Ti-HDRM: Time-indexed Hierarchical Dynamic Roadmaps for Robotic Motion Planning in Environments with Moving Obstacles

Matt Timmons-Brown

4th Year Project Report
Computer Science
Institute for Perception, Action & Behaviour
School of Informatics
University of Edinburgh

2022

Abstract

This work presents Time-indexed Hierarchical Dynamic Roadmaps (Ti-HDRM), a collision-free, joint-velocity-limit-respecting and resolution complete robotic motion planning algorithm for use in highly dynamic environments. By comprehensively reviewing the state-of-the-art robotics literature, an un-addressed specification for safety-critical motion planning is introduced, for which no motion planning method currently fully satisfies. Three naive algorithms (termed Post-Processed-Time HDRM) are created. Through their pitfalls, Ti-HDRM is conceptualised, introducing novel algorithms and a mathematical proof of its overall resolution completeness. Implementations of all four algorithms are created in C++, and through this, Ti-HDRM's supremacy in solving motion in environments with moving obstacles is demonstrated in a variety of experiments with the Nextage humanoid robot, a video of which can be found here: `https://youtu.be/L9aMBA4f8ao`. A nuanced conclusion is provided comparing the full coverage benefits of Ti-HDRM with the increased computational cost of the method.

Acknowledgements

Professionally, I would like to thank my supervisor Professor Sethu Vijayakumar for his guidance, support and all of the opportunities he has provided for me over the last several years as a member of his research group. I'd also like to extend my deep gratitude to Dr Vladimir Ivan for his unwavering help, advice and the time he has dedicated to assisting me in the successful completion of this project.

Personally, I would like to thank my parents, Rebecca and Jeff, without whom my achievements would simply not have been possible. I am endlessly grateful also to my wonderful girlfriend, Maria, who has been my biggest supporter - from making delicious dinners to accompanying me in the Bayes Centre. Thank you for bearing with me. Last, but not least, I express my gratitude to my dear Computer Science colleague Neil for his enduring friendship, discussions and wisdom.

Finally, thank you to Eben, Liz and all of those who invented the Raspberry Pi. I became interested in Computer Science in 2012 when I was first exposed to programming through the Pi, and a decade later I submit this thesis as the final part of my Computer Science undergraduate degree. Without your work, my life and fulfillment would be immeasurably different.

Contents

Chapter 1

Introduction & Background

Alongside the likes of machine-perception and mechatronic design, *motion planning* is one of the pre-eminent problems in the field of robotics. It concerns finding a sequence of poses (*configurations*) that permit a robot to move from one location to another [39].

As humans we perform the task of motion planning every instant of our waking lives, with our movements subconsciously planned by our brain and executed by our muscles. For robots, computers and algorithms instead decide on how to instruct the angles of electric joints. It has proven difficult to develop algorithms similar to our instinctive human motion ability [43].

A core problem is ensuring that robots do not collide with their environment. Increasingly, robots are being used in close proximity to humans - from factories to hospitals [5, 8]. These often combine dexterous robotic arms with a mobile base (see Figure A.1) to allow movement around a space. It is critical that robot motion in these settings is *safe*, i.e. without collision. It is also important that, when a motion is commanded to a robot, it can be executed within the parameters of that robot's hardware - it should respect factors such as the range of motion and capabilities of its motors.

Safety-Critical Motion Planning Specification: To formalise this, a desirable specification for safety-critical motion planning for a robot is as follows:

1. **Free of collision** - the robot should not hit:
    - Itself - as this can cause damage to the robot
    - Static obstacles - objects in a robot's environment that do not move, such as tables and equipment
    - Dynamic obstacles - objects that move in the robot's surroundings when the robot is still, such as other machinery/robots and humans. Equally, dynamic obstacles are static objects that appear to be moving *relative to the robot* when the robot itself is in motion (i.e. it is moving on a mobile base)
2. **Respecting robot hardware limitations** - the robot should not exceed:
    - The range of motion of its joints
    - The maximum intrinsic velocity at which these joints can operate

3. **Motion plan guarantees** - when asked to plan motion between robot poses, an ideal motion planner will be global in its answer, meaning that it will provide one of two outcomes:
   - Solved - it is possible to move between the commanded poses, while respecting the properties above
   - Solution doesn't exist - definitively answer that a motion between the requested poses is not possible to safely execute

This project primarily introduces a new method, named **Time-indexed Hierarchical Dynamic Roadmaps** (shortened to **Ti-HDRM**), that can plan motion that meets this strict specification for robot arms and other similar robots. Before listing the contributions any further, the necessary background is introduced.

1.1 The Terminology of Motion

Motion planning is the high-level problem of finding a robotic motion that permits a robot to travel from a start configuration, to a goal configuration [39]. Specifications that this motion should satisfy, such as avoiding obstacles and respecting robot physical limitations, are referred to as *constraints*. These restrict a free robot system by narrowing its achievable motion possibilities.

When discussing a 'robot', this work is referring to machines with articulated rigid bodies (*links*) connected by joints [12]. More specifically, single-ordered kinematic chains - where each joint is in a series, and is the single child of its parent - are the focus of this thesis. The most common examples of this are robot arms or robot legs (see Figure A.2).

Configuration and Workspace

Every unique configuration of a robot can be represented as a point in its *configuration-space* (*C-space*) [39]. Typically, such a configuration, q, is formed of an array of it's n joint positions: $q = (\theta_1, \ldots, \theta_n)$ where n is the number of joints a robot has. n is also known as the *degrees of freedom* (*DOF*) of a robot. The C-space is therefore n-dimensional, e.g. a 6 DOF robot arm has a 6-dimensional C-space.

This is distinct from the *workspace* of a robot, which is the set of points that can be reached by a robot's end-effector [30]. An end-effector is the final link of a robot - e.g. the hand of a robotic arm. For example, a pen-plotter robot, can only move around a planar surface in x and y, thus its workspace is 2-dimensional. Robot arms can manipulate their end-effectors fully in free-space in both position (x, y, z) and orientation ($roll$, $pitch$, yaw), meaning that their workspace is 6-dimensional.

Joint Limits and Collision Freedom

Rotating *actuators* (motors) make up the joints of a robot and each typically have limitations on their range of motion, expressed as a lower and upper bound. These are often not uniform across the joints of a robot. For example, a 'shoulder' joint could have range $\{-\pi, +\pi\}$, whereas a 'wrist' joint could rotate between $\{-\pi/2, 0\}$.

These position limits, x, are intrinsic properties of the robot introduced at its design. Any configuration outside of these limits is excluded from its usable C-space. Actuators also cannot move at arbitrarily high velocities, and thus have velocity limits too, $\dot{x}$.

Any configuration that results in one of the bodies of the robot making collision with either itself or its environment is not safe (referred to as not *valid*). When the robot makes a collision with itself, this is termed a *self-collision*. When the robot makes a collision with environment, if the obstacle was not in motion, it is a *static obstacle*, and a *dynamic obstacle* otherwise.

Free C-space, $\mathcal{C}_{\text{free}}$, therefore refers to the robot configurations 'where the robot neither penetrates [any type of] obstacle nor violates a joint [position] limit' [39]. See Figure 1.1 for an example of a 2D C-space representation.

1.2 The Motion Planning Problem

The specific motion planning problem this thesis addresses can therefore be formalised as follows (derived from [39]):

Given an initial joint state $= x_{\text{start}}$ and a desired final joint state $= x_{\text{goal}}$ at a given time $= T$, find a set of configurations, qs, such that each configuration $qs_i \in \mathcal{C}_{\text{free}}$ for their respective $t_i \in [0, T]$ and that motion between these configurations does not exceed joint velocity limits, $\dot{x}$.

Motion Planning Methods

As there are variations of both robot design and the motion planning problem, there is not a single motion planner that is applicable to all motion problems. An overview based on [39, 43] of some common methods (detailed further in Section 1.3) are:

- **Analytical methods** - these represent the geometry of the C-space of a robot or of the robot bodies themselves, and then directly solve equations of motion without resolving to numerical methods.
- **Nonlinear optimisation** - by representing a motion planning problem as a series of tunable parameters, e.g. the coefficients of a polynomial, it is possible to minimise a cost function that also satisfies the motion planning constraints.
- **Search methods - sampling & grid** - sampling algorithms tend to use random/deterministic behaviour to chose samples from the free C-space, and then connect these samples locally into a searchable graph data structure that represents the permitted motions that the robot can execute. Similarly, grid-based techniques typically discretise the free C-space (or workspace) of the robot into a connected grid. This can then be searched to find paths between cells.

Completeness

When discussing these methods, *completeness* is important to understand. A motion planning algorithm 'is considered *complete* if, for any input input, it correctly reports whether there is a solution or not. If a solution exists, it must return one in finite time' [37]. This is the strongest form of a motion plan 'guarantee', however there also exists weaker but important derivatives.

If an algorithm is *probabilistically complete*, it means, with enough samples (i.e. as $\lim_{n\to\infty}$ where n is the number of C-space samples), the probability of definitively finding a solution (or reporting that there isn't one) 'asymptotically converges to 1' [44]. This is often used to describe sampling-based algorithms. Similarly, if an algorithm is *resolution complete*, it is able to report a solution or the absence of one in finite time, up to a certain C-space or workspace resolution (where the continuous C-space/workspace has been discretised). If the algorithm reports there is no solution, that does not mean to say that a solution may not exist at a finer resolution however. Grid-based methods often refer to this. In general, resolution completeness provides a stronger guarantee than its probabilistic sibling.

1.3 Understanding the State-of-the-Art

Analytical Methods

Inverse kinematics (*IK*) is the use of kinematic equations to determine the joint angles necessary for a robot to achieve a desired end effector pose [43]. A simple motion planner could then involve compositing many poses, running inverse kinematics on them to determine the joint angles necessary for these poses, and then executing them at some timed intervals with linear interpolation between them.

There is a body of literature related to approaches like this, including Choudhury et al. (2004) [29] that achieves motion plans for kinematically controllable systems in environments cluttered with obstacles. The authors develop an algorithm that exploits the closed-form IK solution of a robot. Some beneficial properties include that the method is both computationally efficient and fully complete.

However, this work relies on the existence of a closed-form solution to the kinematics of a robot. While this is feasible in the simple examples used, solving the IK problem for single-ordered kinematic chains in general is much more complex [43], as the equations to solve the kinematics for most non-trivial robots are generally nonlinear - meaning that a **closed-form solution does not exist**. Even if it does exist, multiple solutions for a given end effector pose may be possible, or even infinite solutions for a kinematically redundant manipulator. Because of this, the IK problem is often solved iteratively, and thus requires an initial configuration guess, and **may get stuck in a local optima** - particularly when there are joint constraints. These methods are **not complete**.

Optimisation

By expressing a motion planning problem as a general nonlinear optimisation, with equality and inequality constraints, a number of standard techniques can be used to solve this. For example, gradient based-methods, such as sequential quadratic programming [42], or non-gradient based methods such as simulated annealing [31, 39].

Schulman et al. (2013) [42] present a gradient approach for incorporating collision avoidance into motion planning. This method scales to higher-DOF robots (like robot arms) - something that analytical methods could not generalise to.

Other methods, such as much-cited 'CHOMP' [41] and its successor 'STOMP' [34], use gradient/stochastic optimisation approaches to generate smooth motion plans. They achieve this by optimising over a range of both robot dynamics and task-based criteria. Further extensions of these methods [40] generalise these algorithms to generate collision-free trajectories in environments with dynamic obstacles by adding a dynamic cost to the optimisation process.

Critically however, while outputs of these methods can be near-optimal solutions, they typically **require an initial guess of parameters for the solution to start from**. As these problems are complex with many DOF, the entire feasible solution space of optimisation problems for robotic arms is usually not convex - meaning that the **optimisation can get stuck far away from an optimal or even local solution** [44]. This undesirable property results in **no completeness guarantees**.

Search Methods - Sampling & Grid

The searching of graphs (collections of nodes with connected edges) is a well-addressed problem. Algorithms, such as *A* search*, introduced by Hart et al. (1968) [33, 1], are able to traverse graphs and return paths with varying properties.

While details are saved for future sections, A* is an informed search algorithm that can return the guaranteed optimal path through a weighted graph. Weights in this context represent the cost travelling from node to node, and the 'optimal' path is the path between a start and goal node with the lowest cost. The completeness, optimality and efficiency of the A* algorithm can be harnessed for motion planning. By representing the free C-space of a robot as a connected graph, this powerful algorithm can be applied to find a sequence of configurations that take a robot from a start to goal pose. A free C-space map of this nature is referred to as a *roadmap* [39].

Probabilistic Roadmaps: One of the first papers to exploit this was Kavraki et al. (1996) [35] , introducing *Probabilistic Roadmaps (PRM)*. The algorithm sets out 2 phases:

- Pre-planning Phase - in this expensive computational phase, the C-space of the robot is sampled. At each sample, the robot configuration is evaluated to detect whether it is in collision. If it is, the sample is discarded. If it is not, it is added to the roadmap.
- Query Phase - once the roadmap has been built, a motion plan between any of the nodes of this safe graph is found with A*.

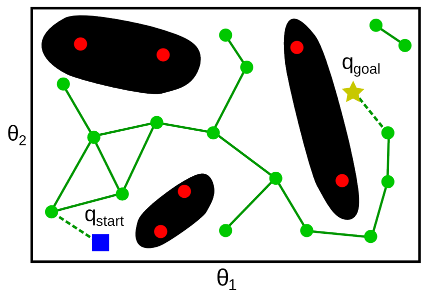


Figure 1.1: Probabilistic Roadmap (PRM) for a 2 DOF robot through $\mathcal{C}_{\text{free}}$ [16]

In a 2D C-space for a robot with 2 joints θ_1 and θ_2, an example robot roadmap could look like Figure 1.1. Obstacles occupy some volume in this C-space, and everywhere there is not an obstacle is $\mathcal{C}_{\text{free}}$.

This method is *probabilistically-complete*, i.e. if the C-space was sampled infinitely, there would be infinite nodes in the roadmap covering the entire continuous C-space. As roadmap generation happens in the Pre-planning Phase, the number of samples is a choice on how many should be generated, and importantly on how much memory a computer has. This is because each robot configuration must be stored - for high-dimensional C-spaces this could easily be millions of configurations.

A key drawback of PRM however is that, **if the robot's environment changes, the expensive Pre-planning Phase must be recomputed** with the positions of the new obstacles. Classic PRM is also **only capable of considering static obstacles**, and generating a sequence of configurations to transit **without any consideration for time or joint velocities**.

Dynamic Roadmaps To address the shortcomings of PRM, Leven et al. (2016) [38] introduced an extension, *Dynamic Roadmaps (DRM)*. This allows for static obstacles to be changed in the workspace, and the precomputed roadmap reused - saving the need to recompute when the environment changes. To achieve this, the key contribution of this work is a unique C-space-to-workspace encoding. Rather than constructing a workspace-specific roadmap that hardcodes the obstacles at compute-time, a 'blank' workspace roadmap for an obstacle-free environment is precomputed.

Alongside this, the workspace of the robot is discretised into cells of fixed sizes (called **voxels**). A data structure maps each voxel to the set of robot configurations that occupy it, as well as the configurations that 'arc' through it. The latter refers to the fact that when a robot moves between one configuration and another, it can 'sweep' through voxels that aren't included in the start or goal configurations - thus it is important to store that, if a certain voxel is occupied during the Query Phase, configurations that starting in, ending in and *transiting* that voxel are no longer possible.

This mapping allows for voxels to be turned on/off, and the corresponding occupying configurations removed from $\mathcal{C}_{\text{free}}$. Workspaces can therefore be reconfigured before running the A* Query Phase - maintaining collision-free path planning without recomputation. It retains the probabilistic completeness of PRM.

While the paper does provide some compression of the encodings and occupation information, it still **requires significant memory to load and store the roadmap and related mapping**. This memory cost scales exponentially as more DOF are added, or as the workspace discretisation gets smaller. Consequently only small roadmaps are feasible, and the lack of density in these roadmaps (still randomly sampled) often causes DRM methods to have a **low planning success rate**. As with PRMs, there is **no support for dynamic obstacles**.

Hierarchical Dynamic Roadmaps Yang et al. (2018) [44] introduced the current roadmap-based state-of-the-art, *Hierarchical-DRM* (*HDRM*). This work extends DRM by developing a novel data structure for the C-space-to-workspace mapping that exploits single kinematic chain robots. By recognising that the joints in a robot arm are organised in series, HDRM adapts the DRM data structure to take advantage of inherent joint hierarchy, reducing the need to store every configuration explicitly. While further details are provided in Section 2.1, this greatly reduces the size of this data structure and allows for the efficient encoding of of 'millions of configurations' [44]. The paper also presents novel indexing algorithms that allow for efficient graph search in roadmaps with millions of nodes.

