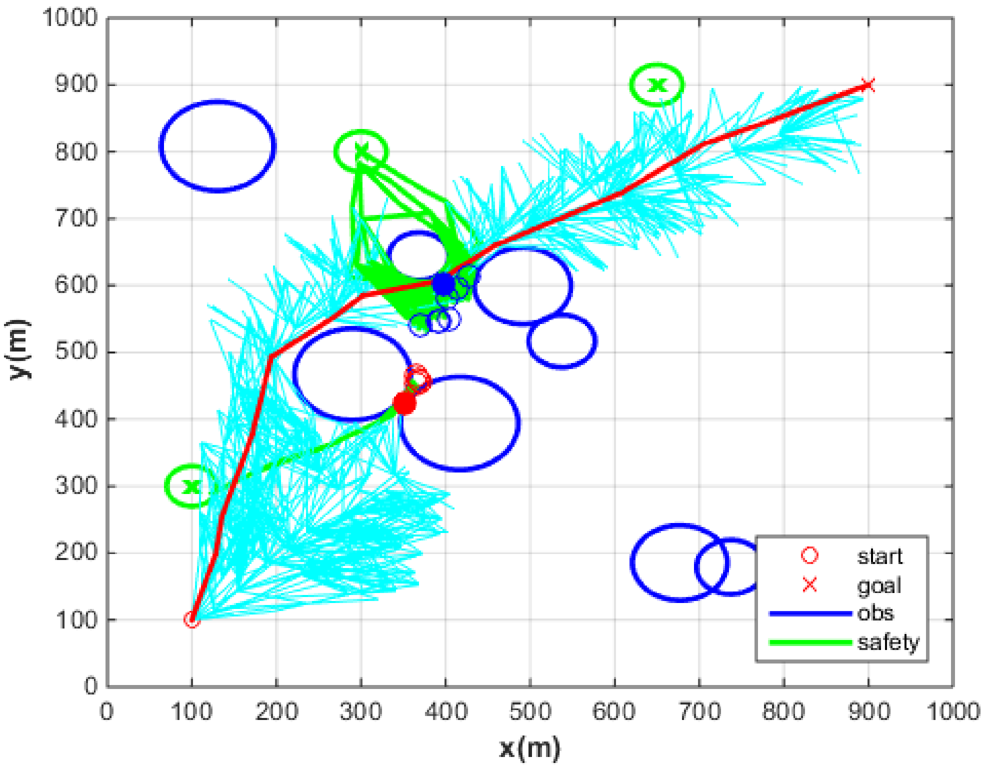


Figure 6.5: Final solution formed by complete algorithm, including both algorithmic extensions. Both the secondary FMTs (in green) and obstacle clusters (scatters circles) are shown along with the final path (red) and FMT (light blue).

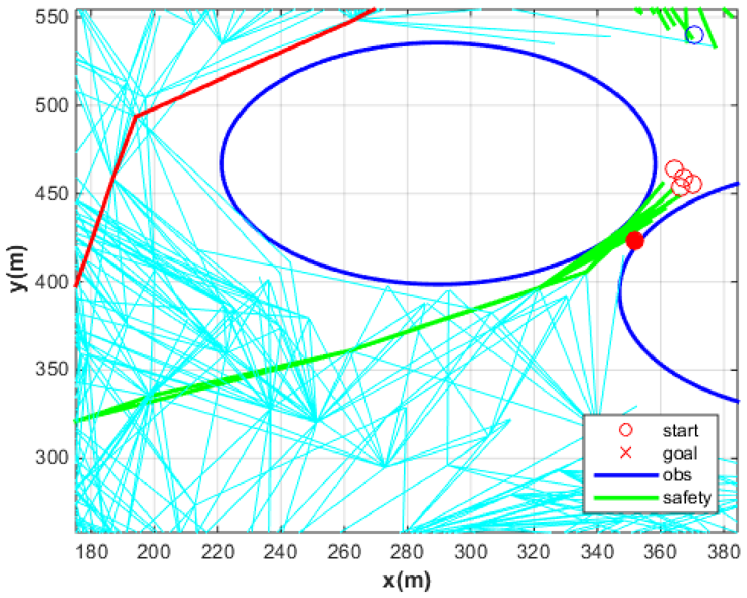


Figure 6.6: Close up look at Figure 6.5 where we see the interaction between the "Obstacle Cluster" and "Path Reconstruction" algorithms.

Result	**Value**
Final path cost	125.51 seconds
Translational time	122.90 seconds
Rotational time	2.61 seconds
Collision checks	76772
Computational runtime	22.96 seconds

Table 6.5: Simulation A: Complete algorithm results with $r_{cluster} = 100m$

The complete algorithm results are observed in Figure 6.5 and Table 6.5. Combining both algorithmic extensions yields roughly the combined improvements of both separately. We see a collision check and computational runtime improvement of 54.31% and 60% respectively. Once again the path cost of the final path is also preserved.

Figure 6.6 shows a close up of the interaction between the "Obstacle Cluster" and "Path Reconstruction" algorithms. We can see that the "Path Reconstruction" does not need to consider the points within the cluster, and therefore all the collision checks associated with checking temporal constraints from the circles in red are ignored. This in turn leads to speed ups within the main FMT since edges towards the cluster are not formed, and therefore the region is largely ignored by the path planner.

6.2 Simulation B (High Complexity)

Simulation A was designed the give the reader a visualisation of the different implementations explored. We could see that for a fairly sparse obstacle space, “Path Reconstruction” was extremely effective, while “Obstacle Cluster” had less effectiveness. Both however made improvements in computational efficiency, while maintaining path quality, and this followed through to the combined algorithm. Simulation B will be a more realistic test of the method capabilities and will explore how these different applications scale with a more complex obstacle space. The simulation parameters are provided below in Table 6.1.

Parameter	**Value**
No. of Samples	1000
Neighbour Radius	100 m
No. of Obstacles	10
Avg. Radius of Obstacles	50 m
v_{xy}	10 m/s
v_z	1 m/s
$\dot{\omega}_{max}$	90 °/s

Table 6.6: Simulation B Parameters

While the parameters seem almost identical to the those from Simulation A, what makes the problem more complex is the alignment of obstacles in relation to the safety zones. This simulation will call for many more secondary paths needing to be formed, which also leaves more room for the algorithmic extensions to show their impact.

6.2.1 Brute Force Implementation

Result	**Value**
Final path cost	122.35 seconds
Translational time	120.06 seconds
Rotational time	2.29 seconds
Collision checks	989554
Computational runtime	343.06 seconds

Table 6.7: Simulation B: Brute-Force algorithm results

Table 6.7 shows the relative complexity of this simulation compared to Simulation A. We can see a significantly higher runtime and number of collision checks needed to form the optimal path using Brute-Force, with both reaching almost 6 times the equivalent values from Simulation A.

