Hard Real-Time Motion Planning for Autonomous Vehicles

A thesis submitted in fulfilment of the requirements for
the degree of Doctor of Philosophy

Andrew B. Walker
B.Eng, B.App.Sc

Faculty of Engineering and Industrial Science
Swinburne University

November 2011

Abstract

Motion planning is the problem of finding a continuous collision free path from an initial configuration (or state) to a goal. On board an autonomous vehicle, the capability to prevent collisions also depends on the sensing and control. Critically, motion planning is a necessary component for the safe operation of an autonomous vehicles. This work focuses on the problem of embedding a motion planner into a software system in which all tasks are required to satisfy hard deadlines. The key contribution of this work is to present a framework that can be used as a basis for incorporating approximately complete motion planning algorithms within a hard real-time system. This framework ensures that deadlines can be satisfied, with only mild constraints on other software. Two motion planning algorithms are presented as a verification of the framework, and are applied to solving dynamic motion planning problems related to the safe and efficient navigation of autonomous systems.

Acknowledgements

Firstly, I would like to thank my supervisory team, Francis Valentinis and Alireza Bab-Hadiashar, for their advice and patience during my candidature. I am particularly indebted to Francis Valentinis, who has continued to support and encourage me, even after a transition from Swinburne to DSTO.

This research program was made possible by Maritime Platforms Division of the Defence Science and Technology Organisation. I would like to thank Janis Cocking and Kevin Gaylor for their leadership with MPD, and Rodger Neil for his leadership of the unmanned underwater vehicles research program, and to all three for supporting me through my candidature.

During my time at DSTO I have been supported by a number of staff members. In particular I appreciate the core trials team, Max Coxhead, Max Bentley, Peter Graham and Paul Dixon, who allowed me to see DSTO's AUVs in action. DSTO are also blessed by an exceptional library staff, and I appreciate all the assistance I have received from the staff at both the Maribyrnong and Fishermans Bend libraries.

I have been enriched by the opportunity to work with a broad range of talented people at DSTO, amongst them Rob Jarvis, John Wharington, Royce Smart, Justin Young, Brad Knox, Tony Travers and Alex Skvortsov. Without their encouragement and willingness to listen, this thesis would have been much poorer and have taken even longer to write. In particular I need to thank Hugh Torresan for making time in his schedule to proof-read so many drafts.

This research was funded in part by an equivalent Australian Postgraduate Award from the Swinburne Faculty of Engineering and Industrial Sciences and a top-up scholarship from the Defence Science and Technology Organisation.

Finally, I need to thank my wife Sharon, without whose love and support none of this would have been possible.

Declaration

Signed

Andrew Walker
November 2011

Contents

List of Figures

Table of Symbols

$\mathcal{U}$	The space of all control inputs
$\mathcal{X}$	State space
$\mathcal{C}$	Configuration Space
$\mathcal{C}_{obs}$	The subset of configuration space that is in collision
$\mathcal{F}$	The subset of configuration space that is collision free
q	A configuration
$\mathcal{X}_{ric}$	A region of inevitable collision
τ	A parameterised path or trajectory
S	The set of tasks in a real time system
T	The period of a real-time task
D	The deadline of a real-time task
C	The execution time of a real-time task

Acronyms

AUV	Autonomous Underwater Vehicle
CFG	Control Flow Graph
CPU	Central Processor Unit
DMP	Dynamic Motion Planning
DSTO	Defence Science Technology Organisation
EDF	Earliest Deadline First
HRT-DMP	Hard Real-Time Dynamic Motion Planning
HRT-RRT	Hard Real-Time Rapidly-exploring Random Tree
ILP	Integer Linear Programming
LPM	Local Planning Method
PRM	Probabilistic Road Map
RM	Rate Monotonic
RRT	Rapidly-exploring Random Tree
RRT-LPM	Rapidly-exploring Random Tree with a Local Planning Method
RTOS	Real-Time Operating System
SLAM	Simultaneous Localisation and Mapping
UUV	Unmanned Underwater Vehicle
UAV	Unmanned Aerial Vehicle
WCET	Worst Case Execution Time

Executive Summary

A challenging problem in the development of unmanned and autonomous systems is to develop navigation strategies that are efficient, by minimising metrics such as transit time and power usage, while also minimising the risk of collisions. This problem is subject to uncertainty and partial information regarding the state of the vehicle, the obstacles, and the responses of the vehicle to inputs. Robust strategies for safe and efficient navigation require replanning to compensate for uncertainty and changes in the environment. The success of such strategies is dependent on the quality of sensing, planning and control, and on the temporal interactions between those tasks.

