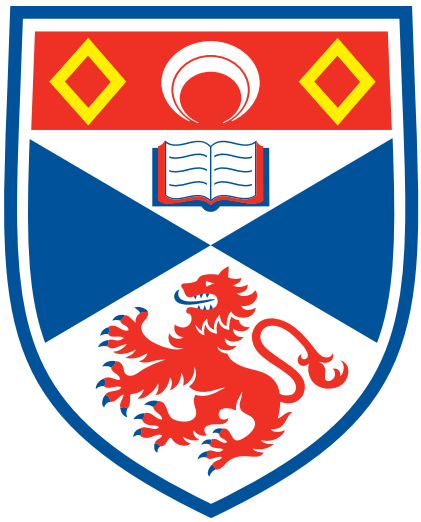
Generating Safe Trajectories in Stochastic Dynamic Environments by Leveraging Information About Obstacle Trajectories



University of
St Andrews

CS4099: Major Software Project

Author:
Alexander Wallar

Supervisor:
Dr. Michael Weir

April 2, 2015

Abstract

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is NNN words long, including project specification and plan.

Contents

List of Figures

Chapter 1

Introduction

Path planning is very important problem in robotics and computer science. It is the problem of generating a path through an environment that if followed by a robot, would move the robot from some initial configuration to a goal configuration without coming into collision with any obstacles on the way. This problem may seem easy for a human to solve, just walk around the obstacles and get to the goal, but for a robot it can be extremely difficult. As humans, we have amazing perception abilities and an unparalleled ability to assess the risk we perceive and plan around it. These capabilities are not as developed in artificial intelligence. Humans can walk around environments which are dynamic and uncertain and (usually) reach wherever they were going without running into moving obstacles or hitting walls. This is because as humans, we can determine automatically where things are going to be in the future by looking at their past locations and use this information to build safe, collision free paths to our goals. Imagine you are driving a car and reach an intersection, you stop because you see a car coming from the right. You have a choice to either move forward crossing over the future path of the other car, or to wait for the car to pass. By judging how far the car is away from you and how fast it is moving, you can automatically determine whether or not is safe to cross the road. Likewise, imagine you are waiter in a busy, hectic restaurant. You have to bring an order to hungry customers. You are able to bring the customers the food by predicting where other waiters and customers are going to be whilst you move the environment in order to avoid spilling the food and sacrificing your tip. Our brains do this planning and prediction automatically in order to generate safe paths through stochastic dynamic environments. The aim of this project is to use given information about the future motion of obstacles by an external system in order to generate safer paths than state of the art planners that have been designed for and operate in dynamic environments.

There has been outstanding progress by the robotics community to develop algorithms that are able to plan the motion of a robot in order for it to reach its goal. From this community, three different paradigms for path generation and motion planning have emerged, geometric algorithms, reactive algorithms, and sampling based approaches [1, 2]. Geometric algorithms such as the bug algorithm [3] or visibility graph algorithm [4], use the geometry of the environment to create exact geometric paths to the goal. These algorithms almost always have no random component, and for a given environment, will return the same path every time. The second paradigm for motion planning, reactive algorithms, move the robot to the next best location at the current time for a given sensing radius. These approaches are vastly dominated by the use of potential fields in order to determine the direction that a robot should move. The approaches use a combination of an attractive potential function to guide the robot towards a goal and a repulsive potential function that keeps the robot from coming into contact with obstacles. The robot moves forward in time by determining the direction it should move that would minimize the combined potentials. Sampling based approaches approximately discretize the environment, also known as the search space, in order to describe its connectivity by sampling plausible configurations and how to move between them. With the discretization, usually in the form of a graph or a tree, paths to the goal are extracted using shortest path algorithms which seek to minimize a given objective function. These approaches are becoming exceedingly popular due their running time and how they are able to scale for high dimensional systems.

This project aims to utilize developments in motion planning, particularly sampling based motion planning in order to move a robot safely from an initial configuration to a goal configuration in a stochastic dynamic environment by leveraging information about the trajectories of dynamic obstacles. By having some idea about where the obstacles are going to move in the future, it is possible to use sampling based techniques that can sample collision free and low risk paths to the goal in space-time. The goal of this project is to create algorithms that can provide low cost paths to the goal by utilizing the information available about how the obstacles in an environment will move. This work introduces a novel representation of dynamic obstacles and a novel algorithm for searching the environment in space-time. Proofs are also provided that can guarantee the completeness of the search such that the algorithms will always provide a path to the goal. This problem is important because if robots are interacting with humans, the robot must move safely in order not to come into contact with and hurt humans as well as minimize the damage that can possibly occur to itself. Likewise, imagine a situation where a robot is deployed to an environment for a long period of time. By generating safe paths (i.e. paths that have a small chance of colliding with an obstacle), the robot can be deployed for longer without maintenance, because it would be less likely that it would get damaged as a result of a collision. A solution to this problem can provide safer paths for the operating environment and the robot by leveraging information that can be extracted about where obstacles are moving. Likewise, this problem is important because there exists systems that are able to predict the motions of obstacles, however there is a lack of systems that use this knowledge to generate safe paths.

Chapter 2

Context Survey

Chapter 3

Software Development Framework

Chapter 4

Objectives

4.1 Primary

The main objective of this work is to develop an algorithm that generates a quantitatively safe trajectory for a robot through an uncertain dynamic environment by utilizing information about how dynamic obstacles are going to move in the future. This can be simply stated as determining the curvature that minimizes the line integral over the dynamic cost distribution for a given set of dynamic obstacles.

$$J(C, A) = \int_C \exp\Big(P(x, y, t_0, t_m, A) + 1\Big)\, \mathrm{d}s \tag{4.1}$$

Eq. 4.1 describes the objective function, J, that needs to be minimized with respect to the curvature in order to determine the safest path through the environment. In Eq. 4.1, the function P is the cost surface for a given time interval and set of obstacles. More description about P is given in Sec. 6.1.2 and is formally defined in Eq. 6.2. More precisely, the objective of this work to develop an algorithm that will provide an approximate solution to Eq. 4.2 which will return the minimum cost path through a environment for a given set of obstacles. The solution is described in Sec. 6.2

$$\Gamma(A) = \arg\min_C \; J(C, A) \tag{4.2}$$

4.2 Secondary

The main secondary objective for this work is to show that the proposed solution can provide safer paths than standard planners such as potential fields by leveraging information about how the obstacles move through the environment. Quantitative and qualitative experiments have been conducted that provide evidence that the proposed solution does indeed produce safer paths based on the safety metrics that have been devised for this work. These results are shown in Ch. 9.

Chapter 5

Design

As with any research project, many different attempts were made to come up with a solution to the objectives state in Ch. 4. Three different techniques were developed in sequence and one technique was theorized but never implemented to try and provide a solution that best matched the sought behaviour for the planner. The theorized technique included using a bug algorithm **??** in space-time to find a path through the dynamic environment. The first attempt was a simple potential field that would take into account the predicted trajectories of the obstacles, and leverage this information to provide safer paths. The second attempt included generating a probabilistic roadmap in relative space-time and using stock graph search algorithms such as Dijkstra's algorithm and Edmonds' algorithm to derive low costs paths through the environment. The last attempt, and the most successful is a planner that uses a two dimensional probabilistic roadmap to sample the search space and then uses Best First Search to expand nodes in space-time to determine the minimum cost path through the dynamic environment. These four attempts are described individually and more detail in this section.

5.1 Space-time Bug Algorithm

The initial design for the algorithm was to use a bug algorithm that operates in space-time. Bug algorithms work by moving the robot towards the goal until it reaches an obstacle. Once an obstacle has been reached, the bug algorithm will move the robot along the edge of the obstacle until the obstacle is no longer between the robot and the goal. The robot will then move to the goal and repeat this process if necessary. An example of this algorithm in practice is shown in Fig. 5.1. This proposed design was going to treat the dynamic obstacles moving in two dimensions as three dimensional static obstacles in three dimensions where time is the third dimension. The robot would use the bug algorithm to move through the environment in three dimensions. A constraint would have been added to the movement of the robot which would be that it could not move backward in the last dimension, time. The benefit of this design is that completeness could be proven geometrically, it is a computationally simple algorithm, and it could be easily implemented for use on a mobile robot. This design was not implemented because of several fundamental flaws discovered during the design phase. Firstly, by thinking of dynamic obstacles as static obstacles in space-time, the robot would have to rely on perfect information about the movement of the dynamic obstacles, i.e. there could be very little uncertainty about the movements or predictions of the movements in the dynamic obstacles. For the planner to be able to handle uncertainty in the motion of the obstacles *a priori*, obstacles would need to be increased in their size (emphasizing the possible positions of the dynamic obstacles) as time increases. This could lead to possibly unrecoverable configurations because obstacles in space-time could be seen as overlapping thus not providing a collision free area for the robot to move through. It was ultimately decided not to continue with this design but instead to leverage a more stochastic and probabilistic definition of the dynamic obstacles and to use planning techniques more suited for stochastic dynamic environments.

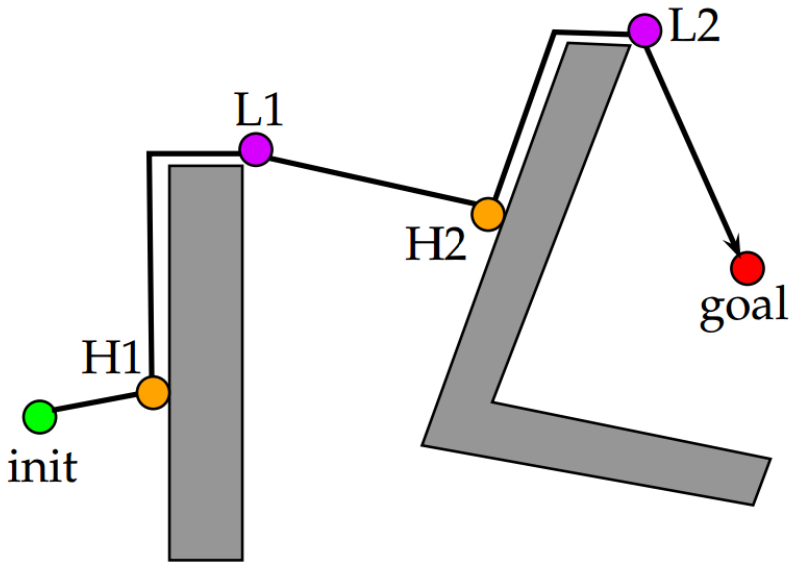


Figure 5.1: Erion

5.2 Potential Fields

Using potential fields was an initial attempt to plan through uncertain dynamic environments since they are frequently used to plan around dynamic obstacles due to their reactive behaviour [5, 6, 7]. The difference between the standard potential field implementations and the one developed for this project was that the repulsive obstacle field at a position (x, y) was proportional to the cost distribution at (x, y) for a given time interval. This is shown more formally in Eq. 5.1.

$$U_{rep}(p, t_0, t_m, A) = k \cdot P(p_x, p_y, t_0, t_m, A) \tag{5.1}$$

In Eq. 5.1, the function P is defined in Eq. 6.2, and $k > 0$ is a constant. The attractive potential field was kept the same as the standard potential field implementation for robotic motion planning as shown in Eq. 5.2

$$U_{att}(p, g) = c \cdot ||p - g||^2 \tag{5.2}$$

In Eq. 5.2, c is a scaling constant such that $c > 0$. The potential field planner would use the sum of these two fields to measure the potential through the environment in order to eventually reach the goal by successively moving to the area within the robot's sensing radius that had the minimal potential. Algo. 1 describes more formally how the potential field planner generates a path through the environment.