These advancements permit for finer workspace discretisations, and therefore the removal of the need to store 'swept' robot configurations as described in DRM. Finally, it provides a proof of resolution completeness - guaranteeing solvable motion plans (or the absence of one) at a given workspace voxel size. However, HDRM, like its predecessors, **only works with static obstacles**.

Dynamic Environments

All of the aforementioned works pertain to static environments. While some generalisation to dynamic environments is possible, in general disregarding the temporal aspect of robot motion planning prohibits common (but complex) motions such as reaching into moving shelves [45, 15]. Fortunately, there is also a body of work addressing this.

In Kindel et al. (2000) [36] a randomised motion planner is presented that procedurally generates small PRMs at each planning query, resulting in sub-graphs of valid motion that can be explored. These graphs are forward-directed and respect the fact that time can only move forwards (referred to as *time monotoncity*).

However, as this approach **still uses PRMs**, all of the corresponding pitfalls apply. Also, as the **algorithm is not end-to-end**, the short horizon of each query means that the method is not able to answer whether a path exists at the start query time.

In Yang et al. (2019) [45], a sampling-based method exploiting *Rapidly-exploring Random Trees (RRT)* was introduced. RRT is an algorithm that randomly samples a high-dimensional space (the robot's C-space) and builds a space-filling tree representing paths through this, see Figure A.3.

While previous methods only discretised the configuration-space (i.e. the spatial dimensions), this work extends to **time-configuration** space (introduced also in [32, 28]) - solving motions in dynamic environments by building and connecting trees in the spatial and time dimensions from start-to-goal and vice-versa. The time dimension has greater constraints than configuration dimensions, such as being monotonic. It also imposes constraints on traversable configurations, as, by adding time, transiting robot states causes velocities at the joints that must be within the limits of the robot. These constraints are respected within this work.

Though the time-configuration representation is ideal for the motion planning problem this thesis addresses, as RRT is a sampling algorithm it is **only probabilistically complete**, rather than the much stronger resolution completeness of a static method like HDRM. Unlike roadmaps too, the **Pre-planning and Query Phase are not separated in this method**, and thus the benefits of an efficient and fast A* search cannot be harnessed.

1.4 Contributions

Revisiting the specification at the start of this chapter gives 3 criteria for an (ideally performant) new method of motion planning: 1) free of collision, 2) joint limit respecting, and 3) complete.

The Missing Link? While time-configuration RRTs [45] clearly get closest to this specification, these methods do not feature the strong resolution completeness of HDRM [44]. For robot motion in safety-critical environments, the guarantee of safe motion (or the inability to perform one) at a given workspace resolution is ideal, as current Computer Vision technology can usually detect obstacles at a certain resolution (i.e. ones no smaller than $5cm^3$) [9].To meet the specification of the problem, and combine the benefits of time-configuration RRTs with those of HDRM, this thesis introduces **Time-indexed HDRM**. The contributions, explained in Chapter 2, are summarised as follows:

- Algorithmic design of 3 versions of a naive way of adding time to HDRM (referred to collectively as *Post-Processed-Time HDRM (PPT-HDRM)*)
- Implementation and analysis of all PPT-HDRM methods
- The algorithmic and conceptual design of fully-fledged Ti-HDRM
- An implementation of full Ti-HDRM in C++ with Python bindings
- Proof of resolution-completeness in time-based extensions of HDRM
- Testing and analysis of Ti-HDRM, with experimentation on a simulated Nextage robot

Chapter 2

Bringing the Temporal Dimension to HDRM

2.1 HDRM In-Depth: Motion in Static Environments

To add time to HDRM, the existing algorithm must first be understood. As a method derived from PRMs (Section 1.3), HDRM can be broken down into 2 steps: 1) Pre-planning Phase 2) Query Phase. In the offline, computationally-expensive Pre-planning, the hierarchical dynamic roadmap is generated - creating a mapping between workspace volumes and C-space volumes. In the online, performant Query, the corresponding invalidations are made for obstacles and the roadmap searched to find a motion path.

2.1.1 Pre-planning

HDRM discretises both the workspace and C-space of a robot. The former causes the continuous environment of the robot to be broken into voxels at resolution s. The latter causes each of the continuous joint limit ranges of the robot to be evenly split across K discrete values. As there are n joints, joint n has evenly distributed values of K_n.

Each configuration of a robot is therefore a unique combination of the discrete values of its joints. For example, a 2 DOF robot with 2 joints and joint limits, $q_1 = \{-\pi, \pi\}$ and $q_2 = \{0, 0.3\}$, with discretisation $K = \{3, 4\}$, would have 3 positions for joint q_1:

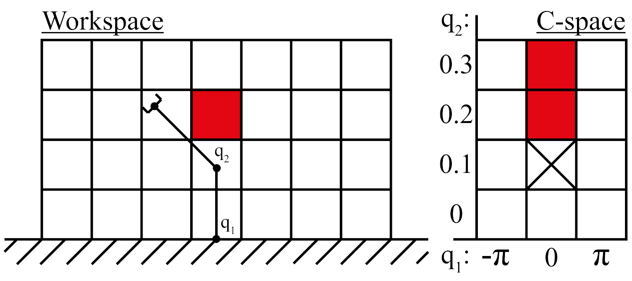


Figure 2.1: Workspace and C-space of 2 DOF robot. $\times$: current robot configuration, red: occupied voxel and corresponding invalid configurations

$\{-\pi, 0, \pi\}$, and 4 positions for joint q_2: $\{0, 0.1, 0.2, 0.3\}$. Combinatorially, there are $K_1 \cdot K_2 = 3 \cdot 4 = 12$ total configurations, seen in Figure 2.1.

For each joint, there is consequently a discrete set of **locations** that it can be set at, from the first position all the way up until the highest: $k_n \in \{1, \dots, K_n\}$. From this, it is then defined that $\mathbf{k}(n)$ is an n-dimensional vector representing the location indices of the joints up until n, see Equation 2.1. This maps onto an actual configuration with Equation 2.2:

$$\mathbf{k}(n) = [k_1, \dots, k_n] \tag{2.1}$$

$$\mathbf{q}(\mathbf{k}(n)) = [q_1(k_1), \dots, q_n(k_n)] \tag{2.2}$$

For example, for the 2 DOF robot: $\mathbf{k}(2) = [3, 2]$ corresponds to an actual robot configuration of $\mathbf{q} = [\pi, 0.3]$. Just specifying $\mathbf{k}(1) = [1]$ instructs the first joint value $q_1 = -\pi$, and corresponds to the set of all possible robot configurations with this fixed first joint.

Representing all of the robot configurations, these distinct arrays of joint locations ($\mathbf{k}(n)$) are each a node in the roadmap for the later Query Phase. Rather than a long array being used as each node, each array of joint locations is converted into a unique index integer, called a **sample**. 2 algorithms are defined [44] to convert from sample $\rightarrow$locations and from locations $\rightarrow$sample - included in Appendix B.1. This structure and indexing allows for the efficient storing and search of all discrete configurations. The final piece to understand in the Pre-planning Phase of HDRM are the **hierarchical occupation lists**.

Safe motion planning in HDRM is feasible by invalidating ('turning off') voxels in the robot's workspace that are occupied by an obstacle. This then invalidates the robot configurations that coincide with that voxel. To achieve this, an occupation list is created - each voxel stores a list of configurations to be removed from the roadmap if the voxel is occupied.

Rather than storing a long list of unqiuely identifying sample numbers for each configuration, HDRM recognises the joint hierarchy present in single kinematic chain robots. If a lower robot link is in collision with a voxel, this will automatically invalidate all higher-up joint permutations. For example, consider a collision of the 2 DOF robot's first link. For this lower q_1 joint configuration there are many possible q_2 configurations - but these will *all* be impossible as the lower robot body is already in collision (see Appendix A.4). Using the hierarchical data structure, HDRM can express that the lower joint configuration is not possible and therefore implicitly that all configuration permutations higher than this are also not possible - massively compressing the number of samples that need to be stored in each voxel's occupation list.

Specifically, at each voxel HDRM stores a pair of indices for each configuration that coincides with it. This pair, (n, i), where n is the robot joint level, and $i \in \prod_1^n K_n$ describes the joint locations using Algorithm 1 from B.1, i.e. the joint locations (as in 2.2) can be retrieved with: $\mathbf{k}(n) = \mathcal{H}(n, i)$.

When all of these have been collected into each voxel, Algorithm 3 (Appendix B.2)[44] sorts through and completes a 'promotion' process - only storing a lower level pair of indices if all of the higher levels are in collision. For example, consider a robot that has value k_1 for its first joint, but differs at all other joints. Suppose the first link of the robot coincides with a voxel that is in collision, due to the k_1 positioning. For a 6 DOF robot, if the subsequent joints $k_2, \ldots, k_6$ all were discretised evenly by 20 between each joint limit (i.e. $K_n = 20$ for all n), there would be $20^5 = 3.2$ million joint locations with the same k_1. Rather than storing all of these at this voxel, HDRM stores a single pair $(1, k_1)$ to represent this collision information [44].

2.1.2 Query

After offline Pre-planning, each voxel has a hierarchical occupation list, forming the blank roadmap. The motion planner is ready to plan between a start pose, q_{start}, and goal pose, q_{goal}, with the following steps:

Collision Update Static obstacles in the voxelised environment of the robot are detected [1], and the corresponding voxels (each of which has a unique **index** number) are marked to be invalidated. The occupation list for each voxel is parsed, and each now-invalid configuration is removed as a node from the roadmap.

Connecting q_{start} and q_{goal} to Roadmap Due to the discretised joints, it is unlikely that q_{start} and q_{goal} align exactly with one of the discrete configurations in the roadmap. Therefore it is necessary to connect the start and goal with the graph. This is achieved analytically by rounding q_{start} and q_{goal} up and down to their nearest discretised neighbours for each joint, and the closest neighbours for each joint picked as the connection node. Note that if any of these neighbours are invalid, then the start and/or goal position is not feasible and the motion will fail.

Returning a Trajectory with A* Search The remaining step is to search through the roadmap to find the shortest path between q_{start} and q_{goal} amongst the valid nodes. The optimal and efficient A* search algorithm is used to accomplish this.

A* search [33], is a best-first search algorithm, meaning that it explores a graph in a direction weighted by a heuristic. It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until finished [1]. At each iteration in its graph traversal, A* seeks to minimise: $f(n) = g(n) + h(n)$. Here, n is the next node, $g(n)$ is the cost of the path from the start to n, and $h(n)$ is the heuristic function that estimates the cheapest path from n to the goal. In this context, the heuristic is simply the Euclidean distance [27] to the goal, i.e. the cost 'left to go'. After all iterations, the path will be identified as the one minimising the cost from the start, and the cost to the goal - therefore the shortest path. The A* algorithm pseudocode is explored in Section 2.4.2.

After the search, the returned list of sample numbers are converted to configurations representing the path through the C-space that the robot should travel to achieve collision-free motion between q_{start} and q_{goal}. Alternatively, if a path through the samples between the start and goal is not possible, then failure (at this resolution) is returned.

With HDRM explained, it is apparent that there is no existing concept of time within the algorithm - it provides purely spatial planning. Consequently there is no support for

[1]Detection of obstacles is a problem in the field of Computer Vision, and not the focus of this thesis. See Experiments in Chapter 3 for more details on potential inputs to HDRM's invalidation stage

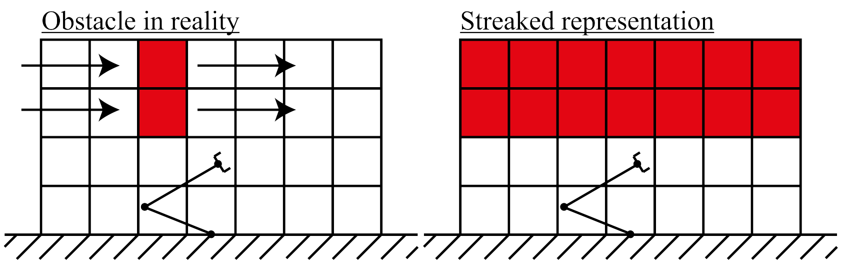


Figure 2.2: L: Obstacle traversing workspace, R: How it appears to Streaked HDRM

solving motion around obstacles that *change their positions through time*, or, equally if a robot changes *its* position with respect to a static environment.

The key question therefore is: how can HDRM be extended to work with changing and dynamic environments? Most naturally, as the later sections of this report explore, this is achieved by explicitly incorporating time into a new HDRM-based algorithm. However, classic HDRM can be extended to work with moving obstacles in a fundamental and simple way - by ignoring time completely.

2.2 Ignoring Time: Streaked HDRM

When an object moves through a discretised workspace, at a given point in time it occupies a certain number and configuration of voxels - essentially forming a rasterised version of the actual object. As time increases, the motion of this object changes the configuration of voxels that are occupied, and, if the object leaves the workspace, the number of voxels occupied too.

Classic HDRM *could* be used to plan safe motion in an environment where the movement of these obstacles is known. The way of achieving this is to 'streak' the obstacles across the workspace, and invalidate every voxel that the obstacle will occupy across its entire movement, as in Figure 2.2:

This representation can be thought of as the most conservative way of robot motion planning around moving obstacles - a robot would avoid everywhere the obstacle has, or will, be. The benefits of this proposed method are that it can be achieved with no modifications to existing HDRM, and the computational complexity will remain the same - the only extra work being that more voxels are invalidated.

However, the clear downside to this **Streaked HDRM** is that it fails to capture the intricacy of movement. Streaking objects across the workspace causes many perfectly feasible motions to no longer be possible - for example, reaching to a point after passing a wall. In this case, while the wall is only transiently in the workspace of the robot, to ensure collision-free motion across its movement, it must take up an entire 3D void. In addition to this, with no formal treatment of time, it is not clear when to remove invalidated voxels - even if the object that invalidated them is long in the past.

While there are use cases where Streaked HDRM is sufficient for the problem (e.g. if the motion of an obstacle can only be determined at one instance), as the Experiments in Chapter 3 show, it is unsatisfactory for motion planning in dynamic environments.

2.3 A Naive Approach to Time: PPT-HDRM

Knowing the clear limitations of ignoring the time completely, how can HDRM be extended to incorporate the notion of temporality?

Whenever the HDRM Query Phase is run, the environment's static obstacles invalidate the voxels that they occupy, and the motion plan query is fulfilled. This thesis now proposes a simple approach to dealing with moving obstacles by **re-running HDRM at any point the environment changes**.

While algorithmically nothing stops the environment instantaneously changing, the nature of motion of objects is continuous - meaning that they move with some velocity, v_{obs}, within the workspace of the robot. If the environment is therefore constantly changing, re-running HDRM strictly 'at any point the environment changes' would result in an algorithm that was infinitely re-run, and therefore practically never commenced.

A way of overcoming this is by introducing **time discretisation**. Just as the continuous spatial dimensions (workspace and C-space) are discretised in HDRM, continuous time can be approximated and discretised as intervals of length dt, for example at $dt = 1$ second or $dt = 0.5$ second intervals.

By approximating time in this way, HDRM's Query Phase can be re-run at the start of each time interval. Originally, motion can be planned between the start and goal, and partially executed until the end of the interval. When this interval expires, the current invalidations are cleared and new invalidations filled out. In the next interval, motion is replanned from a new intermediate start state to the original goal. This can be repeated until the robot reaches q_{goal} at time t_{goal}, or, if it doesn't, then reporting failure.

Post-Processing Time

HDRM is more strictly a *path planner* rather than a *motion planner*, as it provides no instruction on timing or the speed of the outputted trajectory. Consequently, when extending to dynamic environments by re-running an HDRM query at each time interval, the timing of each partially executed plan must be added *after* computation - hence the need for so-called **post-processed time** (**PPT**). From the robot's perspective this can be thought of as: 'how many steps in the current motion plan should I execute before the next time interval, where I recompute the motion plan from scratch?'

There are multiple ways of fitting PPT to a HDRM trajectory. Over the course of this work, 2 methods, named **Linear PPT-HDRM** and **Greedy PPT-HDRM**, have been developed, implemented and tested. Each of these PPT-HDRM algorithms do not need to modify the Pre-planning Phase from the original HDRM paper, and instead only extend the Query Phase. This is natural, as in Pre-planning, we create a map from the workspace to configuration-space of the robot. This relationship is purely spatial and therefore invariant to time. As a result, in the operation of the algorithms in the next sections it is assumed that the hierarchical occupation lists have already been generated by the same methods as classic HDRM.

2.3.1 Linear PPT-HDRM

In this version, after the generation of each motion plan, the outputted plan is considered to be l individual steps spaced evenly, with each step corresponding to a configuration to be transited. For example, if the trajectory at the time of the first query is 20 configurations in length ($l = 20$), and there are 10 seconds left to complete the motion, each step will be given a time budget of $10/20 = 0.5$ seconds to execute. The number of steps available to be executed depends on the value of dt, the interval time. In the case $dt = 1$, 2 whole steps would be feasible in the first interval before replanning.