The reason for this increase in computational complexity is clear via analysing Figure 6.7 below. What is immediately clear is the far left and right safety zones have their immediate surroundings dominated by the obstacle space. This impacts the runtime heavily, since there will very rarely be straight line paths viable for most edges formed in the FMT around these zones. The first safety zone in particular has a notable effect on the cost of the final path, as it forces the red line in Figure 6.7 to divert upwards, instead of through the other gap explored by the FMT. The middle zone is the exception, and hence we expect this section of the simulation to run relatively fast.

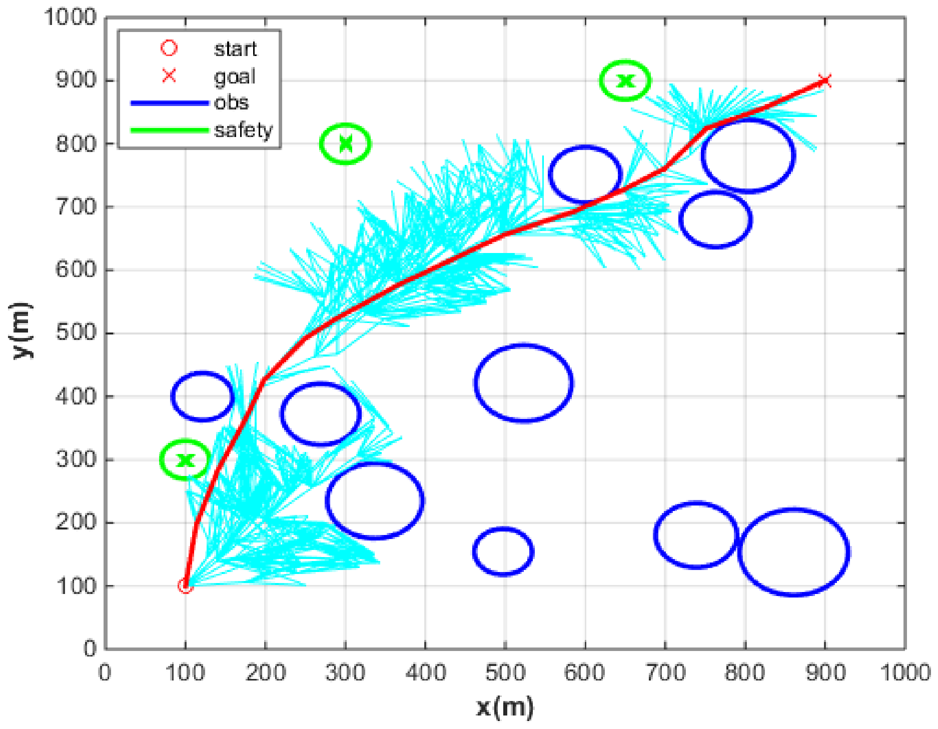


Figure 6.7: Simulation B: 2D visualisation of FMT* structure (light blue) along with final path (red) for brute-force implementation.

A 3D visualisation of the final path relative to the state space has also been provided in Figure 6.8 below. The three emergency paths shown in green are once again indicative of the three key regions of varying computational complexity. The far left and right paths are areas where secondary FMTs will need to be formed towards their respective zones, while the middle path shows a simpler section of the simulation where the FMT is in quite close proximity to its zone.

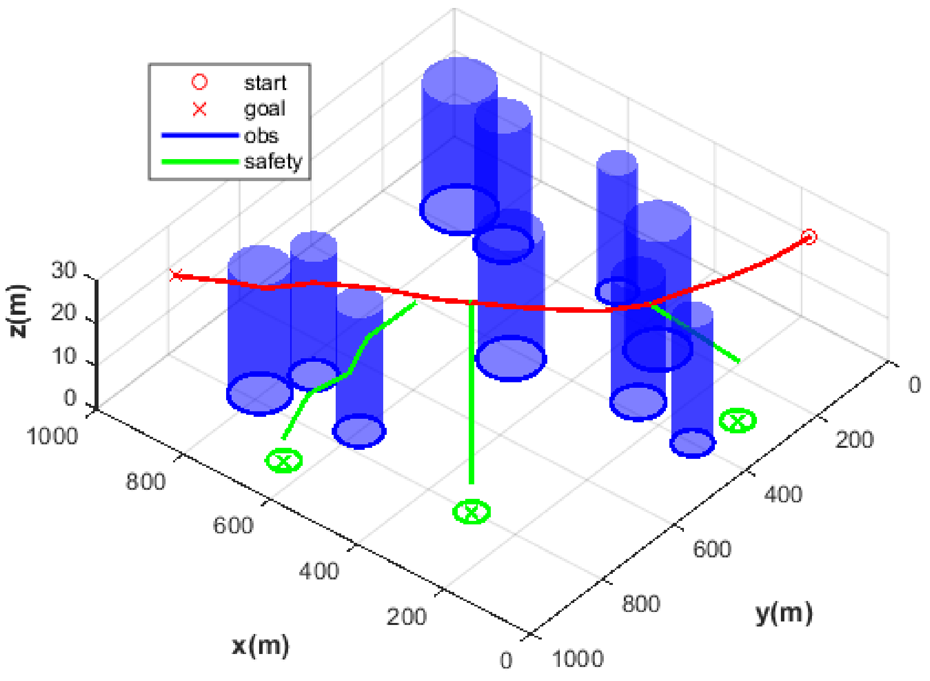


Figure 6.8: Simulation B: 3D visualisation of final path (red) returned for brute-force implementation. Also shows examples of emergency paths (green) taken at three different locations along the path to each of the safety zones.

