ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Risk-based Pathfinding for Drones

Master Thesis

Vilhjalmur Vilhjalmsson

May 1, 2016

Advisors: Prof. Dr. Marc Pollefeys, Lorenz Meier
Department of Computer Science, ETH Zürich

Abstract

In this thesis, the problem of minimizing both distance and risk for paths in graphs is investigated. Various methods for doing so will be considered for general graphs, but also analysed by their compatibility with pathfinding for drones. In particular, we will see how using expected values for risk can greatly simplify the problem in many common cases. Little has been written about searching with expected values in risk-based search and how it compares to the conventional definition of risk, but we will show that by using expected values, risk can be easily introduced into pathfinding for vehicles without sacrificing solution quality.

Additionally, we will show how smoothness can be maximized while maintaining a simple algorithm. These results are then used to give a complete model for minimizing time, energy usage and risk, for a drone with a stereo-camera, using the OctoMap library for maintaining a probabilistic belief of the world.

Finally the model is simulated using the Gazebo simulator and different strategies compared both quantitatively and visually. The strategies are also visually compared to a human pilot to give an interesting insight into what properties are desired when it comes to pathfinding for drones.

Contents

Symbols

Symbols

G	Graph
V	Set of vertices in graph G
E	Set of edges in graph G
$l(e)$	Length of edge e
$r(e)$	Risk of edge e
$w(e)$	Weight of edge e, in terms of $l(e)$ and $r(e)$
s	Source vertex
t	Target vertex
P	Path from s to t
$L(P)$	Length of path P
$R_P(P)$	Probability of failure when traversing path P
$R_E(P)$	Expected number of failures when traversing path P
$S(P)$	Tortuosity of path P, the amount of turning in P
$F(P)$	Cost of path P, in terms of $L(P), R_P(P), R_E(P)$ and $F(P)$

Chapter 1

Introduction

Unmanned aerial vehicles (UAV), or just drones, are an incredibly fast-expanding field of research and development. While previously being mostly known as a hobby for aviation enthusiasts and for their controversial use in war, the field has recently expanded into package delivery, photography, surveillance, and many other fields.

Pathfinding is a mathematical problem which aims to minimize the cost of a path from a starting vertex to a goal vertex. It is one of the most common problems in computer science and specialized algorithms have been developed for pathfinding in computer networks, roadmaps, social networks, etc. Although the problem formally finds the shortest path in a graph, the weights of the edges do not need to represent distance.

Pathfinding for drones aims to find a trajectory in three-dimensional space which minimizes a cost function which can be defined in terms of distance, time, energy usage or other metrics. Traditionally, the goal is split up into two tasks; global planning and local planning. While a global planner finds a high level path to the goal, the local planner translates this path into velocity commands for the robot's motors. Many algorithms have been developed for pathfinding for robotics which can differ in fundamental ways. While some run on discrete grids, others use continuous representations of their environment. In this thesis we will focus on grid-based solutions, and explore how risk can be introduced into the problem.

To analyse algorithms for robotics, simulation is an important part of all robotics research. It enables researchers to see how their solutions works immediately, without risking expensive equipment. In this thesis, we will use the Robot Operating System (ROS) along with the Gazebo simulator to implement and analyze our approach.

Chapter 2

Related work

2.1 Pathfinding with risk

2.1.1 Safest Path

Jun et al. [11] give an algorithm for planning paths for unmanned aerial vehicles in adversarial environments. Using a logarithmic transformation, the safest path problem is transformed into a shortest path problem which is then solved with Dijkstra's Algorithm.

2.1.2 Combining distance and risk

Multiobjective Optimal Paths

Finding Multiobjective Optimal Paths is the problem of finding paths that optimize more than one objective function simultaneously. Hallam et al. [9] give an algorithm which finds approximate solutions for Pareto-optimal paths. The algorithm is used for guiding a submarine through a field of sensors at known positions, within a fixed time period and with minimum probability of detection.

Minimum Cost Reliability Ratio

The Minimum Cost Reliability Ratio (MCRR) problem is the problem of finding the path which minimizes the ratio between the cost of the path and the probability of successfully traversing the path. Greytak and Hoverz [8] give a generalized version of A* that finds the MCRR for holonomic vehicles but the paper does not give any bounds on running time, which seems to be exponential in worst case. To solve the MCRR problem efficiently, Katoh [12] gives a fully polynomial approximation scheme, and Aboutahoun [1] gives a pseudo-polynomial algorithm for solving MCRR with integer input.

Other cost functions

It is also possible to choose the cost function such that it can be solved with Dijkstra's Algorithm. Jun et al. [11] extend the safest path algorithm to penalize distance while maintaining the same asymptotic run-time complexity of Dijkstra's Algorithm. Kothari et al. [13] give an algorithm using Rapidly-Exploring Random Trees (RRT) for Path Planning Algorithm for UAVs. The cost function consists of expected values instead of pure probabilities which makes it easier to compute.

2.1.3 Shortest path with bounded risk

Instead of minimizing both path length and risk, the Constrained Shortest Path Problem (CSP) aims to minimize the distance while guaranteeing that the total risk is below a certain threshold. The CSP path problem is NP-hard but Ono and Williams [17] give an algorithm which is only slightly suboptimal and guarantees that a missions probability of success is above a threshold.

2.2 Pathfinding in partially observed environment

Another type of risk is when some parts of the graph are not known. That is, instead of some edges having a known probability of failure, some edges may or may not exist and whether they do is only known when one is positioned in an adjacent vertex. This problem is known as the Canadian Travellers Problem (CTP) and it is known to be #P-hard. Marthi [15] gives an algorithm for solving the problem as a partially observable Markov decision process (POMDP) in exponential time in the number of possible obstacle locations. Bnaya et al. [4] give an algorithm for solving a generalized CTP to allow for remote sensing actions. The algorithm utilizes heuristics to determine when and where to sense the environment in order to minimize total costs.

Chapter 3

Path Finding

3.1 Shortest Path

3.1.1 Dijkstra's Algorithm

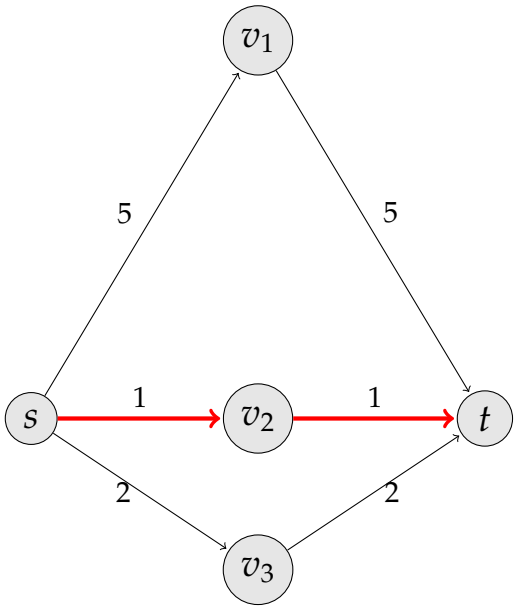


Figure 3.1: Example of a graph with three paths from s to t. The shortest path is shown in red.

Finding the shortest path in a graph is one of the core problems in Graph Theory. Formally, for a graph $G = (V, E)$ and source and goal vertices s and t, we want to find

$$\min_P \sum_{e \in P} w(e), \tag{3.1}$$

where $P = (e_0, ..., e_n) = ((s, v_1), (v_1, v_2), ...(v_n, t))$ for $e_i \in E, v_i \in V$.

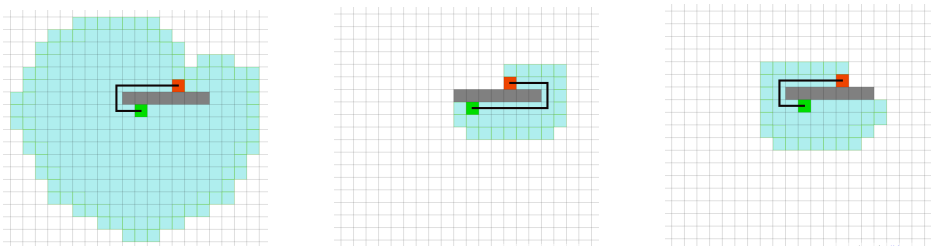
The problem was solved by Edsger Dijkstra with his eponymous algorithm in 1956 [6]. The algorithm's time complexity is originally $O(V^2)$, but a common improvement yields $O(|E| \log(|E|))$. Other more complex algorithms have been discovered with better asymptotic runtime but because of its simplicity, Dijkstra's algorithm remains widely used for general graphs.

The pseudo-code for Dijkstra's Algorithm can be seen in Algorithm 1 below. The loop in line 7 amounts to one iteration of the algorithm, which consists of choosing one vertex u, which has not been chosen before, and for each neighbor v, we check if the distance to u plus the weight of the edge (u, v) is smaller than the smallest distance to v seen so far. This is called expanding the vertex u, and by always expanding the vertex that is closest to s that has previously not been expanded, the algorithm maintains the minimum distance from s.

Algorithm 1 Dijkstra's Algorithm

Require: Graph $G = (V, E)$ with non-negative edge weights
Require: Source vertex s
Require: Target vertex t
Ensure: The shortest path in G from s to t, if one exists

```
1:  for all v ∈ V do
2:      d[v] ← +∞
3:      previous[v] ← undefined
4:  end for
5:  d[s] ← 0
6:  Q ← V
7:  while Q is not empty and t ∈ Q do
8:      u ← vertex in Q with min d[u]
9:      Q ← Q \ {u}
10:     for all edges e = (u, v) outgoing from u do
11:         if d[u] + w(e) < d[v] then
12:             d[v] ← d[u] + w(e)
13:             previous[v] := u
14:         end if
15:     end for
16: end while
```



(a) Dijkstra's Algorithm (b) Best-First search (c) A*

Figure 3.2: A comparison of Dijkstra's Algorithm, Best-First search and A*. The black line is the solution path from the green square to the red square, the gray squares are obstacles and the blue squares are vertices expanded by the search. Note that Best First search does not find the shortest path in this case.

3.1.2 A*

While Dijkstra's algorithm works well for graphs in general, it is often possible to use known properties of the graph to find an optimal solution much faster. For path-finding on a plane, one might, for example, argue that if t is to the right, a vertex to the right is more likely to be part of the shortest path than a vertex to the left. In Figure 3.2(a) we see an example of which vertices are expanded (line 9 in Algorithm 1). Since Dijkstra's Algorithm always expands the vertex that is closest to s that has not been expanded before, it expands all vertices that have a lower distance than t. Looking at the figure it is clear that some improvements can be made.

The problem is that Dijkstra's Algorithm only focuses on the distance from s instead of trying to get closer to t. Best-first search is a search algorithm similar to Dijkstra's Algorithm that expands vertices that are close to t, by some metric. This strategy usually returns a solution much quicker, but there is no guarantee that it is in any way close to optimal. In Figure 3.2 (b) we see what happens if Best-First search is used instead of Dijkstra's Algorithm for the same problem as before. Best-first search expands much fewer vertices than Dijkstra's Algorithm, but the solution is also slightly worse.

A* can be seen as a generalization of Dijsktra's algorithm, which uses heuristics to guide the search in the direction towards t, similarly to Best-First search. In Algorithm 2 we see a pseudo-code for A* which is almost identical to Dijkstra's algorithm, except that it requires a heuristic function, h, which is used in line 9. The heuristic function needs to be admissible, that is, $h(v)$ must be a lower bound on the shortest path from any vertex v to t [5]. Since the straight line distance is a lower bound on the shortest-path

length, it can be used as heuristics. In Figure 3.2(c) we see how A* finds the same path as Dijkstra's Algorithm, which is optimal, while only expanding a few more vertices than Best-First search. Furthermore, A* is guaranteed to be optimal in regards of number of vertices expanded for a given heuristic function. [5]

Algorithm 2 A*

Require: Graph $G = (V, E)$ with non-negative edge weights
Require: Source vertex s
Require: Target vertex t
Require: Consistent heuristic function $h(v), v \in V$
Ensure: The shortest path in G from s to t, if one exists

```
1: for all v ∈ V do
2:    d[v] ← +∞
3:    previous[v] ← undefined
4: end for
5: d[s] ← 0
6: Q ← V
7: while Q is not empty and t ∈ Q do
8:    u ← vertex in Q with min d[u] + h(u)
9:    Q ← Q \ {u}
10:   for all edges e = (u, v) outgoing from u do
11:      if d[u] + w(e) < d[v] then
12:         d[v] ← d[u] + w(e)
13:         previous[v] := u
14:      end if
15:   end for
16: end while
```

A nice feature of the A* algorithm is that it is possible to speed up the search by forfeiting the optimality constraint. This is done by defining a new heuristic function, $h_\alpha(v) = \alpha h(v), \alpha > 1$. This guarantees the solution to be at most an α-factor from the optimal solution, and is referred to as an α-admissible solution [5]. In practice this turns out to be a valuable trade-off as for many cases the decrease in runtime can be significant without a noticeable increase in path length.