In the next interval, the time budget would shrink by a dt and now be $10-(1 \cdot dt) = 9$ seconds. If the next motion plan also had $l = 20$, $20/9 \approx 2.22$ steps along this trajectory would be taken (2 whole steps, and 22% of the 3rd step). This repeats until the time horizon reduces to 0, with the motion succeeding if the robot is at q_{goal} at this time, and failing if not. Refer to Algorithm 1 for full pseudocode.

2.3.2 Greedy PPT-HDRM

According to [7], a *greedy* algorithm is one that makes locally optimal choices at each stage it is run. In this context, a locally optimal motion planning choice would be to execute as much of an HDRM query as is feasible in the current time interval before the environment changes.

Therefore, in this algorithm, after the generation of a motion plan at the start of a time slice, as much of the outputted trajectory is executed as is possible within the time interval. This partial execution is a function of the joint velocity limits - the amount of feasible motion is determined by how far each joint can travel in a time slice. In order for joint velocity limits to not be exceeded, the percentage of the partial execution is limited by the most restrictive joint. For example, in a robot with no joint velocity limits (i.e. its velocity limits are all ∞), every step in the trajectory would be feasible within a single time interval, and the robot would rush to q_{goal} and wait there for t_{goal}.

Similarly, if the 2 DOF robot from Figure 2.1 had realistic joint velocity limits of π rad/second for both joints q_1, q_2, a motion plan that moved both joints by $\pi/4$ at each step would correspond to 4 steps of the trajectory being executed at each time step, with $dt = 1$ second. For another example, if joint q_1 had a velocity limit of π and joint q_2 had a velocity limit of $\pi/2$, a motion plan that tried to move both joints by $\pi/4$ at each step would now correspond to only 2 steps of the trajectory being executed at each time step, for the same $dt = 1$ - the partial execution has been limited by a restrictive joint.

Unlike in the Linear method, Greedy PPT-HDRM *does* take into account and respect the robot's joint limits - meaning that it meets an important part of the specification. The Greedy PPT-HDRM algorithm shares the same structure as the Linear PPT-HDRM, with the only difference being how many steps of each partial trajectory are executed. As such, the **if** statement on line 16 is modified in partial Algorithm 2:

Algorithm 1: Linear PPT-HDRM Query

```
Input: $duration, dt, t_{\text{start}}, t_{\text{goal}}$ :: floats; $q_{\text{start}}, q_{\text{goal}}$ :: arrays of length $n$
$O_v$ :: hierarchical occupation lists for each voxel
Output: SUCCESS or FAIL
1  $time\_slices \leftarrow$ floor$(duration/dt)$;
2  $t_{start} \leftarrow$ floor$(t_{\text{start}}/dt)$;
3  $t_{goal} \leftarrow$ floor$(t_{\text{goal}}/dt)$;
4  foreach $t$ in $t_{start} : t_{goal}$ do
5      resetRoadmap();
6      $voxels \leftarrow$ getOccupiedVoxels(t);
7      foreach $voxel$ in $voxels$ do
8          $configs\_in\_vox \leftarrow O_v(voxel)$;
9          removeConfigurationsFromRoadmap$(configs\_in\_vox)$;
10     end
11     if $q_{start}$'s neighbours are not valid in the roadmap then
12         end Algorithm: FAIL;
13     end
14     connectStartGoalToRoadmap();
15     $traj \leftarrow$ doAStarSearch$(q_{start}, q_{goal})$;
16     if $traj$ is solved then
17         $time\_per\_step \leftarrow$ length$(traj)/(t_{goal} - t)$;
18         $num\_steps \leftarrow time\_per\_step/dt$;
19         executePartialTrajectory$(traj, num\_steps)$;
20         $q_{start} \leftarrow q_{current}$;
21     end
22 end
23 if $q_{start} == q_{goal}$ then
24     end Algorithm: SUCCESS;
25 else
26     end Algorithm: FAIL;
27 end
```

Algorithm 2: Greedy PPT-HDRM Partial Trajectory Diff

```
Input: Same as Linear PPT-HDRM, plus: $joint\_vel\_limits$ :: array of length n
16 if $traj$ is solved then
17     $potential\_steps \leftarrow [...]$;
18     foreach $n$ in $joints$ do
19         $max\_movement \leftarrow joint\_vel\_lim[n] \cdot dt$;
20         $potential\_steps$.append(stepsThroughTraj$(traj, max\_movement)$);
21     end
22     $num\_steps \leftarrow$ min$(potential\_steps)$;
23     executePartialTrajectory$(traj, num\_steps)$;
24     $q_{start} \leftarrow q_{current}$;
25 end
```

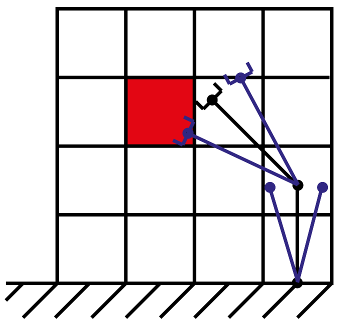


Figure 2.3: The robot configuration is in-between discretised joint locations - all neighbours (blue) must be checked, therefore this is an invalid intermediate start state

2.3.3 Pitfalls of PPT

While these PPT-HDRM methods do somewhat succeed in extending static HDRM to the temporal dimension, there are several pitfalls that can lead to a motion plan failing when another method may discover a solution:

- **Invalid intermediate start states** - for both PPT-HDRM versions, after each time slice and partial trajectory has been executed, the start pose of the robot, q_{start}, is updated to the robot's current position, i.e. how far it got in executing the partial trajectory in the previous interval. As the roadmap has changed, the new intermediate start state's validity must be checked before planning. As a pose in the continuous C-space of the robot, this work has discovered that checking it's validity is **not as simple as finding the nearest valid neighbouring configuration** that is in the roadmap.

  Instead, as there is no guarantee of 'how close' a configuration is to one of its discretised neighbour joint location array, *all* of these neighbours must exist and be valid. This means that, for each joint n, the neighbours plus/minus on either side of n must be valid (checked on line 11 of Algorithm 1, see Figure 2.3). Consequently, some PPT-HDRM motion plans will result in an invalid intermediate start state, causing an entire motion failure. This is a direct result of the stop-start nature of the method. If relaxed, the algorithm would lose its completeness and potentially risk the robot transiting an invalidated configuration.

- **Lack of planning horizon in time** - for some motions, such as reaching into a shelf, it is important for the robot to achieve q_{goal} specifically at t_{goal}. Using the PPT approach is not conducive to this, as the algorithm attempts to get to the goal directly. This can be viewed equivalently as PPT-HDRM methods being *instantaneous planners* - meaning that they plans motion at each instant with no future understanding of how the environment will change. There could be situations where PPT-HDRM manoeuvres a robot to a configuration, not realising that in the next time slice there is an imminent risk of collision with an obstacle and then being unable to move out of the way in time. Similarly, a robot may get trapped behind an obstacle mid-way through the motion, or the goal state may be temporarily occluded causing HDRM Queries to fail and no progress made.

- **Illegal/undesirable velocities** - for Linear PPT-HDRM, as time is evenly chunked and distributed, the joint velocity limits of the robot are not taken into account. Therefore, there is no guarantee that motion between configurations are within the joint velocity limits. For Greedy PPT-HDRM, joint velocity limits are taken into consideration and respected, however the joint velocity in-between steps is essentially fixed to a high value. Consequently, the robot will not take slower routes, even if they are globally more optimal (a similar issue is present in optimisation-based motion planning methods discussed in Chapter 1). This lack of flexibility is undesirable as some motions may be better executed with slower transitions. This 'rushing' also exacerbates the lack of planning horizon in time.

While there are potentially other ways to fit time to a trajectory in the post-processing step (such as tuning an optimisation), some of which may reduce these issues, it is clear that not all problems will be solved - the largest persistent issue being the invalid intermediate start states. Due to this, while PPT-based algorithms have 'resolution completeness' at every instant they are run, they do not qualify for full resolution completeness across entire motion plans - meaning that potentially feasible motions may result in failures, and vice versa. As evidenced in the Experiments in Chapter 3, these naive approaches are unaware of time as a special dimension, and suffer accordingly. In order to achieve the motion planning specification set out in Chapter 1, a better method is needed.

2.4 Incorporating Temporality: Time-indexed HDRM

The fundamental limitations of naive PPT-HDRM are because, at each time interval, the motion planner is only able to form a path through the C-space of the robot. In reality, a dynamic environment has the additional special dimension of time - something that can be represented by so-called **time-configuration space**. By extending HDRM's awareness to this dimension, this work now demonstrates that a fully resolution-complete (both spatially and in time) motion planning algorithm that meets the specification requirements is possible.

2.4.1 The Requirements of Time-Configuration Space

As explained in Section 1.1, the C-space of a robot is an n-dimensional representation of all possible configurations of a robot, where n is the number of robot joints. Section 1.3 extended this by introducing the concept of time-configuration space when discussing Yang et al [45]. First proposed by Fraichard (1993) [32], the idea of time-configuration space is to extend the n-dimensional C-space representation to $(n+1)$-dimensions. In the case of this work, this extra dimension can be used to indicate *which* robot configurations are valid and *when*. To be coupled with this, it is also necessary for the voxelised workspace to have an additional time dimension alongside the 3 existing x, y, z used to identify the coordinates of each voxel. This dimension will represent *when* a voxel is occupied. An example of an obstacle moving through the workspace and time-configuration space of a 1 DOF robot is seen in Figure 2.4.

Just as in PPT-HDRM, discretising this temporal dimension is necessary. Each time

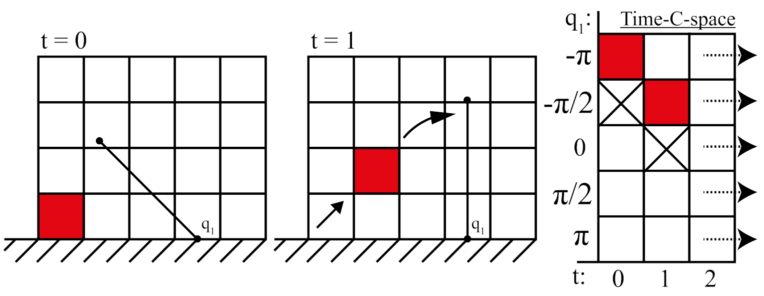


Figure 2.4: Workspaces and corresponding time-configuration space of 1 DOF robot. Time slices 0 and 1 shown.

'slice' in Figure 2.4 represents the workspace state at that time index, and the valid configurations. It can be visualised that each of these individual roadmaps are connected with one another, and that a motion plan can transit configurations and time.

In the configuration dimensions, a motion plan can explore each node without restriction. It is permitted to go backwards and forwards between different configurations, and this is something that the A* search freely does. However, the introduction of time necessitates careful treatment in 2 aspects:

- **Monotonically Increasing** - as per [14], monotoncity refers to the property of a function being either strictly increasing or decreasing. Time is an example of the former - while it is possible for a robot to visit a time-configuration state directly in its future, it is not possible for it to visit time-configuration states that have elapsed. Equally, making jumps forward in time is not possible, i.e. a robot could not go from a state in the roadmap at time index 0 to time index 2, without transiting a state in time index 1.
- **Illegal Velocities** - when a robot travels between 2 time-configurations, its joints can move. In order to achieve the goal state by the start of the following time-index, each joint of the robot will incur a velocity. As $velocity = \frac{distance}{time}$, these velocities are a factor of both the distance that the joint has to traverse and the time interval of this traversal.

When a potential time-indexed motion planner is queried, the start and goal states, q_{start}, q_{goal}, alongside the start and goal times, t_{start}, t_{goal} are provided. Together, these represent 2 unique nodes in the time-configuration roadmap - it is the job of the Ti-HDRM algorithm to forge a path between these.

2.4.2 Conceptual Design of Ti-HDRM

This thesis now describes **Time-indexed Hierarchical Dynamic Roadmaps** (shortened to **Ti-HDRM**), an algorithm that has been designed to achieve motion planning in dynamic environments, by incorporating the temporal dimension and extending HDRM, rather than approximating it or ignoring it as in the Streaked/PPT versions discussed previously. A proof of its resolution-completeness is presented in Section 2.4.3.

Pre-planning and Query - where to add time?

When introducing PPT methods, it was explained that it was only necessary to extend the Query Phase of HDRM, rather than the Pre-planning. This is because Pre-planning involves generating the hierarchical occupation lists for each voxel of the workspace - in other words, the map between the physical workspace and the C-space of the robot.

When considering implementing a temporal dimension, in theory this *could* be added to the Pre-planning Phase. Rather than a mapping between space and configurations, the occupational lists would become a mapping between time-space and configurations. Each voxel would be replicated some number of times to represent different time slices. However, by 'baking in' the time dimension to the Pre-planning, the exact number of time slices, t_{num}, to be stored must be chosen before pre-computation and all stored in memory. It is instead desirable for t_{num} to be specified at Query time. This is because queries could widely vary in the time horizon that they want to plan over. For example, in one problem a user might decide that a coarse discretisation of time with $dt = 1$s over a $duration = 10$s will suffice, necessitating $t_{num} = duration/dt = 10/1 = 10$ time slices. But in another problem $dt = 0.05$s may be necessary for the same $duration$, leading to $t_{num} = 10/0.05 = 200$ time slices.

The time dimension of Ti-HDRM therefore does not extend the hierarchical structure used to encode joint configurations, and exists only at the point of Query. Consequently, Ti-HDRM retains the exact same Pre-planning Phase as classic HDRM, and extends the Query Phase only.

Samples and Indexes

By adding a time dimension, it is necessary to store more state about both the voxels, and the corresponding robot configurations that occupy them. In an implementation of classic HDRM, each voxel has a unique **index** number between $\{0, ..., M-1\}$, where M is the total number of voxels. Distinctly, each configuration of the robot has a unique **sample** number between $\{0, ..., i \in \prod_1^n K_n\}$. While both of these numbers are integers, they are not to be confused and represent the workspace and C-space accordingly. The occupation lists of the algorithm map from index→sample. For example: $\{20 : \{32, 24, 35, 78\}\}$ would correspond to the information that configurations 32, 24 and 35 occupy voxel 20.

Indexing Algorithms & Representing State over Time

Formally incorporating time into the Query Phase of the algorithm requires that a time-indexed data structure be maintained. This data structure stores a record of the configurations that are invalidated at the different time slices by any moving obstacles in the workspace. While the exact data structure that is used is an implementation choice (more details in Section 2.4.4), to enable this, a further integer (alongside sample/index from above) must be introduced: **ti** (for *time-index*).

This time-index augments the classical index numbers from the workspace and the sample numbers from the C-space. By specifying a ti-index, a voxel is uniquely identified in 3D space and time. Similarly, by specifying a ti-sample, a configuration is uniquely identified in C-space and time.

As the underlying standard index and sample numbers will be used and inherited from classic HDRM to use the Pre-planning Phase's occupation lists, it is necessary to create algorithms to convert between ti-sample $\leftrightarrow$ sample, as well as ti-index $\leftrightarrow$ index. These depend on how the indexing is implemented, and, to remain agnostic to this, the ones this work has developed are described in Section 2.4.4.

Creating the Time-Indexed Roadmap

With the 3 integers (index, sample, ti), the roadmap can now be prepared for searching by performing the workspace voxel invalidations for the specific motion problem. In Ti-HDRM, invalidating voxels for a certain time slice will mark the robot configurations that occupy them for that time slice as invalid, but not affect the same configurations in other time slices.

Specifically, by combining ti and sample into a ti-sample (this could be as simple as a tuple data structure of $\{ti, sample\}$), a unique configuration in time is referenced. Similarly, by combining ti and index into a ti-index, a unique voxel in time is referenced. This encompasses the new augmentation with the additional ti integer. The user can specify a list of voxels at certain times to be invalidated as a list of ti-indexes. In turn, this marks a list of ti-samples as invalid. This is achieved with Algorithm 3.