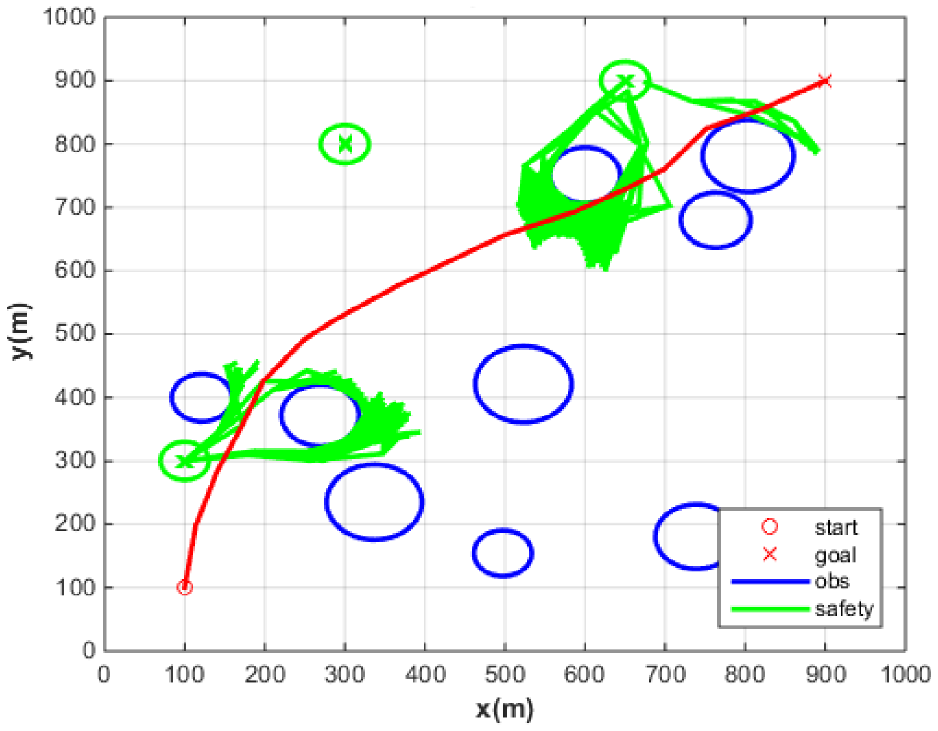
6.2.2 Path Reconstruction Implementation

Result	Value
Final path cost	122.35 seconds
Translational time	120.06 seconds
Rotational time	2.29 seconds
Collision checks	555020
Computational runtime	195.71 seconds

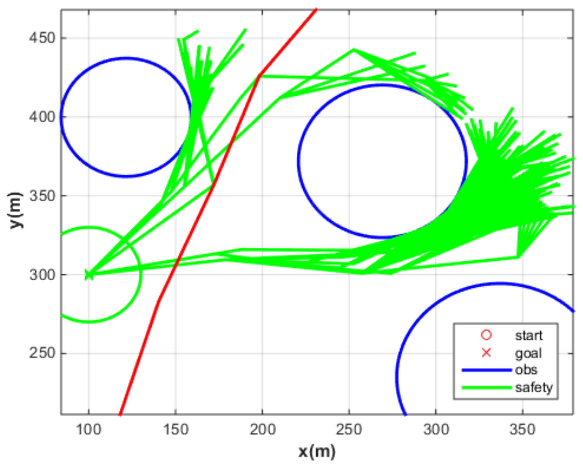
Table 6.8: Simulation B: "Path Reconstruction" algorithm results

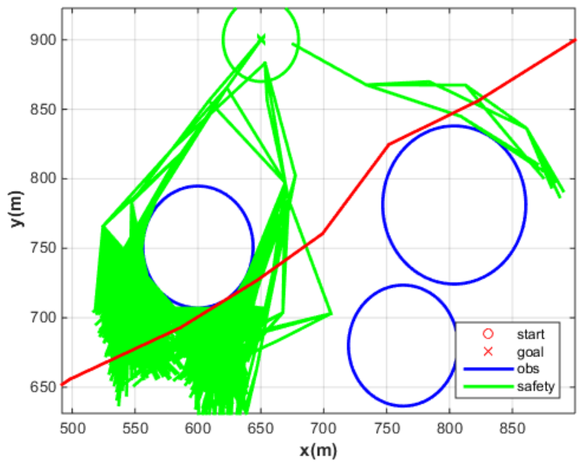
The results from implementing "Path Reconstruction" in Simulation B are provided above in Table 6.8. We observe a 42.95% computational runtime and 43.9% collision check improvement using this algorithmic extension over the Brute-Force solution. This indicates that the algorithm has not scaled significantly with the added complexity in the obstacle space, as we observed a 50% improvement in Simulation A.

Interestingly however, by analysing Figure 6.9, we might expect even better computational performance than what was returned. Note that in this simulation, the density of the secondary FMTs surrounding the obstacles in Figures 6.9b and 6.9c is much greater than the equivalent Figures 6.3b and 6.3c relating to Simulation A. This indicates that "Path Reconstruction" experienced heavier usage in these cluttered areas, which should reflect through a greater performance improvement.



(a)





(b) (c)

Figure 6.9: Visualisation of secondary FMTs formed by "Path Reconstruction". Figure 6.3a shows the overall path in red, which is the exact same as the one formed by Brute-Force in Figure 6.1. Figures 6.3b and 6.3c show close ups of the sheer density of interconnecting paths surrounding each of the obstacles.

One possible reason for this limitation is eluded to by observing Line 14 of Algorithm 5. Each concatenation needs to be tested just like a node connection against the obstacle space, which will require additional collision checks. Due to the sheer density of common paths, and the fact that they tightly wind themselves around the circular obstacles, the chances of an invalid concatenation increase. Even with this slight drawback, we still see a significant overall improvement in efficiency without sacrificing path cost, which is a highly desirable result for a much more complex problem space.

6.2.3 Obstacle Cluster Implementation

Result	**Value** ($r_{cluster} = 100m$)	**Value** ($r_{cluster} = 25m$)
Final path cost	122.35 seconds	122.35 seconds
Translational time	120.06 seconds	120.06 seconds
Rotational time	2.29 seconds	2.29 seconds
Collision checks	422213	730391
Computational runtime	144 seconds	259.04 seconds

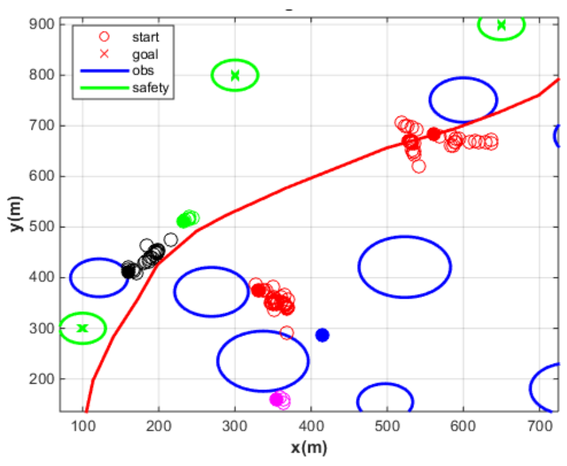
Table 6.9: Simulation B: “Obstacle Cluster” algorithm results

Table 6.9 above shows the results from applying the “Obstacle Cluster” algorithm to Simulation B. Once again the results have been split between the same two cluster radii. We still consider the same maximum radius value as Simulation A since the average diameter of the obstacle space in Simulation B is very close to that of A.