3.2 The Safest Path

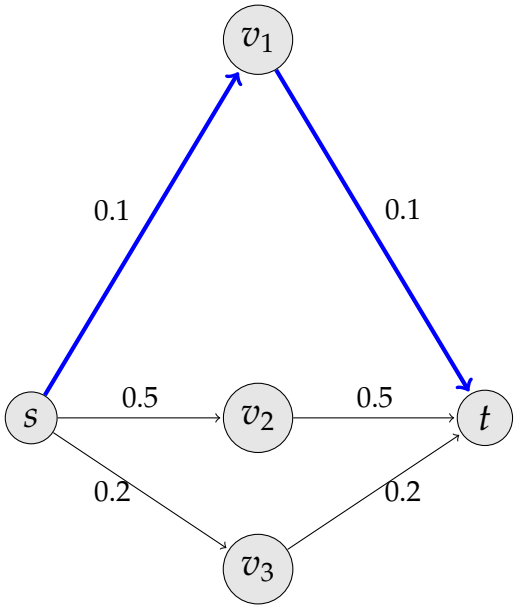


Figure 3.3: The safest path, which minimizes the probability of failure, is shown in blue.

In order to incorporate risk into path-finding, the first question to ask is how would one find the safest path. Let $r(e) > 0$ be the probability of failure while traversing an edge e where $r(e)$ and $r(e')$ are independent for every $e \neq e' \in E$. Then the probability of successfully traversing an edge is $1 - r(e)$, and the probability of successfully traversing the whole path is

$$f(P) = \prod_{e \in P} (1 - r(e)), \tag{3.2}$$

which we want to maximize.

3.2.1 Transformation to Dijkstra's Algorithm

Since the cost function is now a product of edge weights, instead of a sum, we cannot use Dijkstra's Algorithm as before. However, by using a logarithmic transformation, it is possible to transform the problem into a shortest path problem [11]. Since the logarithm of a product is a sum of logarithms we can define a new weight function

$$w(e) = -\log(1 - r(e)). \tag{3.3}$$

Now the cost of path P in terms of w is

$$
\begin{aligned}
f'(P) &= \sum_{e \in P} w(e) \\
&= \sum_{e \in P} -\log(1 - r(e)) \\
&= -\log(\prod_{e \in P} 1 - r(e)) \\
&= -\log(f(P)).
\end{aligned}
\tag{3.4}
$$

Since the logarithm is monotonically increasing, maximizing the logarithm of a function yields an equivalent solution to maximizing the original function. That is,

$$
\begin{aligned}
\operatorname*{argmax}_P f(P) &= \operatorname*{argmax}_P \log(f(P)) \\
&= \operatorname*{argmin}_P -\log(f(P)) \\
&= \operatorname*{argmin}_P f'(P).
\end{aligned}
\tag{3.5}
$$

Now we have shown that to maximize the path safety, $f(P)$, we can minimize $f'(P)$. And since $f'(P)$ is a sum of edge weights, we can use Dijkstra's Algorithm to find $f'(P)$. The only thing that remains to be shown is that the edge weights are non-negative. And since $r(e) \in (0, 1]$ we have that $w(e) = -\log(r(e)) \in [0, \infty)$.

3.3 The Best Path

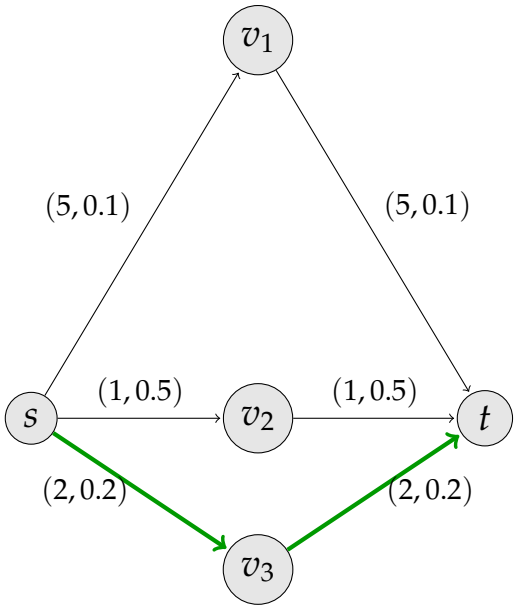


Figure 3.4: An example of a path that tries to minimize both distance and probability of failure, is shown in green. Note that the solution is dependent on how the objective function is defined.

Now that we know that both the shortest path and the safest path can be found efficiently with Dijkstra's Algorithm, we wonder whether we can find a good solution which incorporates the best of both worlds. For an edge e we now have two weights, length $l(e)$ and risk $r(e)$. The weight of e is then the tuple $w(e) = (l(e), r(e))$, as seen on Figure 3.4. The length of path P is defined as

$$L(P) = \sum_{e \in P} l(e) \tag{3.6}$$

and the probability of failure is

$$R_P(P) = 1 - \prod_{e \in P} (1 - r(e)). \tag{3.7}$$

$R_P(P)$ will also be referred to as the risk of P. Ideally, we would want an algorithm that can solve

$$\min_P F(P) = \min_P F(L(P), R_P(P)) \tag{3.8}$$

efficiently for any function F. Since there is no obvious way to define F we would like to experiment with different cost functions in order to find one which suites the problem. Unfortunately, no such algorithm exists. The

simplest and most obvious cost function, a linear combination of length and risk, $F(P) = L(P) + \lambda R_P(P)$, is NP-hard. Although NP-hard problems can often be solved efficiently in practice for reasonable inputs, such problems will not be considered in this thesis.

3.3.1 Choosing the cost functions

Pareto-optimality

Since length and risk are inherently bad properties for a path, we can safely assume that the cost function $F(P)$ should be increasing in both $L(P)$ and $R_P(P)$. Then a path P can be a solution only if there is no other path, P', for which $L(P') < L(P)$ and $R_P(P') < R_P(P)$, because otherwise P' would be better than P in both risk and length. Such paths are called Pareto-optimal paths and finding all such paths is NP-hard [3], as there can be exponentially many such paths (in regards to $|V|$).

Minimum Cost-Reliability Ratio

A popular choice for penalizing both cost and risk is the Minimum Cost-Reliability Ratio (MCRR),

$$F_{MCRR}(P) = \frac{L(P)}{1 - R_P(P)}. \tag{3.9}$$

Although it is NP-hard in general, it is possible to approximate it in polynomial time [12], and for integer inputs, a pseudo-polynomial algorithm exists [1].

Despite having been used for numerous different problems, MCRR is better suited for dealing with communication networks than vehicle paths. If we imagine that we want to send a packet from A to B and if it has not arrived by the time it should, we resend the packet. In this case, MCRR computes the expected time it takes to successfully send a packet from A to B [8]. Since resending packets is a cheap operation, the risk in this case is only due to the uncertainty of how long it takes to successfully send the packet.

For vehicles, on the other hand, the risk is a completely different cost than just variations in finishing time. If a failure occurs during traversal of a path, it might mean damage to the vehicle or even aborting the mission completely. In that sense MCRR provides no means for us to decide the importance of successfully finishing the mission or the cost of losing the vehicle, as risk is only potentially added time.

Tailor Fitted Cost Functions

To convert the problem to something tractable, some have tried to tailor $F(P)$ to the logarithmic transformation to get a cost function that is easy to compute [11]. One way to do that is

$$F(P) = \frac{C^{L(P)}}{1 - R_P(P)}. \tag{3.10}$$

Using the logarithmic transformation we get

$$\begin{aligned} F'(P) &= \log_C(F(P)) \\ &= \log_C(C^{L(P)}) - \log_C(1 - R_P(P)) \\ &= L(P) - \log_C(1 - R_P(P)) \\ &= \sum_e l(e) - \log_C(\prod_e (1 - r(e))) \\ &= \sum_e l(e) - \sum_e \log_C(1 - r(e) \\ &= \sum_e (l(e) - \log_C(1 - r(e)). \end{aligned} \tag{3.11}$$

By setting

$$w(e) = l(e) - \log_C(1 - r(e))$$

the problem can be solved efficiently by Dijkstra's Algorithm. The drawback is that there is no argument for why exponentiating the distance yields a relevant cost function, and it is easy to imagine a scenario where this cost function would be minimized by a bad path.

Suppose there are two available paths, one has length 100 and risk 0.5 and the other has has length 102 and risk 0. If $C > 2$ then the shorter path with 50% success rate is chosen instead of the slightly longer one with 100% success rate. In fact, adding just one unit of distance is equivalent to dividing the success rate with C. Of course, reducing C can resolve the issue in this particular scenario but the problem remains that for long distances, small changes in distance are equivalent to great changes in risk. And even if C is chosen with great care the algorithm will behave very differently for short paths compared to long paths.

3.3.2 Probability vs. Expected Value

Another way to allow Dijkstra to consider both risk and distance is to use the expected number of failures instead of the probability of one or more failures. Let $R_E(P)$ be the expected number of failures while traversing path P,

$$R_E(P) = \sum_{e \in P} r(e). \tag{3.12}$$

By choosing the cost function to be a linear combination of distance and expected number of failures, we get

$$\begin{aligned} F(P) &= L(P) + \lambda R_E(P) \\ &= \sum_{e \in P} l(e) + \lambda \sum_{e \in P} r(e) \\ &= \sum_{e \in P} (l(e) + \lambda r(e)). \end{aligned} \tag{3.13}$$

And by setting $w(e) = l(e) + \lambda r(e)$ we get

$$F(P) = \sum_{e \in P} w(e), \tag{3.14}$$

which can be solved by Dijkstra's Algorithm. Choosing F to be a linear combination of distance and expected number of failures also allows us to tune how risky the algorithm should be. In fact, λ can be seen as the cost of failure in terms of distance, which is an intuitive way of defining the relationship between risk and distance.

Union Bound

Now that we have a cost function that can be easily optimized, we wonder whether it is actually useful. One of the features of the expected value is that the expected number of occurrences is an upper bound on the probability of one or more events occurring. This is formally known as Boole's inequality or the Union Bound,

$$P\Big(\bigcup_i A_i\Big) \leq \sum_i P(A_i). \tag{3.15}$$

where $A_1, A_2, A_3, \ldots$ are a countable set of events. In our case, this means that the expected number of failures is an upper bound on the probability of one or more failures,

$$R_P(P) \leq R_E(P). \tag{3.16}$$

This inequality holds regardless of any correlation which may be between the risks of two edges, but as you may recall, when defining the risk of a path we assumed that the edge-risks would be independent. This is not a non-trivial assumption as real-world environments are far from random. To illustrate this point we can imagine that given an image from a stereo-camera, an algorithm determines that two adjacent vertices have each 50% chance of containing an obstacle. If the drone now visits one of these vertices and finds that it does not contain an obstacle, it strongly increases the probability of the other vertex containing an obstacle, because something was, in fact, seen by the stereo-camera. In this case the individual risks would be

negatively correlated and the $R_P(P)$ would be greater than anticipated by the independence assumption.

Although this example does not prove that the risk is negatively correlated in general, and neither is it supposed to, it serves as a reminder that the risk of a path assuming that all edge-risks are independent may not be what we want after all. Using the expected value frees us from having to assume anything about correlations and gives us the comfort of knowing that our solution is at least not riskier than we assume.

Expectation as an approximation

Suppose the cost function is defined in terms of expected number of failures, $F(P) = R_E(P)$, and the optimal path P is found using Dijkstra's Algorithm. An interesting question is, what does a particular $R_E(P)$ tell us about the risk, or given a particular $R_E(P)$, what is the range of values $R_P(P)$ can take?

Let $P_1 = (e_1)$ be a path of one edge, e_1. Then we have

$$R_E(P_1) = R_P(P_1) = r(e_1) \tag{3.17}$$

For a path of two edges, $P_2 = (e_1, e_2)$, we get

$$R_E(P_2) = r(e_1) + r(e_2) \tag{3.18}$$

and

$$\begin{aligned} R_P(P_2) &= 1 - (1 - r(e_1))(1 - r(e_2)) \\ &= r(e_1) + r(e_2) - r(e_1)r(e_2), \end{aligned} \tag{3.19}$$

so $R_E(P_2) > R_P(P_2)$. It looks like distributing the risk to many edges decreases $R_P(P)$ while having no effect on $R_E(P)$.

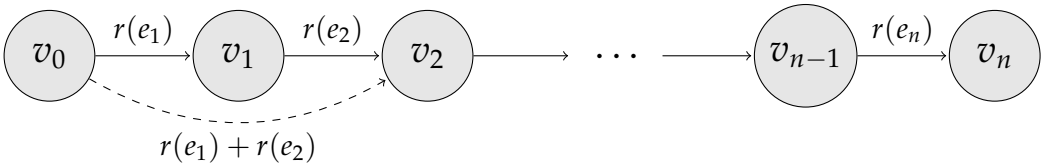


Figure 3.5: Example of how the first two edges of a path are merged into a new edge. The resulting path contains the dashed edge instead of the edges between v_0 and v_1, and v_1 and v_2.