Algorithm 3: Time-indexed Invalidations to Setup Roadmap

```
Input:
$duration, dt, t_{\text{start}}, t_{\text{goal}}$ :: floats
$q_{\text{start}}, q_{\text{goal}}$ :: arrays of length $n$
$O_v$ :: hierarchical occupation lists for each voxel
$obs$ :: list of obstacles and their geometries
$pos\_obs$ :: list of lists of obstacle positions in x,y,z space
$len(obs) == len(pos\_obs)$
1  $time\_slices \leftarrow$ floor$(duration/dt)$;
2  foreach $ob$ in $obs$ do
3      foreach $ti$ in $time\_slices$ do
4          $ti\_index\_obs \leftarrow$ to_ti_index$(pos\_obs[ti])$;
5          foreach $ti\_index$ in $ti\_index\_obs$ do
6              $ti\_samples \leftarrow O_v(ti\_index)$;
7              invalidate_ti_samples$(ti\_samples)$;
8          end
9      end
10 end
```

Querying the Time-Indexed Roadmap

Users can now query the time-indexed roadmap to solve motion plans between ti-samples. Algorithmically, this is achieved by searching the graph with A* [33]. Because of the nature of representing each of the robot configurations as a single integer ti-sample, these form the vertices of the graph to search. What requires careful consideration is determining the neighbourhood of each of these ti-sample vertices - these are the edges of the graph. The A* search algorithm pseudocode can be found at Appendix B.3

A* operates by using a priority queue [17] (**frontier**) to perform a repeated selection of minimum estimate cost nodes to explore. At each iteration, the node with the lowest $f(x)$ value (see Section 2.1.2) is popped from the queue, and the f and g values of its neighbours are updated and added to the frontier to be explored. The algorithm terminates when it removes the goal node from the frontier. The f value of the goal node is the cost of the shortest path. Every time a node is popped from the frontier, it is added to a **closed set** that keeps track of what nodes have been visited (and their current costs), and can then be used to retrace and return the shortest path.

For Ti-HDRM these nodes are each ti-samples, and $h(x)$ is the Euclidean distance in n-dimensional joint space between ti-sample x and the goal ti-sample. In classic HDRM, the **potential** neighbours of each sample are the $2 \cdot n$ joint configurations on either 'side' of each joint of the sample (each of these is also an integer sample). The **actual** neighbours are those that have not been removed during invalidation. At every iteration of A*, the `getNeighbours` function is run to determine the current sample's valid neighbouring configurations.

For Ti-HDRM, the neighbourhood of a ti-sample is significantly more complicated. Being a 'neighbour' of a ti-sample corresponds to the statement 'the robot can travel from this configuration to this neighbouring configuration'. Clearly, by adding the temporal dimension, the algorithm must be careful to not suggest neighbours for a ti-sample, $t_i q_a$, that are:

- Are in a previous time slice ($< t_i$), or in a time slice in the future that is greater than the succeeding time slice ($> t_{(i+1)}$)
- Suggest a neighbouring ti-sample $t_{(i+1)} q_b$ that will incur an illegal velocity when moving from $t_i q_a$

Searching the Neighbourhood

To only explore neighbours that are possible, and to ensure the temporal resolution completeness proved in Section 2.4.3, the `getNeighbours` function in Algorithm 4 has been designed. The inputs to this function are the ti-sample being explored, as well as the A* search's closed set, `close`. `close` is a hashmap, $\{ti\_sample \rightarrow \{parent\_ti\_sample, cost\}\}$, representing both the previous ti-sample in the explored path, as well as the cost of travelling to it.

In summary, the neighbourhood of a ti-sample can be up to 3 other ti-samples for each joint of the robot, i.e. a 6 DOF robot could have up to a total of 18 neighbouring time-indexed configurations. **The resolution completeness of these options matches the proof established in Section 2.4.3**. These 3 ti-samples for each joint correspond to the following:

Algorithm 4: `getNeighbours` function of A* search

```
Input:
ti_sample :: integer
close :: dictionary of {integer → {integer, float}}
Output:
neighbours :: list of ti-samples
dt_streak ← countPrevImmobileSlices(ti_sample, close);
current_ti, sample ← tiSampleToSample(ti_sample);
neighbours ← [...];
foreach joint n of robot do
    next_ti_sample ← sampleToTiSample(sample, ti + 1);
    if next_ti_sample is valid then
        neighbours.append(next_ti_sample);
    end
    config_right ← joint_value[n, +1];
    config_left ← joint_value[n, −1];
    ti_sample_r ← configToTiSample(config_right, ti);
    ti_sample_l ← configToTiSample(config_left, ti);
    if ti_sample_r is not out of joint range then
        joint_totals ← calculateTotalJointMovementInTi(ti_sample_r, close);
        if ti_sample_r is valid and does not exceed joint_totals then
            j ← dt_streak;
            still_valid ← True;
            while j > 0 && still_valid do
                prev_ti_sample_r ← configToTiSample(config_right, ti − j);
                still_valid = is prev_ti_sample_r valid?;
                j − −;
            end
            if still_valid then
                neighbours.append(ti_sample_r);
            end
        end
    end
    if ti_sample_l is not out of joint range then
        joint_totals ← calculateTotalJointMovementInTi(ti_sample_l, close);
        if ti_sample_l is valid and does not exceed joint_totals then
            j ← dt_streak;
            still_valid ← True;
            while j > 0 && still_valid do
                prev_ti_sample_l ← configToTiSample(config_left, ti − j);
                still_valid = is prev_ti_sample_l valid?;
                j − −;
            end
            if still_valid then
                neighbours.append(ti_sample_l);
            end
        end
    end
end
```

1) Staying stationary, but transitioning to the next time slice Provided that the configuration in the next time slice is valid, it is always possible for the robot to stay stationary - effectively only moving forwards in time, with no movement in C-space. By limiting moving in time to only this option, the time-indexed roadmap remains forward directed and time monotonicity is respected.

Both the start ti-sample, $_{t_i}q_a$ and end ti-sample $_{t_{(i+1)}}q_a$ must be checked for their validity. In Algorithm 4, it is assumed that the ti-sample currently being queried has *already* been validated and verified as collision-free. This is because, if it hadn't, then it would not have been added to the previous parent ti-sample's neighbourhood and the A* algorithm's frontier to search. Because of this, and as the robot is not moving simultaneously in time and C-space (referred to as **moving along a curve in time-configuration space**) to another robot configuration, it is sufficient to only check the validity of the end point, $_{t_{(i+1)}}q_a$. The logic for this can be seen on lines 5-8.

In A*, each neighbour has a potential cost to visit that can be increased/decreased with the heuristic function h - this is usually the Euclidean distance to that node in the graph. For classic HDRM, this is C-space distance. For Ti-HDRM, the heuristic is a free choice and most simply can remain as C-space distance - meaning that transitioning to the next time slice has 0 cost - or can incorporate a penalty on time transitioning - potentially encouraging slower motion or even a desired velocity. h is a free parameter in the implementation of this work.

2) Moving to discretised joint configuration -1 of current, but staying in same time slice Provided that the joint is not on the boundary of the range of motion, with the right velocity and validity checks, it can transition to the configuration directly adjacent to its current within the same time slice. This can be thought of as moving 'left', or moving to the configuration that is -1 of the current.

Firstly, outside of verifying that this proposed configuration is not invalid, the algorithm must check whether there is a velocity budget in the current time slice to execute the transition between a proposed $_{t_i}q_a$ and $_{t_i}q_b$. It is possible (and in fact likely with a coarse dt) for the ti-sample's parent to be within the current time slice, and if this is the case it will have already moved the joint by some amount. This amount must be calculated and the proposed neighbour, when added to this total, must not exceed the velocity limit of the joint. For example, consider a joint with position range $\{-\pi, \pi\}$ discretised into 8 steps with a velocity limit of π rad/s. For $dt = 1$s, if the queried ti-sample had a chain of parents in the current time slice, each of which had moved the joint $\pi/4$, and the chain was of length 3 there would have been $3\pi/4$ rad covered - meaning a remaining velocity budget for 1 more movement of up to $\pi/4$. If the chain was of length 4, there would be no velocity budget - and the only neighbour of the ti-sample would be from point (1) above. This check is done on lines 13-15. The function call that achieves this is covered in the Section 2.4.4.

For finer time discretisations, with smaller dt, a motion between 2 of the discretised joint states may incur a large velocity that is not feasible in a single time slice. For example, consider the joint of the 1 DOF robot having a discretisation of $K_0 = 3$, with a range of motion between $\{-\pi, \pi\}$. Consider also a time slice of $dt = 0.05$s. Moving

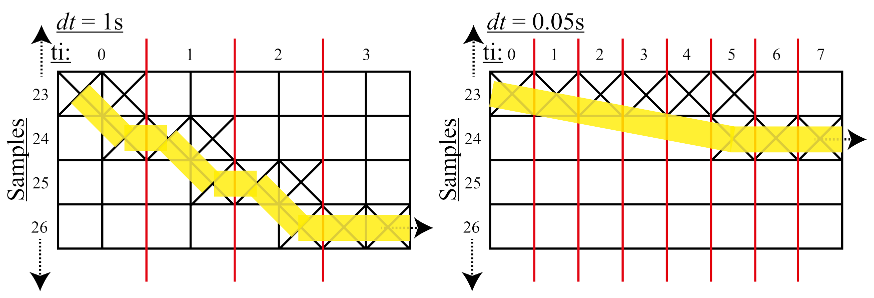


Figure 2.5: Exploring the time-configuration space at different dt. L: Coarser time discretisation R: Finer time discretisation with backtracking. $\times$: explored configuration, yellow: actual joint path

from joint location 0 ($-\pi$) to joint location 1 (0) would lead to a joint velocity of $\pi/0.05 = 62.8$ rad/s. Joint velocity limits of real robot motors typically lie in the region between 0 and 8 rad/s, and therefore this would not be feasible.

In order to achieve this motion, it must be stretched along multiple time slices, as in Figure 2.5. This increases the effective dt of the motion. To continue the example, 10 time slices of 0.05s each would lead to a 0.5s delta in time: $\pi/0.5 = 6.28$ rad/s, which would be feasible within typical joint velocity limits when executed over this stretch.

The algorithm therefore must check that there is such a time budget, dt_streak, that the movement can be executed over. This is calculated by looking back through the history of the parent ti-samples that have been used to get to the current ti-sample - if all of these are the same underlying configuration, only increasing in time, it shows that the robot has been 'immobile' for a number of time slices. By identifying this, instead of being stationary over this period, the joint can transition over this entire dt_streak from $t_{(i-dt\_streak)}q_a$ to t_iq_b. Identifying the dt_streak is also an implementation detail, see Section 2.4.4.

As part of the resolution completeness of Ti-HDRM, see Section 2.4.3, **every ti-sample either side of the curve in time-configuration space of the** dt_streak **must be checked.** This is explicitly checking that the motion can take place over this streak of time slices. If a single one of these is invalid, the motion is not safe to perform. This backtracking is performed on lines 17-22 with a while loop that drops out as soon as an unsafe ti-sample is identified. If all are safe, the neighbour is added.

3) Moving to discretised joint configuration +1 of current, but staying in same time slice By the same logic as (2), the neighbouring ti-sample to the 'right'/+1 is considered on lines 28-42 of the algorithm.

Continuous Path Reconstruction

With this, the A* search can swiftly navigate the time-configuration space, only considering travelling through robot poses that are collision-free and velocity-safe. When an A* search terminates, the `close` set must be re-traversed to reconstruct the shortest path, and then return this list of time-indexed configurations for the robot to travel through. See Algorithm B.4 for the general `reconstructPath` subroutine of an A* search.

While the `close` set will contain a path of ti-samples that can be performed on the robot, by the nature of Algorithm 4, for small *dt* these ti-samples will typically feature large 'jumps' where a joint appears to remain in the same position over a number of time slices, and then suddenly shifts up/down. This is an artefact of `getNeighbours` and the A* search only outputting the feasible time-indexed configurations, without interpolating over each dt_streak. Another hashmap can be used to track of each of these ($\{ti - sample \rightarrow dt\_streak\}$), and an interpolation algorithm, Algorithm 5, has been designed that returns the final output of Ti-HDRM.

Algorithm 5: `reconstructPathInterpolation` function of A* search

```
Input:
$dt\_streaks$ :: hashmap of $\{integer \rightarrow integer\}$
$close$ :: hashmap of $\{integer \rightarrow \{integer, float\}\}$
Output:
$traj$ :: list of ti-configurations
1  $reconstructed\_path \leftarrow$ reconstructPath($close$);
2  $traj \leftarrow [...]$;
3  $prev\_ti\_config = reconstructed\_path[0]$;
4  foreach ti_config in reconstructed_path[1:] do
5      $steps \leftarrow dt\_streaks[ti\_config]$;
6      if steps > 0 then
7          foreach j in steps do
8              $step =$ interpolateTiConfigs($prev\_ti\_config, ti\_config, j$);
9              $traj$.append($step$);
10             $prev\_ti\_config = step$;
11         end
12     else
13         $traj$.append($ti\_config$);
14     end
15 end
```

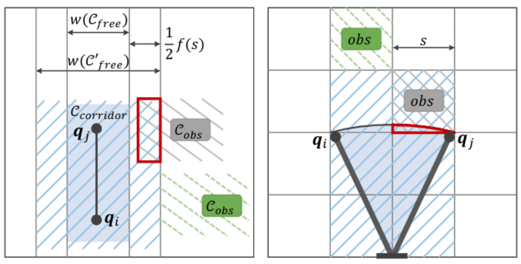
2.4.3 Formal Resolution Completeness of Time Indexing

This work now establishes a proof of resolution completeness for general time-indexed extensions of HDRM, of which Ti-HDRM satisfies.

Existing Spatial Resolution Completeness in HDRM

For HDRM, Yang et al. [44] provide a proof of resolution completeness of deterministic roadmaps *with discretised workspaces*. This builds upon LaValle et al.'s (2004) [37] proof that deterministic roadmaps are resolution complete. *Deterministic* here refers to a C-space is one that has been uniformly sampled.

To understand the HDRM proof of resolution completeness, a function $\Phi(x)$ for $x \in (0, \infty)$ is introduced. This is used to represent the set of all free C-space with the *width* of this free C-space: $w(\mathcal{C}_{\text{free}}) \geq x$. Best understood graphically in Figure 2.6, the



(a) Configuration Space (b) Workspace

Figure 2.6: C-space and workspace view of obstacles in the passable corridor, from [44]

width x is the minimum width of a ‘passable corridor’ in the collision free portion of the C-space. If all motion exists within this corridor, it can be considered safe at that resolution. In Figure 2.6 there are 2 obstacles: green and grey. In the case of the green obstacle, neither q_a or q_b occupy it, and the *swept volume* between the configurations does not either. The swept volume refers to the voxels that the robot occupies when moving between 2 configurations. However, while the grey obstacle is outside of q_a and q_b, it can be observed that it is impacted as part of the swept volume.

LaValle, Yang et al.’s proof combined states that, after M iterations, a deterministic dynamic roadmap (such as HDRM), sampled on a uniform grid, is resolution complete for all free robot configurations $\mathcal{C}_{\text{free}}$:

$$\mathcal{C}_{\text{free}} \in \Psi\left(4M^{-\frac{1}{N}} + f(s)\right) \tag{2.3}$$

M is the number of samples (the total number of represented configurations), N is the dimension of the configuration space and s is the resolution of the workspace (i.e. the length of a side of a square voxel). $f(s)$ is a robot-specific function that defines the extra width of the corridor that is necessary to ensure collisions do not occur in the swept volumes. This depends on the robot’s geometric shape, as well as the parameters M and s.

To calculate this, Yang et al. [44] describe that the largest width corridor must be considered. The paper then goes on to show that, in a discretised workspace made out of adjacent voxels, there is no need to consider swept volumes between configurations provided that the discretisation is fine enough.

Figure A.5 shows the edges between 2 occupation voxels, O_a and O_b. If the swept volume edge $\varepsilon(a,b)$ stretches across other occupation voxels, it must be stored, however if it is so short that the vertices and edges all fall within the same 2 occupation voxels, i.e. $O_{a,b} = O_a \cup O_b$, then the need to explicitly store the edge is removed at this resolution and discretisation: $\varepsilon(a,b)$ is collision-free iff a, b are collision-free, and vice-versa [44].

To achieve this density, the joints K_n must be discretised by at least some lower bound. This is derived in the paper and provided in equation 2.4. In this, $L_n^k = \sum_{j=n}^{j=k} l_j$ is the fully extended length of the robot arm from its base link n to its end-effector k, r_n is the radius of each robot link and θ_n is the range of motion of joint n. This equation forms the relationship between the voxel size s and the joint discretisation K_n. In general, as s

gets smaller, the necessary K_n gets larger.

$$K_n = \left\lceil \frac{\theta_n}{\Delta_n} + 1 \right\rceil = \left\lceil \theta_n \left(\sup_{n \leq k \leq N} \left\{ \frac{s + \sqrt{2} r_k}{L_n^k} \right\} \right)^{-1} + 1 \right\rceil \tag{2.4}$$

Finally, as HDRM is a uniform grid of configurations, a configuration's connected neighbours in the roadmap are precisely the joint configurations on either side of each joint, meaning that every n-dimensional configuration (apart from the boundaries of the range of motion) has $2 \cdot n$ neighbours it could traverse to.

Resolution Completeness in Time-Configuration Space

As this work extends HDRM to the time dimension, it must also extend HDRM's resolution completeness proof to upkeep this desirable property for time-indexed HDRM extensions.