Through some qualitative testing, these types of potential fields were still leading the robot into unsafe areas and caused the robot to collide with the dynamic obstacles regardless of velocity of the obstacles and the velocity of the robot. After some manipulation of the constants used for the repulsive and attractive potentials, there was only a nominal improvement which lead the author to move towards sampling based motion planning techniques which are outlined in Sec. 5.3 and Sec. 5.4.

5.3 Space-time Roadmap

The second attempt at devising a solution to the primary objective in Ch. 4 was to create a three dimensional probabilistic roadmap that can capture the connectivity of a two dimensional surface in space-time.

Algorithm 1 PF(q, g, O, A, R)

$q_{min} \leftarrow q$
$p_{min} \leftarrow \infty$
$\theta \leftarrow 0$
while $\theta \leq 2\pi$ **do**
  $q' \leftarrow q + \delta t \cdot s \cdot \text{ROT}(\theta)$
  $p \leftarrow U_{rep}(q', O \cup A) + U_{att}(q', g)$
  if $p < p_{min}$ **then**
    $p_{min} \leftarrow p$
    $q_{min} \leftarrow q'$
  $\theta \leftarrow \theta + \delta\theta$
  for all $a \in A$ **do**
    STEP(a)
if $||q_{min} - g|| < R$ **then**
  return $\{p_{min}\}$
return $\{q_{min}\} \cup \text{PF}(q_{min}, g, O, A, R)$

A spatio-temporal probabilistic roadmap (PRM) is a directed, weighted graph, (V, E), that represents the spatio-temporal connectivity of the search space by randomly sampling points and connecting them such that if $((i, t), (j, t')) \in E$, then both i and j must not collide with an obstacle at times t and t' respectively, $||i - j|| \leq d$ where d indicates the maximum distance away connected nodes can be from one another, there must not be a collision with any obstacle along the edge from i and j in the time interval $[t, t']$, $|t - t'| < \delta t$, and $t < t'$ [8, 9]. For the attempted space-time PRM, each node would be a vector, (x, y, t), which represents a two dimensional location, (x, y), at a certain absolute time t, and instead of randomly sampling a point in the environment, a node in the graph would be randomly selected and propagated forward in time by some random change in time such that the constraints for the roadmap are still satisfied. This algorithm is shown more formally in Algo. 2.

Algorithm 2 TEMPORALROADMAP$(N, d, \delta t, s, p, O, A)$

Input:
n: Maximum number of samples
d: Maximum distance between neighbouring nodes
O: Set of obstacles
Output:
An weighted directed graph of points describing where it is possible for the robot to move from its initial configuration.

$V \leftarrow \{(p, 0)\}$
for $i = 1$ **to** N **do**
  $(n, t) \leftarrow$ RANDOMSELECTION(V)
  $t' \leftarrow t +$ UNIFORMRANDOM$(\varepsilon, \delta t)$
  $\theta \leftarrow$ UNIFORMRANDOM$(0, 2\pi)$
  $q \leftarrow n + s \cdot t' \cdot \text{ROT}(\theta)$
  if $\bigwedge_{o \in O} \neg$COLLISION(o, q) **then**
    for all $(v, \tau) \in V$ **do**
      if $||v - q|| < d \wedge |\tau - t'| < \delta t$ **then**
        if $\tau < t'$ **then**
          $E \leftarrow E \cup \{((v, t), (v, t'), C(v, q, \tau, t', A))\}$
        else
          $E \leftarrow E \cup \{((q, t'), (v, t), C(q, v, t', \tau, A))\}$
    $V \leftarrow V \cup \{(q, t)\}$
return (V, E, W)

With the generated roadmap, the first thought was to use a graph search algorithm such as A* [10] or Dijkstra's algorithm [11] to find the path through the environment that had the lowest overall weight. The weight for an edge, (i, j), defined as the line integral over the cost surface for a given set of dynamic obstacles with a time interval of $[i_t, j_t]$. The notion of a cost distribution is described in Ch. 6 and the

formal equation for this line integral is given in Eq. 6.5. This space-time roadmap approach yielded mixed results. The robot would sometimes evade the obstacles, but with the incidence, the planner would lead the robot directly into a collision with a dynamic obstacle. After some testing, it was discovered that since graph search algorithms seek to find the path with minimum combined weight through the graph, these algorithms are biased to return paths with a smaller number of vertices. This is because paths with a larger number of vertices will have a higher overall weight. Since shortest path algorithms try to minimize this overall weight, paths which may be safer but may take longer not be returned by these algorithms.

To overcome this, instead of searching over the entire graph, the search could be occur over the minimum spanning tree of the graph. Since the roadmap is a directed graph, Edmonds' algorithm [12] was used since other minimum spanning tree algorithms such as Kruskal's algorithm [13] and Prim's greedy algorithm [14] only work on undirected graphs. Using the minimum spanning tree would minimize the maximum cost associated with a path from the initial configuration to the goal configuration thus moving the robot away from high cost areas in space-time. Through qualitative analysis, this approach was shown to still lead the robot to high cost areas and even into collisions with obstacles.

This approach using a three dimensional probabilistic roadmap did not work in practice regardless of the search algorithm because of the number of nodes that need to be sampled in space-time in order for it to be effective. The roadmap indicates where in the environment the robot is able to travel to from a starting location in space-time. If the number of nodes is too small the, the goal may not even be in the graph, and thus the robot will never reach it. Also, the less nodes in the graph the less optimal the generated path is, but the more nodes added to the graph, the more computationally difficult it becomes to search. Lastly, the main flaw with this approach is that biases the sampling to areas that already have a high sample density and therefore may not sample nodes in the goal area without having a high number of nodes in the graph.

5.4 Probabilistic Roadmap With Best First Search

After consideration for other sampling based motion planning techniques such as rapidly exploring random trees [15] and expansive space trees [16] the author chose to explore using a custom graph search algorithm over a two dimensional probabilistic roadmap due to the lack of ability for classical sampling based techniques to deal with time-dependent edge costs. The idea was to use a probabilistic roadmap to capture the connectivity of the two dimensional environment and to generate a search tree through the roadmap that encodes the temporal information for each point in a tree node and is therefore able to account for time-dependent edge costs. This graph search algorithm is a temporal analogue of best-first search which tries to expand the best current node in a search tree based on some heuristic [17]. The heuristic in this project is to expand the node in the search tree that has the minimum cost as defined in Eq. 6.5. A more complete and in depth description of this method is given in Ch. 6.

Chapter 6

Planner Methodology

As described briefly in Sec. 5.4, the final planner design uses a probabilistic roadmap to capture the connectivity of the environment and uses a novel variant of best first search to find a safe, low cost path the goal. This section describes the algorithm as well as formalizes the definition of a dynamic obstacle and its associated cost distribution and motion.

6.1 Dynamic Obstacles

As a major component of this work, dynamic obstacles need to be designed such that one can describe their trajectories, initial configurations, and their levels of uncertainty. This section introduces the definition of a dynamic obstacle used throughout this work along with how one is represented to the planner and its simulated & predicted equations of motion. Specifically, the formal definition of a dynamic obstacle will be introduced, a novel way of representing obstacles using a cost distribution will be formalized, and their predicted and simulated motion will be described.

6.1.1 Definition

A dynamic obstacle is defined as a 5-tuple, $a = (I, \dot{\zeta}, \epsilon, \xi, T)$ where I is the initial configuration of the obstacle, $\dot{\zeta}$ is a function, $\dot{\zeta} : \mathbb{R}^+ \rightarrow \mathbb{R}^2$, representing the velocity of the obstacle, ϵ is used to define a random variable $\rho \sim \mathcal{U}(-\epsilon, \epsilon)$ that is used to inject noise into an obstacle's trajectory shown in Eq. 6.4 where $\mathcal{U}$ is a uniform distribution, ξ is the current configuration used for prediction, and T is the time that the obstacle was in configuration ξ. The variables ξ and T are dynamic variables and are updated throughout the execution of the algorithm and are used to determine when it is appropriate for the algorithm replan and find a new path through the environment using more up to date information. This is explained in Sec. 6.2. The variables ξ and T are initially set to I and 0 respectively. It is assumed that the robot has access to this information about the dynamic obstacles and will use it to safely around them. Information such as the initial configuration and the velocity equation for each dynamic obstacle are assumed to be determined by an external system that is either using a machine learning technique to deduce these properties by using information about where the dynamic obstacle has been before or by an external planning system such as in a warehouse that is commanding the velocities and configurations of these dynamic obstacles. This is the same assumption as that made by Phillips et al [18] and Narayanan et al [19].

6.1.2 Cost Distribution

Unlike in the previous work, dynamic obstacles are represented by cost distributions that resemble probability density functions. The difference being is that these cost distributions do not have a unit integral.

These cost distributions are used to describe where the obstacle is going to be in within a time interval and can be generated by a third party system, such as a motion capture system. There is an assumption that for a given interval, $\mathcal{T} = [t_0, t_m]$, the highest cost with the smallest uncertainty will be at $t = t_0$ and the lowest cost with the highest uncertainty will be at $t = t_m$. Under this assumption, the cost function models how the obstacle may diverge from its current trajectory as time increases. With these assumptions, the cost function, $P_a : \mathbb{R}^2 \times (\mathbb{R}^+)^2 \to \mathbb{R}$, represents represents the cost surface for a given dynamic obstacle, a, within a given time interval. Eq. 6.1 formally defines the cost function for a single obstacle.

$$P_a(x, y, t_0, t_m) = \frac{1}{t_m} \cdot \int_{t_0}^{t_m} \mathcal{N}(\zeta_a(t), \alpha \cdot (t - t_0)^2 + \beta, x, y) \cdot (t_m - t)^\gamma \, \mathrm{d}t \tag{6.1}$$

In Eq. 6.1, $\mathcal{N}(\mu, \sigma^2, x, y)$ is the evaluation of a 3D normal distribution centered at (μ_x, μ_y) with a variance of σ^2 at (x, y) and α, β, and γ are scaling constants such that $\alpha > 0$, $\beta > 0$, and $\gamma \geq 1$. Fig. 6.1 shows an example of $\mathcal{N}$. The normal distribution works as a basis function to describe the possible trajectories of the dynamic obstacle. This equation models how the uncertainty of obstacle trajectory prediction increases over time by increasing the standard deviation of the Gaussian distribution as the time increases. Likewise, this function multiplies the Gaussian distribution by a factor of $(t_m - t)^\gamma$ where $\gamma \geq 1$ which gives higher costs to times closer to t_0.