The consequences of failing to perform sensing, planning or control within a suitable deadline can result in failure to observe obstacles, produce safe paths or accurately track a path. Consistent failure to meet deadlines can have serious effects including poor performance with respect to mission objectives or loss of vehicle, such as when a collision occurs.

This work focuses on the demonstrating that it is possible to find safe and efficient motion plans in dynamic environments, subject to hard limits on execution time. This is a significant addition to the field of motion planning, where historically algorithms were not designed to satisfy hard real-time constraints.

This claim is addressed by developing a framework that can be used to incorporate approximately complete motion planning algorithms into a hard real-time system. By varying the amount of search performed by such algorithms, it is possible to find an acceptable tradeoff between search quality and time required.

This framework is demonstrated by using a novel planning algorithm - the *Hard Real-Time Rapidly-exploring Random Tree* (HRT-RRT) to find safe and efficient plans when subject to time constraints. This result is achieved by using execution time measurements and static analysis to predict maximum resource consumption patterns.

Further results are used to demonstrate techniques that can be used to improve capability to produce safe and efficient plans. This is achieved by adopting planning techniques that can be shown to make better use of the time available for planning. By estimating elapsed execution time while such planners are running, it is possible to maximise efficiency, while still satisfying hard deadlines.

This work is significant in that it presents a novel approach to dynamic motion planning which removes common assumptions regarding temporal interactions between motion planning, sensing and control. Furthermore, it provides new planning strategies that can be used to optimise the quality of search performance, even in the presence of hard deadlines.

CHAPTER 1

Introduction

The last decade has seen a great increase in the adoption of remotely operated and unmanned systems by both defence and commercial users [Dep04, BAN09]. Such systems have the benefit of allowing a human operator to perform a task in potentially hazardous working environments, and have the potential to reduce the man-power and cost required for operations.

Ongoing research in the field of unmanned systems has a strong focus on developing autonomous systems, which require less direct human supervision and control than remotely operated vehicles. The intention of this work is to develop systems that can operate independently, over extended periods of time, to achieve objectives selected by a human operator. Operators can then make better use of their time by focusing on control and co-ordination, further reducing costs and improving the usefulness of unmanned systems.

In Australia, the Defence Science and Technology Organisation (DSTO) is researching novel autonomous systems which have the potential to support future generations of navy platforms. Ongoing research at DSTO has included the development of Autonomous Underwater Vehicles (fig. 1.1), and research into safe and efficient navigation for such vehicles. This work has the potential to reduce risk exposure of naval personnel and equipment, and to extend and complement existing capabilities.

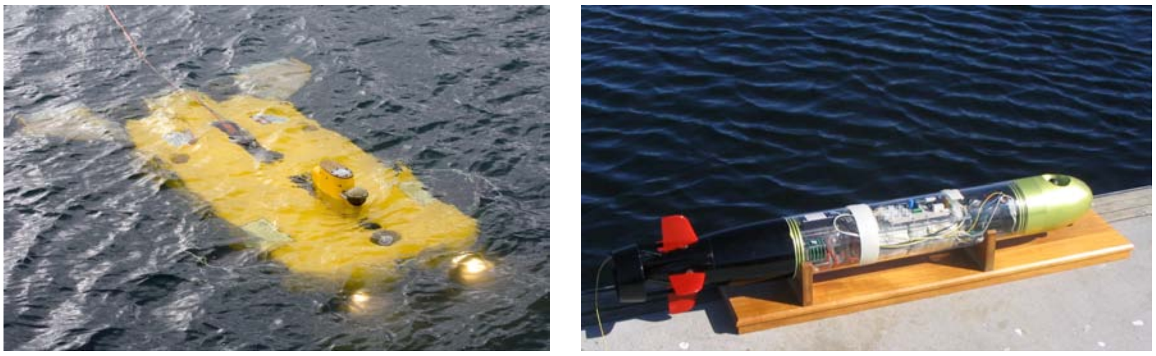

Figure 1.1: DSTO's Autonomous Underwater Vehicles. Left: Wayamba; Right: Mullaya. Images courtesy of DSTO