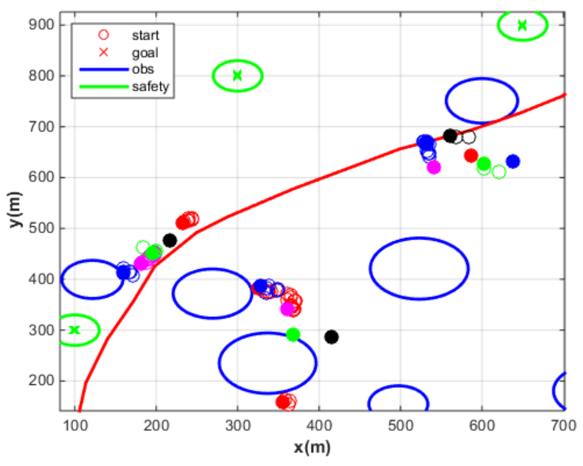
For $r_{cluster} = 100m$, we see an extremely significant improvement in terms of computational efficiency, without sacrificing the cost of the final path. We observe a 58% runtime improvement and 57% collision check improvement over the Brute-Force solution. This stands out even more due to the somewhat limited effect the “Obstacle Cluster” algorithm had in Simulation A. This implies that it performs better with more complex problem instances.

Another trend evident via this computationally harder simulation is the effect of decreasing the $r_{cluster}$. From Simulation A, we noted only a slight decrease in computational efficiency as a result of reducing the cluster radius by a factor of 4. For this simulation however, we observe a much more substantial difference, as the results for $r_{cluster} = 25m$ only yield a 24.5% and 26.2% improvement in runtime and collision checks respectively, which is less than half what was achieved for $r_{cluster} = 100m$.

Figures 6.10a and 6.10b give a clearer picture as to why a larger cluster radius improves performance significantly. We see in Figure 6.10a that the numbers of clusters is limited,



(a) $r_{cluster} = 100m$



(b) $r_{cluster} = 25m$

Figure 6.10: Visualisation of Obstacle Cluster algorithm shown with varying $r_{cluster}$. Note each cluster is shown in a different colour, and the filled circles represent the local minimum of each cluster.

but the size of each cluster is large. This is a desirable outcome since firstly, it creates less computational overhead and runtime during linear cluster searches when points are assessed for their proximity to existing clusters. Secondly, and more importantly, every time a new cluster is formed, we need to validate that the node is in fact invalid, which means a full FMT* search would need to be performed. Therefore in Figure 6.10b, we see several small size clusters, each of which would have had to have gone through this process. It also creates unnecessary overlap between clusters, especially considering that nodes in the same general vicinity are targeting the same zone.

6.2.4 Complete Algorithm Implementation

The results for the complete algorithm implementation for Simulation B are summarised in Table 6.10 and Figure 6.11.

Result	**Value**
Final path cost	122.35 seconds
Translational time	120.06 seconds
Rotational time	2.29 seconds
Collision checks	225898
Computational runtime	81.61 seconds

Table 6.10: Simulation B: Complete algorithm results with $r_{cluster} = 100m$

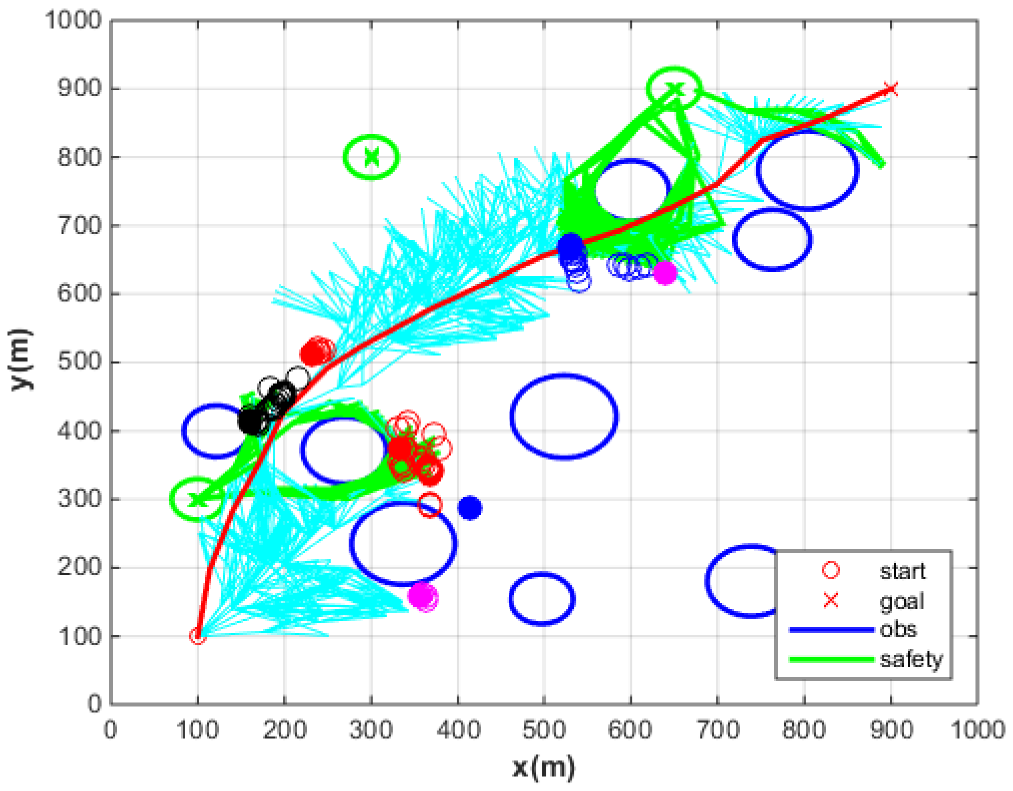


Figure 6.11: Final solution formed by complete algorithm, including both algorithmic extensions. Both the secondary FMTs (in green) and obstacle clusters (scatters circles) are shown along with the final path (red) and FMT (light blue).

After seeing the results of the individual algorithmic extensions, it should come as no surprise to see that the combined solution incorporating both did much better computationally than the Brute-Force solution as well as maintaining path cost. Quantitatively, we see a 76.21% runtime and 77.17% collision check improvement, which is extremely significant and shows the positive scaling of the complete algorithm with complexity of the problem instance.

The key factor in why this is the case is analysing how the "Obstacle Cluster" and "Path Reconstruction" algorithms work together. By introducing a more complex obstacle space, we form large clusters of invalid nodes. Since certain areas are now dominated by clusters, the FMT path planner can effectively ignore them. This also means the "Path Reconstruction" does not need to attempt to concatenate paths from these problematic areas, and consequently we save a lot of computational time and collision checks. Note also that by bounding the cluster radius as a function of the obstacle space, we are able to maintain the correctness of the algorithm.