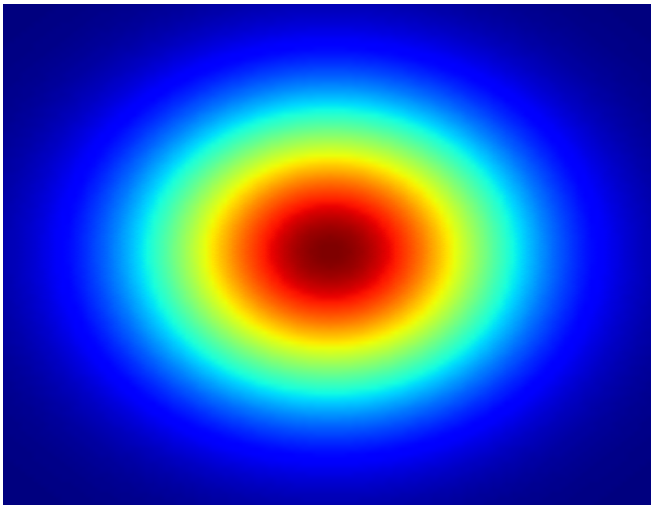


Figure 6.1:

A cost function is also needed that can incorporate the cost distributions for multiple dynamic obstacles within the environment. The cost function used in this work, $P : \mathbb{R}^2 \times (\mathbb{R}^+)^2 \times \mathcal{A} \to \mathbb{R}$ where $\mathcal{A}$ is the set of all possible sets of dynamic obstacles, calculates the average cost at a point (x, y) within a given time interval for a given set of dynamic obstacles, A. This is shown formally in Eq. 6.2.

$$P(x, y, t_0, t_m, A) = \frac{\sum_{a \in A} P_a(x, y, t_0, t_m)}{|A|} \tag{6.2}$$

An example of how P changes over time is shown in Fig. 6.2. In that example, two dynamic obstacles are placed in the scene and given sinusoidal velocities. In this example the time interval, δt, is kept constant throughout the simulation, i.e. $t_m = t_0 + \delta t$, for all $t_0 \in [0, T - \delta t]$ where T is the length of the simulation. Since the velocity does not remain constant in the example, the cost distribution elongates an shrinks based on the acceleration of the obstacle. For instance, in the first and last images in Fig. 6.2, the cost is contained to a small area due to the velocity equations of the obstacles being at their minimum

and in the fourth image,the cost is more spread out through the environment because the velocity is at its maximum.

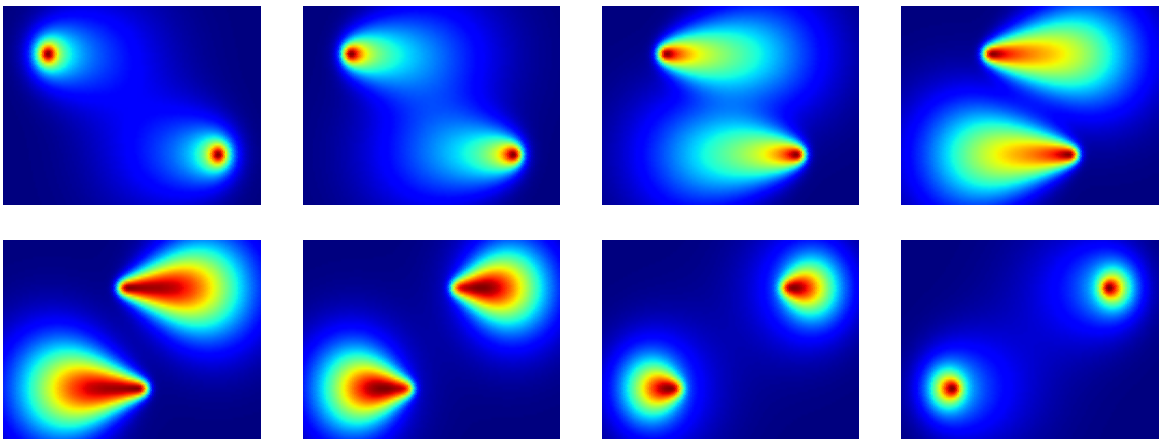


Figure 6.2: Cost distributions indicating the likelihood that an agent will be at a certain location within a given time interval. These figures show how this distribution changes over time (left to right, top to bottom)

6.1.3 Equations of Motion

The motion of a dynamic obstacle is defined by the velocity equation, the initial configuration, the amount of uncertainty. Defining the obstacle's trajectory in terms of its velocity makes it easier to model when creating scenes. The equation of motion for the dynamic obstacle is shown in Eq. 6.3.

$$\zeta_a(t) = \begin{cases} \xi_a + \int_{T_a}^{t} \dot{\zeta}_a(\lambda)\,\mathrm{d}\lambda & \text{if } t \geq T_a \\ \tilde{\zeta}_a(t) & \text{if } t < T_a \end{cases} \tag{6.3}$$

In Eq. 6.3, $\tilde{\zeta_a}$ represents the observed trajectory of the obstacle whereas ζ_a corresponds to the predicted trajectory of the obstacle. This disambiguation is needed because the planner needs to be able to extrapolate the future movements of a dynamic obstacle. The variables, ξ_a and T_a are dynamically updated when the planner replans and are initially set to I_a and 0 respectively. The need for this and how it is designed will be discussed in Sec. 6.2

For the experiments, the motion of the obstacles is simulated by adding a random variable, $\rho \sim \mathcal{U}(-\epsilon, \epsilon)$ to the trajectory during the numerical integration of the velocity equation. This form of stochasticity allows the obstacle to diverge from its specified trajectory whilst maintaining the same velocity equation. This means that the obstacle will not exhibit random, Brownian motion around its specified path, but rather is able to diverge completely. Also, by adding the random variable to the velocity equation during the numerical integration, the obstacle will not "jump" to a new location, but will gradually diverge off of its specified path. The definition of $\tilde{\zeta_a}$ is shown in Eq. 6.4. For this equation, it is assumed that the function only computes a value for any given time value, t, only once.

$$\tilde{\zeta_a}(t) = \begin{cases} \tilde{\zeta_a}(t - \delta t) + \dot{\zeta_a}(t) \cdot \delta t + \rho & \text{if } t > 0 \\ I_a & \text{if } t = 0 \end{cases} \tag{6.4}$$

In Eq. 6.4, δt is a constant where $\delta t > 0$ and is used for the numerical integration of the velocity. Numerical integration is used in order to allow the path to diverge. Since a recursive function is used for determining the current position of the obstacle, the obstacle's next location will be determined by its last location, its velocity, and a certain error factor ρ. This will allow the obstacle to exhibit more

than simply Brownian motion along a path and it will be able to diverge completely from its specified path. This type of uncertainty is important because it is exhibited in real-world scenarios such as error propagation in controls and by simply not having enough information about the actual velocity equation for a given obstacle. The full extent of $\tilde{\zeta_a}$ is not known by the planner and this equation is only used to simulate the motion of the obstacles for experimentation and evaluation of the developed planner.

By increasing the level of stochasticity, ϵ, for a given dynamic obstacle, it will be more likely to diverge from its current path. This means that the uncertainty of a given dynamic obstacle can be parametrized by its value for ϵ.

The motion of a dynamic obstacle is described by it's velocity and a starting location in order for the planner to be able to continue to predict an obstacle's motion even when it diverges from its current path. If the obstacle's motion was described by parametric equations of its position, it would not be possible to continue to predict where it is going to move once it is no longer following its prescribed path since the obstacle may diverge from this path completely.

6.1.4 Available Information

It is important to note what information is available to the planner. It is not assumed that the planner has perfect information about the motion of the obstacles or their associated definitions. The planner only has access to the obstacles' velocity equations and their positions throughout the execution of path. The planner does not know the amount of noise, ϵ, being injected into the obstacles' velocities as described in Sec. 6.1.3. It is also assumed that the robot or an external system is evaluating the cost distribution based on the predicted motion of the dynamic obstacles in the future. Even though these assumptions may seem unrealistic, there are currently methods being developed that are able to extrapolate and predict the motion of obstacles based on their past locations or based on the algorithm used to dictate their behaviour [20, 21, 22, 23, 24]. Also, as described in Sec. 6.2.4, the planner does not need to have a perfectly precise equation for the obstacles' motion because the planner has the ability to construct a new plan through the environment if the prediction of the obstacles' motion does not accurately depict where the obstacle actually is.

6.2 Planning Algorithm

The planning algorithm for Dodger has three major components. First, a two dimensional probabilistic roadmap is constructed over the search space in order to capture the spatial connectivity of the environment. The planner then uses a best-first (BestFS) graph search algorithm to determine the safest path from the initial position to the goal position by creating a temporal search tree through the graph in order to account for time-dependent edge costs. Once the robot has an initial path to follow, it will incrementally pursue this path whilst updating its information about the location of the dynamic obstacles. If any of the obstacles deviate from their predicted path, the planner will generate a new path (replan) for the robot using this information. These three components allow the planner to reduce the number of samples over time by reusing the same two dimensional roadmap and allows the planner to account for dynamic obstacles with stochastic motion by replanning. These are improvements to the first sampling based technique described in Sec. 5.3

6.2.1 Building the Roadmap

The first component of the planning algorithm is the underlying two dimensional roadmap which represents the spatial connectivity of the environment. This roadmap is constructed using a standard variant of the probabilistic roadmap algorithm created by Kavraki et al [8]. A probabilistic roadmap is a undirected graph, (V, E), created by randomly sampling n configurations in the configuration space of the robot and connecting them such that if $(i, j) \in E$, then $||i - j|| < d$, both i and j are not within any of the static obstacles, and there is no collision along the geometric edge from i to j. In this work the configuration

space is $\mathbb{R}^2$ and the edge between two nodes is a straight line. The nodes in the roadmap represent the possible configurations of the robot and in this work, points along the edge between two nodes represent geometric transitions from one node to another. The complexity of the graph is parametrized by the number of samples, n, and the maximum distance between nodes, d. By increasing the number of samples, the density of samples increases and therefore the average degree for a node in the graph increases. Likewise, if the maximum distance between nodes increases, so does the average degree for a node in the graph. Pseudocode is provided for constructing a probabilistic roadmap in Algo. 3.