In practice, the capability to satisfy mission objectives depends on the interactions and performance of sensing, planning and control. Experience with real-world vehicles such as the Predator Unmanned Aerial Vehicle (UAV) and the REMUS [ASA^{+}97] Autonomous Underwater Vehicle (AUV) has demonstrated that the task of developing robust autonomous systems is notoriously challenging.

Failing to satisfactorily achieve the necessary level of sensing, planning or control means an autonomous system may either fail to perform acceptably with regard to the mission objectives or collide with an obstacle, causing damage to the vehicle.

1.1 Safe Navigation

Safe navigation depends on being able to plan paths or trajectories that prevent collisions with obstacles. In the case of autonomous vehicles these obstacles can be either static, such as terrain or buildings, or dynamic, such as other moving vehicles.

In cases where an environment contains only static obstacles the task of finding safe paths can be addressed by solving a motion planning problem [LaV06]. A plan produced in this way is only safe if the environment is fully observable and does not change over time, a case which does not occur except in highly structured workspaces.

More realistic environments would include obstacles that move over time and obstacles that must be detected using sensors. This more difficult task can be addressed by replanning when the environment changes, a process known as dynamic motion planning.

Using an analogy from control theory, motion planning can be thought of as *open-loop*, while dynamic motion planning can be considered its *closed-loop* equivalent. This highlights that the replanning is used to minimise errors resulting from unforeseen environmental changes. It is also useful to consider the task of safe navigation as a cycle of sensing, planning and control (actuation) (Figure 1.2).

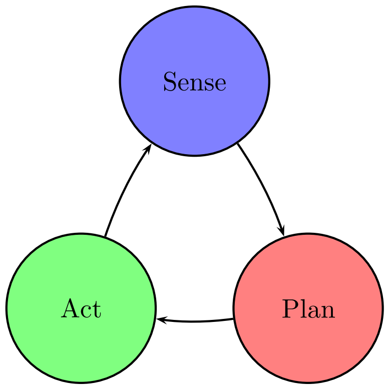

Figure 1.2: Sensing, planning and control tasks

The sensing task is responsible for mapping and localization. Mapping involves measuring, and in some cases predicting, locations of obstacles based on sensor observations. Localization is the task of finding the configuration (location and pose) of the vehicle. The quality of localization and mapping estimates is dependent on the types and properties of the sensors used. A significant external factor which influences localization and mapping relates to how quickly new observations are acquired, as compared to how quickly the vehicle moves.

The actuation task takes the sensed location of the vehicle and the path output of the motion planner, and produces inputs to the actuators. These inputs are chosen so as to minimise path-tracking errors. The capability to track a path accurately depends on a number of factors including: the magnitude of environmental disturbances, sensors, actuators and the implementation of the controller, including the rate at which the controller is updated at.

The planning process uses a single mapping and localization estimate to produce a collision free path from the current configuration, to a goal configuration, determined based on the current mission objective. While planning is being conducted, no new information can be taken into account.

The capability to navigate safely depends on the quality of sensing, planning and control, and the interaction between those tasks. Interactions between tasks include direct coupling due to transfers of inputs and outputs, and indirect coupling that results from time taken for each task to execute. Excessive delays by any task could result in a collision.

1.2 Time Critical Planning

The combination of minimal computational resources, high cost of failure through collision and inter-coupling between the software tasks necessitate careful consideration of software design alternatives on-board autonomous vehicles.

The need to support long-term missions with minimal human interaction imposes constraints on the design of autonomous systems. In particular, vehicle designers must find an appropriate trade-off between payload, energy storage, endurance, sensing, actuation and computing capability. This work focuses on the impact that constraints on the quantity and quality of computing resources impose on the software on-board autonomous systems.