6.3 Monte-Carlo Results Summary

Table 6.11 below shows the Monte-Carlo simulation results where each simulation was repeated 15 times for each algorithm. The values shown are the mean percentage performance improvements over the Brute Force FMT* algorithm. Note that this summary is included to verify that the effects of the random sampling do not significantly alter the results of each individual simulation.

Algorithm	**Simulation**	**ΔCost**	**ΔCollision**	**ΔRuntime**
Path Recon	A	0%	50%	56%
Path Recon	B	0%	48%	50%
Obst Clust	A	0%	3.2%	14%
Obst Clust	B	0%	57%	58%
Complete	A	0%	53%	59%
Complete	B	0%	77%	76%

Table 6.11: Monte-Carlo Performance Comparison

Evidently, the random sampling does not effect the cost of the final paths produced since each algorithm obtained the exact same path as the Brute Force FMT* implementation.

We also see that the trends that were observed in the previous two sections are also evident here. Namely, the "Path Reconstruction" algorithm provides an approximate 50% improvement in number of collision checks performed, and this remains relatively constant even for the high complexity simulation. "Obstacle Cluster" contrastingly has an extremely limited effect for Simulation A, but then scales significantly to provide a 58% runtime improvement in Simulation B.

The complete algorithm, which combines both algorithmic extensions, inherits the effects of both and correspondingly yields an average 77% reduction in collision checks and 76% reduction in runtime over the Brute Force FMT* implementation for Simulation B. It is also seen to scale relatively well between Simulations A and B.

6.4 Complete Algorithm with Wind

This section of the results will analyse the effects of wind on the complete algorithm and in particular assess the robustness of the final solution. This means judging whether correctness is still maintained while obeying the constraints of the problem. We shall consider two cases; a constant wind field, and a variable wind field.

6.4.1 Constant Wind Field

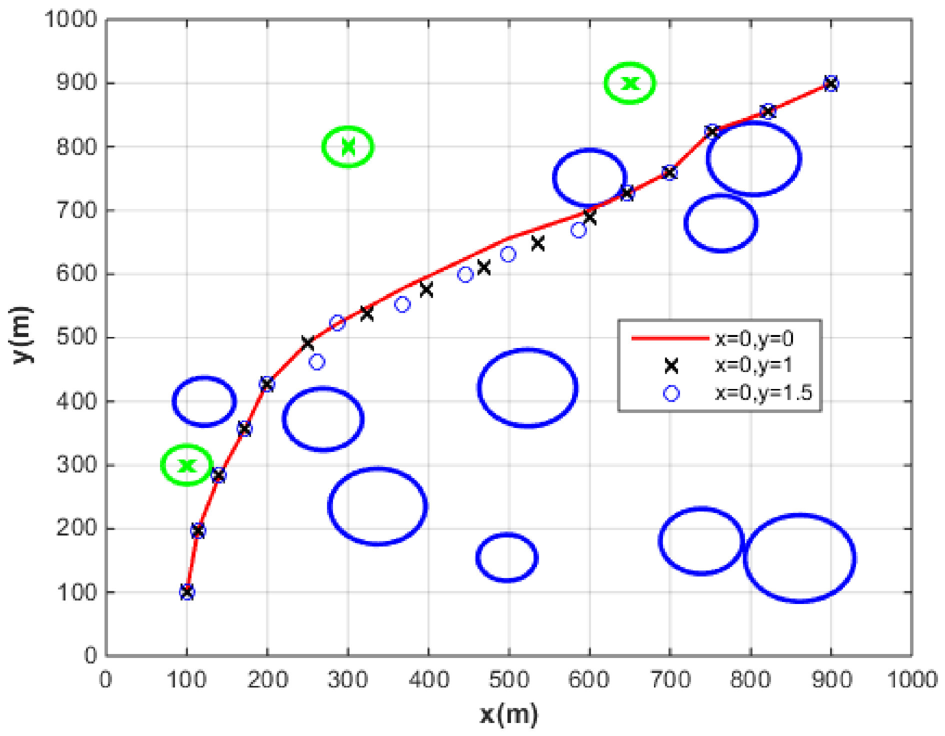


Figure 6.12: Shows the final paths obtained by implementing the complete algorithm on the same obstacle field as Simulation B with constant wind fields. The wind fields and associated paths can be seen in the legend in the Figure.

The parameters from Simulation B were loaded with three different constant wind fields measured in m/s: $(x_{wind} = 0, y_{wind} = 0)$, $(x_{wind} = 0, y_{wind} = 1)$ and $(x_{wind} = 0, y_{wind} = 1.5)$. The final paths are seen in Figure 6.12 above.

The biggest discrepancies we expect to see in the paths are areas which are on the boundary of not satisfying the temporal constraint. We observe this occurring in the middle portion of the paths since this is the section which is furthest away from its associated zone (the middle most zone). We can see that due to the effect of the positive y wind, the optimal path shifts downwards slightly, which is expected since it is given more of a push towards its zone when checking the temporal constraint. This importantly allows it to take a shorter, and therefore more desirable path towards the goal, while still obeying the constraints of the problem.

Another point of interest is the path relating to the $y_{wind} = 1.5$ (blue circles in Figure 6.12). We see that just after the path has crossed the gap between the first two obstacles, it takes a sudden dip downwards, and then goes straight back up towards the black and red paths. This sudden activity shows a transition between the path targeting the first and second zones. The dip is because the wind is too strong going upwards for it to reach the first zone successfully while following the standard path. Hence it diverts away from the main path in order to overcome the effects of the wind and reach the left most zone, and then proceeds back to the main path to target the middle zone.

6.4.2 Variable Wind Field

Once again, the state space from Simulation B will be used to test the capabilities of the complete algorithm, this time with a variable wind field. Figure 6.13 shows the specific wind field which will be tested.