Algorithm 3 ROADMAP(n, d, w, h, O)

Input:
n: Maximum number of samples
d: Maximum distance between neighbouring nodes
O: Set of obstacles
Output:
An unweighted graph of points describing the connectivity of the environment

1: **for** $k = 1$ **to** n **do**
2: $q \leftarrow$ RANDOMPOINT2D(w, h)
3: **if** $\bigwedge_{o \in O} \neg$COLLISION(o, q) **then**
4: $V \leftarrow V \cup \{q\}$
5: **for all** $i \in V$ **do**
6: **for all** $j \in V$ **do**
7: **if** $i \neq j \wedge ||i - j|| \leq d \wedge \bigwedge_{o \in O} \neg$COLLISION$(o, i, j)$ **then**
8: $E \leftarrow E \cup \{(i, j)\}$
9: **return** (V, E)

In Algo. 3, first at most n points are sampled and added to the set of vertices, V, if and only if there is not a collision with any of the static obstacles. Essentially the Cartesian square of V is iterated over and any two points that are not the same, whose distance is less than d, and if there is no collision along the edge between these two points are added to the edge set, E. Finally, the vertices and edges are returned as a tuple representing the graph. Fig. 6.3 shows a constructed probabilistic roadmap with 300 nodes.

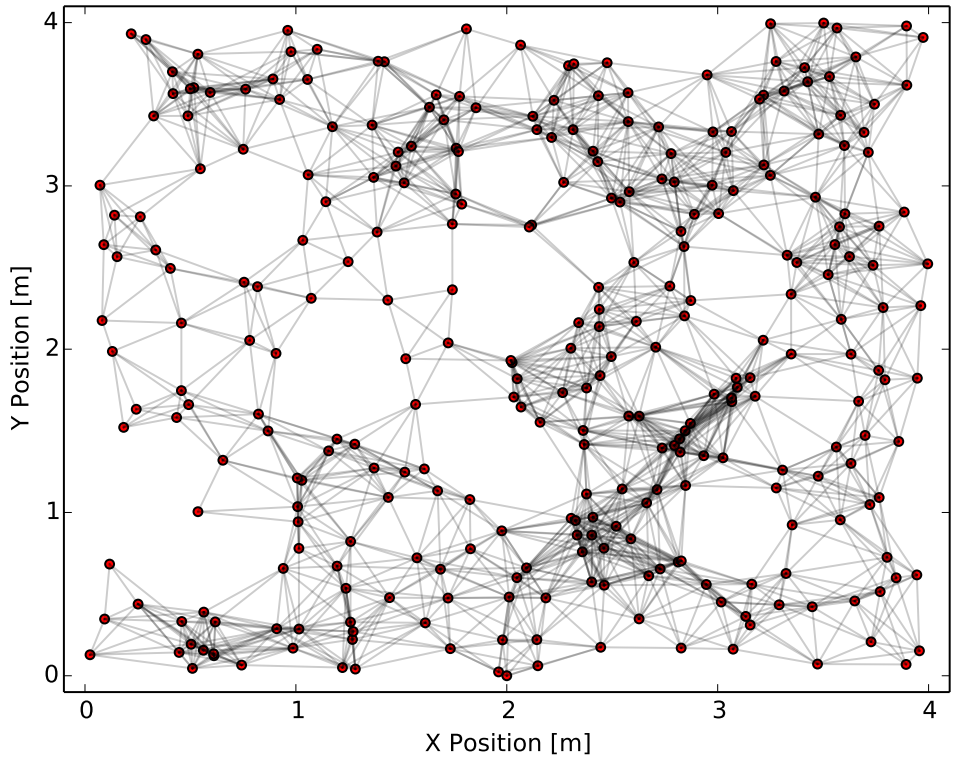


Figure 6.3:

6.2.2 Searching the Graph

To search the generated two dimensional probabilistic roadmap in space-time, a novel best first search (`tBestFS`) algorithm has been developed that creates a search tree that expands the current node not only in space but also in time until the current node is within an acceptance radius the goal. This algorithm works by using priority queue to store to store the expanded nodes. The priority is based on two things, how many times a two dimensional node in the probabilistic roadmap has been expanded and the cost to reach the node in space-time in the queue. A dictionary indexed by the node and mapped to an integer is used to keep track of how many times the node has been visited. The cost between two nodes in space-time, (i, t) and (j, t') is defined by the line integral from i to j over the exponential of the cost distribution P for the time interval $[t_0, t_m]$ and set of dynamic obstacles A. This function uses the exponential of the cost distribution in order to emphasize higher costs along the path. This is formally shown in Eq. 6.5.

$$\begin{aligned} C(i, j, t_0, t_m, A) &= \int_0^1 \exp\Big(P(x(\lambda), y(\lambda), t_0, t_m, A) + 1\Big) \cdot \sqrt{\Big(\frac{\mathrm{d}x}{\mathrm{d}t}\Big)^2 + \Big(\frac{\mathrm{d}y}{\mathrm{d}t}\Big)^2}\,\mathrm{d}\lambda \\ &= \int_0^1 \exp\Big(P(x(\lambda), y(\lambda), t_0, t_m, A) + 1\Big) \cdot \sqrt{(j_x - i_x)^2 + (j_y - i_y)^2}\,\mathrm{d}\lambda \\ &= \int_0^1 \exp\Big(P(x(\lambda), y(\lambda), t_0, t_m, A) + 1\Big) \cdot ||i - j||\,\mathrm{d}\lambda \end{aligned} \tag{6.5}$$

In Eq. 6.5, C is a function $C : \mathbb{R}^2 \times \mathbb{R}^2 \times (\mathbb{R}^+)^2 \times (\mathbb{R}^+)^2 \times \mathcal{A} \to \mathbb{R}$ where $\mathcal{A}$ is the set of all possible sets of dynamic obstacles, and $x(\lambda) = (j_x - i_x) \cdot \lambda + i_x$ and $y(\lambda) = (j_y - i_y) \cdot \lambda + i_y$ are the parametric equations of the line from i to j. Eq. 6.5 increases exponentially as the maximum cost along the line increases and therefore makes edges with larger maximum costs more costly than edges with lower maximum costs. To combine the two cost formulations for an edge, $((i, t), (j, t'))$, a weighted sum is computed that adds the number of times node j has been visited with the line integral over the cost distribution from i to j for the given time interval. This total cost function is formally described in Eq. 6.6.

$$TC(i, j, t_0, t_m, A, D) = \psi \cdot C(i, j, t_0, t_m, A) + \omega \cdot D_j \tag{6.6}$$

In Eq. 6.6, ψ and ω are scaling constants. It is important to incorporate a penalty for returning to a location that has already been visited even if this penalty is small because the tBestFS algorithm may choose to only expand nodes in a safe area and thus not reach the goal since there is not a heuristic to sample nodes closer to the goal in order give preference to safer paths instead of shorter ones.

Since the algorithm operates in both space and time, the search tree is expanded at a given node, i at a time t, by determining how long it would take the robot to reach all of the neighbours of i and not only adding each neighbour of i into the priority queue with its associated cost, but also adding the time at which the robot would reach each neighbour. This is accomplished by assuming the robot will travel at a constant speed, s, through the environment which makes it easy to compute how long it would take the robot to reach a neighbour, $j \in \textsc{Neighbours}(i)$. The time it would take for the robot to reach a neighbour, j from i is $||i - j||/s$ which would make the robot reach j at time $t + ||i - j||/s$. Also, since the algorithm is also operating in time, the robot is able to stay at the same location for a given wait time δt if it is less costly than moving forward to any of its neighbours.

The algorithm progresses forward by expanding the best current node in the priority queue. This is the node that has the smallest combined cost in the queue. Once a node, i at time t, is expanded and its temporal neighbours determined, the parent for each of these neighbours is defined as i at time t. By storing the parents of each of the neighbours, it becomes possible to backtrack from the goal to the starting position once the goal is reached and construct a path to the goal configuration. The algorithm terminates either when there are no more nodes in the priority queue or once the node popped out of the queue is less than a certain distance away from the goal node. Once the node popped out of the priority queue is within the goal region, the path to the goal is returned. The algorithm used for searching the graph is described formally in Algo. 4.

Algorithm 4 `tBestFS`(V, E, R, A, p, g, T)

$Q \leftarrow$ PriorityQueue()
$D \leftarrow$ Dictionary()
$\mathcal{P} \leftarrow$ Dictionary()
Insert(Q, p, T)
while $|Q| > 0$ **do**
  $(q, t) \leftarrow$ Pop(Q)
  if $||q - g|| \leq R$ **then**
    return BacktrackPath$(p, q, t, \mathcal{P})$
  $S \leftarrow \emptyset$
  $N \leftarrow$ Neighbours$(V, E, q) \cup \{q\}$
  for all $n \in N$ **do**
    if $q \neq n$ **then**
      $t' \leftarrow ||q - n||/s + t$
    else
      $t' \leftarrow t + \delta t$
    $\mathcal{P}_{(n,t')} \leftarrow (q, t)$
    $c \leftarrow \psi \cdot C(q, n, t, t', A) + \omega \cdot D_n$
    $D_n \leftarrow D_n + 1$
    $Q \leftarrow$ Insert$(Q, (n, t'), c)$

From the algorithm in Algo. 4, it is evident that picking a good wait time, δt, is important for generating safe paths through the environment. Having a large value for δt can lead to very suboptimal paths because even though staying in the same position for a given period of time may be safer than moving to any of the neighbours, staying there for too long could lead the planner to miss safer viable paths. For instance, say the planner has expanded a node, i at time t and $TC(i, i, t, t + \delta t, A, D) < TC(i, j, t, t + ||i - j||/s, A, D)$ for all $j \in$ Neighbours(i). There may be a neighbour j and some time t' such that $t < t' < t + \delta t$ where $TC(i, j, t, t', A, D) < TC(i, i, t, t + \delta t, A, D)$. Moving the robot to node j at time t' from i at time t would lead to a safer path than staying at node i until time $t + \delta t$. This is why it is important for the safety of the generated paths to use a small value for δt in order to not miss sampling safe paths. By using a smaller value for δt, the chance that a safer path would be missed decreases, but the number of nodes in the search tree would increase.

Fig. 6.4 shows how the search tree for `tBestFS` is being built in space-time as the algorithm progresses. The sequence for this figure is left to right, up to down. The last plot in the Fig. 6.4 also shows the path in red that was returned by `tBestFS`. Fig. 6.5 shows how the dynamic obstacle cost distribution changes over time for the search tree in Fig. 6.4. In Fig. 6.4, the tree is being expanded into areas that have low cost in space-time and that is why the tree is being expanded on the left of the scene. It is evident that the tree is essentially moving around the dynamic obstacles when they are all near each other on the right side of the scene.

Completeness

It is important to prove certain properties about the completeness of the `tBestFS` algorithm in order to guarantee that the algorithm will always return a path in a finite amount of time. For the completeness proof it is first important to prove properties about the bounds of the two functions that comprise the total cost function for an edge. The proof for completeness rests on the fact that the added penalty of returning to a node will be monotonically increasing as the number of times the node has been visited increases and that the cost of an edge is strictly finite for all acceptable parameters. The proof uses these properties to show that in a finite amount of time, it will be more favourable to expand towards the goal node than to stay in the same location.