Constraints on computational resources tend to introduce trade-offs between performance and other factors that tend to be less critical in traditional software design. For example, an open issue in energy aware computing involves finding a suitable compromises between the power consumption and speed required to execute a piece of software [Tiw94, Kan08]. Similarly, safety critical software systems are required to produce results within a time limit or risk catastrophic failure. With regard to planning for autonomous vehicles, failure to identify new plans within an acceptable time limit can potentially result in a collision, and for this reason, the thesis focuses solely on the hard real-time aspect of the computing constraints.

The temporal inter-coupling which occurs between sensing, planning and actuation is a phenomenon that is frequently ignored in planning literature. The rare examples in which the problem is considered [BV02, HKLR02], make the assumptions that minor collisions are acceptable. In this work, the cost of a collision of an autonomous system makes such an assumption unacceptable.

Perhaps the best known results for planning under time constraints are those based on anytime algorithms [BD89]. These algorithms progressively refine an initial solution, improving quality as time permits. Such an approach is less applicable in cases like motion planning subject to kinematic and dynamic constraints, where finding an initial solution is time consuming.

Rigorous techniques for the verification of safety critical systems have been developed for systems with multiple concurrently executing tasks [BW01]. The limitation of these approaches is that they are typically only applied in cases where there is minimal variation in timing performance of each task.

1.3 Claim

The lack of algorithms for implementing dynamic motion planning subject to hard real-time deadlines highlights a significant deficiency in the current state of the art for dynamic motion planning.

Work on real-world autonomous systems is targeted towards deployment in complex, unstructured, time-variant and partially observable environments. The need to achieve safe navigation in these types of environments necessitates the use of dynamic motion planning algorithms.

Safe operation of an autonomous system requires that critical tasks (sensing, planning and control), execute in a timely manner. Failure to do this can lead to errors in localization (vehicle state), mapping (obstacle state and geometries), path tracking, and most importantly for this work, path generation. Sufficiently large errors in any of these areas can result in degradation in mission performance, or in the worst case, failure due to collisions.

In order to be used as a component of a safety critical system, a hard real-time dynamic motion planner must satisfy limits imposed on computational resources such as memory and execution time. Furthermore, to ensure the all tasks in the system will be able to satisfy deadlines, those resource limits are *hard* - failure to satisfy a resource limit is considered to be unsafe.

The claim of this thesis is that *it is possible to produce motion planning algorithms which are both safe and efficient in dynamic environments that satisfy hard limits on available computational resources*.

1.4 Contributions

This thesis makes contributions related to bridging of concepts from hard real-time systems and dynamic motion planning. This contribution is a significant extension to existing work in the field of motion planning because the time required to find a plan can limit its safety and effectiveness. In a safety critical system, failure to satisfy hard time constraints is considered unsafe. Similarly, failure to perform efficient level of search can result in a planner being considered unsafe.

The first contribution of this work is to provide a formal definition for this new special case of the dynamic planning problem, *Hard Real-Time Dynamic Motion Planning (HRT-DMP)*. This class of problem describes both the planning requirements and the available computational resources, with a particular focus on execution time. Planners that can satisfactorily address this problem are then considered capable of safe planning in dynamic environments, subject to hard constraints on the available computational resources.

The major contribution of this thesis is a framework for producing dynamic motion planning algorithms that can be proven to satisfy hard real-time constraints and achieve an acceptable level of search. This framework addresses two key challenges. Firstly, it describes an approach for embedding a motion planner into a real-time task, with measurable computational resource consumption. Critically, this technique must address both the interactions between the planner and other tasks, as well as the performance of the planner in isolation. Secondly, it describes a series of metrics for evaluating planning performance. By varying the parameters of the planner it is possible to identify the case which maximises performance.

In order to demonstrate the framework in practice, a novel planning algorithm, the *Hard Real-Time Rapidly-exploring Random Tree* (HRT-RRT) is presented. HRT-RRT is an example of an algorithm which addresses the HRT-DMP problem, and is therefore an example of a motion planning algorithm that provides safe plans in dynamic environments when subject to time constraints. This result is achieved by using execution time measurements and static analysis to predict maximum resource consumption patterns. By varying parameters of the planner, a new method for maximising search while respecting a bound on execution time is demonstrated.