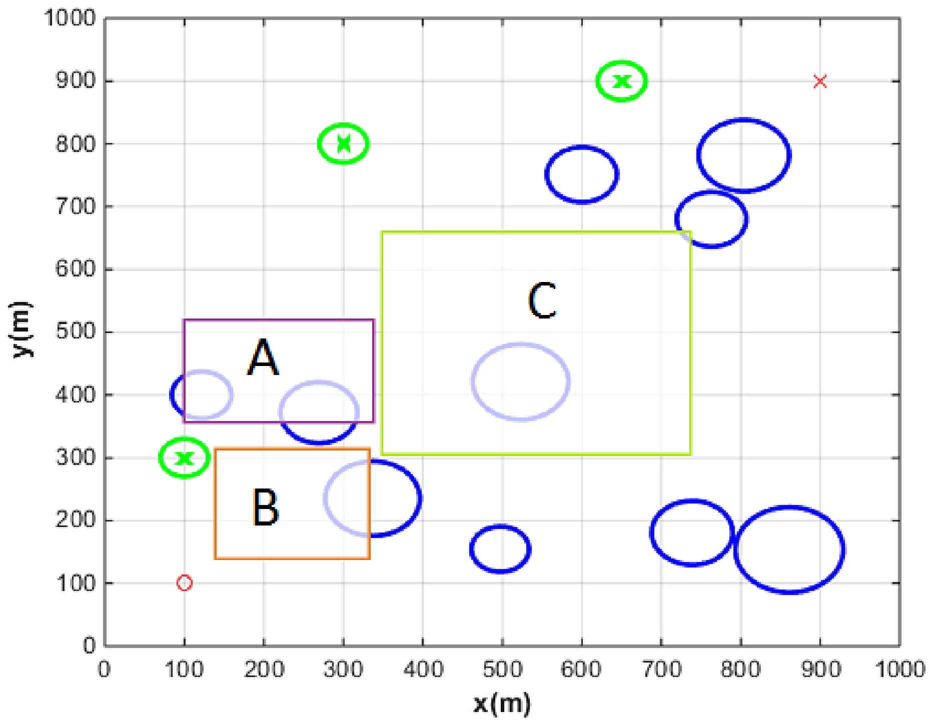


Figure 6.13: Wind field will consist of three areas, observed by the three coloured boxes A, B and C. Each will posses a different wind velocity in the x and y axis. Any region lying outside these boxes will have no wind.

The three specific areas defined in Figure 6.13 have the following characteristics:

- Wind field A: $x_{wind} = -5m/s, y_{wind} = -5m/s$
- Wind field B: $x_{wind} = -0.5m/s, y_{wind} = 0m/s$
- Wind field C: $x_{wind} = 0m/s, y_{wind} = 2.5m/s$

Any region outside these boxes will have no wind associated. The final path generated by the complete algorithm with the above wind field has been provided below in Figure 6.14.

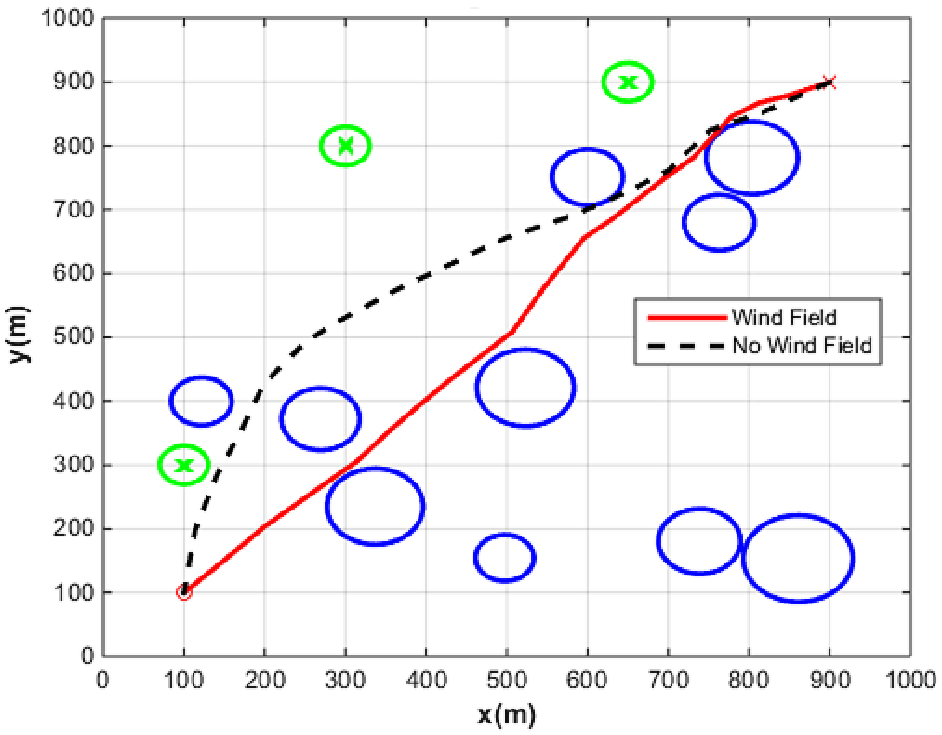


Figure 6.14: Final paths obtained by applying the complete algorithm on the wind field specified above. The red line defines the path taken considering the wind field, and the dashed black line is the one without considering the wind field.

We can see that the final path changes completely due to the effect of the wind field. The most significant change is attributed to Wind field A, which modelled an area of extremely high turbulence. This prevented the FMT from expanding through the same gap as was taken by the dashed black line in Figure 6.14. The only other available route was moving through Wind field B and towards C. Note that Wind field B is significant because it pushes the UAV towards the farthest left zone, allowing it to pass the temporal constraint. Wind field C is also important for the next set of temporal constraints, as it pushes the UAV up towards the middle zone. Without this upwards y velocity, it would be situated too far from any of the zones, and therefore lead to a termination of the algorithm. Therefore, we see that the complete algorithm is robust even with a variable wind field.

6.5 Summary

This chapter presented results from applying various combinations of the Brute-Force implementation of FMT* along with the novel extensions proposed in this thesis: “Path Reconstruction” and “Obstacle Cluster”. These were compared between two simulations of varying difficulty.

It was found that for the easier problem, “Path Reconstruction” had more of an impact on improving computational efficiency than “Obstacle Cluster”. The combined algorithm inherited the path quality of both individual solutions, as well as the significant reduction in number of collision checks required. For the more complex simulation, the opposite was found, as “Obstacle Cluster” scaled much better to the harder problem than “Path Reconstruction”. The net effect once again was a complete algorithm which also shared the similar positive scaling as “Obstacle Cluster”, and improved computational runtime significantly while maintaining path cost.

The complete algorithm was then also tested for robustness in a static and varying wind field. It was found that in both cases, the algorithm held strong while ensuring that the constraints of the problem were not overlooked in an effort to minimise the time objective.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Thesis Summary

This thesis was addressed at solving the UAV engine-out problem. This is a path planning problem based around minimising the time taken along a path while abiding by spatial constraints from an obstacle space and temporal constraints relating to proximity to pre-designated safety zones. The inherent difficulty of the problem is creating a computationally efficient algorithm which is able to handle the non-linearities of the constraints, integrating UAV motion dynamics, and handling wind disturbances.