The proof presented here uses HDRM's as a basis - all of the existing equations related to the workspace/C-space discretisation relationship continue to hold. The the proof can be split in 2 parts, and this thesis will now show **that any algorithm (such as Ti-HDRM) that satisfies both parts 1 and 2, as well as the discretisation properties from HDRM, can be considered to be spatially and temporally resolution complete**.

1) Resolution Completeness in Time-Indexed Static Environments

Before considering dynamic obstacles, it is important to determine how resolution completeness is maintained when the time dimension is added, but the environment is kept static as in classic HDRM.

When a robot travels from a configuration q_a, to another configuration q_b in the classic HDRM grid-based C-space roadmap, both the configurations individually, as well as the continuous set of points on the edge $\varepsilon(a,b)$ must be collision-free. With HDRM's fine workspace discretisation it suffices to check that just q_a and q_b are valid. For time-configuration space, movement in time and configuration are treated separately - only one occurs at a time.

Moving Forwards in Time By the same premise, motion between time indexed configurations $t_i q_a$ and $t_{(i+1)} q_a$ (where t_i represents a configuration belonging to a given time slice i) through time indexed roadmaps maintaining the same spatial resolution properties in HDRM can be considered valid if each of these end points are valid and the edge $\varepsilon(a,b)$ is valid, see the first horizontal red arrow moving to time-configuration A in Figure 2.7. By restricting moving forwards in time to only occur without movement in C-space, the path in time-configuration space does not curve, and, provided that the time discretisation is fine enough that it satisfies the bounds given in part (2), it is sufficient to check the validity of just $t_i q_a$ and $t_{(i+1)} q_a$ - guaranteeing safe movement forwards in time at this temporal resolution, as it is not possible for an obstacle to come between the $t_i q_a$ and $t_{(i+1)} q_a$.

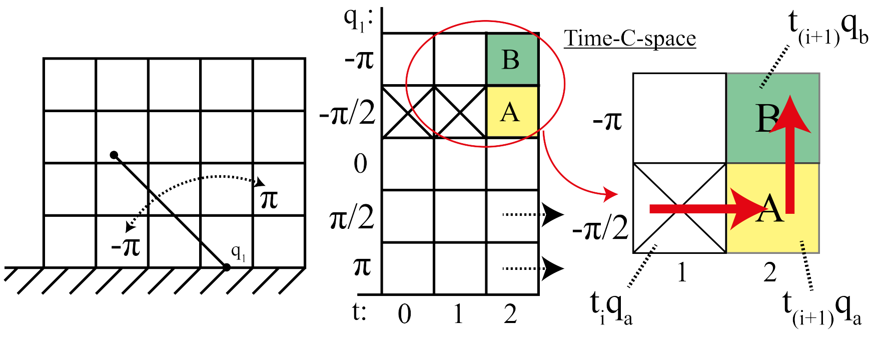


Figure 2.7: Travelling between time slices and then moving in C-space within a time slice. Moving to A is achieved in time, moving to B in C-space

Moving Sideways in C-space By restricting a robot to only move in its C-space *within a time slice*, the spatial resolution completeness properties from HDRM continue to hold, as each time slice can be considered its own HDRM problem. Within a time slice, the robot can move around as much as is feasible within its joint velocity limits (as described in Section 2.4.2) - though this is a concern of the algorithm and not a formal requirement for the temporal resolution completeness proof. See the vertical red arrow in Figure 2.7 for an example of this, where the robot moves to time-configuration B from A. In fact, for $dt = duration$ there will be only a single time slice, and in this case transitions across time slices will not occur and classic HDRM is recovered: HDRM is a subproblem of Ti-HDRM.

Finally, the special case of small dt must be explicitly handled. As described in Section 2.4.2, a robot may not be able to move to a neighbouring configuration within a single time slice if dt is short as it would incur an illegal joint velocity. Instead, the movement must be considered across multiple time slices, referred to as a dt_streak. In this case, the movement between configurations must be able to traverse a curve in time-configuration space, illustrated in Figure 2.8. In general, to guarantee that such a stretch is valid, **all of the configurations on the time-configuration space diagonal between the start and goal time slice must be checked, leading to $2 \cdot l$ validity checks, where l is the number of time slices the motion must be completed over**. If any of these checks are not made, there is no confirmation that there is not an obstacle that has intermediately popped up and invalidated a configuration, and resolution completeness will be lost.

2) Temporal Resolution Completeness and the Velocity of Dynamic Obstacles

In classic HDRM, the resolution of the workspace, s, defines the length of one of the sides of the uniformly-shaped voxels. This means that any obstacle from an infinitesimal point up to a cube of length s would invalidate a single voxel if placed in its centre.

Consider an infinitesimal, called *obs*, that is moving across the workspace of the robot at velocity v_{obs}. This point moves distance d in every time slice. To maximise the distance spent travelling in a single dimension, in the worst case *obs* travels parallel to an axis, as moving across multiple dimensions would split d into components - all less than d. In the most basic example of a 1D workspace with the 1 DOF robot, across

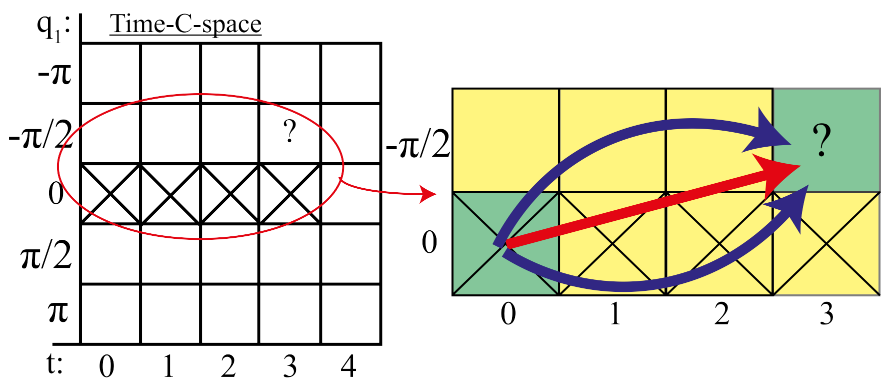


Figure 2.8: Stretching across small *dt* requires all configurations along the curves to be validated

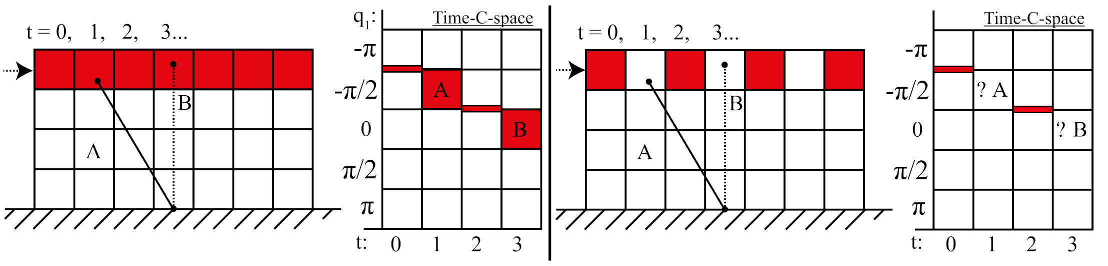


Figure 2.9: L: Trailing obstacle through workspace and C-space, R: Obstacle tunneling. Small red bars in C-space represent where the obstacle is in-between states

several time slices this motion would look like the trail seen in L of Figure 2.9. Here, the point causes invalidations in a perfect trail - meaning that its position in each time precisely succeeds the previous.

However, consider that the velocity of the point has increased, and now it invalidates across the workspace as in R of Figure 2.9.

In this scenario, *obs* is moving too fast to appear in the directly succeeding voxel in the directly succeeding time slice. Instead, **it appears to 'tunnel' through space at this time resolution. This is an issue as, when viewed in time-configuration space as seen in R of Figure 2.9, configurations that should be considered invalid are not**. If this is ignored then any time-indexed HDRM will not be resolution complete, and a robot may hit *obs* in one of the labelled 'gaps'.

This occurs when v_{obs} causes the distance d travelled in a time slice to exceed the voxel size s. In the worst case, the infinitesimal point could be infinitely close to the left of a voxel boundary in the preceding time slice, and then infinitely close to the right of a voxel boundary in the next time slice, see L in Figure 2.10.

In order to capture the information that *obs* transits voxels A, B, C in Figure 2.10, it is necessary for a Ti-HDRM algorithm to store the end points where *obs* occurs at each time index (A, C), as well as the swept volume in-between them (B). Storing this is analogous to traditional DRM methods that stored robot configurations as well as the volumes that were swept when moving between a pair of configurations.

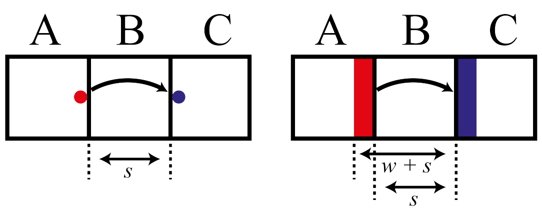


Figure 2.10: L: Infinitesimal R: Arbitrary width. Red is object before tunneling, blue is after

It can now also be shown that storing swept volumes of the obstacles such as *obs* is not necessary if the time discretisation is fine enough. Based on the worst case of the infinitesimal point moving parallel to an axis and being infinitely close to the voxel of size s boundary, it will appear in the succeeding voxel in the succeeding dt time slice iff:

$$v_{obs} \leq \frac{s}{dt} \tag{2.5}$$

If this is satisfied, at resolution s it is sufficient to only store the invalidated voxels, and not consider swept volumes. In the case of the infinitesimal, its width w can be considered to be 0. More generally, to guarantee temporal resolution completeness without swept volumes in 3D space, in a dynamic environment with R obstacles of any shape or size, the smallest width of each must be considered. Each of these obstacles will have a thinnest section width measuring w_r in length - for example, this could be a pinch-point of the shape or it could be constant across all 3 dimensions (in the case of a cube). As seen in R of Figure 2.10, this entire width must pass the width of a voxel before it will ‘tunnel’. Therefore, the time-configuration roadmap is resolution complete iff the following velocity upper bound is not exceeded by each of the obstacles (each obstacle will have its own upper limit, based on its respective thinnest width):

$$\max(v_r) = \frac{(w_r + s)}{dt} \tag{2.6}$$

2.4.4 Implementation of Ti-HDRM

RobotsInMotion Codebase

As part of this project, access to an implementation of classic HDRM was provided by the project supervisors. This codebase is the intellectual property of their startup company *RobotsInMotion Ltd.* [20], and therefore an NDA was signed that limits this report’s ability to go deep into the specific implementation details. This report is authorised however to discuss the general structure of the codebase, as well as the extensions that have been authored.

The codebase consists of an efficient implementation of both the Pre-planning and Query Phases of HDRM, using the algorithms detailed in [44]. Primarily, the code is written in C++ for performance, but a set of Pybind11 [10] Python bindings are available, meaning that motion plans can be setup and solved using Python within

Jupyter notebooks [18] The following parts were provided, those underlined are the parts that required extensive additions and edits to realise Ti-HDRM:

- **Pre-planning** - code to generate the hierarchical occupation lists.
- **HDRM Setup** - code that parses the occupation lists and creates data structures that represent the validity of the configurations.
- **A* search** - code that searches through the configurations.
- **Testing and visualisation** - a basic set of functionality, based on the Python MeshCat [19] 3D visualiser, that allows for robot configurations and occupied workspace voxels to be viewed in a WebGL [26] window.

HDRM Setup

In the classic HDRM codebase, after the occupation lists have been precomputed, the mapping that they contain is stored in memory and, as the user invalidates voxels (by referencing their unique index number), the samples (representing configurations) are marked as valid/invalid for use in the roadmap search.

In the new Ti-HDRM implementation, the occupation list is replicated across all of the time slices, so that the validity of configurations at different times can be assessed - creating a $(n+1)$-dimensional `ValidityMap`. This is indexed into with the previously described introduction of a 3rd integer, ti.

In order to invalidate ti-voxels and therefore ti-configurations, it is necessary to map the 3 spatial dimensions (x,y,z) and time slice to a single ti-index. This ti-index is then used to query the `ValidityMap` and 'turn off' all of the corresponding configurations at that ti. The function in Figure 2.11 allows the user to specify the location and time of an invalidated voxel, and it outputs a ti-index formed from multiplying and squashing the dimensions onto a single linear axis [2]:

```
int to_ti_index(std::tuple<int, int, int, int> ti_location) {
    return std::get<0>(ti_location) * z_len * y_len * x_len +
           std::get<1>(ti_location) * y_len * x_len +
           std::get<2>(ti_location) * x_len +
           std::get<3>(ti_location); }
```

Figure 2.11: Each field of ti_location corresponds to: ti, x, y, z

This ti-index can then be used to identify the list of configurations in the corresponding time slice in the `ValidityMap`. As the Pre-planning Phase has not been changed, these will be represented by a list of samples, rather than ti-samples. A similar re-indexing to ti-sample from sample is necessary - essentially offsetting the standard sample by a time slice multiple, see Figure 2.12:

[2]Note that the code shown here is a focused subset of the authored C++/Python, and has been sanitised in compliance with the NDA.

```
void sample_to_ti_sample(int ti, int sample, int& ti_sample) {
    ti_sample = sample + (ti * total_config_num); }
```

Figure 2.12: sample to ti-sample

A* Search

With considerable technical detail abstracted due to the NDA, at this point in the implementation, for a given motion query, there is now a collection of valid ti-samples, ready to be searched through with A*. Because each ti-sample is just the vertex of a graph, the A* search code is a direct line-by-line translation of the standard pseudocode in Algorithm B.3. One of the primary works of the implementation is the completely redesigned `getNeighbours` function of the A* - conceptually described in Algorithm 4. By design, the C++ code for this algorithm is also a direct translation of the pseudocode presented. Algorithm 4 did however rely on 2 important function calls that are implementation specific: `countPrevImmobileSlices` - for calculating the explored ti-sample's *dt_streak* (see Figure 2.13), and `calculateTotalJointMovementInTi` - for working out the remaining velocity budget in the current time slice (see Figure B.5).

```
int count_prev_immobile_(int ti_sample, CloseList& close) {
    int orig_sample, parent_ti, ti, sample,
        parent_sample, prev_ti_sample;
    int count = 0;
    ti_sample_to_sample(ti_sample, ti, orig_sample);
    bool found_start_ti = false;
    while ((!found_start_ti) ||
                (close.find(ti_sample) != close.end())) {
        if (ti_sample != std::get<0>(close[ti_sample]))
            ti_sample = std::get<0>(close[ti_sample]);
        else
            found_start_ti = true;
        ti_sample_to_sample(ti_sample, ti, sample);
        if (sample == orig_sample)
            count++;
        else
            found_start_ti = true;
    }
    return count; }
```

Figure 2.13: `countPrevImmobileSlices` function

Both of these work similarly by traversing backwards through the closed set. For `countPrevImmobileSlices`, at each iteration, the parent ti-sample's underlying configuration is checked to see if it is different from the one being considered. If not, a count is increased and the next iteration begins, if yes (or the start ti-sample is found), the count is returned.

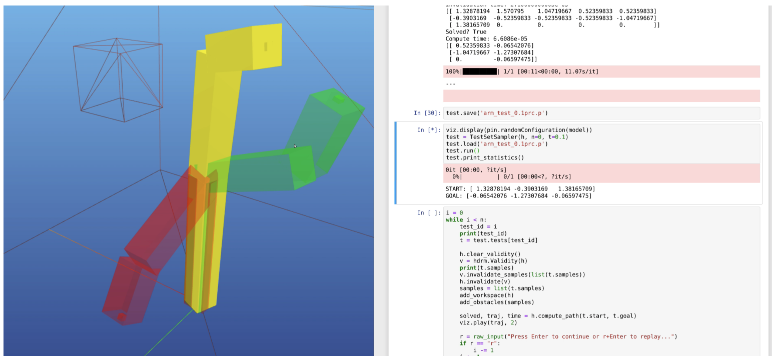
Similar backtracking is used for `calculateTotalJointMovementInTi`. A recursive function is used to identify the first parent ti-sample of the current ti in the explored path.

Then, this ti-sample is converted into the joint angles of the robot, and the difference in movement in these joint angles is summed over its children ti-samples - calculating a total amount of joint movement thus far. When divided by the dt, this generates a velocity that is checked to be below the joint velocity limits of the robot. If there is a velocity budget, the neighbour can be considered as in the conceptual design. Due to length, see Figure B.5 for this function.