A second demonstration of the framework is made in the form of a generalisation of HRT-RRT which includes popular planning optimisations. The contribution of this work is to show that naive application of these optimisations in a hard real-time system has minimal performance benefit. By using the results of instrumentation it is possible construct new types of planners which exhibit good average search performance, while still satisfying hard deadlines.

These contributions demonstrate that it is indeed possible to perform safe and efficient motion planning in dynamic environments, even when there is hard bound on the time available for planning. The proposed approach offers a practical method for managing the trade-off between the time available for planning, and the quality of the search results that can be produced.

1.5 Overview

The thesis is broadly organised as follows. The early chapters present relevant background material. They are followed by a chapter outlining the approach to hard real-time motion planning presented in this thesis. This approach is then applied by considering specific hard real-time motion planners. The final chapter concludes the discussion. A description of each chapter in the thesis is given below.

Chapter 2 presents the background material relevant to the planning portion of this thesis. Firstly the chapter introduces the background and mathematical nomenclature required for understanding this work. This is followed by a catalogue of planning methods which have been used for, or influenced, the development of dynamic planning strategies.

A discussion of scheduling and real-time systems is presented in Chapter 3. This work focuses on presenting the requirements on any task in a hard real-time system, particularly with regard to estimating worst-case execution time.

Chapter 4 presents a framework for developing hard real-time motion planning algorithms. This work also includes a discussion on measures of performance which should be considered when developing hard real-time planners.

A concrete example of hard real-time planning algorithm based on the rapidly exploring random tree (RRT) planning algorithm is presented in Chapter 5. The analysis of this algorithm considers the implications of using this algorithm in dynamic environments. Finally, validation of the temporal characteristics are presented.

Chapter 6 focuses on considering how to optimise the search characteristics described as a hard real-time search. This is demonstrated by showing the impact a common optimisation has on time required and search performance of the RRT algorithm. A final planner is then presented which overcomes the problems associated with scheduling planning algorithms with complex execution time profiles.

CHAPTER 2

Motion Planning

Research into path and motion planning has origins in a wide range of fields including optimal control, artificial intelligence, computational geometry and robotics. Although many of the tasks presented in this document have been studied for nearly 50 years, this area remains the focus of a significant amount of research. Reviews of the most significant parts of planning literature are presented in *Robot Motion Planning* [Lat91], *Planning Algorithms* [LaV06], and *Principles of Robot Motion Planning* [CMH$^+$05].

Safe and efficient navigation is a task of significant interest in robotics, and particularly mobile robotics. It involves finding a continuous path that can be followed from an initial configuration (or state) to a goal configuration. A safe path is one that prevents collisions with obstacles, and an efficient path is one which minimises cost. In real-world environments, safe navigation depends on *sensing*, *motion planning* and *control*.

One of the challenges of safe navigation relates to the quantity and reliability of information available about the environment. Put simply, if there are inaccuracies in the estimated location of the vehicle or obstacles, then there is risk that a collision will occur. To ensure that an accurate model of the environment is available, vehicles are equipped with sensors capable of measuring the workspace around the vehicle. Sensor observations are used to identify the current state of the vehicle, *localisation*, and to identify the location of portions of the workspace which contain obstacles, *mapping*. The combined tasks of localisation and mapping are typically described as Simultaneous Localisation and Mapping (SLAM) [LDW91], and are a major topic in robotics research today.

At a given time, given the current best estimate of the location of the vehicle and obstacles, it is possible to consider the task of finding an open-loop collision-free path from the current vehicle state to a goal state. Any errors in localisation or mapping could result in paths being planned through obstacles. Furthermore, old plans can become invalidated if new obstacles are observed, or if the vehicle cannot accurately track a path.

Vehicle control involves attempting to accurately follow a planned path or trajectory. To this end, the outputs to the actuators are determined so as to minimise the deviation of the state of the vehicle from the specified path. There are several well known approaches for solving this problem, including modern and classical control techniques [Oga01]. In the field of control, other excellent resources are available that consider the issues related to the control for flight [Bla91, SL03], and underwater [Fos94] vehicles.