It was found after a thorough literature review that FMT*, which is a recently proposed Sampling Based Planner, was the most suitable method for solving the path planning problem. However, it suffers from limitations, mainly in its ability to deal with multi-layered constraints in a time efficient manner. This motivated extensions to be made to the canonical form which came in the form of two novel algorithmic approaches. The key elements which were used to form the final core algorithm were:

- FMT* algorithm: Utilised to form the framework of the core algorithm from which to build upon
- "Path Reconstruction" algorithm : Novel algorithmic extension designed to significantly lower the cost of the temporal constraint checking within the core algorithm
- "Obstacle Cluster" algorithm: Novel algorithmic extension designed to filter out invalid points from the path planner based on their proximity to the pre-existing obstacle space.

This core algorithm was then tested using several MATLAB simulations in Chapter 6. Emphasis was placed on assessing the impact the algorithmic extensions had on the path cost, computational efficiency and robustness of the final solution. The key results observed were:

- FMT* proved an efficient framework from which to build upon, as it allowed for differential motion constraints from the UAV to be integrated with a dynamically growing cost tree. It used a lazy heuristic to determine the validity of nodes, which had varied success for more complex obstacle spaces, however it inherently assured asymptotic optimality of the final path.
- "Path Reconstruction" resulted in superior computational efficiency, while maintaining the correctness of the canonical Brute-Force approach. The efficiency improvement remained relatively consistent between simulations, and did not significantly improve or worsen under more complex problem scenarios.
- "Obstacle Cluster" had similar improvements in efficiency, without an increase in cost. However in contrast, it showed much superior scaling to "Path Reconstruction", as it went from having limited improvement in a simple problem instance to a significant improvement in the complex instance. It relies on a user defined variable $r_{cluster}$, which also had a scaling impact on the efficiency results (large $r_{cluster}$ yielded better performance).
- The combined algorithm incorporating both algorithmic extensions inherited the advantages from the individual algorithms, and similar to the "Obstacle Cluster" results, showed a positive scaling for more complex problem instances.
- The core algorithm was robust to both static and varying wind fields. The final paths returned were able to minimise the time objective, whilst satisfying the spatial and temporal constraints of the thesis problem.

7.2 Summary of Contributions

- Non-holonomic path planning solution to UAV engine-out problem using recently proposed Sampling Based planner FMT*. This thesis exposed the method to an inherently difficult problem with multi-layered constraints, while incorporating effects of wind disturbances.
- Introduced the "Obstacle Cluster" algorithm, which is a novel extension to the FMT* framework. The method aimed to filter invalid nodes via analysing the spatial and

temporal obstacle space, and was shown to save significant computational time on the way to returning an feasible path. Previous work in the field had achieved a similar result via analysis of a purely geometric space, however this thesis extended the state-of-the-art by analysing a space which is non-geometric.

- Introduced "Path Reconstruction" algorithm, which is another novel extension to FMT* framework which once again aimed to save computational runtime during the path planning process. It did this by reusing paths which were previously generated by the path planner so future temporal constraint checks became significantly shortened.

7.3 Future Work

- Extend the UAV dynamic model to include turn radius and velocity constraints and allow the trajectory between nodes to not necessarily traverse straight lines. Allowing an arbitrary trajectory would be non-trivial and would be cause for reworking the lazy cost determination of FMT*.
- Generalise "Obstacle Cluster" for any obstacle space. Currently, it relies on circular obstacles in 2D space and using the average diameter to obtain a maximum bound for $r_{cluster}$. Investigating different bounds for an arbitrary obstacle space would be an interesting extension.
- Using knowledge of the wind field and obstacle space to engineer a more informed cost determination of nodes, which currently involves forming a straight line cost heuristic towards the goal.

BIBLIOGRAPHY

[1] A. of Unmanned Vehicle Systems International, "Projected annual sales of unmanned vehicles." `www.iii.org/insuranceindustryblog/?cat=8`, 2015. [Online; accessed 25-May-2015].

[2] D.Rathbun, S.Kragelund, A.Pongpunwattana, and B.Capozzi, "An evolution based path planning algorithm for autonomous motion of a UAV through uncertain environments," *Digital Avionics Systems Conference*, vol. 2, pp. 8D2:1–12, 2002.

[3] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *International Journal of Robots Research*, vol. 5, pp. 90–98, 1986.

[4] A. Richards and J. How, "Aircraft trajectory planning with collision avoidance using mixed integer linear programming," *American Control Conference 2002*, vol. 3, pp. 1936–1941, 2002.

[5] R. Bellingham, A. Richards, and J. How, "Receding horizon control of autonomous vehicles," *Massacusetts Institute of Technology*, pp. 1390–1397, 2002.

[6] A. Patel, "Introduction to a*." `theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html`, 2015. [Online; accessed 29-May-2015].

[7] S. LaVelle, "Rapidly-exploring random trees: A new tool for path planning," *Department of Computer Science, Iowa State University*, 1998.

[8] S. Choudhury, S.Scherer, and S.Singh, "RRT*-AR: Sampling-based alternate routes planning with applications to autonomous emergency landing of a helicopter," *IEEE International Conference*, vol. 8, pp. 3947–3952, 2013.

[9] J. Denny and N. Amato, "Toggle prm: Simultaneous mapping of c-free and c-obstacle - a study in 2d -," *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference*, 2011.

[10] L. Janson, E. Schmerling, A. Clark, and M. Pavone, "Fast marching tree: a fast marching sampling-based method for optimal motion planning in many dimensions," *16th International Symposium on Robotics Research*, 2015.

[11] R. Rhoad, G. Milauskas, and R. Whipple, "Geometry for enjoyment and challenge," *rev. ed.*, 1984.

[12] P.Chandler, M.Pachter, and S.Rasmussen, "UAV cooperative control," *American Control Conference*, pp. 50–55, 2001.

[13] X. Song, "Fly the automated skies: Drones and the rise of the civilian UAV sector." `www.datafox.co/blog/2014/04/fly-the-automated-skies-drones-and-the-rise-of-the-civilian-sector`, 2014. [Online; accessed 25-May-2015].

[14] CASA, "Visual flight rules guide," *Canberra: Civil Aviation Safety Authority Australia (CASA)*, 2007.

[15] C.Redelinghuys, "A flight simulation algorithm for a parafoil suspending an air vehicle," *AIAA Journal of Guidance, control and dynamics*, vol. 28, no. 4, pp. 801–811, 2007.

[16] P. C. S. Eng, *Path planning, guidance and control for a UAV forced landing.* PhD thesis, Australian Research Centre for Aerospace Automation, School of Engineering Systems, Queensland University of Technology, 2011.

[17] B. Anderson and J. Moore, "Optimal control - linear quadratic methods," *Department of Systems Engineering, Australia National University, Canberra*, 1989.

[18] C. Hajiyev and S. Vural, "Lqr controller with kalman estimator applied to UAV longitudinal dynamics," *Scientific Research:Positioning*, vol. 4, pp. 36–41, 2013.