Testing and Visualisation

To aid in the testing and verification of Ti-HDRM, this project has created a suite of visualisation tools, extending the basic MeshCat functionality provided in the codebase. As an overview, support has been added for:

- Loading and manipulating complex robots and 'ghosts' to represent start and goal states for motion - using the Pinocchio rigid body dynamics library [23], the ability to add and manipulate robot models has been added. A variety of trajectory playback software has been authored to allow for the execution and visualisation of trajectories generated by both PPT-HDRM and Ti-HDRM methods.
- Dynamic obstacle and point cloud support - objects can be added and moved across the workspace of the robot at different velocities. Point clouds, similar to the ones provided by depth cameras [9], are also supported and can be used as an input to invalidating portions of the workspace.
- Graphing capabilities - to check trajectory velocities and joint positional limits are respected, live graphing with Matplotlib [13] has been implemented for easy visualisation of what the robot is being commanded.
- Comprehensive unit testing - as a complex software project, test-driven development practices [24] were followed in feature development: both unit and module testing of the algorithms and their components was undertaken. As a testbed, a simple 3 DOF robot (see Figure 2.14) with a limited range of motion (and therefore smaller set of ti-samples to explore) was used to verify components of the algorithms were working.



```
[[ 1.32878194  1.570795    1.04719667  0.52359833  0.52359833]
 [-0.3903169  -0.52359833 -0.52359833 -0.52359833 -1.04719667]
 [ 1.38165709  0.          0.          0.          0.        ]]
Solved? True
Compute time: 6.6086e-05
[[ 0.52359833 -0.06542076]
 [-1.04719667 -1.27307684]
 [ 0.         -0.06597475]]
100%|██████████| 1/1 [00:11<00:00, 11.07s/it]
---

In [30]: test.save('arm_test_0.1prc.p')

In [*]: viz.display(pin.randomConfiguration(model))
        test = TestSetSampler(h, n=0, t=0.1)
        test.load('arm_test_0.1prc.p')
        test.run()
        test.print_statistics()

0it [00:00, ?it/s]
  0%|          | 0/1 [00:00<?, ?it/s]
START: [ 1.32878194 -0.3903169   1.38165709]
GOAL: [-0.06542076 -1.27307684 -0.06597475]

In [ ]: i = 0
        while i < n:
            test_id = i
            print(test_id)
            t = test.tests[test_id]

            h.clear_validity()
            v = hdrm.Validity(h)
            print(t.samples)
            v.invalidate_samples(list(t.samples))
            h.invalidate(v)
            samples = list(t.samples)
            add_workspace(h)
            add_obstacles(samples)

            solved, traj, time = h.compute_path(t.start, t.goal)
            viz.play(traj, 2)

            r = raw_input("Press Enter to continue or r+Enter to replay...")
            if r == "r":
                i -= 1
```

Figure 2.14: 3 DOF robot test bed, alongside a snippet of the Jupyter notebook used for testing and development

Chapter 3

Evaluation

This chapter evaluates Ti-HDRM and the naive methods that have been developed in this work. Both qualitative visualisation-based evaluation is conducted, as well as quantitative automated testing, simulation and code performance profiling. **A video of the experiments is available here:** `https://youtu.be/L9aMBA4f8ao`

3.1 Qualitative Visualisation Comparison

Part of experimentally proving that a new motion planning algorithm functions correctly is achieved by planning and executing trajectories on a robot model in a dynamic environment computer visualisation. By qualitatively inspecting the motion, the pitfalls of PPT methods can be witnessed, and Ti-HDRM's success in those settings verified.

A seminal task in a dynamic environment (seen in [45, 15]) is to reach into a moving shelf. As the walls of the shelf are in motion, the free space window within the different sections is constantly changing. This problem is equivalent to a robot on a mobile base reaching into a static shelf as it moves past, as the motion is relative to the robot itself.

The experiment setup for this uses the Nextage bi-manual robot. As HDRM is limited to work on single kinematic chains, the left arm, chest and head joints are fixed, whilst the 6 DOF of the right arm are free to move. The Pre-planning Phase is performed for this arm and the hierarchical occupation lists are generated for the robot's 0.8m $\times$ 0.8m $\times$ 1.0m workspace with voxel size $s = 0.05$m. For spatial resolution completeness, the joint discretisation K_n for the Nextage at this resolution is: $\{19, 21, 11, 6, 6, 1\}$. A model of a KALLAX IKEA shelf [11] is also used. A point cloud (generated with CloudCompare [2]) of 300 points from this model is over layed on top of it to mimic a depth sensor input - these points are used as the input for voxel invalidations. The thinnest part of this model has $w = 0.01$m, and a time discretisation of $dt = 0.05s$ is chosen. By temporal resolution completeness equation 2.6, the maximum velocity of this shelf can be: $\max(v) = (0.01 + 0.05)/0.05 = 1.2$m/s. The chosen velocity for the shelf is 0.5m/s and it moves parallel to the front of the Nextage at a constant distance of 0.05m. The movement target (red ghost) corresponds to joint angles within a section of the shelf. The start joint angles were randomly chosen (green ghost).

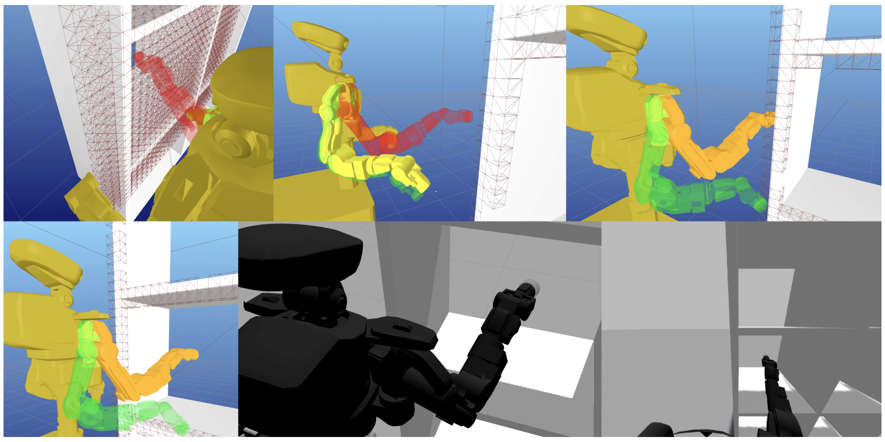


Figure 3.1: TL: Streaked failure, TC: Linear invalid intermediate TR: Greedy rushed goal, BL: Ti-HDRM success, BC: Gazebo Ti-HDRM, BR: Gazebo head-mounted view

Streaked HDRM: As seen in the image at the top-left in Figure 3.1, in the Streaked experiment, the red voxels show the continuous block of space that the front face of the moving shelf occupies when time is ignored and the moving obstacle is streaked across the workspace. Consequently the goal pose is occluded, and the motion fails.
Linear PPT-HDRM: An improvement can be seen in Linear PPT-HDRM in that it progresses somewhat into the motion plan, with the voxels being reset at each time slice. Unfortunately, an invalid intermediate start state occurs and the motion fails, seen in the top-centre of Figure 3.1.
Greedy PPT-HDRM: Greedy PPT-HDRM fares better than its Linear brother, managing to avoid any invalid intermediate start states. It races to the goal pose and arrives there seconds before the shelf is due to pass. Blissfully unaware that it must retract before re-entering the shelf, it waits here right up until the moment the shelf is about to collide with it. The motion fails because there is not enough time for the retraction to take place, see the top-right of Figure 3.1.
Ti-HDRM: Ti-HDRM, with its full knowledge of time-configuration space, is able to solve the motion, and it orients the arm up and to the side, waiting for the placement window to arrive. When it does, it successfully reaches into the shelf, see bottom-left of Figure 3.1.

Overall, this experiment clearly demonstrates the shortcomings of the classic and naive methods. Both Streaked HDRM and Linear PPT-HDRM either did not execute or were only able to solve a fraction of the motion. Greedy PPT-HDRM rushed to the goal, but resulted in a collision with the moving shelf as it quite literally did not ‘see it coming’! Ti-HDRM solved the motion correctly, reaching into the shelf safely and accurately.

3.2 Full Simulation in Gazebo

While the 3D visualisation is a useful tool to understand the path of the trajectories of the Nextage, it is not a full physics simulation. From visualisation there is confidence in the instructed robot configurations, but testing that joint velocities and the physical limitations of the robot is harder to determine. Theoretical velocity graphs can be plotted, but compliance with hardware limits is best tested in full simulation.

To do this, *ROS* [21] and *Gazebo* [6] are used as the basis of simulation experiments. ROS is a set of software libraries and tools that provide an interface between the outputs of motion planning algorithms like Ti-HDRM and actual robots - either hardware or software. Gazebo is a physics and rigid body simulator. The crux of using ROS in these experiments consisted of composing and sending network messages with robot instructions to be executed on the Nextage in Gazebo. The trajectory was computed using Ti-HDRM, and the shelf's point cloud data was again used to perform invalidations across the workspace. The simulation was run on a Ubuntu 18.04 machine with ROS Melodic, see bottom-centre and bottom-right of Figure 3.1. Just as in the visualisation, the Nextage successfully reached into the shelf. This action was completed smoothly and the velocity limits were not exceeded during the multiple reaching attempts made.

3.3 Automated Test Suite Experiments

From these promising experiments, a hypothesis emerges that Ti-HDRM is able to definitively answer all feasible motion queries, while Streaked and PPT methods may succeed sometimes, but are likely to fail in the majority of cases - primarily due to invalid intermediate start states or illegal velocities along the way. To test this hypothesis, this report now presents a suite of automated tests (tested with all HDRM-derived methods) involving a randomised shelf moving across the workspace, for different dt.

3.3.1 Complex Dynamic Environment - Randomised Moving Shelf

For this experiment, in each test the dynamic environment is setup with the shelf moving across the Nextage's workspace. The parameters of this movement are randomly selected, including its start location, size and velocity (within the stated temporal resolution bounds). The start and goal poses are also randomly chosen, and the test software iterates these until a valid pair is found. Without running a motion planner, it is not possible to know whether this randomised environment does present a motion planning problem that **is** solvable - for example, just because the start and goal poses are valid does not mean that there is a safe trajectory between them. In fact, if successful, Ti-HDRM would be the baseline used to determine this! When comparing Ti-HDRM with Streaked/PPT methods however, it is of interest to **see if there are any problems where Ti-HDRM fails, but any of the other methods succeed**.

For each randomised setup, this experiment runs Streaked HDRM, Linear/Greedy PPT-HDRMs and Ti-HDRM. The outcomes of each of these methods are either: solved or failed, however failure is split into 'true fail', 'invalid intermediate start state' and 'illegal velocity' to provide statistics on how often each event occurs. 3 dt values ($0.01s, 0.025s, 0.04s$ were used) all with a fixed duration of 5s (for 500, 200, 125 time slices respectively).

For each dt, the experiments were run 1000 times, with **not a single instance found where Ti-HDRM failed to solve a motion but one of the other succeeded**. Out of the 1000 randomly generated environments, Ti-HDRM was successful approximately half of the time - suggesting that only half of the environments actually had solvable motion plans. After this, the experiments were re-run until there were 500 successful Ti-HDRM solves, and the outcomes of each method saved for each of these. For each dt, the results were similar and thus $dt = 0.01$ is provided at Figure 3.2, with $dt = 0.025$ and $dt = 0.04$ available at Appendix C.1 and C.2 respectively.

$dt = 0.01$	True Fail	Invalid Int.	Illegal Vel.	Solved	% Success
Streaked HDRM	468	–	–	32	6.4%
Linear PPT-HDRM	0	181	318	1	0.2%
Greedy PPT-HDRM	58	334	0	108	21.6%
Ti-HDRM	0	0	0	500	**100.0%**

Figure 3.2: $dt = 0.01$ complex shelf experiment results

These results show that, for dynamic problems such as moving shelves, the Streaked and PPT methods perform poorly, with low success rates. As expected, Greedy PPT-HDRM is the best performing, with no illegal velocities generated, but still only an average success rate of 19%.

While these results could further be compared to other motion planners, as detailed in Chapter 1, Ti-HDRM is a unique motion planning method for the criteria outlined in the specification. To the author's knowledge, no others provide the mixture of collision-free, velocity-limit-respecting and resolution-complete motion planning in time-configuration for multi-DOF robots. As such, comparing this work to other methods, even ones that partly fulfill the specification (such as [45]), has proven to be akin to comparing apples to oranges! It is for this reason that all of the HDRM extensions have been developed, and comprehensive testing against them conducted.

3.3.2 Simple Dynamic Environment - Slow Moving Cube

The previous experiment tests the performance of the algorithms in a highly dynamic environment involving a moving shelf with numerous sections that can be reached in and out of. This stress testing largely benefits Ti-HDRM. To explore the performance in a more simple dynamic environment, a similar experiment with a moving cube was performed. The solid cube measured 4 voxels^3, and, in each test run, moved across the workspace at the same fixed distance away from the Nextage, and at a random velocitiy between 0m/s - 0.25m/s. The same 3 dt values were used with a fixed duration of 5s, with the $dt = 0.01$ simple experiment found in Figure 3.3. $dt = 0.025$ and $dt = 0.04$ results can be found at Appendix C.3 and C.4 respectively.

$dt = 0.01$	True Fail	Invalid Int.	Illegal Vel.	Solved	% Success
Streaked HDRM	334	–	–	276	55.2%
Linear PPT-HDRM	74	181	215	30	6%
Greedy PPT-HDRM	37	149	0	314	62.8%
Ti-HDRM	0	0	0	500	**100.0%**

Figure 3.3: $dt = 0.01$ simple cube experiment results

As expected, all PPT and Streaked methods performed substantially better in this more simple experiment, though none approach the coverage that Ti-HDRM offers. As the next section explores however, the computational cost of Ti-HDRM is greater than classic/Streaked HDRM, and the decision to use one or the other is more nuanced.

3.4 Profiling and Computational Analysis

With the excellent 100% coverage of Ti-HDRM demonstrated, what is the computational price of this? Figure 3.4 shows the average time taken for the successful solves of each method in the densest time-configuration roadmap, $dt = 0.01$, of the complex shelf experiment in Figure 3.2 - the tables for time taken for other dt tell a similar story and are included in Appendix C.5 and C.6. A breakdown of time spent on invalidations vs time spent on searching is provided. Note that these were measured from Python and do also contain the overhead of other code, including function calls manipulating the shelf:

$dt = 0.01$	Invalidations (s)	Search (s)	Total (s)
Streaked HDRM	0.97	0.25	1.22
Linear PPT-HDRM	2.28	0.98	3.26
Greedy PPT-HDRM	2.59	0.97	3.56
Ti-HDRM	3.67	1.31	4.98

Figure 3.4: Average elapsed time for $dt = 0.01$ complex shelf experiment

In all instances, the invalidation step of the algorithms was the most expensive. This is expected as the point cloud representing the shelf consists of 300 points that must be parsed, and then invalidated across 400 dt! It also makes sense that the time for these invalidations increases as the methods become more complicated. As Streaked HDRM is the same algorithm as in the original paper, the invalidations are only performed once across a single set of hierarchical occupation lists. For the PPT methods, there is an extra layer of computation as the occupation lists are reset and then re-invalidated at every time step. Finally, for Ti-HDRM, the introduction of a time index also adds another layer of translation and indexing.

Again, with Streaked HDRM, only a single A* search of a relatively small C-space roadmap is performed - leading to a quick solution. PPT methods also search this small C-space roadmap, but must repeat this between the start and goal pose times - in this instance that could be up to 400 A* searches! For Ti-HDRM, it only performs one A* search, but through a considerably larger time-configuration space roadmap that has 400 times more states than the others.

3.4.1 Quantifying Ti-HDRM Performance

This section will now discuss the computational performance of Ti-HDRM. Roadmap methods, like HDRM and now Ti-HDRM, suffer from the *curse of dimensionality* [3], meaning that, as the dimensionality of the C-space increases with added DOF, so does the sheer number of configurations that must be stored, indexed and explored. Therefore, a robot arm with 8 DOF vs one with 3 DOF will traditionally lead to slower HDRM planner times. By adding time to HDRM, a 6 DOF robot arm like the Nextage now becomes a 7 DOF - however this is an underestimate of the additional work.

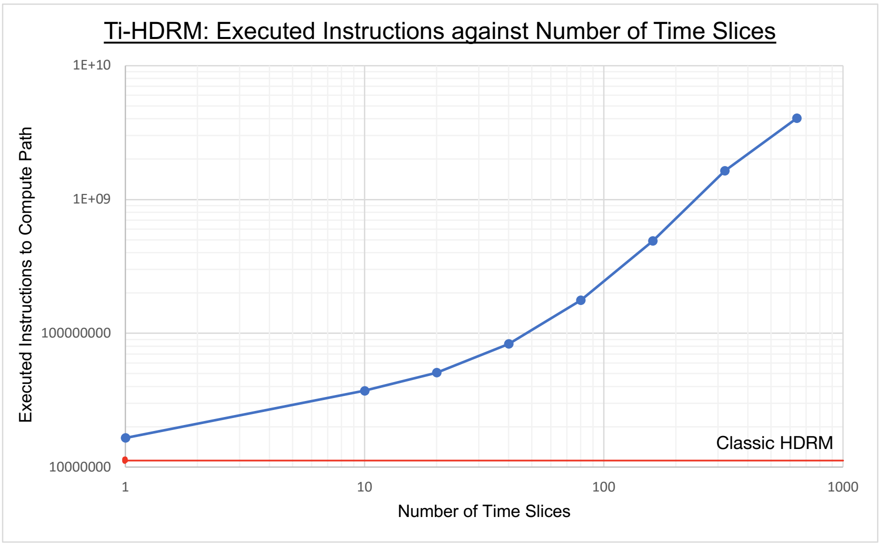


Figure 3.5: Ti-HDRM: Instructions executed during `computePath` (log scales)

While the typical discretisation of a joint for the Nextage is as high as 20, in Ti-HDRM, even for a short planning duration of 5s, for small *dt* this can easily lead to 100s-1000s of time slices - greatly increasing the amount of time-configurations to explore. In addition to this, the `getNeighbours` function, called in every iteration of the A* search, must now check the validity of many more ti-samples than in HDRM, in order to maintain resolution completeness, explained in Section 2.4.3.