Lemma 1. *The added penalty of returning to a node in the probabilistic roadmap is monotonically increasing.*

Proof. Since the added penalty for returning to a node increases by one each time the node is visited, as

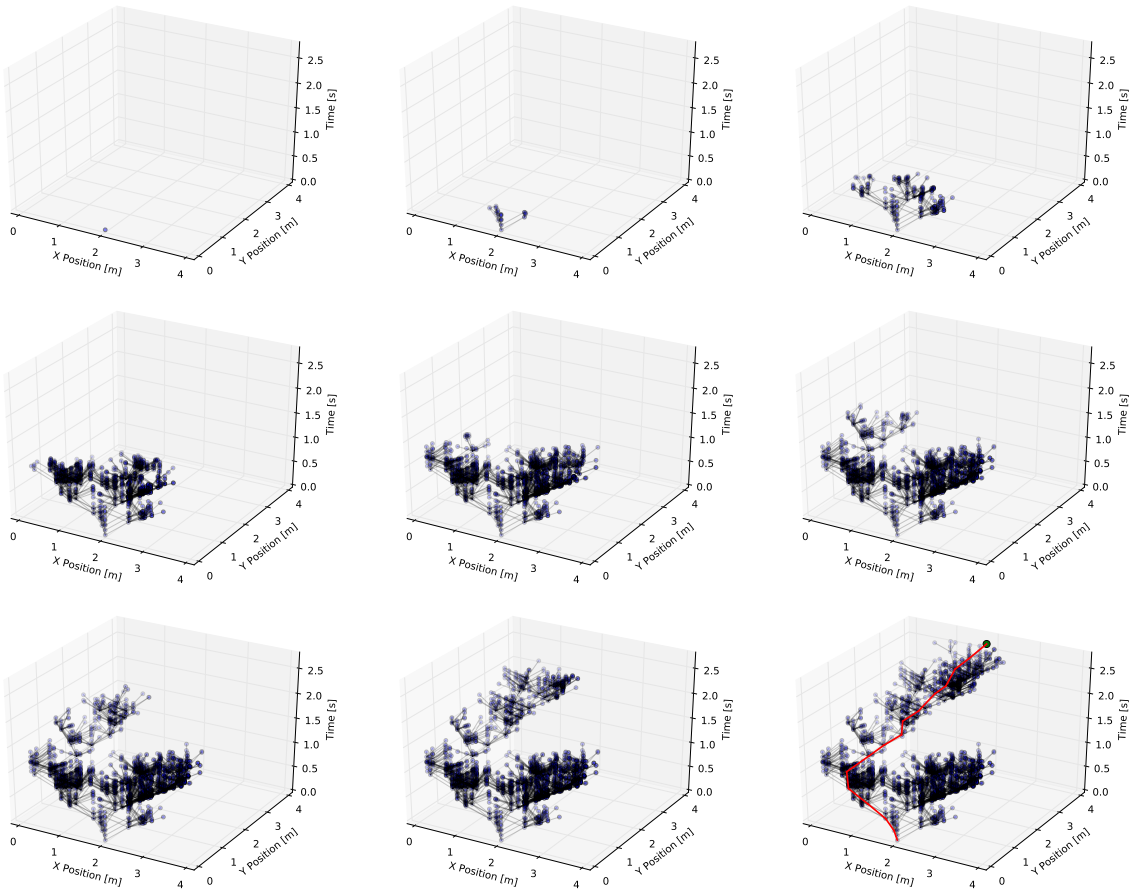


Figure 6.4:

the number of times it has been revisited increases to infinity, so does the added penalty.

□

Corollary 1. *The total cost, TC, is a monotonically increasing function.*

The second property of the functions that make up the edge cost is shown in Lemma 2 which describes that the cost function, C will always have a value less than infinity for all permissible parameters.

Lemma 2. *The cost for any $(m, n) \in E$ for a time interval $[t, t']$ such that $t' - t < \infty$ is strictly less than infinity and $0 < t < t'$.*

Proof. The function, C, can only increase to infinity if the two nodes on the edge are an infinite distance apart or if the cost distribution in Eq. 6.1, P, increases towards infinity. For the former, two nodes cannot be an infinite distance apart because the connection distance, d, is set to be strictly less than infinity and because the $\forall i, j \in V : ||i - j|| \leq \sqrt{w^2 + h^2}$ where $0 < w < \infty$ and $0 < h < \infty$ are the width and height of the scene respectively. Therefore C cannot tend to infinity because of the distance between nodes since no two nodes are an infinite distance away. For the latter, P can only tend to infinity if the normal distribution, $\mathcal{N}$ reaches infinity. This is only possible if the variance approaches 0. However the variance cannot reach 0 in case because the constant β is always greater than zero. Therefore it is impossible for the cost along an edge, C, to tend towards infinity.

□

Using Lemmas 1 and 2, a proof can be constructed for Theorem 1 which states that `tBestFS` is complete and will always return a path.

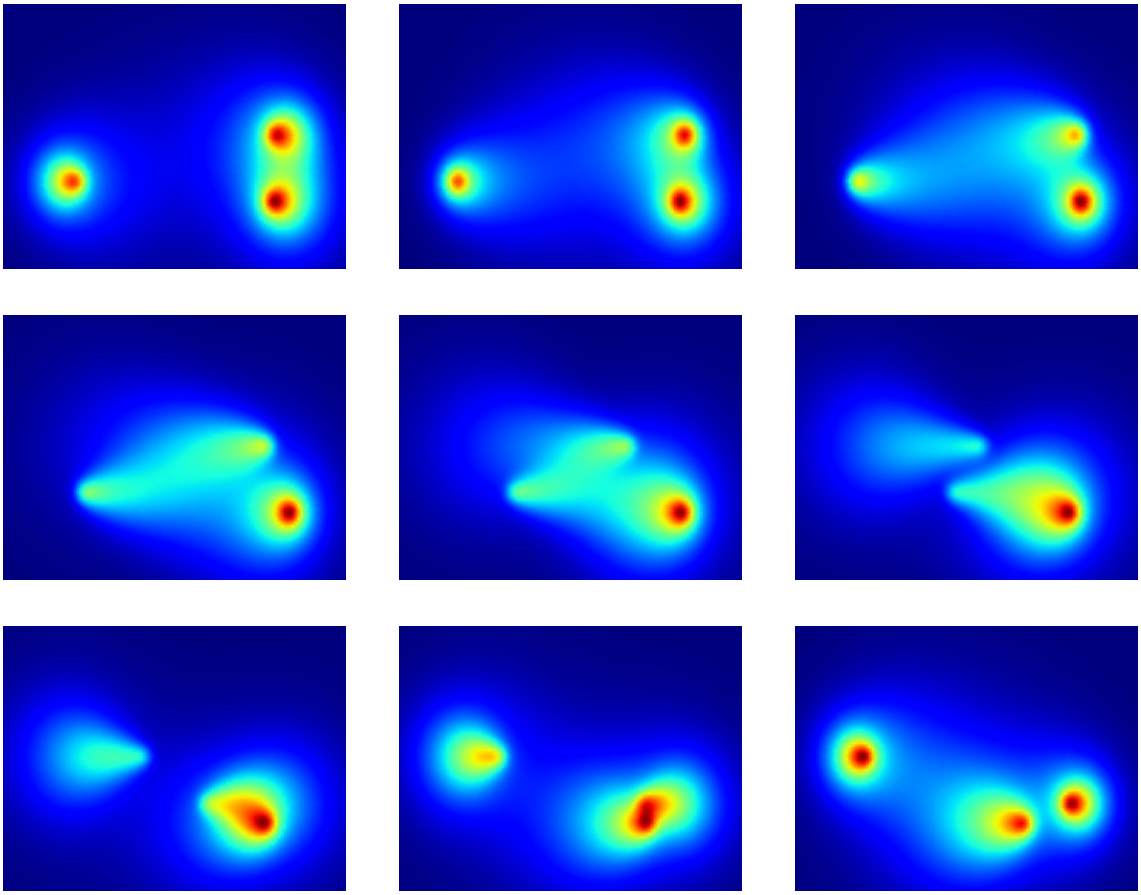


Figure 6.5:

Theorem 1. *`tBestFS` will always return a path in a finite amount of time.*

Proof. First, it is important for the proof to note that the penalty for returning to a previously visited location increases monotonically to infinity as the number of visits for a particular node approaches infinity. Also, the cost function, C, is bounded strictly below infinity for all possible parameters. Now assume the pathological case where $\forall(m,n) \in E, \forall i \in \text{NEIGHBOURS}(g) : C(i,g,t,t',A) > C(m,n,t,t',A)$ for all time intervals $[t,t']$ and where g is the goal node in the probabilistic roadmap. This means that the cost to reach the goal will be larger than any edge cost at any time in the graph. Therefore, without the added penalty for returning to a previously visited location, the goal will never be popped out of the priority queue for expansion. However, since there is a penalty for returning to the same geometrical location, as the algorithm progresses, $\forall(m,n) \in E, \forall i \in \text{NEIGHBOURS}(g) : TC(i,g,t,t',A,D) < TC(m,n,t,t',A,D)$ because each node in the probabilistic roadmap except the goal node would have been expanded one or more times, thus increasing the added penalty. Since the goal node has not been expanded, its total cost would only be the value for C and since C is bounded below infinity and the added penalty increases monotonically towards infinity, all edges to the goal node would have a lower cost than any edge to any other node and thus the goal would be expanded. If the goal is expanded, a path to the goal would be returned. The path is returned in a finite amount of time because the number of nodes in the graph is finite.

□

6.2.3 Replanning

In order to generate safe paths in uncertain, stochastic environments, the planner is not able to simply provide an *a priori* plan from the `tBestFS` algorithm. It is necessary for the planner to generate new paths from the current configuration of the robot as it progresses through the environment if the actual trajectory of a dynamic obstacle deviates too much from its predicted trajectory. This planning occurs in real-time during the execution of the path in the environment. This framework for regenerating paths is called *replanning*. Replanning for this work occurs once the model of movement for a given dynamic obstacle can no longer predict within an acceptable error the actual position of the dynamic obstacle. When replanning, the same probabilistic roadmap is used and the connectivity of the environment is not resampled.

Replanning works by first generating a preliminary path through the environment using the `tBestFS` algorithm. The robot will then follow this path by moving at its prescribed constant speed, s, in a straight line from node to node in the path. Once it reaches a node, i at time t in the path, the robot checks whether the actual locations of the obstacles either sensed by the robot or by an external system differ more than an acceptable amount, δ, from the predicted locations of the dynamic obstacles at time t i.e.

$$\bigvee_{a \in A} ||\tilde{\zeta_a}(t) - \zeta_a(t)|| > \delta$$

Where A is the set of dynamic obstacles. If this proposition is true, there is substantial deviation in the obstacle positions, the planner will update the value of the dynamic variables, T and ξ for the obstacles which are used for predicting their trajectories to t and $\tilde{\zeta_a}(t)$ respectively. These variables are part of the 5-tuple defining the dynamic obstacles which is described in Sec. 6.1.1. Once these variables are updated, the graph is searched using the `tBestFS` algorithm starting from the current position of the robot, i with a starting time of t. It is also important to note that the same dictionary used for storing how many times a node has been visited when researching the graph. The actual path of the robot is a union of the past locations of the robot based on the partial paths generated whilst traversing the environment. Algo. 5 describes the processes of replanning due to the deviations in the obstacle locations.