[19] A. Eiben and J. Smith, *Introduction to Evolutionary Computing.* Springer.

[20] M. Gerke, "Genetic path planning for mobile robots," *American Control Conference 1999*, vol. 4, pp. 2424–2429, 1999.

[21] U.Cekmez, M.Ozsiginan, M.Aydin, and O.K.Sahingoz, "UAV path planning with parallel genetic algorithms in cuda architecture," *Proceedings of the World Congress of Engineering 2014*, vol. 1, 2014.

[22] S. Sanci and . Veysi, “A parallel algorithm for uav flight route planning on gpu,” *International Journal of Parallel Programming*, vol. 39, no. 6, pp. 809–837, 2011.

[23] J. Andrews, “Impedance control asa framework for implementing obstacle avoidance in a manipulator,” *Massachusetts Institude of Technology, Department of Mechanical Engineering*, 1983.

[24] K. Sigurd and J. How, “UAV trajectory design using total field collision avoidance,” *Massachusetts Institute of Techology, Cambridge*, 2003.

[25] Y.Koren and J.Borenstein, “Potential field methods and their inherent limitations for mobile robot navigation,” *IEEE conference on Robotics and Automation*, pp. 1398–1404, 1991.

[26] A. Bemporad and M. Morari, “Control of systems integrating logic, dynamics and constraints,” *Automatica*, no. 35, pp. 407–427, 1999.

[27] H. Williams and S. Brailsford, “Computational logic and integer programming,” pp. 249–281, 1996.

[28] M. Elbanhawi and M. Simic, “Sampling-based robot motion planning: A review,” *Access IEEE*, vol. 2, pp. 56–77, 2014.

[29] E. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[30] D. Delling, P. Sanders, D. Schultes, and D. Wagner, “Engineering route planning algorithms,” *Algorithmics of large and complex networks*, pp. 117–139, 2009.

[31] R. Dechter and J. Pearl, “Generalized best-first search strategies and the optimality of a*,” *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 502–536, 1985.

[32] B. Meng and X.Gao, “UAV path planning based on bidirectional sparse A* search algorithm,” *Intelligent Computation Technology and Automation (ICICTA)*, vol. 3, pp. 1106–1109, 2010.

[33] L. Ji and X. Sun, “A route planning's method for unmanned aerial vehicles based on improved a-star algorithm [j],” *Acta Armamentarii 7*, pp. 788–792, 2008.

[34] D. Eppstein, “Finding the k shortest paths,” *35th Annual IEEE Symposium on Foundations of Computer Science (FOCS'94)*, pp. 154–165, 1994.

[35] R. Geisberger, M. Kobitzch, and P. Sanders, “Route planning with flexible objective functions,” *ANLENEX*, vol. SIAM, pp. 124–137, 2010.

[36] I. Abraham, D. Delling, A. Goldberg, and R. Werneck, "Alternative routes in road networks," *9th Annual International Symposium on Experimental Algorithms (SEA'10)*, pp. 23–34, 2010.

[37] M. Kothari, I. Postlethwaite, and D. Gu, "Multi-uav path planning in obstacle rich environments using rapidly-exploring random trees," *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference*, pp. 3069–3074, 2009.

[38] J. Amin, J. Boskovic, and R. Mehra, "A fast and efficient approach to path planning for unmanned vehicles," *In Proceedings of AIAA Guidance, Navigation, and Control Conference and Exhibit*, pp. 21–24, 2006.

[39] Karaman, Sertac, and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[40] T. Yomchinda, J. Horn, and J. Langelaan, "Autonomous control and path planning for autorotation of unmanned helicopters," *American Helicopter Society 68th Annual Forum*, 2011.

[41] L. Kavraki, P. vestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *Robotics and Automation, IEE Transactions*, vol. 12, no. 4, pp. 556–580, 1996.

[42] F. Yan, Y. Zhuang, and J. Xiao, "3D PRM based real-time path planning for UAV in complex environment," *Robotics and Biometrics (ROBIO) 2012*, pp. 1135–1140, 2012.

[43] O. Salzman, D. Shaharabani, P. Agarwal, and D. Halperin, "Sparsification of motion-planning roadmaps by edge contraction," *International Journal of Robots Research*, 2014.

[44] J. Denny, K. Shi, and N. Amato, "Lazy Toggle PRM: A single-query approach to motion planning," *Robtics and Automation (ICRA)*, pp. 2407–2414, 2013.

[45] K. Hauser, "Lazy collision checking in asymptotically-optimal motion planning," *Robtics and Automation (ICRA), to appear*, 2015.

[46] E. Frazzoli and S. Karaman, "Sampling-based algorithms for optimal motion planning," *International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[47] O. S. University, "Dot products and vector projections." `math.oregonstate.edu/home/programs/undergrad/CalculusQuestStudyGuides/vcalc/dotprod/dotprod.html`, 1996. [Online; accessed 30-September-2015].

APPENDIX A
LINE SEGMENT AND CIRCLE INTERSECTION

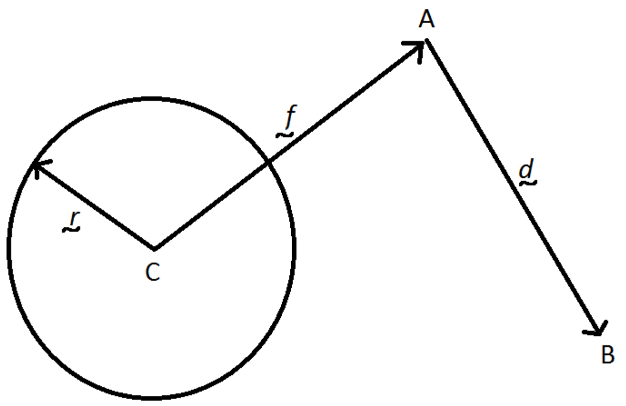


Figure A.1: Circle and Line Segment intersection in 2D Cartesian space. A and B correspond to ends of the line segment, while C is the center of the circle. $\underline{f}$ is the direction vector from center of circle to start of line segment, $\underline{d}$ is the line segment and $\underline{r}$ is the radius.

$$(\underline{d}.\underline{d})t^2 + 2(\underline{f}.\underline{d})t + (\underline{f}.\underline{f} - r^2) = 0 \tag{A.1}$$

Equation A.1 will yield two solutions t_1 and t_2, which must abide by one of the following restrictions for an intersection to occur:

- $0 \leq t_1 \leq 1$
- $0 \leq t_2 \leq 1$
- $t_1 \leq 0$ and $t_2 \geq 1$

Note this mathematical formulation is derived in reference [11]