Lets now show that merging the first two edges of any path P_n, as shown in figure 3.5, increases $R_P(P_n)$. First we have

$$\begin{aligned} R_P(P_n) &= 1 - \prod_{i=1}^{n} 1 - r(e_i) \\ &= 1 - (1 - r(e_1))(1 - r(e_2)) \prod_{i=3}^{n} 1 - r(e_i) \\ &= 1 - (1 - r(e_1) - r(e_2) + r(e_1)r(e_2)) \prod_{i=3}^{n} 1 - r(e_i). \end{aligned} \tag{3.20}$$

Now let P'_n be P_n after the first two edges have been merged. That is $P'_n = (e_{1,2}, e_3, \ldots, e_n)$ where $e_{1,2}$ is the merged edge and $r(e_{1,2}) = r(e_1) + r(e_2)$. Then we have

$$\begin{aligned} R_P(P'_n) &= 1 - (1 - r(e_{1,2})) \prod_{i=3}^{n} 1 - r(e_i) \\ &= 1 - (1 - r(e_1) - r(e_2)) \prod_{i=3}^{n} 1 - r(e_i) \\ &> 1 - (1 - r(e_1) - r(e_2) + r(e_1)r(e_2)) \prod_{i=3}^{n} 1 - r(e_i) \\ &= R_P(P_n). \end{aligned} \tag{3.21}$$

Now we know that merging the first two edges of a path increases the probability of failure, while keeping the expected number of failures constant. Note that the formula is independent of the order of the edges so merging any two edges has the same effect. Then we know that for a particular value of $R_E(P)$, the probability of a failure is maximized if P has only one edge, in which case $R_P(P) = R_E(P)$. That is the same result as we got from Boole's inequality, $R_P(P) \leq R_E(P)$. The more interesting question is, what is the minimum value $R_P(P)$ can take?

Since merging two edges increases $R_P(P)$, clearly the opposite holds as well. That is, taking some edge of P and splitting it up into two edges decreases $R_P(P)$. Therefore $R_P(P)$ is minimized when P has infinitely many edges with infinitely small risks. Let $P_n = (e_1, ..., e_n)$ and $r(e) = \varepsilon, e \in P, \varepsilon \in (0, 1]$. Then

$$R_E(P_n) = n\varepsilon \tag{3.22}$$

and by plugging in equation 3.22 we get

$$\begin{aligned} R_E(P_n) &= 1 - (1-\varepsilon)^n \\ &= 1 - (1 - \frac{R_E(P_n)}{n})^n \\ &= 1 - (1 - \frac{R_E(P_n)}{n})^{n\frac{R_E(P_n)}{R_E(P_n)}} \\ &= 1 - \big((1 - \frac{R_E(P_n)}{n})^{\frac{n}{R_E(P_n)}}\big)^{R_E(P_n)}. \end{aligned} \tag{3.23}$$

Then using the equation

$$\lim_{n\to\infty}(1 - \frac{1}{n})^n = \frac{1}{e} \tag{3.24}$$

we get

$$\lim_{n\to\infty} R_P(P_n) = 1 - \frac{1}{e^{R_E(P_n)}}. \tag{3.25}$$

So, now we know that

$$1 - \frac{1}{e^{R_E(P_n)}} \leq R_P(P_n) \leq R_E(P_n). \tag{3.26}$$

In figure 3.6 we see a comparison on the difference between the lower and upper bounds on $R_P(P_n)$ for a particular $R_E(P_n)$. Notice that the bounds are tight when $R_E(P_n)$ is small, which means that $R_E(P_n)$ is also an approximation for $R_P(P_n)$, for low risk paths. In our scenario it was deemed infeasible to send a drone on a risky mission, so the decision was made to assume that $R_E(P_n) < 0.1$. In that case we know that $|R_E(P_n) - R_P(P_n)| < 0.005$.

While the bounds give a theoretical guarantee for the worst case, we would also like to know what the difference is in a more likely scenario. In figure 3.7 we see the average difference in the probability of failure depending on whether the probability of failure was minimized directly or if the expected number of failures was minimized. The difference was calculated for 3×3 grids with random edge-risk, but 3×3 grids were found to have the greatest average difference. Although the maximum difference between $R_E(P_n)$ and $R_P(P_n)$ is 0.005 when $R_E(P_n) = 0.1$, the average difference in random grids is less than 10^{-6}. This tells us that the two strategies tend to find very similar paths and that using $R_E(P_n)$ instead of $R_P(P_n)$ is unlikely to affect the solution significantly, even for riskier missions.

Remarks

We have now shown that we can use the expected number of failures to get a cost function that is easy to compute, and we have also shown that

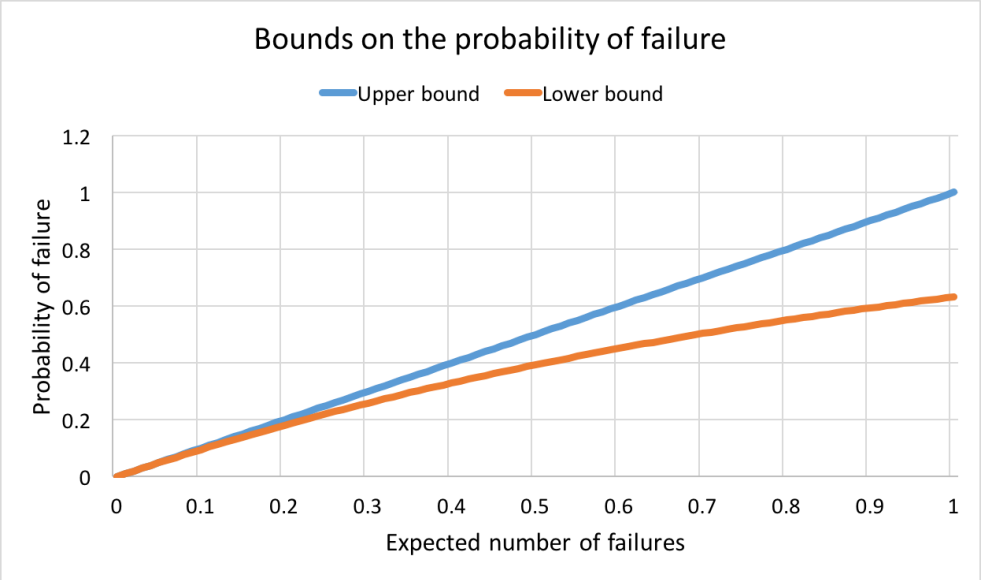


Figure 3.6: The relationship between the expected number of failures and the probability of failure. The two curves are the upper and lower bounds on $R_P(P)$ for a given $R_E(P)$.

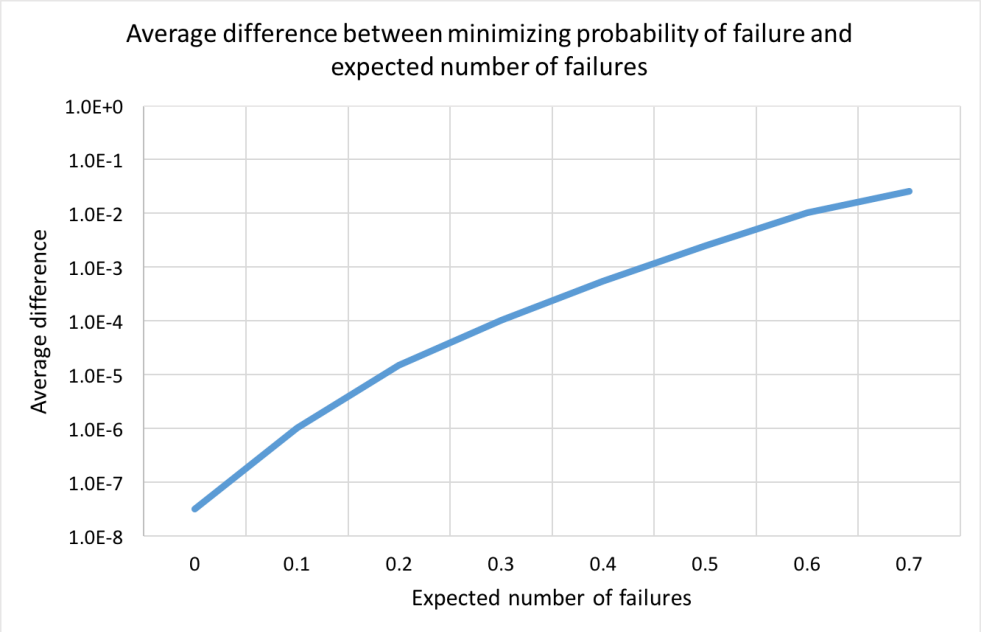


Figure 3.7: The average difference in the probability of failure, depending on whether it was minimized directly or if the expected number of failures were minimized.

the expected number of failures is an approximation to the probability of a failure. But most importantly, we have argued that a linear cost function of path length and expected number of failures is at least as reasonable as the other cost functions when it comes to pathfinding for vehicles. It gives an upper bound on the risk without any assumption on correlations and provides a simple and intuitive way of relating cost of risk to the cost of path length.

This is interesting due to the fact that despite finding numerous papers about various ways of combining probability of failure with path length, only one paper was found that used expected values, although it was for a completely different algorithm. For the remainder of this thesis the risk of a path shall refer to the expected number of failures for traversing the path.

3.3.3 Iterative consistency

Since path-finding algorithms for vehicles are generally run iteratively, the solution should not change if the algorithm is run again for only a part of the path unless there is new information. For example, if halfway through the path, the algorithm is run again, we would not expect the solution to change unless the graph had changed.

In figure 3.8 we see an example where the MCRR-path is found first from v_1 to v_3 and then again from v_2 to v_3. When we have reached v_2 most of the distance has already been covered, which makes the algorithm more prone to taking risks. The eventual solution will thus have more risk (and higher MCRR-cost) than the initially planned path, even though no information was gathered during travelling. This means that whether it is better to turn left or right at some vertex in the path is dependent on previous parts of the path.

This behaviour can, however, be perfectly reasonable for sending packets over the internet. That is, because in the case of failure, the packet must be resent from it's original sender. In that case, the consequence of failure at vertex v is that a new packet must travel from s to v again. Therefore the cost of failure at v should be related to the path from s to v. For vehicles, however, a failure has completely different consequences as the mission is either aborted, or in case of partial damage, the vehicle will try to continue from where it actually failed. In that case the cost of failure cannot be defined in terms of the previous distance, but rather the cost of aborting the mission or the cost of damaging the vehicle. Of course we might want longer missions to have more weight than shorter ones, but that is not a trivial assumption and the algorithm should allow for the importance of the mission to be defined explicitly, instead of assuming implicitly.

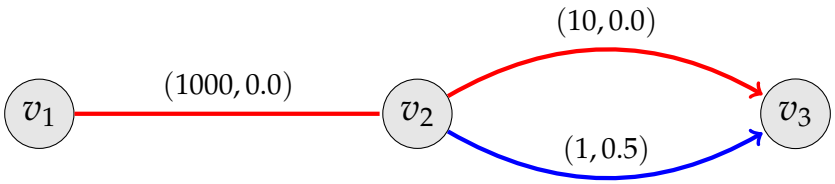


Figure 3.8: The red path minimizes the MCRR for a path from v_1 to v_3, and the blue path is similarly the optimal path starting at v_2.

Upper bounds

There is, however, one reason for why we might want the search to make different decisions at vertex v, depending on previous parts of the path. That is when the drone has limited amount of energy. In figure 3.9 we see a case where the risk is minimized under the constraint that a path can be of length at most 10. The optimal path from v_1 to v_3 is in red, and the paths from v_1 to v_2 and from v_2 to v_3 are blue and green respectively. Here the red path is completely different from the two other paths, even though they consist of the same vertices.

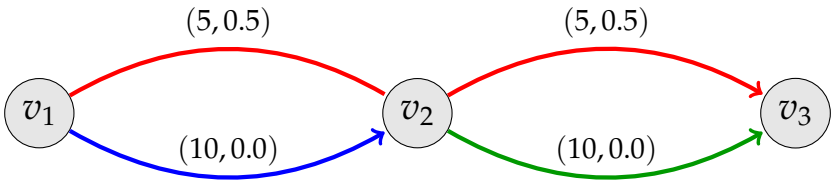


Figure 3.9: Three optimal paths, with different start and goal vertices, where risk is minimized under the constraint that distance can be at most 10.

This is known as a Constrained Shortest Path problem and it is known to be NP-hard [19]. That means that finding the shortest path with risk less than p or the safest path with distance less than d are both intractable and not a viable approach in our case. However, constraining the risk is very important in practice. If the drone determines that the only path is through a window with 0.99 risk, we would not want the drone to automatically try it.

3.4 Risk of exploration

So far we have defined risk to be due to imperfect maneuvrability and sensory, which may ultimately lead to a crash. That is, some edges have a fixed and known probability of causing failure when traversed. However, for partially known graphs there is another type of risk; risk of the path being blocked. In figure 3.10 we have two possible paths but we only know whether the edge from v to t is blocked when we are already at v. While the cost of the red path is c_1, the cost of the blue path is undefined as it may be blocked. To analyse the blue path we need to define a strategy S_b for deciding what happens if the blue path is blocked. In this case we can only go back to s and then take the red path to t. Now we can calculate the expected cost of S_b

$$F(S_b) = p_b(c_2 + c_3) + (1 - p_b)(2c_2 + c_1) \tag{3.27}$$

where p_b is the probability of the blue path being free. That is, with probability p_b we take the blue path which has cost $(c_2 + c_3)$. But with probability $1 - p_b$ we take the blue path from s to v and then turn around back to s, and finally take the red path to t, for a total cost of $(2c_2 + c_1)$. Intuitively, this is a hard problem to optimize because the solution might be a strategy with a unique path for every subset of unobserved edges.