Even though extra invalidations need to be performed in Ti-HDRM, the invalidation procedures are largely unchanged from HDRM. As more ti-indexes are invalidated, the invalidation function calls scale linearly. On the other hand, the A* search has significant extra complexity as it must backtrack and check many more configurations in Ti-HDRM than HDRM - **the key added computational complexity of Ti-HDRM**.

To quantify the A* search slowdown, a blank workspace with no obstacles was created. Motion was then planned with Ti-HDRM for the Nextage with a *dt* ranging from $dt = 10$s to $dt \approx 0.015$s, for a fixed $duration = 10$s - leading to between 1 and 640 time slices. As *dt* decreases, the number of time slices and therefore time-configurations to explore and backtrack increases. The start and goal poses for the motion were fixed in all experiments, and the start and goal times corresponded to the first and last time slice for each *dt*. Measured in C++, both the time to search the roadmap was measured, and then Valgrind's [25] `callgrind` tool was used to profile the program and determine the number of instructions that were executed in the A* search (inclusive of `getNeighbours` and adding/popping from the frontier/closed set). Experiments were run on a desktop with Intel Core i9-9900K CPU and 64GB of RAM. Figure 3.5 shows the results for the number of instructions, and Appendix C.8 shows the same but for elapsed time. Raw data is available in Figure C.7. Both charts include a baseline of the same motion (without time consideration) calculated by classic HDRM.

With over 3 million instructions, classic HDRM's A* search is more performant than the completely equivalent Ti-HDRM problem with only a single time slice, which executed 16 million instructions. While the output paths by each method were confirmed to be identical, this performance difference is related to the increased overhead of indexing with the extra ti index, as well as repeated calls to `countPrevImmobileSlices` (even though there is only one time slice).

As the number of time slices increase, so does the number of instructions executed in the A* path computation. This relationship starts out roughly linear (up until 40 time slices) - meaning that doubling the time slices doubles the number of executed instructions on the CPU. Interestingly, as the number of time slices surpasses 40, this relationship starts to accelerate - for example, 80 time slices takes 17 million instructions, while 160 time slices takes almost 50 million. This is due to the fact that, as the number of time slices in this experiment increased, the *dt* decreased. When *dt* is small, the requirement to backtrack and check the validity of previous time slices before making any movement is exacerbated, explained in Section 2.4.3. These extra validity checks incur extra operations. Note however that, even in the case of 640 time slices, the A* search was completed in less than 750ms. Also, whilst this trend is due to extensions in the algorithm, absolute performance can be improved by optimising the C++ codebase.

Clearly, while Ti-HDRM has many exceptional qualities including temporal resolution completeness, if performance is valued above all else and the environment is only lightly dynamic, a user may make the informed choice to use classic/Streaked HDRM.

3.5 Conclusion and Future Work

Overall this project has designed and implemented Ti-HDRM - a motion planning algorithm that meets the safety criteria outlined in Chapter 1. Along the way, various unsatisfactory other methods have been designed, and Ti-HDRM's supremacy over these proven and evaluated. Full physics simulation has been undertaken to showcase the theoretical performance of Ti-HDRM in reality. Finally, a formal proof of resolution completeness has been derived. With the success of this project, it is the author's intention to contribute Ti-HDRM as either a conference or journal paper. A reflection on personal lessons learned can be found at Appendix D. As an undergraduate project with limited time, Ti-HDRM presents interesting further explorations for future work:

- **Full hardware integration testing** - while this thesis focused on an algorithmic contribution to the field of motion planning, with simulation-based verification, a full showcase of Ti-HDRM on physical hardware is a natural next step. Integrating the inputs to Ti-HDRM with Computer Vision methods and creating the pipeline for a full hardware demo is probably another undergraduate project in itself!
- **Optimisations** - to maintain completeness properties, considerable extra state and time-configurations must be stored and checked in Ti-HDRM. While the implementation already presents extremely performant data structures, more efficient encodings may be possible.
- **Extension to multiple kinematic chains** - a fundamental limitation of HDRM, and consequently Ti-HDRM, is that the hierarchical encoding limits the method's operation to single kinematic chains like robotic arms, as opposed to multi-chains such as quadrupeds. An extension that generalised the method would have to combat the curse of dimensionality [3] that comes with higher DOF systems.

Bibliography

[1] A* search algorithm - Wikipedia. https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-nilsson-4.

[2] CloudCompare - home. https://www.cloudcompare.org/main.html.

[3] Curse of dimensionality - Wikipedia. https://en.wikipedia.org/wiki/Curse_of_dimensionality.

[4] Forward kinematics — ros robotics. https://www.rosroboticslearning.com/forward-kinematics.

[5] The Future of Manufacturing: Human and Robot Collaboration. https://www.reliableplant.com/Read/31352/human-robot-collaboration.

[6] Gazebo. http://gazebosim.org/.

[7] Greedy algorithm - Wikipedia. https://en.wikipedia.org/wiki/Greedy_algorithm.

[8] Harmony – assistive robots for healthcare. https://harmony-eu.org/.

[9] Intel® RealSense™ Technology. https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/realsense-overview.html.

[10] Intro — pybind11 documentation. https://pybind11.readthedocs.io/en/stable/.

[11] KALLAX white, Shelving unit, 77x147 cm - IKEA. https://www.ikea.com/gb/en/p/kallax-shelving-unit-white-80275887/.

[12] The Kinematics of Machinery: Outlines of a Theory of Machines - Franz Reuleaux - Google Books. https://books.google.co.uk/books?id=WUZVAAAAMAAJ&printsec=frontcover&dq=kinematics+of+machinery&hl=en&sa=X&ei=qpn4Tse-E9SasgLcsZytDw&redir_esc=y#v=onepage&q=kinematics%20of%20machinery&f=false.

[13] Matplotlib — Visualization with Python. https://matplotlib.org/.

[14] Monotonic function - Wikipedia. https://en.wikipedia.org/wiki/Monotonic_function.

[15] Nextage Interactive Collision-Free Bi-Manual Manipulation | InfWeb. https://web.inf.ed.ac.uk/slmc/research/projects-and-grants/kawada.

[16] Path planning · introduction to open-source robotics. http://www.osrobotics.org/osr/planning/path_planning.html.

[17] Priority queue - Wikipedia. https://en.wikipedia.org/wiki/Priority_queue.

[18] Project Jupyter | Home. https://jupyter.org/.

[19] rdeits/meshcat-python: WebGL-based 3D visualizer for Python. https://github.com/rdeits/meshcat-python.

[20] ROBOTSINMOTION LIMITED overview - Find and update company information - GOV.UK. https://find-and-update.company-information.service.gov.uk/company/SC680083.

[21] ROS: Home. https://www.ros.org/.

[22] Rrt graph1 - rapidly-exploring random tree - wikipedia. https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree#/media/File:RRT_graph1.png.

[23] stack-of-tasks/pinocchio: A fast and flexible implementation of Rigid Body Dynamics algorithms and their analytical derivatives. https://github.com/stack-of-tasks/pinocchio.

[24] Test-driven development - Wikipedia. https://en.wikipedia.org/wiki/Test-driven_development.

[25] Valgrind home. https://valgrind.org/.

[26] WebGL: 2D and 3D graphics for the web - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.

[27] J. P. Ballantine and A. R. Jerbert. Distance from a line, or plane, to a point. *American Mathematical Monthly*, 59:242–243, 4 1952.

[28] Massimo Cefalo, Giuseppe Oriolo, and Marilena Vendittelli. Task-constrained motion planning with moving obstacles. *IEEE International Conference on Intelligent Robots and Systems*, pages 5758–5763, 2013. ISBN: 9781467363587.

[29] Prasun Choudhury, Benjamin Stephens, and Kevin M. Lynch. Inverse kinematics-based motion planning for underactuated systems. *Proceedings - IEEE International Conference on Robotics and Automation*, 2004(3):2242–2248, 2004. Publisher: Institute of Electrical and Electronics Engineers Inc.

[30] John J Craig, Pearson Prentice, and Pearson Prentice Hall. Introduction to Robotics Mechanics and Control Third Edition. 2005.

[31] Marcos de Sales Guerra Tsuzuki, Thiago de Castro Martins, and Fábio Kawaoka Takase. ROBOT PATH PLANNING USING SIMULATED ANNEALING. *IFAC Proceedings Volumes*, 39(3):175–180, January 2006. Publisher: Elsevier.

[32] Th Fraichard. Dynamic trajectory planning with dynamic constraints: A 'state-time space' approach. pages 1393–1400, 1993. Publisher: Publ by IEEE ISBN: 0780308239.

[33] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[34] Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. STOMP: Stochastic trajectory optimization for motion planning. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 4569–4574, 2011. ISBN: 9781612843865.

[35] Lydia E. Kavraki, Petr Švestka, Jean Claude Latombe, and Mark H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[36] Robert Kindel, David Hsu, Jean Claude Latombe, and Stephen Rock. Kinodynamic motion planning amidst moving obstacles. *Proceedings-IEEE International Conference on Robotics and Automation*, 1:537–543, 2000.

[37] Steven M. LaValle, Michael S. Branicky, and Stephen R. Lindemann. On the Relationship between Classical Grid Search and Probabilistic Roadmaps:. *http://dx.doi.org/10.1177/0278364904045481*, 23(7-8):673–692, July 2016. Publisher: SAGE Publications.

[38] Peter Leven and Seth Hutchinson. A Framework for Real-time Path Planning in Changing Environments:. *http://dx.doi.org/10.1177/027836490202101200l*, 21(12):999–1030, July 2016. Publisher: SAGE PublicationsSage UK: London, England.

[39] Kevin M Lynch and Frank C Park. MODERN ROBOTICS MECHANICS, PLANNING, AND CONTROL. 2017. ISBN: 9781107156302.

[40] Jiayu Men and Jesus Requena Carrion. A Generalization of the CHOMP Algorithm for UAV Collision-Free Trajectory Generation in Unknown Dynamic Environments. *2020 IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2020*, pages 96–101, November 2020. Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781665403900.

[41] Nathan Ratliff, Matt Zucker, J. Andrew Bagnell, and Siddhartha Srinivasa. CHOMP: Gradient optimization techniques for efficient motion planning. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 489–494, 2009. Publisher: Institute of Electrical and Electronics Engineers Inc.

[42] John Schulman, Jonathan Ho, Alex Lee, Ibrahim Awwal, Henry Bradlow, and Pieter Abbeel. Finding Locally Optimal, Collision-Free Trajectories with Sequential Convex Optimization. *undefined*, January 2013. Publisher: Robotics: Science and Systems Foundation.

[43] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. Robotics. 2009. Publisher: Springer London Place: London ISBN: 978-1-84628-641-4.

[44] Yiming Yang, Wolfgang Merkt, Vladimir Ivan, Zhibin Li, and Sethu Vijayakumar. HDRM: A Resolution Complete Dynamic Roadmap for Real-Time Motion Planning in Complex Environments. *IEEE Robotics and Automation Letters*, 3(1):551–558, January 2018. Publisher: Institute of Electrical and Electronics Engineers Inc.

[45] Yiming Yang, Wolfgang Merkt, Vladimir Ivan, and Sethu Vijayakumar. Planning in Time-Configuration Space for Efficient Pick-and-Place in Non-Static Environments with Temporal Constraints. *IEEE-RAS International Conference on Humanoid Robots*, 2018-November:893–900, January 2019. Publisher: IEEE Computer Society.

Appendix A

Additional Figures

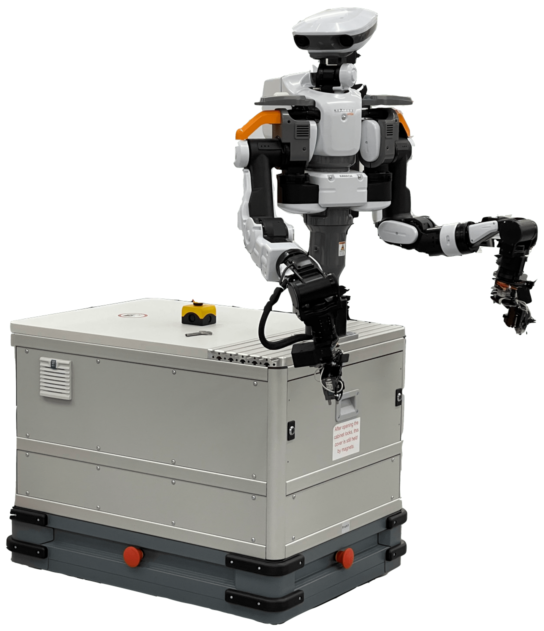


Figure A.1: Nextage Bi-manual robot atop an omni-directional mobile base

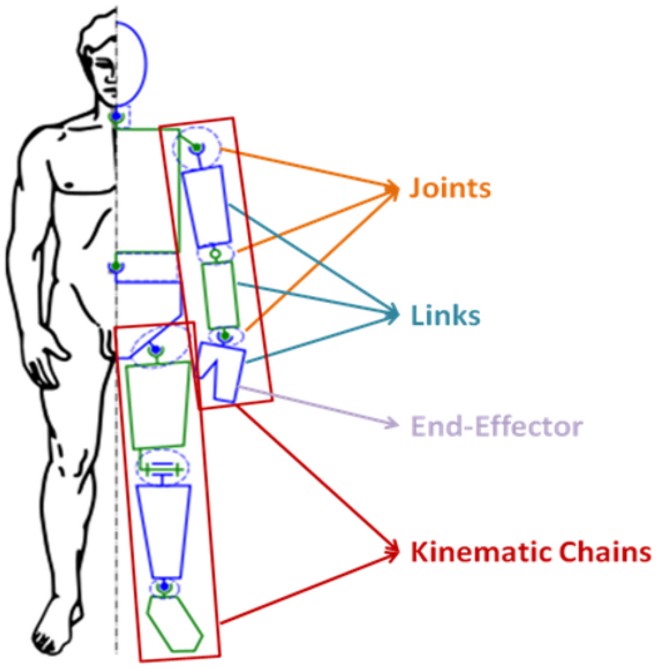


Figure A.2: Single kinematic chains in a human, from [4]

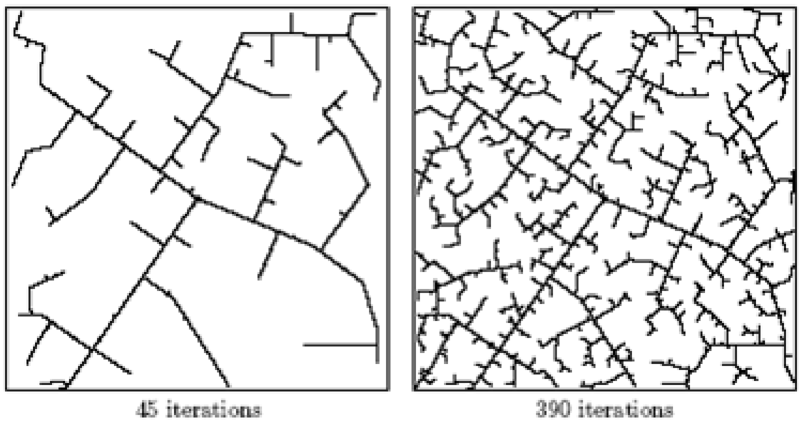


Figure A.3: RRT example, from [22]

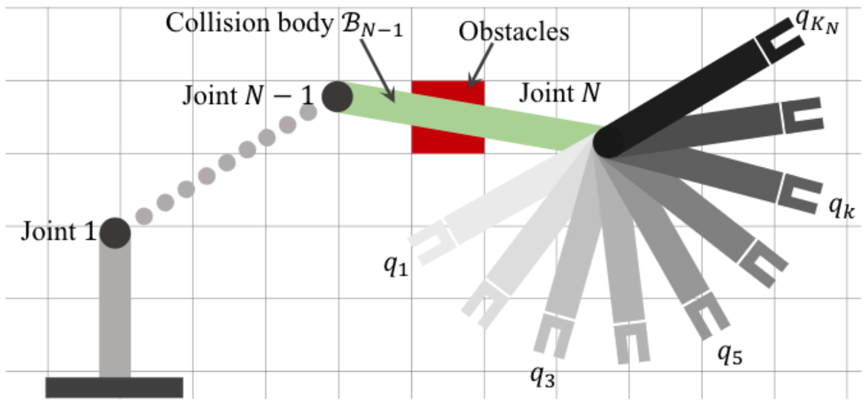


Figure A.4: A collision in a lower joint automatically excludes all permutations of higher joints, from [44]