Replanning is necessary because it allows the planner to redirect the robot to new, safer paths as the environment changes. This is important because the initial path generated by the `tBestFS` algorithm uses the predicted motion of the obstacles and assumes they will stay on this predicted trajectory for the execution of the path. The `tBestFS` is able to account for some uncertainty in the motion of the obstacles as described by the cost distribution in Sec. 6.1.2 but this distribution is not able to fully account for path divergences. By updating the information for the dynamic obstacles and re-searching the probabilistic roadmap using `tBestFS`, safer overall paths can be generated. It is possible to regulate the number of times the planner will need to replan and thus the safety of the plan by adjusting the value for δ. The larger the value for δ, the more the obstacles the will need to diverge from the prescribed trajectories in order to replan and vice-versa.

Completeness

Theorem 2. *The robot will always reach the goal in a finite amount of time given the proposed replanning scheme outlined in Algo. 5 for $\delta > 0$.*

Proof. □

6.2.4 Discussion

The final algorithm developed is able to generate low cost paths through the environment by using the available information about the obstacles' motion. The planner is able to do this by searching through

Algorithm 5 GetPath$(n, d, w, h, \delta, p, g, O, A, R)$

Input:
n: Maximum number of samples for the roadmap
d: Maximum distance between neighbouring nodes in the roadmap
w: Width of the scene
h: Height of the scene
Output:

1: $(V, E) \leftarrow$ Roadmap(n, d, w, h, O)
2: $\Pi \leftarrow \emptyset$
3: $q \leftarrow p$
4: $t \leftarrow 0$
5: **while** $||$Back$(\Pi) - g||_2 > R$ **do**
6: $\pi \leftarrow$ SearchGraph(V, E, R, A, q, g, t)
7: **for all** $(i, t') \in \pi$ **do**
8: $\Pi \leftarrow \Pi \cup \{i\}$
9: **for all** $a \in A$ **do**
10: Step(a)
11: **if** $\bigvee_{a \in A} ||\tilde{\zeta_a}(t') - \zeta_a(t')|| > \delta$ **then**
12: **for all** $a \in A$ **do**
13: $T_a \leftarrow t'$
14: $\xi_a \leftarrow \tilde{\zeta_a}(t')$
15: $q \leftarrow i$
16: $t \leftarrow t'$
17: **break**
18: **return** Π

space-time over a two dimensional graph using the `tBestFS` algorithm which uses the cost distribution described in Sec. 6.1.2 to weight the edges of the search tree. The algorithm is also able to adapt and create new plans through the environment when its' prediction of the obstacles' motion is no longer able to accurately determine where the obstacles are going to be in future. This replanning stage of the algorithm allows it to be used in real-time in stochastic dynamic environments. This mimics the reactive behaviour of potential fields, with the added benefit of provable completeness and not being a solely reactive planner. Likewise, the replanning ability allows the information given to the planner about the motion of the obstacles not to be perfect. Even if the equations of motion that the motion prediction system provides in now way describe the actual motion of the obstacles, the algorithm will simply replan at every time step and instead of using the cost distributions to determine paths through the environment, the planner will simply move to the next best node in the probabilistic roadmap. This means that the algorithm is able to continuously plan through stochastic dynamic environments utilizing the available information to move the robot to the goal.

Chapter 7

Implementation

7.1 Initial

Due to the ease of development and the author's experience with the language, Python was chosen as the initial language for the implementation. All three attempts described in Ch. 5, were implemented and a suite of tests scripts and visualization mechanisms were created in order to incrementally assess how each planner was behaving. However after the implementation was completed for the final design described in Sec. 5.4, it was discovered that Python could not produce solution paths (without replanning) through dynamic environments in a real-time scenario. Since Python could not search the graph within an acceptable amount of time, it could be only used for *a priori* planning and therefore could not be used in stochastic environments in which replanning would be needed. Due to this drawback, the author decided to rewrite the entire implementation in C++. Since this initial attempt is complete, it is publicly available at [25].

7.2 Final

The final implementation was written in C++ due to the tremendous speed improvement which made planning in real-time viable. Due to the object oriented structure of C++, the planner and the associated data structures were encapsulated in classes which made it very easy for people to use the code as an API. While designing the software, extra effort was put into making the code easily usable by other people who are not necessarily experts in robotics or motion planning. The system has an easy import mechanism and several examples on how to use the different planners that have been implemented.

In order to visualize the paths that the planner has generated, the C++ code can exports the paths of the robot and the dynamic obstacles to a JSON file and a Python script then can read and parse the generated JSON and display the paths using either Matplotlib or RViz. Matplotlib is an open source library for creating plots from data [26]. It is very similar to Matlab or Mathematica except it runs natively in Python. RViz is a core component of the Robotic Operating System (ROS) which can render dynamic scenes in three dimensions and can visualize native geometrical and navigation messages from ROS [27]. Since the paths are exported to JSON, a standard format with parsers in many languages, third party software can be developed that can parse the generated JSON file and analysis or visualized the generated paths. Likewise, the `Dodger`library can be imported by a third party program to control robots from the generated paths or to visualize the paths using other visualization tools such as OpenGL. A diagram of this tool-chain is shown in Fig. 7.1.

By separating the visualization code completely from the planner code, the software is more easily portable to different visualization frameworks and ensures that the user does not need to have a certain graphical software or ROS installed for the software to be able to compile and use the library. This allowed the planner code to be entirely self contained and simple to compile with a provided CMake file

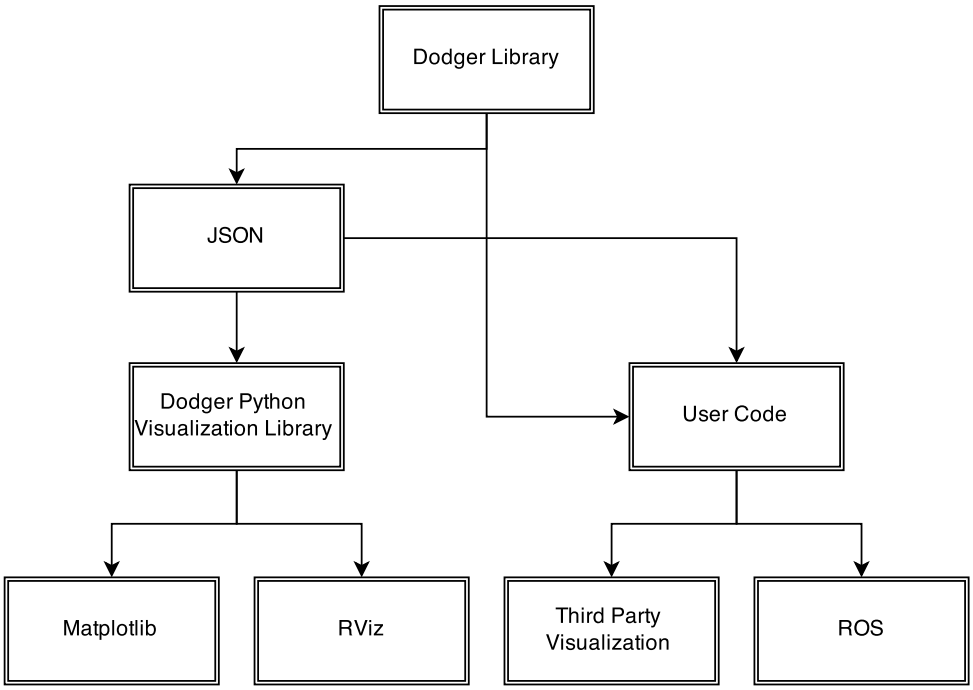


Figure 7.1:

which compiles the C++ code into a static library. Likewise, this design decision in the implementation allows the generated paths to be saved and either analysed or visualized at another time. Also, since the planner had no dependencies, experiments could be run on the servers provided without needing root access. This code is also made publicly available at [28].

Chapter 8

Experimental Setup

8.1 Design

In order to quantify the performance of the developed planner and to judge how well it performs, experiments needed to be run that can determine the degree of safety present in the generated paths and how long it takes to generate these paths. To gather this data, three different scenes were created with differing number dynamic obstacles with different trajectories. The independent variables, or parameters, for each scene consisted of the amount of noise, ϵ for the dynamic obstacles and the speed of the robot, s. For the experiments, a given value of ϵ is the same for all of the dynamic obstacles in the scene. The amount of noise ranges from 0.002 to 0.01 with an increment of 0.002 and the speed ranges from 1.0 to 4.5 meters per second with an increment of 0.5 meters per second. Due to the stochasticity of the planner and the dynamic obstacles, each set of parameters for each scene was run 20 times. The values for each metric recorded is averaged and the standard deviation for each set of parameters for each scene is presented.

In order to judge how the developed planner compares to a standard algorithm developed to plan in stochastic dynamic environments, a slight variant to the potential fields planner described in Algo. 1 was implemented. The difference is that the implemented potential fields planner used a different repulsive potential function described in Eq. 8.1.

$$U_{rep}(p, t, A) = \max_{a \in A} \frac{k}{||\tilde{\zeta_a}(t) - g||^2 + \varepsilon} \tag{8.1}$$

In Eq. 8.1, k is a scaling constant where $k > 0$, and ε is constant where $0 < \varepsilon < k$ that ensures that the function does not exhibit a singularity. This variant in the potential fields ensures that the planner has no information about the motion of the obstacles and can thus be used to compare how this information can benefit a planner. Also, since this planner is not sampling based, it can be used to see how a complete sampling based planner can be more or less effective. The results from this experimental setup are described in Ch. 9.

8.2 Metrics

To quantify the performance of the planners, four metrics are used, three to measure the safety and one for computational time. Along with these metrics, the standard deviation is collected for each scene and set of parameters in order to gauge the reliability of the planner under different circumstances. These metrics are described in the sections below.