This is a variant of the Canadian Traveller Problem (CTP) and is known to be #P-hard [18], meaning that it is at least as hard as NP-hard and finding an optimal solution is not feasible in practice. The CTP problem was originally defined in [18] as a game for two players where one player tries to minimize the path length while the other places obstacles in order to make the first player take a path which is much worse than the optimal. Both variants of the problem are hard to compute and as of yet, there has been no significant progress on approximation algorithms [16].

[16] and [4] give a heuristics approaches for solving the problem and in later chapters we will look at simple heuristics for our case.

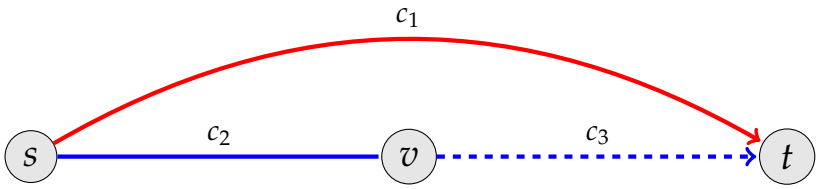


Figure 3.10: An example of a graph with risk due to unknown edges. The dashed edge may or may not be traversable but it can only be determined by visiting v.

3.5 The smoothest path

So far the cost function has been dependent on length and risk of the path, but having a smooth path is also desirable for vehicles [2]. Not only does each turn increase the total travelling time, energy usage and risk of crashing, but a straight path is also more pleasing to the eye and easier to understand than a tortuous one.

The cost of turning may be very different depending on both the mission and the vehicle, so the cost function should have a way to express the difference. The difference in time and energy usage for turning with different drones is also not necessarily linear in the angle of the turn. For example, one drone might make 45° turns with ease while 90° take more than twice as long as a single 45° turn. Another drone might find the need to stop before turning and therefore take roughly the same time for any turn.

The sensory system is also an important factor as sharp turns can lead the drone into unknown environment. For example, a camera with a field of view (FOV) of 90° can see 45° to the left and right, which means that a path with a single 45° turn will be in the cameras FOV the whole time, so the drone might not need to slow down at all. In figure 3.11 we see how the red path is contained within the FOV while the blue path is not.

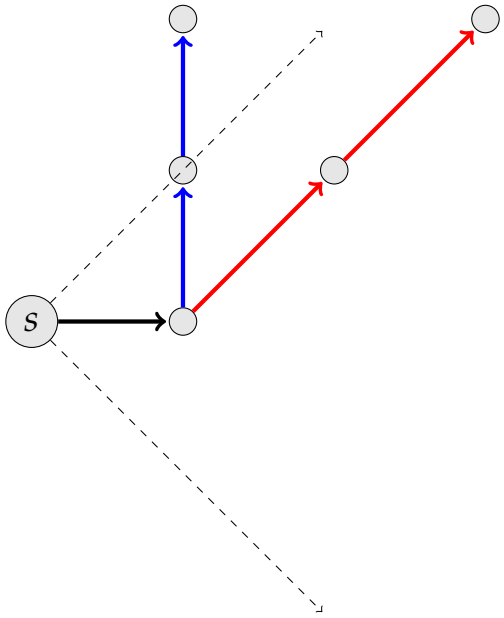


Figure 3.11: Example of the relationship between Field Of View (FOV) and cost of turning. The vehicle starts at s facing right, the dashed lines represent its FOV.

To make the model capable of handling the aforementioned versatility, the cost of $45°, 90°, 135°$ and $180°$ turns can be set independently.

Finding smoothness with Dijkstra's Algorithm

A key element of Dijkstra's algorithm is the property that the cost of a path from s to t that goes through vertex w, is the sum of the cost of going from s to w and from w to t. Unfortunately, this is not true if the cost of the path is the amount of rotation needed to traverse the path. On figure 3.12 we see two paths from s to w. The red path has three $45°$ turns ($135°$ in total), while the blue path has two $45°$ turns and one $90°$ turn, which amount to $180°$ in total. Since we want to minimize the total rotation, the red path is the optimal path from s to w. However, the optimal path from s to t is obtained by taking the blue path to w, even though it is not the optimal path from s to w.

Let $P_{s,w}$ be a path from s to w, let $P_{w,t}$ be a path from w to t, and let $P_{s,t}$ be a concatenation of $P_{s,w}$ and $P_{w,t}$. Let $S(P)$ be the sum of the total rotation of path P, and we will also call $S(P)$ the tortuosity of P. Now we have that

$$S(P_{s,t}) = S(P_{s,w}) + S(P_{w,t}) + angle(P_{s,w}, P_{w,t}) \tag{3.28}$$

where $angle(p_{s,w}, p_{w,t})$ is the rotation needed to get from the end of $p_{s,w}$ to the beginning of $p_{w,t}$. For Dijkstra's Algorithm to work, we need the cost to be

$$F(P) = F(P_{s,w}) + F(P_{w,t}) \tag{3.29}$$

If we can now make sure that $angle(p_{s,w}, p_{w,t}) = 0$ we have the property from equation 3.29, and then we could use Dijkstra's Algorithm to minimize $S(P)$. To do so we let a vertex represent a tuple of position and direction.

Let $G = (V, E)$, be a graph as before and $G' = (V', E')$ be G transformed. Now for each vertex in V we have eight vertices in V', one for each direction. Now for a path P in G' we have that $angle(P_{s,w}, P_{w,t}) = 0$ since w encodes both position and orientation. In figure 3.13 we see the graph in figure 3.12 after the transformation. Notice that the red and blue paths do not end in the same vertex any more as they don't end with the same direction.

Effects on running time

For a particular position, Dijkstra's Algorithm now needs to keep track of the shortest distance to that position ending in each of the possible directions. In our case there are 8 horizontal directions and 2 vertical. That is, diagonal movement is only permitted in the XY-plane. Therefore the number of nodes in the transformed graph, G', is tenfold that of the original graph.

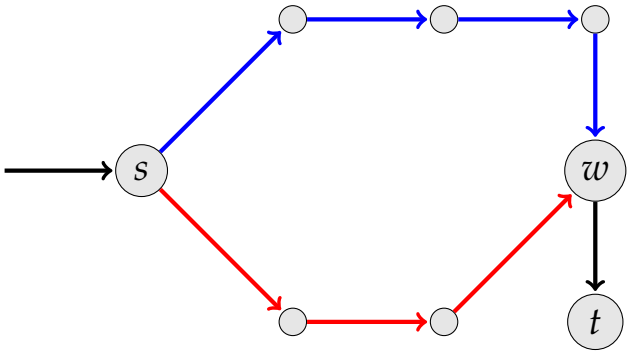


Figure 3.12: Example of a graph where the smoothest path from s to t does not contain the smoothest path from s to w, even though the former goes through w. Note that the incoming edge to s determines the initial direction at s.

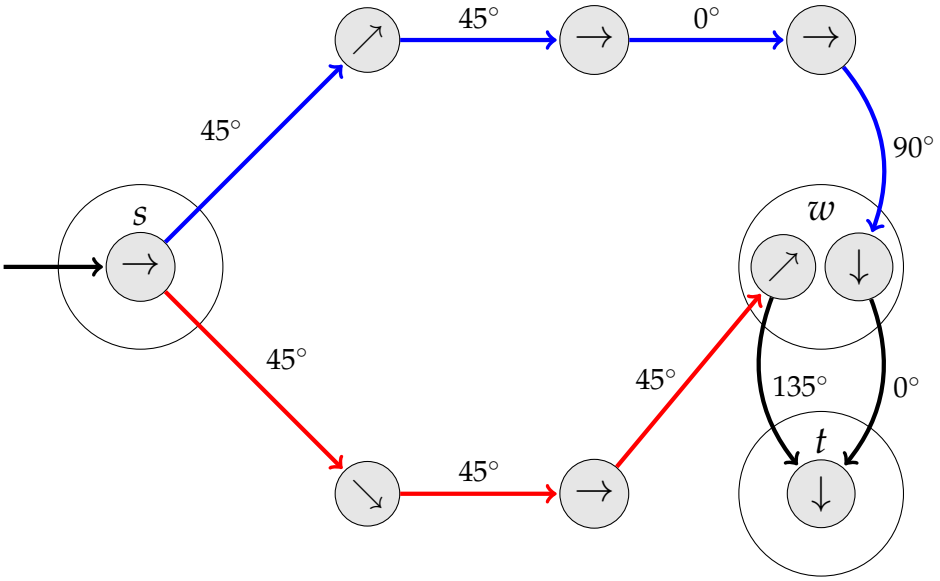


Figure 3.13: The graph from figure 3.12 transformed such that each vertex represents a position and a direction. For each vertex in figure 3.12 there are 8 vertices in the new graph, one for each possible direction. Note though, that isolated vertices have be omitted from this figure. The weight of an edge is the difference of the directions of its vertices in degrees.

Asymptotically, Dijkstra's Algorithm has the complexity $\Theta(n \log n)$ for $n = |E|$. The transformed graph has ten times as many vertices (and edges) as the original graph, and since the complexity is super-linear the runtime can be even worse than tenfold. Although worst case analysis is important for understanding the algorithm it is worth noting that eventually A* will be

used, which is highly dependent on the quality of its heuristics. In later chapters the runtime will be measured empirically.

3.6 Dynamic Search Algorithms

Pathfinding in robotics often deals with the problem of finding the shortest path over and over again. As the robot moves closer to the goal it gains information about its environment which can render its current path obsolete. Dynamic algorithms are used to avoid having to compute the shortest path for multiple graphs that differ only slightly. Algorithms like D*, D* lite and Anytime Dynamic A* assume that t will stay constant while s can change since the vehicle is the only thing that is moving. Since the sensors also tend to measure the space near the drone's position, the algorithms can utilize the fact that the changes in the graph will occur close to s. By searching from t to s and building up the shortest path tree from t, most of that tree will still be valid even though the graph changes close to s. In figure 3.14 we see how the robot moves and some branches of the search tree become invalid and need to be recalculated. If the range of the robot's sensor is much smaller than the length of the path, then intuitively only a small part of the shortest-path-tree needs to be fixed.

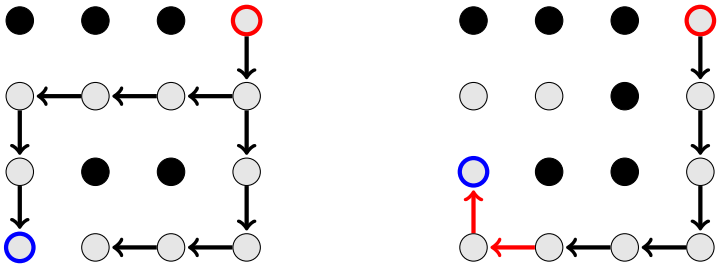


Figure 3.14: Example of a dynamic search where the blue vertex is the source, s, the red vertex is the goal, t, and the black vertices are obstacles. On the left is the shortest-path-tree after searching in the opposite direction, from t to s. After moving up, an obstacle is observed on the current path so a new shortest-path tree needs to be calculate. On the right we see the new shortest-path-tree where the red edges are the only ones that need to be calculated.

Despite the obvious benefits of dynamic search there are quite a few drawbacks which need to be weighted carefully against the potential gains. One drawback is that all the data gathered during the search needs to be stored, although that is not a significant drawback in our case. A significant drawback is, however, the fact that the transformation from chapter 3.5 creates 10 vertices for each vertex in the original graph. That means that for a single

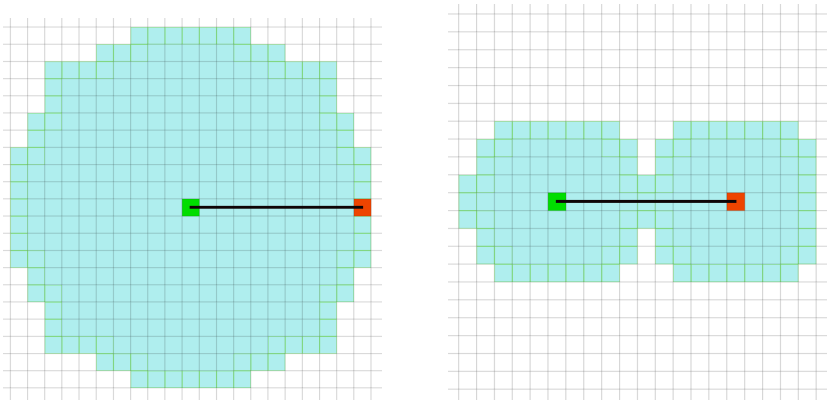
measurement we have up to ten updates in the search-graph. Finally, the algorithm might want tune parameters in the cost function regularly, but any change in the cost function renders the whole search-tree obsolete. In order to incorporate that into a dynamic algorithm one would need to keep track of multiple trees and again multiply the number of updates needed per measurement.