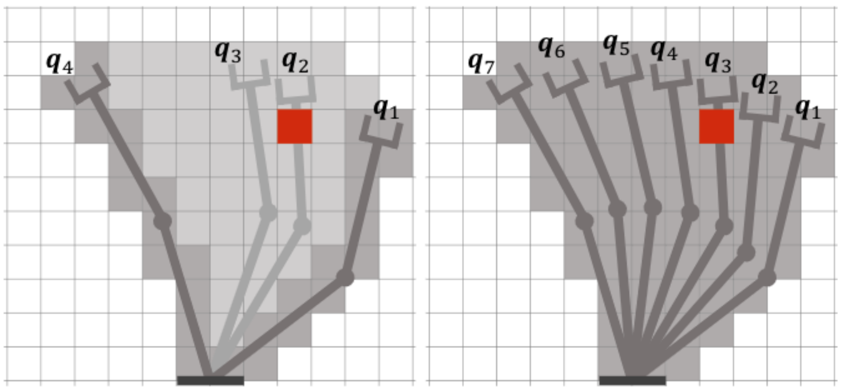
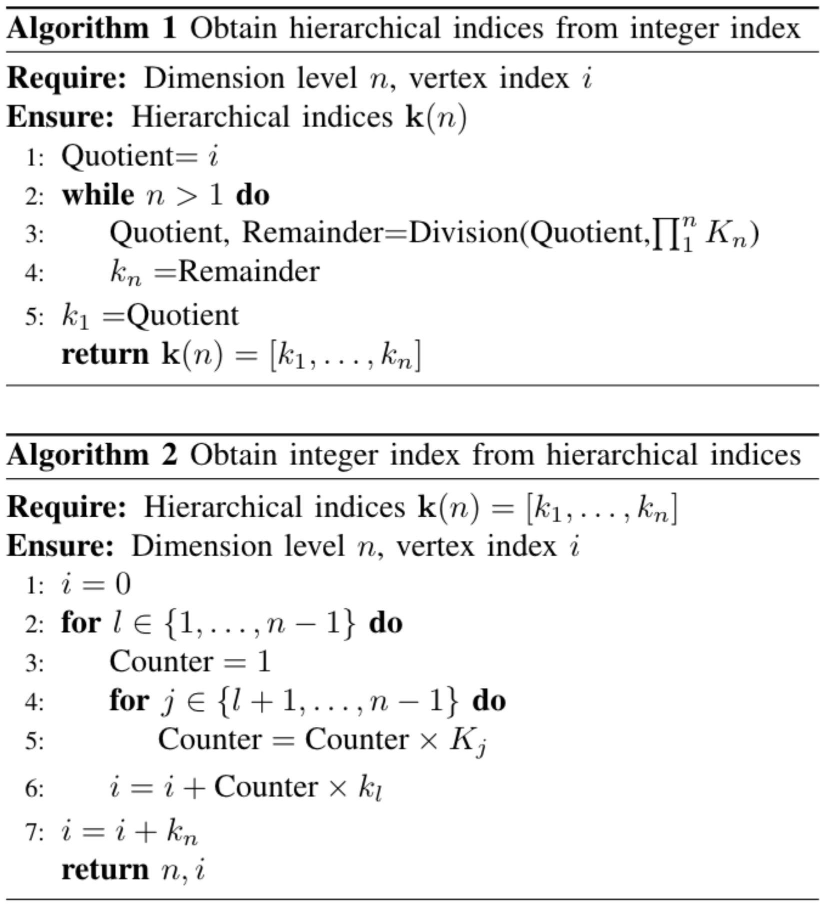


Figure A.5: Swept volumes between configurations [44]

Appendix B

Further Algorithms



Algorithm 1 Obtain hierarchical indices from integer index

```
Require: Dimension level $n$, vertex index $i$
Ensure: Hierarchical indices $\mathbf{k}(n)$
1: Quotient$= i$
2: while $n > 1$ do
3:     Quotient, Remainder=Division(Quotient,$\prod_1^n K_n$)
4:     $k_n$ =Remainder
5: $k_1$ =Quotient
   return $\mathbf{k}(n) = [k_1, \dots, k_n]$
```

Algorithm 2 Obtain integer index from hierarchical indices

```
Require: Hierarchical indices $\mathbf{k}(n) = [k_1, \dots, k_n]$
Ensure: Dimension level $n$, vertex index $i$
1: $i = 0$
2: for $l \in \{1, \dots, n-1\}$ do
3:     Counter $= 1$
4:     for $j \in \{l+1, \dots, n-1\}$ do
5:         Counter $=$ Counter $\times K_j$
6:     $i = i +$ Counter $\times k_l$
7: $i = i + k_n$
   return $n, i$
```

Figure B.1: Algorithms 1 and 2 from the HDRM paper [44]

Algorithm 3 Generate hierarchical occupation lists

```
Require: Robot model $\mathcal{R}$, voxelized workspace $\mathbb{V}$
Ensure: Hierarchical occupation lists $\mathcal{O}_v, v \in \mathbb{V}$
for $v \in \mathbb{V}$ do
    Occupation list $\mathcal{O}_v = \emptyset$
for $n \in \{1, \ldots, N\}$ do
    for $i \in \{1, \ldots, \prod_1^n K_n\}$ do
        $\mathbf{k}(n) = \mathcal{H}(n, i)$
        Set first $n$ joints of $\mathcal{R}$ to $\mathbf{q}(\mathbf{k}(n))$
        if $\{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ are NOT in self-collision then
            $V =$ findBodyOccupiedVoxels$(\mathbb{V}, \mathcal{R}, \mathcal{B}_n)$
            for $v \in V$ do
                $\mathcal{O}_v = \mathcal{O}_v \cup \{(n, i)\}$
        else
            Set vertex $(n, i)$ as default invalid
for $v \in \mathbb{V}$ do
    for $n = N$ to $1$ do
        $\mathbf{O} =$ extractListOfDimension$(\mathcal{O}_v, n)$
        Remove duplicated indices and sort $\mathbf{O}$
        for $O_i \in \mathbf{O}$ do
            if $O_i \bmod K_n = 0$ & $O_{i+K_n} = O_i + K_n$ then
                $\mathcal{O}_v = \mathcal{O}_v \backslash \{(n, p) | p \in [O_i, \ldots, O_{i+K_n}]\}$
                if $n > 1$ then
                    $\mathbf{k}(n) = [k_1, \ldots, k_n] = \mathcal{H}(n, O_i)$
                    $\mathcal{O}_v = \mathcal{O}_v \cup \{(n-1, \prod_{p=1}^{p=n-1} k_p)\}$
                $i = i + K_n - 1$
```

Figure B.2: Algorithm 3 from the HDRM paper [44]

```
// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost
// to reach goal from node n.
INPUTS: start, goal, h

// The priority queue of discovered nodes
// Initialised to start
openSet := { start, 0.0 }

// The closed set: < value, cost >
close[start] := { start, 0.0 }

while openSet is not empty
    current := the node in openSet with highest priority
    if current = goal
        return success

    cost = close[current][1]
    openSet.Remove(current)
    neighbors = getNeighbors(current)
    for neighbor in neighbors:
        // d(current,neighbor) is the weight of the edge
        // from current to neighbor
        new_cost := cost + d(current, neighbor)
        if new_cost < cost or neighbor not explored:
            // Path to neighbor is better than any
            // previous one.
            priority = new_cost + h(neighbor)
            close[neighbor] := { current, new_cost }
            openSet.add({ neighbor, priority })

// Open set is empty but goal was never reached
return failure
```

Figure B.3: Generic A* Search pseudocode, heavily inspired from [1]

```
function reconstruct_path(close, current)
    total_path := {current}
    while current in close:
        current := close[current][0]
        total_path.prepend(current)
    return total_path
```

Figure B.4: Generic A* reconstruction pseudocode, from [1]

```
void get_total_abs_movement_since_init_sample(IndexType ti_sample,
                Eigen::VectorXdRef joint_totals, CloseList& close)
{
    IndexType parent_sample, ti, sample, parent_ti_sample, parent_ti;

    if ((close.find(ti_sample) == close.end()) ||
                (std::get<0>(close[ti_sample]) == ti_sample)) {
        return;
    } else {
        parent_ti_sample = std::get<0>(close[ti_sample]);
    }

    ti_sample_to_sample(ti_sample, ti, sample);
    ti_sample_to_sample(parent_ti_sample, parent_ti, parent_sample);

    Eigen::VectorXd q(data_->params.n);
    Eigen::VectorXd parent_q(data_->params.n);
    get_sample_configuration(sample, q);
    get_sample_configuration(parent_sample, parent_q);

    for (int i = 0; i < data_->params.n; i++) {
        joint_totals(i) = joint_totals(i) + abs(q(i) - parent_q(i));
    }

    if (parent_ti == ti - 1){
        // Found
        return;
    } else {
        // Recurse
        return get_total_abs_movement_since_init_sample
                (parent_ti_sample, joint_totals, close);
    }
}
```

Figure B.5: `calculateTotalJointMovementInTi` function

Appendix C

Further Results

$dt = 0.025$	True Fail	Invalid Int.	Illegal Vel.	Solved	% Success
Streaked HDRM	474	–	–	26	5.2%
Linear PPT-HDRM	10	285	203	2	0.4%
Greedy PPT-HDRM	95	341	0	64	12.8%
Ti-HDRM	0	0	0	500	**100.0%**

Figure C.1: $dt = 0.025$ complex shelf experiment results

$dt = 0.04$	True Fail	Invalid Int.	Illegal Vel.	Solved	% Success
Streaked HDRM	475	–	–	25	5.0%
Linear PPT-HDRM	12	192	293	3	0.6%
Greedy PPT-HDRM	65	320	0	115	23.0%
Ti-HDRM	0	0	0	500	**100.0%**

Figure C.2: $dt = 0.04$ complex shelf experiment results

$dt = 0.025$	True Fail	Invalid Int.	Illegal Vel.	Solved	% Success
Streaked HDRM	188	–	–	312	62.4%
Linear PPT-HDRM	52	211	204	33	6.6%
Greedy PPT-HDRM	21	150	0	329	65.8%
Ti-HDRM	0	0	0	500	**100.0%**

Figure C.3: $dt = 0.025$ simple cube experiment results

$dt = 0.04$	True Fail	Invalid Int.	Illegal Vel.	Solved	% Success
Streaked HDRM	165	–	–	335	71.0%
Linear PPT-HDRM	39	222	197	42	8.4%
Greedy PPT-HDRM	22	130	0	348	69.6%
Ti-HDRM	0	0	0	500	**100.0%**

Figure C.4: $dt = 0.04$ simple cube experiment results

$dt = 0.025$	Invalidations (s)	Search (s)	Total (s)
Streaked HDRM	0.86	0.24	1.10
Linear PPT-HDRM	2.08	0.86	2.94
Greedy PPT-HDRM	2.23	0.78	3.01
Ti-HDRM	3.26	1.01	4.27

Figure C.5: $dt = 0.025$ average time taken results

$dt = 0.04$	Invalidations (s)	Search (s)	Total (s)
Streaked HDRM	0.77	0.26	1.03
Linear PPT-HDRM	1.98	0.83	2.81
Greedy PPT-HDRM	2.18	0.79	2.97
Ti-HDRM	2.99	0.99	3.98

Figure C.6: $dt = 0.04$ average time taken results

dt	Time Slices	Solution Time (ms)	Instructions Executed
Classic HDRM	0	0.50	3,424,735
10	1	2.32	16,584,299
1	10	5.02	37,228,073
0.5	20	6.95	50,943,382
0.25	40	10.93	83,240,382
0.125	80	22.92	488,502,026
0.03125	320	202.69	1,634,030,399
0.015625	640	738.00	4,044,156,546

Figure C.7: Ti-HDRM: Solution times and executed instructions for A* search for different *dt* and slices. All with $duration = 10$s

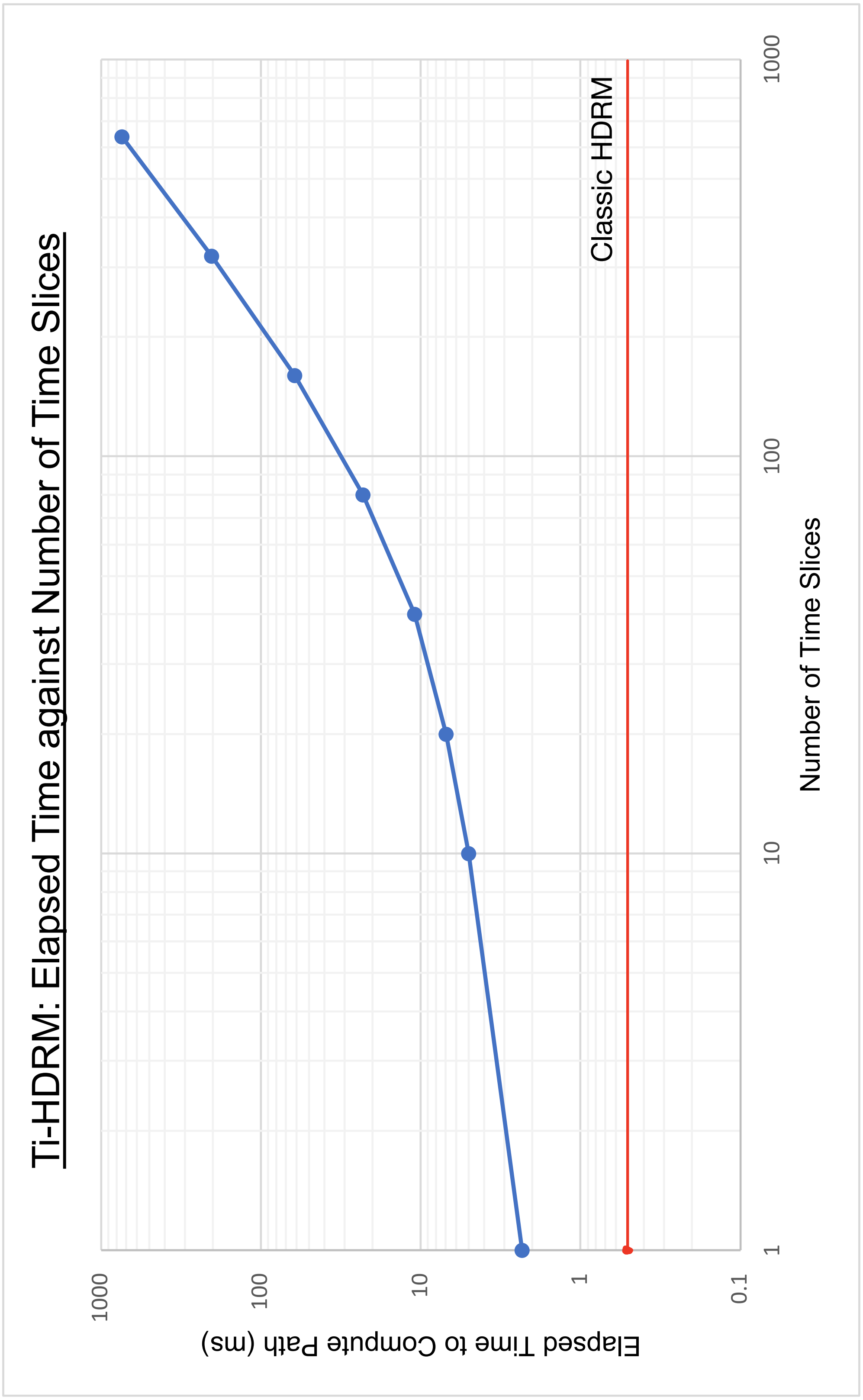


Figure C.8: Ti-HDRM: Elapsed time during `computePath` (log scale)

Appendix D

Reflection

As the author's first experience in undertaking novel research and extending a state-of-the-art method, much of this project was challenging. Fulfilling the criteria of a method that is resolution complete required deep thought and analysis when proposing and working on the algorithm modifications detailed in this thesis - extending Ti-HDRM to the time dimension would be simple if one did not need to take care of completeness!

Working in the large HDRM C++ codebase was also challenging at times, as the code optimisations implemented are not necessarily conducive to the easiest-to-understand (but less performant) vanilla implementation. From grappling with this however, a huge amount was learnt and the author leaves this work as a better C++ programmer.