8.2.1 Safety

In order to quantify the safety of a given path Π, three metrics were devised. These metrics need to be used together to measure the safety of a path and each represent a component of what it means for a path to be "safe". The first metric is the most straightforward and what is probably the first metric to come to mind, the minimum distance to any dynamic obstacle at any given time. This metric represents for a given path, what was the closest the robot came to coming into contact with a dynamic obstacle. Since collisions are more likely to occur as the robot approaches an obstacle due to the uncertainty in its motion, this metric serves to provide a simple way of quantifying the safety without having to account for the motion of the obstacles. This metric is defined formally in Eq. 8.2.

$$\mathit{MinDist}(\Pi, A) = \min_{t \in \mathcal{T}} \min_{a \in A} ||\zeta_a(t) - \Pi(t)|| \tag{8.2}$$

In Eq. 8.2, Π is the path of the robot, $\mathcal{T}$ is the time interval for the path, and A is the set of dynamic obstacles in the scene. Since this metric does not account for the motion of obstacles, it cannot be used as the sole quantification of safety. For example, if the robot moved near a dynamic obstacle, but was moving in the opposite direction of the obstacle, the path taken by the robot would still be safe, because there would be a smaller chance of the robot actually coming into contact with the obstacle. The robot could have a smaller cost over its path even if it moved near an obstacle than a robot that was farther away from an obstacle but moved directly into its trajectory.

Another metric used to compute the safety of a path is the maximum cost incurred by the robot along the path. This is what the planner described in Sec. 6.2 is trying to minimize. This metric describes how risky a certain path is by determining the likelihood that the robot's path would intersect with the trajectory of a dynamic obstacle. A formal definition of this metric is shown in Eq. 8.3.

$$\mathit{MaxCost}(\Pi, A) = \max_{t \in \mathcal{T}} P(\Pi(t), A) \tag{8.3}$$

Since the potential field planner implementation that is used to compare with the planner created in this work does not utilize the information given about the cost distribution associated with dynamic obstacles, this metric also indicates how access to this information can contribute to generating safer paths through dynamic environments. Since this metric is used to measure the safety of both the potential fields planner and the developed planner, this metric can be used to quantify how information about the motion of dynamic obstacles can either improve or have no effect on the generated paths.

The last metric used to measure the safety of a generated paths is the average cost associated with the robot along the path. This metric does not provide the same indication of safety as in Eq. 8.3 because the overall safety of a path is not just the average safeness along the path. For instance if the majority of a path taken by a robot has a relatively low cost, but then the robot comes into collision with an along the path, this path cannot be labelled as safe. However, this metric combined with the metrics described in Eq. 8.3 and Eq. 8.2 can be used to gauge the level of safety for a given path. This last metric is described formally in Eq. 8.4.

$$\mathit{AvgCost}(\Pi, A) = \frac{1}{\max \mathcal{T}} \cdot \int_{\mathcal{T}} P(\Pi(t), A)\, \mathrm{d}t \tag{8.4}$$

Since experiments were conducted with a given set of parameters multiple times due to the stochasticity of the dynamic obstacles and the developed planner, each of the metrics were gathered for each individual run. The data gathered for each of these metrics was then averaged and the standard deviation of the dataset was determined. As such, it is the averages and standard deviations of these metrics that are used to quantify the safety of a generated path.

8.2.2 Computational Time

In order to determine the feasibility of the paths generated by both planners, the amount of computational time needed for each planner was collected. Collecting this data can indicate how well the developed planner can perform in a real-time scenario. Since the time it takes to compute a path with replanning, the computational time also represents how long it would take for the robot to execute the path because it is replanning in real-time.

8.2.3 Variance

For each set of parameters, the experiments were conducted multiple times due to the stochasticity of the dynamic obstacles and the planner. The standard deviation for each metric and each set of parameters were recorded and used to indicate how the noise injected into the trajectories of the dynamic obstacles and the speed of the planner affect the variance in the results. This metric provides an indication of how consistent the planners will be for different sets of parameters and can contribute to judging how well each planner will work in a real world scenario over time.

Chapter 9

Results

9.1 Safety

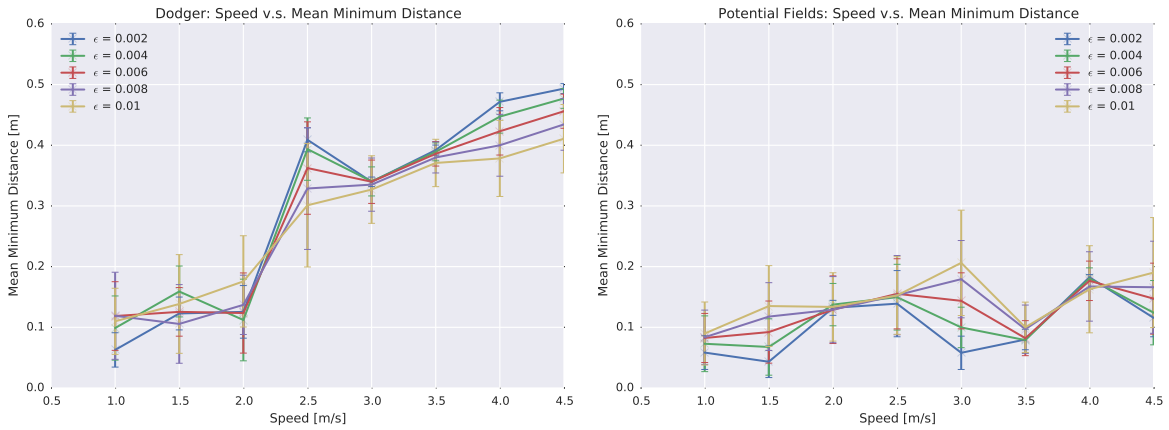


Figure 9.1: Plots showing how the average minimum distance to the obstacles changes as the speed increases for various amounts of obstacle position uncertainties

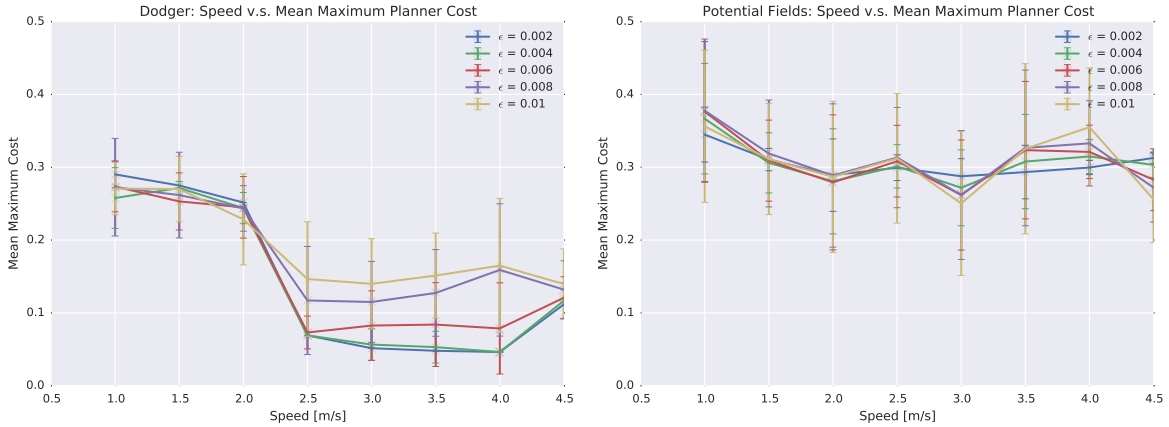


Figure 9.2:

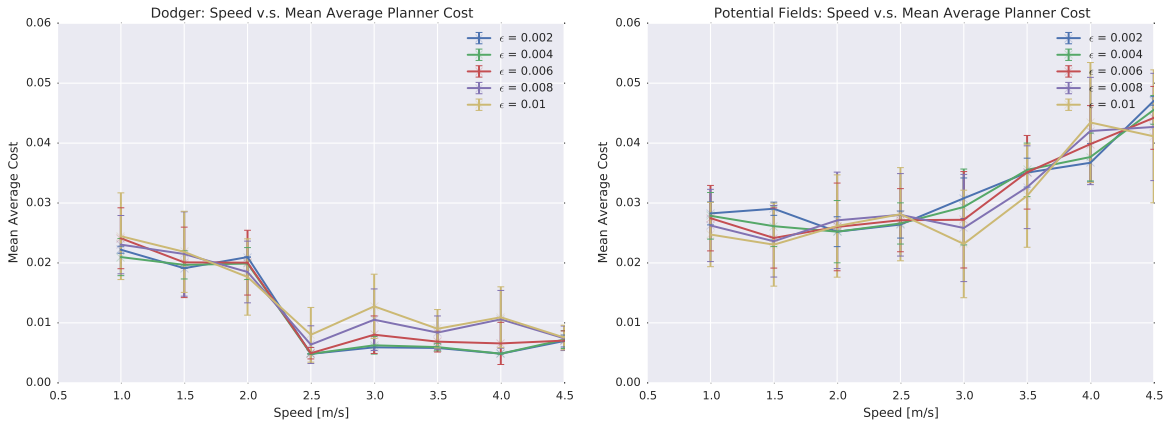


Figure 9.3:

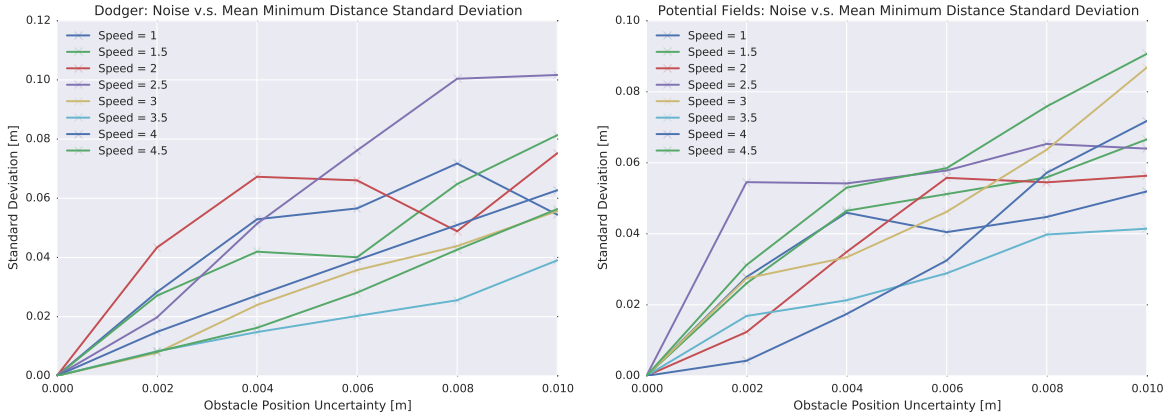


Figure 9.4:

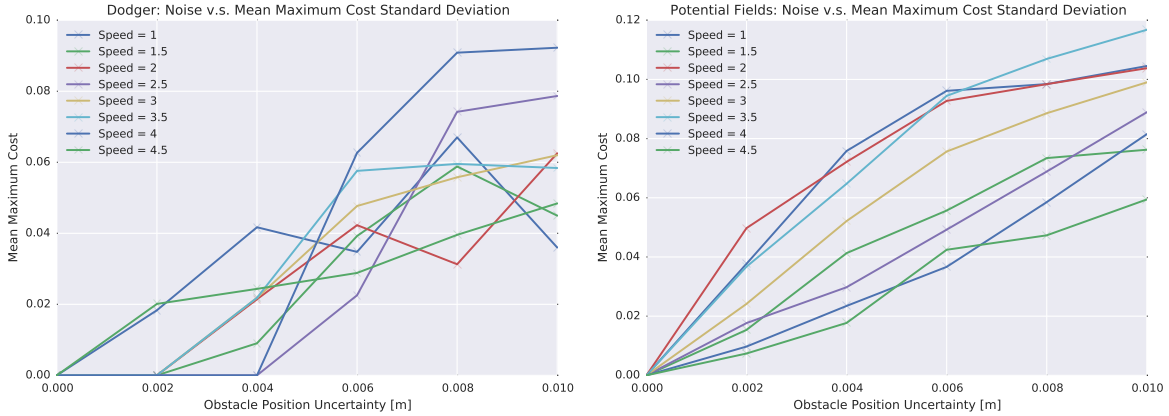


Figure 9.5:

9.1.1 Variance

9.2 Computational Time

9.2.1 Variance

9.3 Behaviour

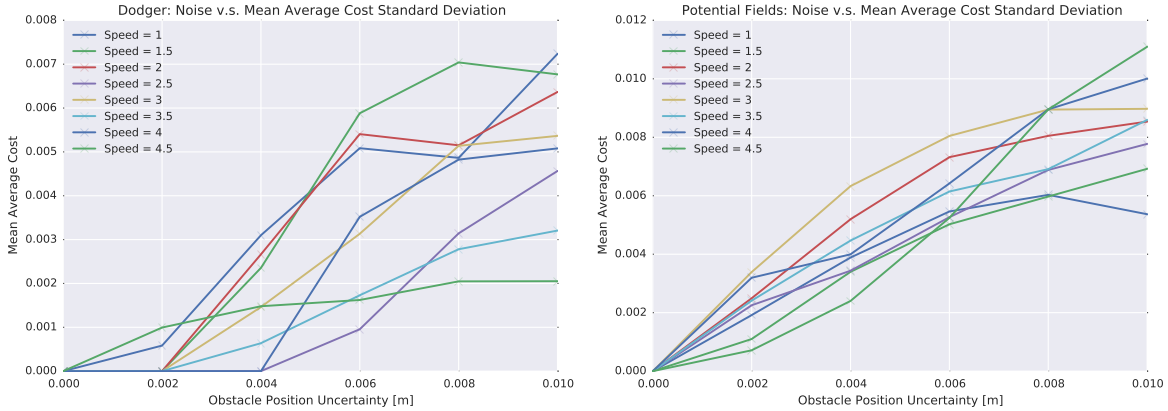


Figure 9.6:

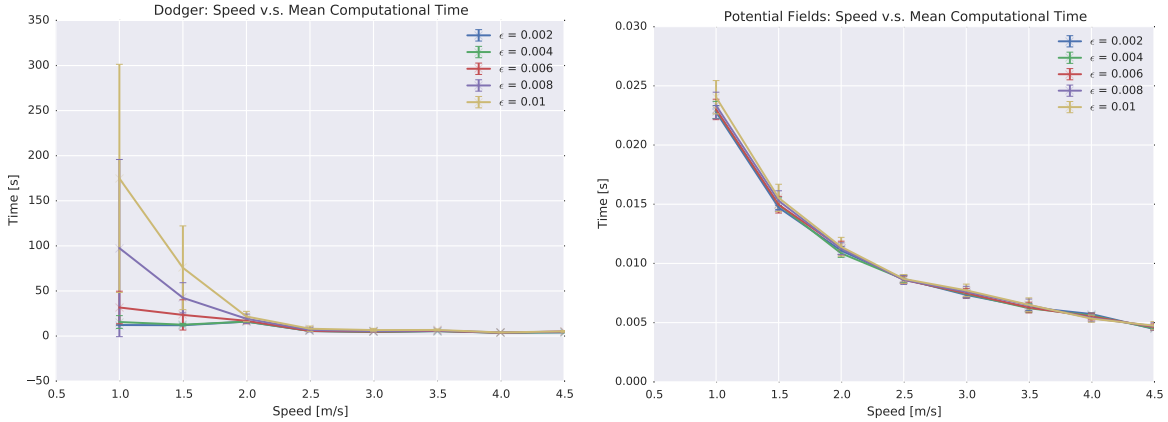


Figure 9.7: Plots showing how the computational time changes as the speed increases for various amounts of obstacle position uncertainties

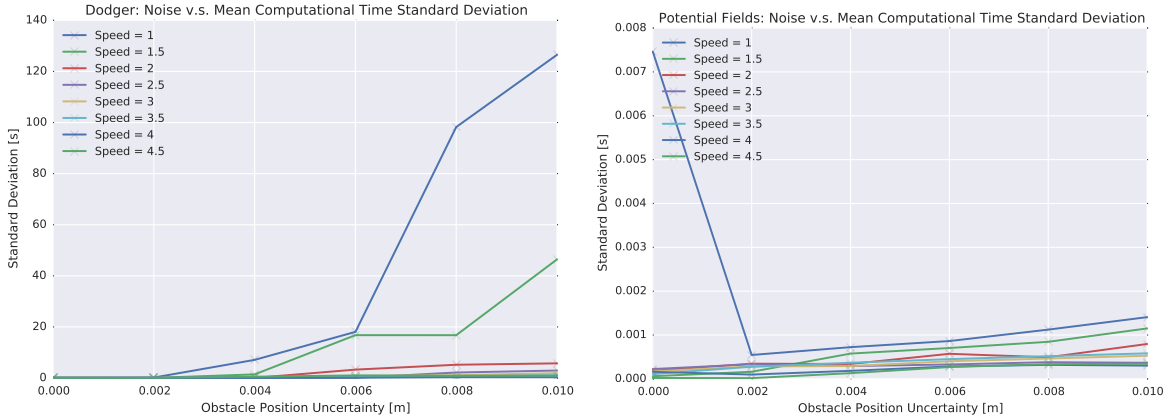


Figure 9.8:

Chapter 10

Discussion

10.1 Conclusions

10.2 Future Work

Chapter 11

Ethics

Since this project does not use any personal or classified information and the algorithms developed have not been tested with humans as obstacles, this project does not raise any ethical concerns. One could argue that the algorithms developed could be used in the robot uprising, but that is not the intention of the work and the robot uprising is not set to come for another few decades. A possible vessel for this uprising is shown in Fig. 11.1 [29].

Figure 11.1: A picture of a possible vessel for the robot uprising

Chapter 12

Acknowledgements

The author would like to thank Dr. Michael Weir and Dr. Erion Plaku for their valuable insight throughout the development of this project and the School of Computer Science for a truly phenomenal undergraduate education. The author would also like to thank his wife, Jessica, for putting up with the late nights and limited contact whilst this project slowly enveloped his life.

Chapter 13

Appendix

Bibliography

[1] H. M. Choset, *Principles of robot motion: theory, algorithms, and implementation.* MIT press, 2005.

[2] S. M. LaValle, *Planning algorithms.* Cambridge university press, 2006.

[3] M. Weir, A. Buck, and J. Lewis, "Potbug: A minds eye approach to providing bug-like guarantees for adaptive obstacle navigation using dynamic potential fields," in *From Animals to Animats 9* (S. Nolfi, G. Baldassarre, R. Calabretta, J. Hallam, D. Marocco, J.-A. Meyer, O. Miglino, and D. Parisi, eds.), vol. 4095 of *Lecture Notes in Computer Science*, pp. 239–250, Springer Berlin Heidelberg, 2006.

[4] T. Lozano-Pérez and M. A. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles," *Commun. ACM*, vol. 22, pp. 560–570, Oct. 1979.

[5] Y. Koren and J. Borenstein, "Potential field methods and their inherent limitations for mobile robot navigation," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pp. 1398–1404, IEEE, 1991.

[6] A. Wallar and E. Plaku, "Path planning for swarms by combining probabilistic roadmaps and potential fields," in *Towards Autonomous Robotic Systems*, pp. 417–428, Springer, 2014.

[7] A. Wallar and E. Plaku, "Path planning for swarms in dynamic environments by combining probabilistic roadmaps and potential fields," in *Swarm Intelligence (SIS), 2014 IEEE Symposium on*, pp. 1–8, IEEE, 2014.

[8] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *Robotics and Automation, IEEE Transactions on*, vol. 12, pp. 566–580, Aug 1996.

[9] G. Alankus, N. Atay, C. Lu, and O. Bayazit, "Spatiotemporal query strategies for navigation in dynamic sensor network environments," in *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pp. 3718–3725, Aug 2005.

[10] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, pp. 100–107, July 1968.

[11] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[12] J. Edmonds, "Optimum branching," *Journal OF Research of the National Bureau of Standards*, November 1967.

[13] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, p. 4850, 1956.

[14] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36, p. 13891401, 1957.

[15] S. M. LaValle, "Rapidly-exploring random trees a new tool for path planning," 1998.

[16] J. M. Phillips, N. Bedrossian, and E. Kavraki, "Guided expansive spaces trees: A search strategy for motion-and cost-constrained state spaces," in *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, vol. 4, pp. 3968–3973, IEEE, 2004.

[17] R. E. Korf, "Linear-space best-first search," *Artificial Intelligence*, vol. 62, no. 1, pp. 41–78, 1993.

[18] M. Phillips and M. Likhachev, "Sipp: Safe interval path planning for dynamic environments," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 5628–5635, IEEE, 2011.

[19] V. Narayanan, M. Phillips, and M. Likhachev, "Anytime safe interval path planning for dynamic environments," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 4708–4715, IEEE, 2012.

[20] C. Sung, D. Feldman, and D. Rus, "Trajectory clustering for motion prediction," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 1547–1552, Oct 2012.

[21] P. P. Choi and M. Hebert, "Learning and predicting moving object trajectory: a piecewise trajectory segment approach," *Robotics Institute*, p. 337, 2006.

[22] T. Liu, P. Bahl, and I. Chlamtac, "Mobility modeling, location tracking, and trajectory prediction in wireless atm networks," *Selected Areas in Communications, IEEE Journal on*, vol. 16, no. 6, pp. 922–936, 1998.

[23] F. T. Pokorny, M. Hawasly, and S. Ramamoorthy, "Multiscale topological trajectory classification with persistent homology," in *Robotics: Science and Systems X*, 2014.

[24] F. Pokorny, M. Hawasly, and S. Ramamoorthy, "Topological trajectory classification with filtrations of simplicial complexes and persistent homology," *International Journal of Robotics Research*, 2015.

[25] A. Wallar, "Racer: Path planning in stochastic dynamic environments in python." `https://github.com/wallarelvo/racer`, 2015.

[26] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[27] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, 2009.

[28] A. Wallar, "Dodger: Path planning in stochastic dynamic environments in c++." `https://github.com/wallarelvo/Dodger`, 2015.

[29] R. McKee, "Skynet: 5 Things You Didn't Know." `http://uk.askmen.com/entertainment/special_feature_300/339_skynet-5-things-you-didnt-know.html`.