3.7 Bidirectional Search

Adding smoothness to the cost function introduces a new dimension to our previously three dimensional search problem, which could prove to have a very bad effect on the runtime. Although the dimension is constrained such that each position can have at most ten directions, it still means that there are 10.000 vertices corresponding to a 10 m wide cube.

For graphs with a high branching factor, it is often beneficial to use bidirectional search algorithms. Searching from s to t while simultaneously searching from t to s can reduce one long search to two shorter ones. In figure 6.4 we see vertices expanded by a bidirectional version of Dijkstra's Algorithm compared to the original. Each search grows its own shortest path tree until the two trees meet. Geometrically, the intuition is that when searching for a path in a plane, the number of visited vertices tends to increase with the square of the distance between s and t. By searching in both directions simultaneously, instead of one large disc, we get two smaller discs with half the radius of the large one. Due to the quadratic growth of visited vertices compared to the distance, the number of visited vertices is halved by this approach. For four dimensional space, a bidirectional version of Dijkstra could be up to 8 times as fast as the original. It is worth noting though, that two of the dimensions (altitude and direction) are limited, which reduces the potential speed up.

Although Dijkstra's Algorithm tends to build up a sphere of expanded vertices around s, the same cannot be said about A*. By using overestimated heuristics to speed up the search, it is hard to say whether searching bidirectionally will prove to give a significant increase in performance. In (ALT reference) A* was used to find the shortest path on large roadmaps where distance from preprocessed landmarks were generated using the Triangle inequality. Searching bidirectionally was found to increase the algorithms efficiency significantly [7]. In later chapters, the effect of searching bidirectionally will be measured empirically.



(a) Unidirectional search (b) Bidirectional search

Figure 3.15: A comparison of vertices expanded by a bidirectional variant of Dijkstra's Algorithm and the original unidirectional algorithm.

Chapter 4

Simulation Environment

4.1 Software Components

4.1.1 ROS

The Robot Operating System (ROS) is a set of software frameworks for robot software development which provides standard operating system services. A ROS-based process is called a node, and different nodes can communicate by sending messages through ROS. For example, the pathfinding algorithm was implemented as a ROS node and communicates with other parts of the system that also have ROS nodes.

4.1.2 Gazebo

Gazebo is a 3D simulator which generates sensor feedback and physical interactions between robots and other objects, and it is often used with ROS to simulate a robot defined by ROS. Gazebo can simulate various sensors, and we will use a depth camera to inform the algorithm about the environment. The depth camera measures the distance to points in its FOV and publishes the data as a point cloud (set of points in 3 dimensions). In figure 4.1 we see a screenshot of the Gazebo simulator.

4.1.3 PX4

PX4 is an autopilot hardware and software and the drone simulated in Gazebo is controlled by the PX4 autopilot software. The autopilot communicates with the rest of the system through a ROS node called Mavros, using the MAVLink communication protocol.

4.1.4 OctoMap

In order to track possible obstacles, the OctoMap library is used. OctoMap implements a probabilistic 3D occupancy grid which is particularly suited for robotics [10]. The octomap_server implements an OctoMap as a ROS node. In figure 4.1 we see a visualization of a OctoMap representation and the corresponding environment in Gazebo.

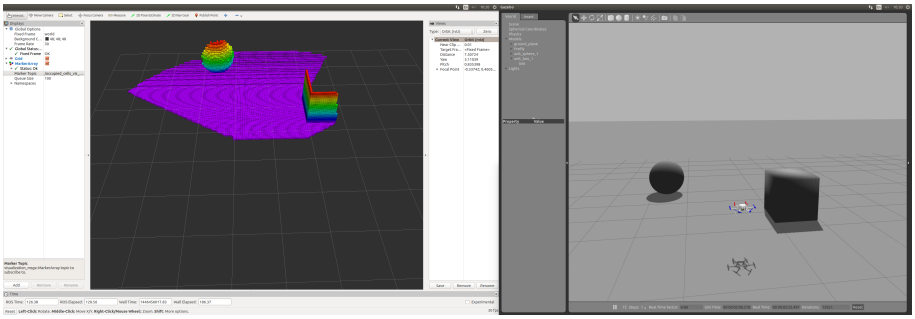


Figure 4.1: On the left we see a visualization of the OctoMap for the Gazebo simulation on the right. The program Rviz was used to visualize the OctoMap.

Data Structure

The probability of each unit of space being occupied is stored efficiently using an octree, as shown in figure 4.2. That is, each node in the tree represents a cubic three-dimensional subspace and keeps track of the probability of the space containing an obstacle.

The tree structure allows the OctoMap to merge together 8 siblings with the same probability of being occupied for memory efficiency. OctoMap defines a maximum and minimum probability of occupancy and when free space has been observed for a sufficient amount of time, many leafs reach the minimum probability and can subsequently be merged.

Algorithm

For every obstacle point, the Octomap traces a ray from the camera to the point, decreasing the probability of all the voxels in between, but increasing the probability of the voxel containing the point.

To keep track of the probability of a voxel being occupied, the OctoMap uses Recursive Binary Bayes Filter. For efficient updating OctoMap uses log-odds which are easy to compute. That is, for prior probability $p_{1:t-1}$ and update

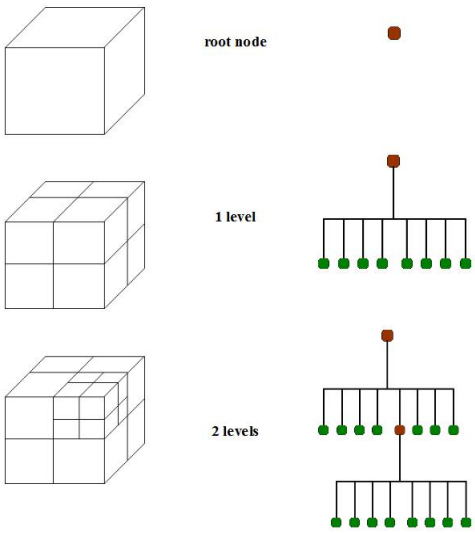


Figure 4.2: An example of discrete three-dimensional space with varying resolution represented with octrees [14].

probability p_t at time t, the update rule is

$$L(p_{1:t}) = L(p_{1:t-1}) + L(p_t) \tag{4.1}$$

where L is the log-odds function,

$$L(p) = \log(\frac{p}{1-p}) = \log(p) - \log(1-p). \tag{4.2}$$

In the model p can only take two values, which allows the OctoMap to do probabilistic updates using only sums.

4.2 Communications

ROS provides the means of communication between different parts of the system and in figure 4.3 we see a graph of the relationships between ROS nodes. Gazebo and OctoMap have dedicated ROS nodes for sending and receiving messages while the PX4 autopilot uses the Mavros node for communicating.

4.2.1 global_planner_node

The global_planner_node runs and handles communication for the global planner. That is, it receives information about the current position, goal position and position of obstacles, and runs the planning algorithm when needed. When a new path is found it publishes new waypoints to path_handler_node.

4.2.2 path_handler_node

The path_handler_node handles communications between the global_planner_node and Mavros, which is a communication driver for autopilots. It gets new paths from the global_planner_node and while listening to the current position it streams the next waypoints to Mavros. The path_handler_node also keeps track of the path back which can be used in emergencies.

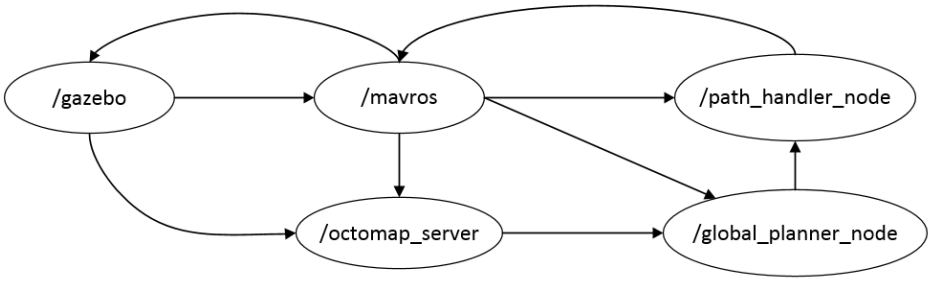


Figure 4.3: A graph which shows how the nodes of the system communicate

Chapter 5

Implementation

5.1 Cost function

When it comes to drones, there are three criteria that we want to minimize; traversal time, energy needed to traverse the path and the risk of traversing the path. But so far, we have only seen how to minimize path length, tortuosity and risk. Luckily, it turns out that traversal time and energy usage can be expressed in terms of path length and tortuosity.

5.1.1 Distance

The path length affects both time and energy in very similar ways. We assume that each meter of horizontal movement has a fixed cost, which is due to the time and energy that is needed to move one meter horizontally. For vertical movement, however, it is more expensive to move upwards than downwards. Although moving upwards and downwards may or may not take the same time, the energy consumption for upwards movement is always greater.

Let $L_h(P)$ be the horizontal distance of P, that is $L(P)$ excluding vertical movement, and let similarly $L_u(P)$ and $L_d(P)$ be the upwards and downwards distance of P respectively. Now we just need to define the constants c_h, c_u and c_d to represent the cost of horizontal, upwards and downwards distance respectively. Then we have the following cost of distance,

$$F_d(P) = c_h L_h(P) + c_u L_u(P) + c_d L_d(P). \tag{5.1}$$

We could imagine c_h as being composed of

$$c_h = c_t t_h + c_e e_h \tag{5.2}$$

where t_h and e_h are the time and energy needed to move one meter horizontally, and c_t and c_e are the unit-costs of time and energy. Then the cost of distance can be transformed into just time and energy.

5.1.2 Smoothness

Similarly to the path distance, each turn increases both time and energy usage. As discussed in chapter 3.5, the cost of a turn does not need to be linear in the number of degrees of the turn. The tortuosity of path $P = (e_1, \ldots e_n)$ is defined as

$$F_S(P) = \sum_{i=1}^{n-1} f_s(angle(e_i, e_{i+1})) \tag{5.3}$$

where *angle* returns the rotation needed to traverse two adjacent edges. Then $f_s(\theta)$ is the cost of turning θ degrees, which, like before, can be seen as

$$f_s(\theta) = c_t t_s(\theta) + c_e e_s(\theta) \tag{5.4}$$

where $t_s(\theta)$ and $e_s(\theta)$ are the time and energy needed to make a θ degree turn.

For simplicity and to penalize sharp turns, the cost of one turn was defined to be quadratic in the number of degrees of the turn. That means that a 90° turn costs four times as much as a 45° turn, which pushes the algorithm to choose two 45° turns instead of a single 90° turn.

5.1.3 Risk

The risk is the most complicated part of the cost function. This is because the probabilities given by the OctoMap are the probabilities of a vertex being occupied, not the risk of traversing the vertex. The risk of traversal needs to be inferred from the occupation probability, and tested empirically. The risk of traversal is assumed to be composed of the following probabilities.

OctoMap occupation probability

The OctoMap supplies the algorithm with the probability of a voxel containing an obstacle. Although travelling through an occupied voxel does not necessarily end with a crash, it is defined as a crash in the cost function.

OctoMap neighborhood risk

Since the drone has imperfect sensors and maneuvrability, being close to an obstacle is risky, since an obstacle might be misplaced by the sensor, or the drone might find itself in a different position than planned. The neighborhood risk is simply the sum of the occupation probabilities of the adjacent voxels multiplied by constant, p_n. p_n can be seen as the probability of the drone being in a neigboring vertex. The better the sensors are, and the better the drone can follow a path, the less risk is induced by the neighborhood.

Risk for unexplored space

As discussed in Chapter 3.4, unexplored space has a different kind of risk than explored space. Essentially, planning through unexplored space induces risk of having to plan a new path, and the new path might be very costly in terms of the previously defined costs. To get a useful estimate of this risk we use a prior probability $p_{prior}(v_z)$ of a vertex v containing an obstacle depending on its altitude. In figure 5.1 we see how higher altitudes have a lower probability of being occupied and the risk of having to recalculate the path is lowered by flying at high altitudes. The altitude prior is essentially the risk of blind traversal, so it does not encapsulate the risk of having to plan a new path. It does, however, prove to be very useful, as we will see in the next chapter.

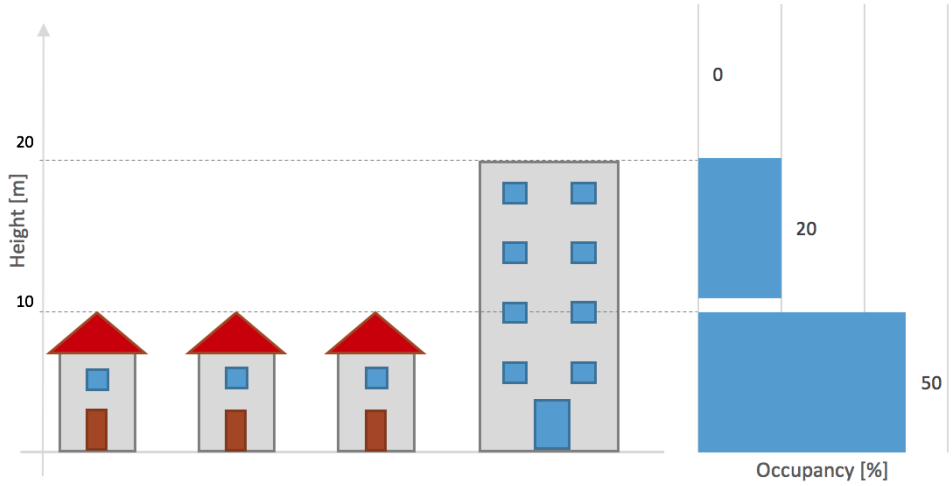


Figure 5.1: An illustration of the height prior probability, which is the proportion of occupied space at a particular height. In this example 50 % of all space below 10 m, and 20 % of all space between 10 m and 20 m, is occupied.

5.2 Heuristics

What makes A* so popular and gives its speed in practice is its use of case specific heuristics. In theory, the heuristics must be a lower bound on the cost function for the solution to be optimal. But to unleash its true power, the optimality constraint can be dropped to make way for good heuristics that are not necessarily lower bounds. Since the cost function is defined in terms of distance, tortuosity and risk, the heuristic function should reflect that.

5.2.1 Distance

The 'straight-line distance' is the most popular heuristics for pathfinding but there are a few ways to define a lower bound on distance. The euclidean distance can be used but it does not reflect path distances on grids and it is relatively hard to compute due to its square root. The Manhattan-distance is easy to compute but it is not a lower bound, since it does not consider diagonal moves. To get a tight lower-bound on paths with diagonal moves, the following formula can be used,

$$h_d(u) = |u_x - t_x| + |u_y - t_y| + |u_z - t_z| - (2 - \sqrt{2}) \min(|u_x - t_x|, |u_y - t_y|) \tag{5.5}$$

where $u = (u_x, u_y, u_z), t = (t_x, t_y, t_z)$ are two points on the grid, and t is the goal as per usual. Intuitively, the first three terms are the Manhattan-distance, while the fourth subtracts what can be gained by using diagonal moves. Note that diagonal moves are only allowed in horizontal directions.

5.2.2 Smoothness

A lower bound on the tortuosity can be found by calculating the number of 45° turns that are needed to get from the current pose to the goal. Since the cost of rotation is quadratic the cost is minimized by always taking 45° turns. In figure 5.2 we see how one 45° turn is not enough to reach t, but two are. Then we know that the cost of tortuosity is

$$F_S(P) \geq 2 f_s(45°). \tag{5.6}$$

If our current position is not at the same altitude as the goal and is not directly above or below it, then we also need to make at least one change between vertical and horizontal movement.

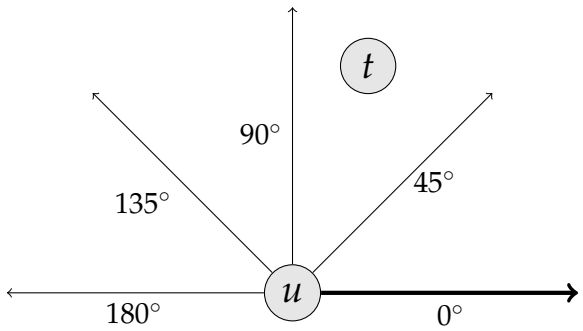


Figure 5.2: An example of heuristics for smoothness. When positioned in u and facing right, at least two 45° turns are needed to reach t.

5.2.3 Risk

Since there is no limit for how little risk a voxel can have, there is unfortunately no lower bound for the risk of the safest path. If there exists a path through completely known empty cells, the cost due to risk is 0. But since risk is such an integral part of the cost function (and this thesis), we will let a non-admissible heuristic function suffice. Now, since we can choose the heuristic function, we will use the most common case; a path through completely unknown space. That is, the heuristics for the cost of risk will be the cost of risk for the shortest path where every voxel is unknown. We use the shortest path that leads horizontally to the xy-coordinates of the goal and then vertically to the goal. In that case, the risk-heuristics for vertex u is

$$h_r(u) = p_{prior}(u_z)h_d(u). \tag{5.7}$$

Since the goal tends to be of low altitude, or at least close to an obstacle, it is often a cell with high risk. Therefore, the algorithm will look at other options until it finds that it can't get rid of this high risk. A nice way to counter that is to add the risk of t to the heuristic function. Then we get

$$h_r(u) = p_{prior}(u_z)(h_d(u) - 1) + risk(t). \tag{5.8}$$

where $risk(t)$ is the risk of traversing t using the previously defined cost function. Now we assume that we pay the unknown cost for every cell of the shortest path up to but excluding t, plus the risk of t.

Note that although we are not guaranteed to find the optimal solution using these heuristics, we are using the cost of a path with no obstacles so intuitively the non-optimal solutions will come up when the cost of the path is low anyway.

5.2.4 Tuning

An important parameter in the heuristic function is the overestimation factor, α.

$$h_\alpha(u) = \alpha h_d(u) + h_s(u) \tag{5.9}$$

By choosing $\alpha > 1$ the search may disregard a vertex u on the optimal path because the heuristic function gives it a false lower bound on the cost needed to get from u to t. However, the solution can be at most an α factor from the optimal. Note though, that we already have a non-admissible heuristic function, $h_r(P)$, for the risk and, in the next chapter the use of $h_r(P)$ will be compared to tuning α.

5.3 Pipe-Line

Since the algorithm can require numerous iterations, it is important to have some way of tuning the algorithm such that it finds a path quickly, even though it may be far from optimal. To do so, the algorithm runs the search several times with a decreasing overestimation factor, but with a limited number of total iterations. Since the overestimation only overestimates the distance heuristics, a large α will mostly disregard the cost of tortuosity. In that case, there is no point in doing the smoothness transformation from Chapter 3.5, since it would only slow down the search. The first few searches use a function called $search3D()$ which only considers the distance and risk. With each search, α gets smaller, yielding a better solution. And when $\alpha < 2$, the function $search4D()$, which also considers smoothness is called instead. If the time runs out before any solution is found, a 2-dimensional search is called at the maximum altitude. This is because 2-dimensional search is much faster and it is only in rare cases where it is not possible to find a path at high altitudes. A pseudo-code for the pipe-line can be seen in Algorithm 3.

Algorithm 3 RunSearch

```
1: α ← α_max
2: foundPath ← False
3: while α > α_min and numIterations < maxIterations do
4:    if α > 2 then
5:       foundPath ← search3D()
6:    else
7:       foundPath ← search4D()
8:    end if
9:    α ← α/2
10: end while
11: if not foundPath then
12:    search2d()
13: end if
```

Chapter 6

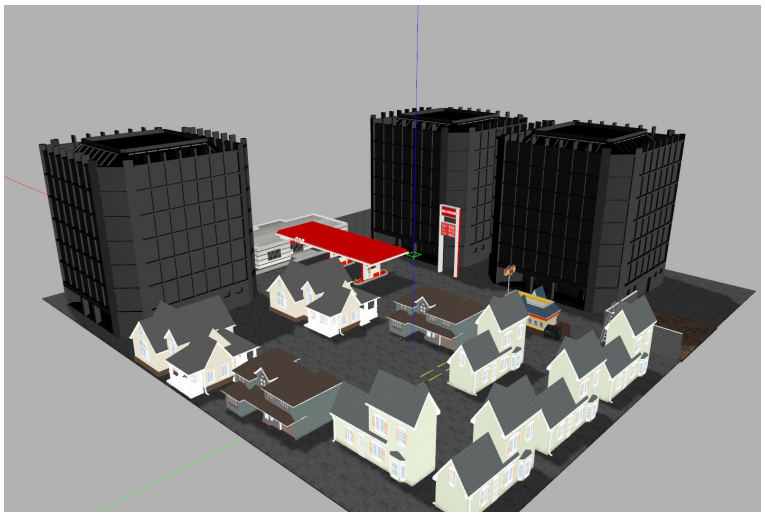
Results

6.1 Test settings

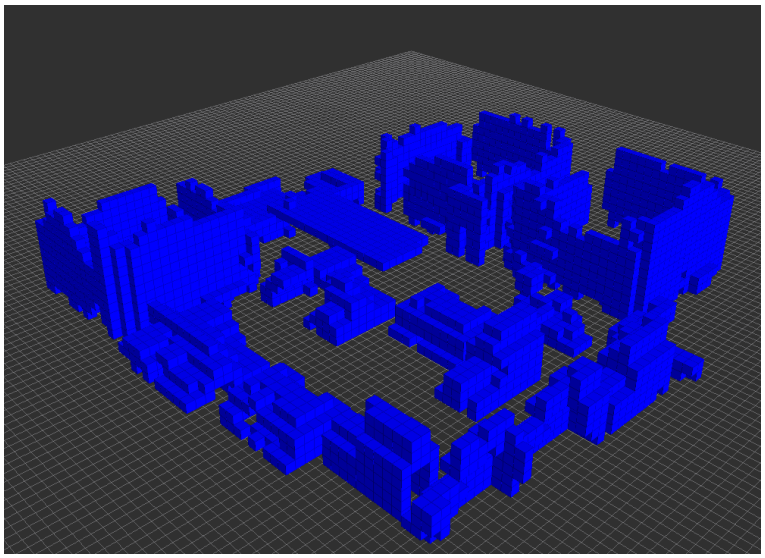
In this chapter, different search strategies are compared using simulation. An 80 m $\times$ 80 m 'world' was created to capture both realistic and hard scenarios for the algorithm, and a screenshot of the world can be seen in figure 6.1. The three office buildings provide an important challenge because of their large surface area. They are all of the same size and a single facade is about 200 m^2. Two of the office buildings are positioned close together to test a common decision problem of whether it is better to take a short risky path or a longer and safer path.

The algorithm was fed 25 random points, which were known to be reachable from the start position, and as soon as one point was reached a path was planned to the next point. A distance sensor was also mounted on the drone to measure the distance to the nearest obstacle. If the drone was within 10 cm distance to an obstacle at any point, the drone was considered to have crashed and the mission aborted.

The cost function was defined such that safety was most important, then traversal time, and finally the energy usage was defined as least important.



(a) The world



(b) OctoMap

Figure 6.1: (a) The 'world' in which all the simulated tests were run. (b) A vizualization of the three dimensional grid of the data gathered during a simulation given by OctoMap. The blue cubes represent positions which have a probability of being occupied above a certain threshold.

6.2 Comparison of different strategies

6.2.1 Comparison of search in two and three dimensions

In figure 6.2 we see a comparison on the traversal time, energy usage, failure rate and cost between 2D and 3D search in the simulated world. The 2D search only looks at paths at 2 m altitude while the 3D search considers paths beneath 10 meters. The 2D search is worse than the 3D search in every aspect, even in expanded vertices. This is because the environment has much more obstacles at 2 m than at higher altitudes, which makes it much harder to find a path. Although the 3D search did much better than the 2D search in terms of traversal distance and time, it did not reduce the failure rate significantly.

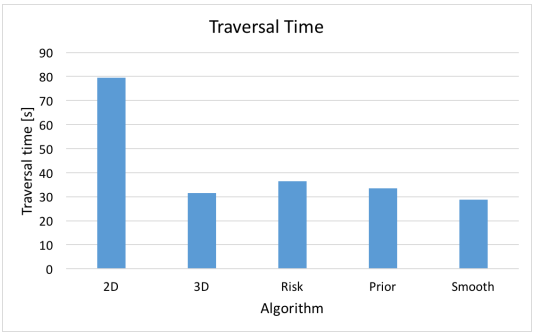
6.2.2 Searching with risk

Adding risk to the cost function immediately reduced the failure to about a quarter of the failure rate for the original 3D algorithm. In figure 6.2, Risk represents the results where risk has been added to the cost function. As we can see in the figure, the traversal time slightly increased, but that was to be expected since the algorithm was no longer finding the shortest paths.

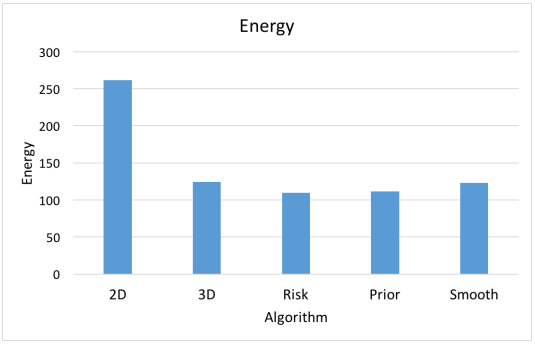
6.2.3 Flight Altitude

Using the height prior defined in Chapter 5, the drone will prefer certain altitudes. In our case, the prior decreases with height, so flying at high altitudes was considered safer than flying close to the ground. But since gaining altitude uses a lot of energy, there is a trade-off between decreasing risk by increasing altitude, and reducing energy usage by keeping current altitude. Whether it is worth it to spend energy on increasing the altitude is dependent on how far away the goal is. In figure 6.3 we see a plot of the altitude chosen by the algorithm for a particular prior, depending on the distance to the goal. This is assuming the drone starts at 1 m above ground and has no information about the world. This allows the drone to spend energy to go above an obstacle early on in the mission, since it knows that the higher altitude is beneficial in the long term.

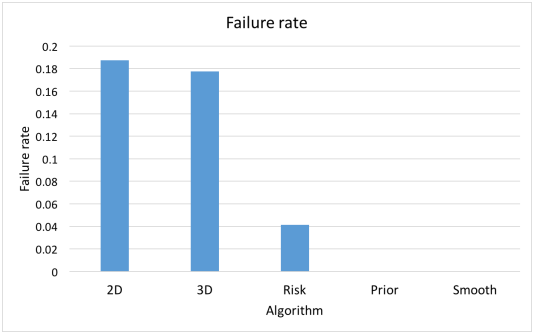
In figure 6.2, Prior represents the results after adding the prior to the cost function. The height prior decreased the traversal time and the failure rate, while slightly increasing the energy usage, which resulted in an overall decrease in cost. This is because the prior provides information about which altitudes are more likely to have good paths, and reaching a high altitude early on can save a lot of time.



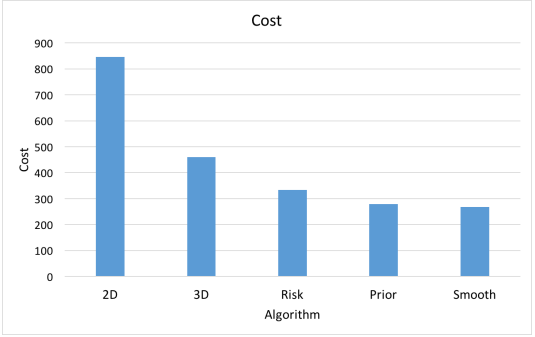
(a) Traversal time



(b) Energy usage



(c) Error rate



(d) Cost

Figure 6.2: A comparison of the traversal time, energy usage, error rate and cost, for different search strategies.

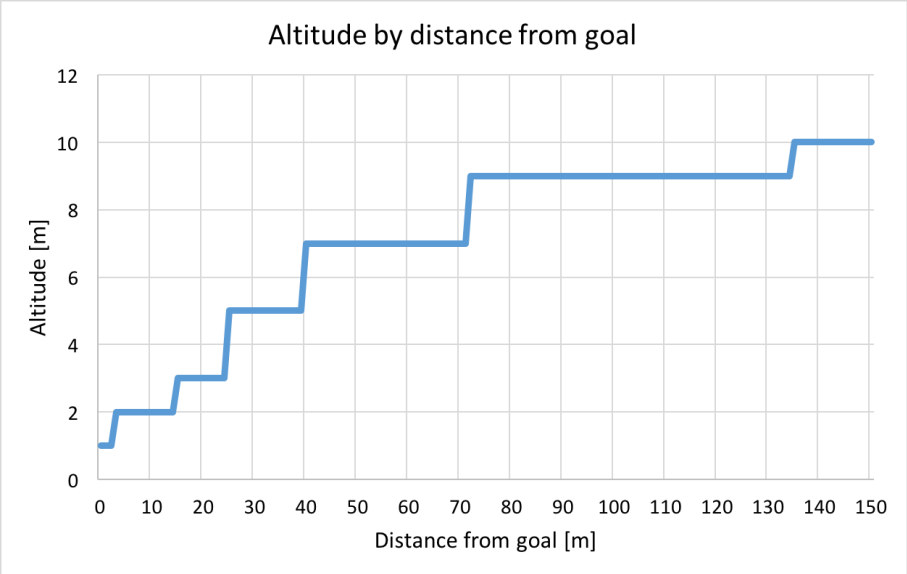


Figure 6.3: The flying altitude by the distance to goal. This is assuming nothing is known about the world and both the starting position and the goal are at 1 m above ground.

6.2.4 Smoothness

In figure 6.2, Smooth represents the results after adding smoothness to the cost function. By increasing the smoothness of the path, both traversal time and total cost were decreased. As expected, optimizing smoothness also proved to increase the number of iterations significantly.

6.3 Visual comparison of different strategies

Although not part of the cost function, aesthetic properties of the strategies should also be noted. A planner that chooses paths that are intuitive and predictable is preferable as it is arguably less likely to generate a corner case for the controller.

The three strategies were run on a simple scenario which showcases their difference. In figure 6.6 we see how Risk flies over the first house, but instead of keeping its altitude, it starts to descend and then needs to fly up again when it reaches the second house. This happens because of how the stereo camera and the OctoMap work together. The OctMap can only determine space to be free if the stereo-camera sees an obstacle behind it. Since there are more obstacle below the drone, and close to the ground, the drone goes down again where the risk is slightly lower. With the prior probability, the algorithm keeps its altitude as it knows that it is less likely to encounter

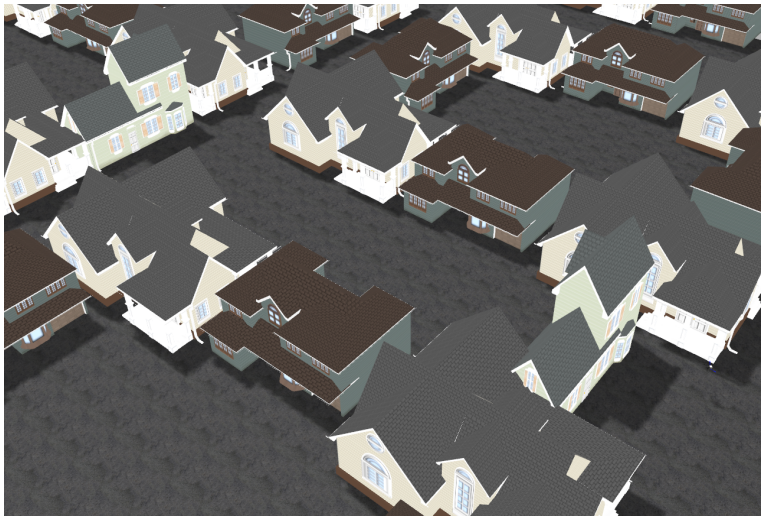


Figure 6.4: The world used for comparing the strategies visually.

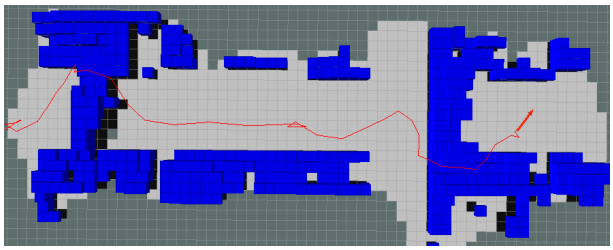
problems that way.

Looking at figure 6.5 we see that the Risk and Prior turn into the middle of the street which means that it again needs to turn to go around the second house. The middle of the street is slightly safer, but not enough for Smooth to alter its direction. In figure 6.6 we see how Risk decreases altitude before it reaches the second house while Prior and Smooth maintain current altitude.

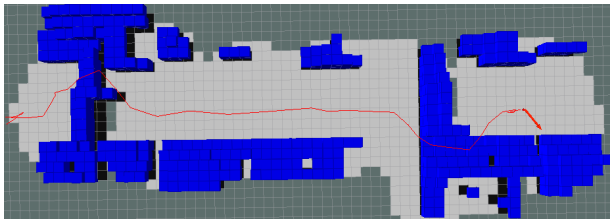
Human Pilot

For comparison, the same test was conducted with a human pilot. Since the planner can only make decisions based on data from sensors, it is solving a different problem than a human that can see the whole map. Therefore two tests were conducted; first where the pilot could only see the obstacle map, and then secondly where the pilot knew the environment and could see the whole map. As we can see on figures 6.5(d) and 6.6(d), the planner did better than the human pilot when the pilot did not know the environment, but the difference might be due the problem being unnatural for humans. In figures 6.5(e) and 6.6(e) we see that when the human knows the environment, it chooses a smooth path that goes straight over the houses.

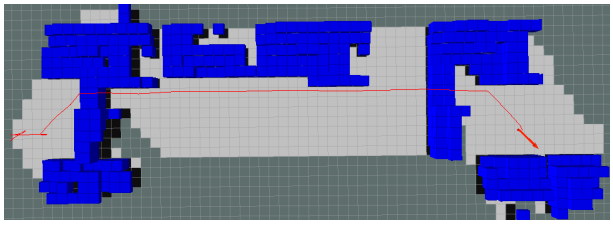
Of course, this analysis only scratches the surface of a complicated problem, but it does give some insight into the differences between the strategies, and which features are desirable despite not being part of the cost function.



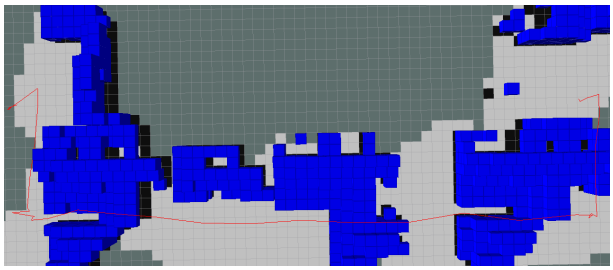
(a) Risk



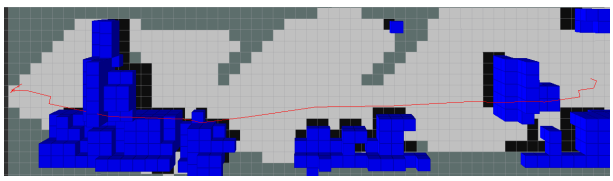
(b) Prior



(c) Smooth

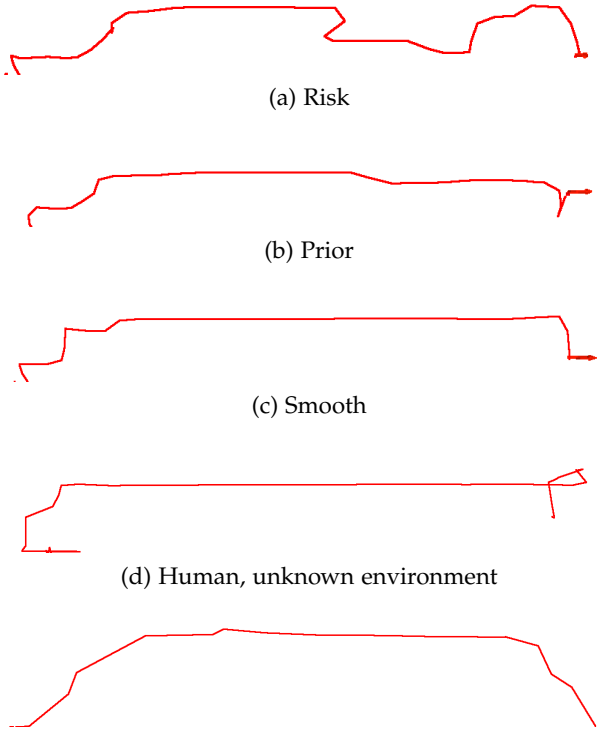


(d) Human, unknown environment



(e) Human, known environment

Figure 6.5: Visual comparison of the trajectories for the planner and Human pilot.



(a) Risk

(b) Prior

(c) Smooth

(d) Human, unknown environment

(e) Human, known environment

Figure 6.6: Height plots for the tests in image 6.5

6.4 Heuristics

6.4.1 Overestimated Heuristics

Recall that A* can be seen as a mixture of Dijkstra and Best First search, and as long as the heuristics are not overestimated, an optimal solution is guaranteed. With overestimated heuristics, A* acts more like Best-first search and greedily expands vertices near t. While this eliminates the optimality guarantee it can greatly reduce the number of expanded vertices and therefore speed up the algorithm. On figure 6.7 we see how the search only expands vertices close to obstacles. This is because the vertices that are not explored due to overestimated heuristics are part of paths with similar cost as the one returned by the solution. This makes the search focus on getting past obstacles instead of finding small changes in the path which could reduce the cost slightly.

On figure 6.8 we see a comparison of the number of nodes expanded and solution cost, depending on how much the heuristics were overestimated. Clearly, a large α leads to very high costs, but by choosing α carefully, a

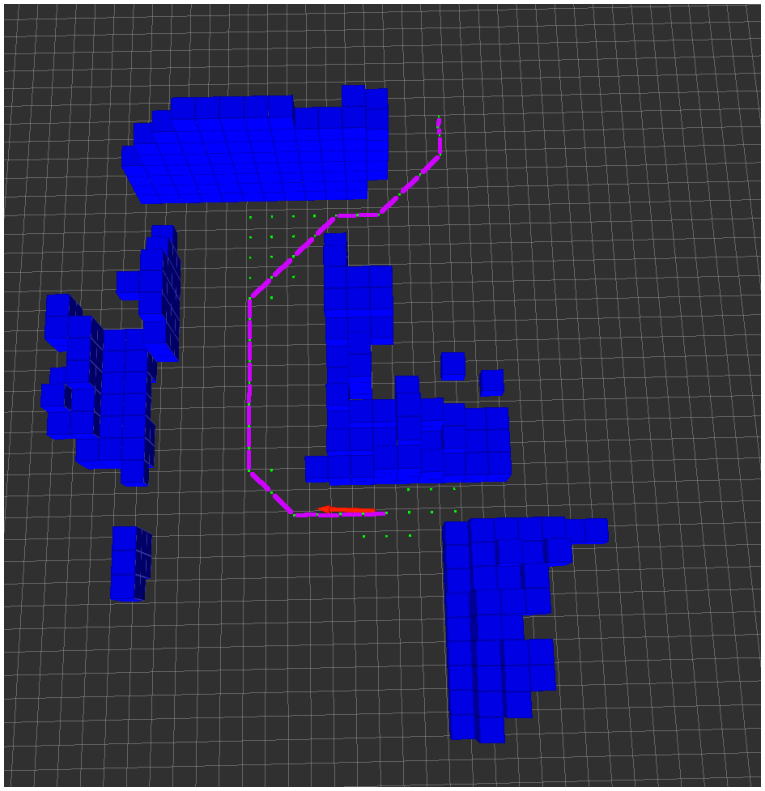
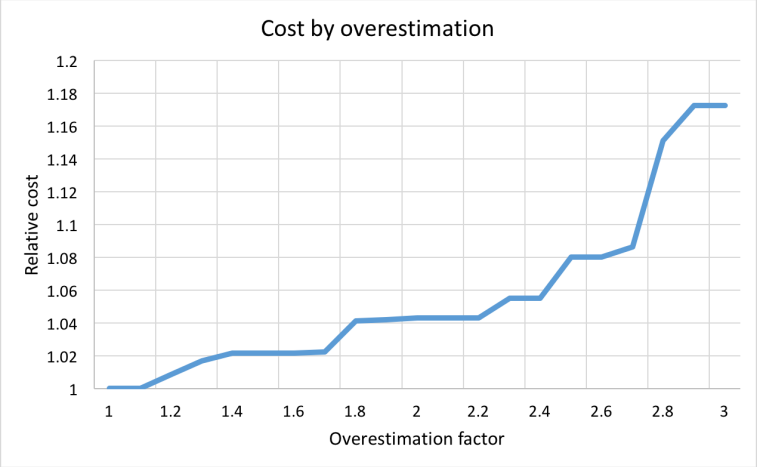


Figure 6.7: A visualization of vertices expanded by A* with overestimated heuristics.

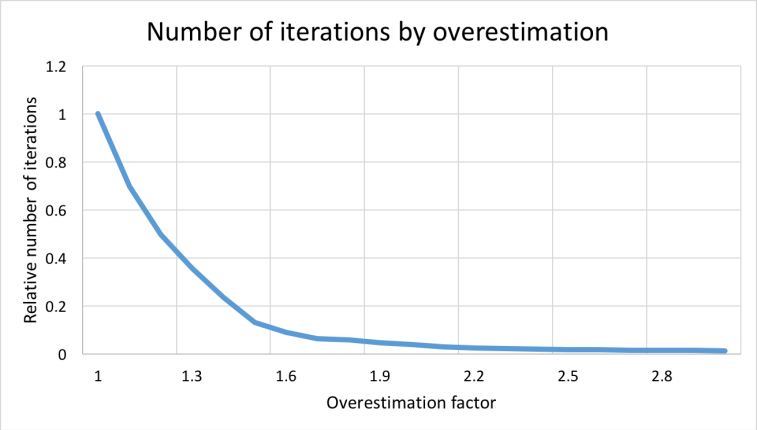
significant speed-up can be gained without a significant decrease in solution quality.

6.4.2 Risk Heuristics

Recall that there is no lower bound for the risk, but we can forfeit the optimality constraint and use the risk for unknown environment as heuristics to gain a speed-up, similarly to using overestimated heuristics. In figure 6.9 we see the two graphs in figure 6.8 merged into one. Since we want to minimize both cost and number of iterations we want to choose α such that we are closest to the origin of the graph. The red dot is the result for the risk heuristics, and we see that it effectively finds the 'knee' for α. So, by using the risk heuristics we get a similar result to the overestimated heuristics, but we don't need to tune α.



(a) Cost



(b) Number of iterations

Figure 6.8: The two graphs show the effect of using overestimated heuristics. A higher overestimation factor reduces the number of iterations needed to find a solution, but also reduces the quality of the solution.

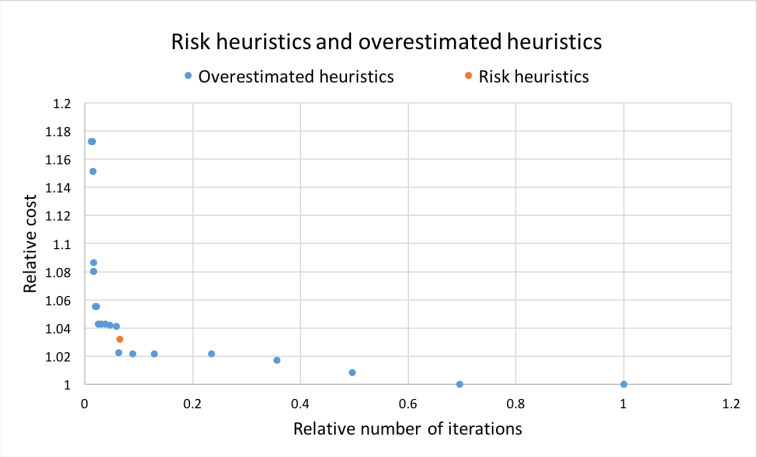


Figure 6.9: A comparison of risk heuristics to overestimated heuristics from figure 6.8 (a) and (b) merged into one graph. The blue points represent cost and number of iterations for different values of the overestimation factor, α. The single red point represents the result of using risk heuristics without overestimation.

6.5 Bidirectional Search

The bidirectional search was found to work badly due to the effect of overestimated heuristics. An early trial exposed a very frequent worst-case scenario where the bidirectional search takes about double the time of the original search, instead of an eighth, as we hoped. When one big obstacle is positioned between s and t, e.g. a wall, the search will expand a lot of vertices in front of the wall to find a way through, but when t is in sight, it will expand only the vertices on the path from the wall to t. On figure 6.10 we see an example of this where most of the work is done right in front of the wall. A search from t to s will similarly spend most of its work analysing the back of the wall, effectively doubling the work. In figure 6.11 we see an example where a bidirectional search expands roughly twice as many vertices as the original search. This scenario was found to be so frequent that further development of making the algorithm bidirectional was aborted.

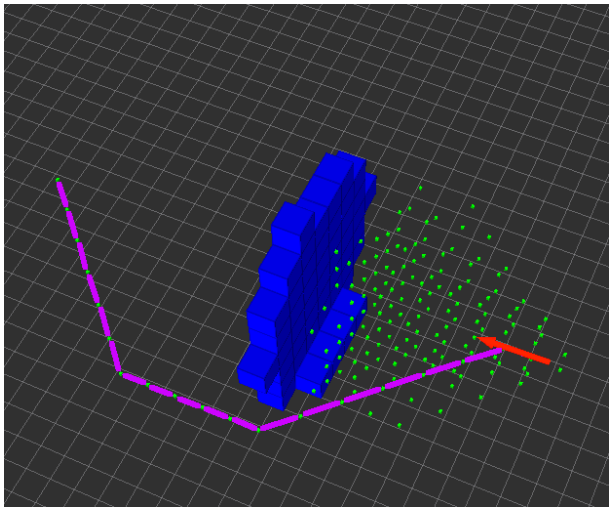


Figure 6.10: A visualization of which positions were considered during a search. The red arrow represents the drone, the blue cubes are obstacles, the magenta lines represent the path returned by the search and the green dots are the positions considered by the search.

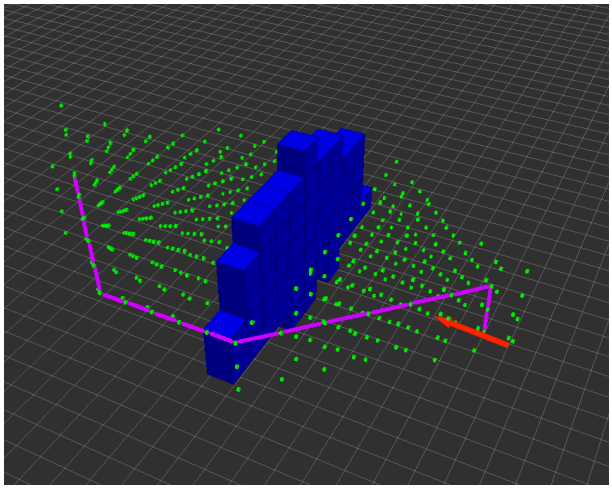


Figure 6.11: A visualization of the same search as in figure 6.10 using bidirectional search.

6.6 Discussion

6.6.1 Future Work

Despite successful incorporation of risk and smoothness in to the algorithm, there is still a lot of room for improvement. Modelling risk can be done in many different ways and in real life there are case-specific scenarios which

need to be addressed. For example, a drone with a single front facing camera might find horizontal movement to be safer than vertical movement. The costs and benefits of exploration are also an open problem. The height prior provides a nice way of estimating the benefits of gaining altitude, but the prior needs to be known in advance. An adaptive model which estimates how hard it is to find paths in the current environment could be an interesting extension.

Dynamic search where information from previous search is used could also be beneficial. A fully dynamic algorithm that keeps track of the shortest path tree at all times is currently not feasible due to the amount of computation needed, but by updating at constant intervals or using old paths as heuristics the total amount of work could be reduced. By running continuously the latency between a request for a new path and a new solution could be reduced, allowing the algorithm to find better paths in the same time frame.

Chapter 7

Conclusion

We have presented an algorithm for efficiently finding paths in three dimensions which minimize distance and risk, while maximizing the path's smoothness. Using the expected number of failures proved to be a good way to incorporate risk into the pathfinding, without making the problem computationally hard. By choosing the cost function to be in terms of path length and expected number of failures, the problem could be solved by either Dijkstra's Algorithm or A*. That also allowed us to introduce smoothness into the cost function without sacrificing the algorithm's efficiency.

We also showed how a cost function in terms of time and energy can be transformed into distance and smoothness, which can then be optimized with A*, and using a linear cost function allowed us to relate the different criteria together. The risk was defined in terms of occupancy probabilities derived from sensors, and prior probabilities dependent on the flight altitude.

Using simulation, the algorithm was found to be able to help the drone get to its destination, despite the drone's imperfect maneuvrability. While adding risk to the cost function decreased the error rate significantly, the height prior and the path smoothness also contributed to reducing the risk.

Bibliography

[1] Abdallah W Aboutahoun. Minimum Cost-Reliability Ratio Path Problem. 9(3):1633–1645, 2012.

[2] Faez Ahmed and Kalyanmoy Deb. Multi-objective optimal path planning using elitist non-dominated sorting genetic algorithms. *Soft Computing*, 17(7):1283–1299, 2012.

[3] Esther M Arkin, Joseph S B Mitchell, and C D Piatko. Bicriteria shortest path problems in the plane. *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 153–156, 1991.

[4] Zahy Bnaya, Ariel Felner, and Solomon Eyal Shimony. Canadian traveler problem with remote sensing. *IJCAI International Joint Conference on Artificial Intelligence*, pages 437–442, 2009.

[5] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality af A*. *Journal of the ACM*, 32(3):505–536, 1985.

[6] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[7] a V Goldberg and C Harrelson. Computing the Shortest Path: {A^*} meets Graph Theory. *16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[8] Matthew Greytak and Franz Hover. Motion planning with an analytic risk cost for holonomic vehicles The MIT Faculty has made this article openly available . Please share Citation Accessed Citable Link Detailed Terms Motion Planning with an Analytic Risk Cost for Holonomic Vehicles. 2013.

[9] Christina Hallam, K.J. Harrison, and J.a. Ward. A Multiobjective Optimal Path Algorithm. *Digital Signal Processing*, 11(2):133–143, 2001.

[10] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.

[11] Myungsoo Jun and Raffaello D Andrea. Path Planning for Unmanned Aerial Vehicles in Uncertain and Adversarial Environment. *Cooperative Control: Models, Applications and Algorithms*, pages 95–111, 2003.

[12] N Katoh. A fully polynomial time approximation scheme for minimum cost-reliability ratio problems. *Discrete Applied Mathematics*, 35(2):143–155, 1992.

[13] Mangal Kothari and Ian Postlethwaite. A Probabilistically Robust Path Planning Algorithm for UAVs Using Rapidly-Exploring Random Trees. *Journal of Intelligent & Robotic Systems*, 71(2):231–253, 2013.

[14] Lukasz Gwizdz. Octree, 2010. [Online; accessed May 1, 2016].

[15] Bhaskara Marthi. Navigation in Partially Observed Dynamic Roadmaps. *ICAPS POMDP Workshop*, 2010.

[16] E Nikolova and D R Karger. Route planning under uncertainty: the \mbox{C}anadian traveller problem. *In Proc. the 23rd AAAI Conf. on Artificial Intelligence, Chicago, Illinois*, 2008.

[17] Masahiro Ono and Brian C. Williams. An efficient motion planning algorithm for stochastic dynamic systems with constraints on probability of failure. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 1376–1382, 2008.

[18] Christos H. Papadimitriou and Mihalis Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84(1):127–150, 1991.

[19] Ying Xiao, Krishnaiyan Thulasiraman, Guoliang Xue, and a Jüttner. The constrained shortest path problem: algorithmic approaches and an algebraic study with generalization*. *AKCE International Journal of ...*, pages 1–38, 2005.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Risk-Based Pathfinding for Drones

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Vilhjalmsson	Vilhjalmur

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

2.5.2016

Signature(s)

Vilhjalmur Vilhjálmsson

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.