In the context of planning, safe navigation involves an incremental process of sensing, planning and control. Failure of any of those subsystems to provide accurate or timely results can result in a collision.

Having provided a brief overview of the context in which autonomous planning occurs, the remainder of this chapter focuses on the fundamentals of the motion planning.

2.1 Definitions

The geometry of a motion planning problem is described, at least initially, in terms of a workspace. The workspace, $\mathcal{W}$, can be thought of as a mathematical representation of the physical space, in which the boundaries of both the vehicle and the known obstacles are encoded. The workspace is normally represented as a three dimensional Euclidean space $\mathbb{R}^3$.

The geometry of vehicles and obstacles within the workspace can be represented in a variety of ways. Typically these methods include either polygonal models such as triangular meshes, or semi-algebraic models.

Within the workspace, the position of the vehicle can be encoded as a *configuration*, q. A vehicle's configuration can be described by an n-dimensional vector where the vehicle has n degrees of freedom. For example, a rigid body moving in the plane has a configuration vector with three elements $[x, y, \theta]^T$ - a two dimensional position and a heading; a rigid body in three dimensions can be represented as a configuration vector with six elements $[x, y, z, \phi, \theta, \psi]^T$, a three dimensional position, and an orientation specified in terms of Euler angles (yaw, pitch and roll).

For a given vehicle configuration, the vehicle occupies a certain portion of the workspace, $\mathcal{A}(q) \subset \mathcal{W}$. The portion of the workspace occupied by a set of m stationary obstacles can be described as $\mathcal{B}_i \subset \mathcal{W}$ for $i \in [1, 2, ..., m]$. The total affect of all of the obstacles in the workspace can be expressed as $\mathcal{B} = \bigcup_{i\in[1,m]} \mathcal{B}_i$. This leads to the definition that a *collision* occurs if the area occupied by the vehicle would intersect with that of any of the obstacles, that is if $\mathcal{A}(q) \cap \mathcal{B} \neq \emptyset$.

The breakthrough work in path planning was the paper by Lozano-Pérez [LP83], who showed that path planning problems can be formulated in terms of a *configuration space*, $\mathcal{C}$, which allows the geometric and topological complexities of all planning problems to be modelled using a consistent set of mathematical tools. Configuration space consists of all of the possible valid configurations, but it can be further refined into two subsets, the set of configurations that the vehicle can safely occupy, which is the free space $\mathcal{F}$, and the set of configurations which, when occupied, would result in collision, $\mathcal{C}_{obs}$.

By transforming the problem from one of planning in the workspace to a configuration space, the task of motion planning becomes one of finding a continuous path. A *path* can be expressed as a parameterised continuous map, $\tau : s \rightarrow q$, where the initial configuration is q_i at $\tau(0)$ and the final configuration is q_f at $\tau(1)$. In order for a path to be collision free, all configuration $\tau(s)$, for all $s \in [0, 1]$, must be in $\mathcal{F}$. In cases where the configuration space changes over time, it is customary to refer to a path parameterised by time as a *trajectory*.

If a system to be modelled is subject to kinematic or dynamic effects, then it is necessary to consider the derivatives of configuration during planning. For example, a model that includes kinematics and dynamics will result in a state space $\mathcal{X}$, where each state, $x \in \mathcal{X}$ encodes a configuration, q, velocity, $\dot{q}$, and acceleration, $\ddot{q}$ such that $x \in (q, \dot{q}, \ddot{q})$.

2.1.1 Motion Planning

Motion planning, is the task of finding a continuous collision free path, where that path starts at an initial configuration, and ends at a goal configuration. The basic motion planning problem is sometimes known as the piano movers problem [SS83b, SS83a].

Definition 1. *Given the geometry of the vehicle, $\mathcal{A}$, and the obstacles, $\mathcal{B}$, motion planning is the problem of finding a path, $\tau(s)$, from an initial configuration, $\tau(0) = q_{init}$, to a final configuration, $\tau(1) = q_{goal}$, such that all configurations on the path are not in a collision state, that is, $\tau(s) \in \mathcal{F}$ for all $s \in [0, 1]$, or correctly report that no such path exists.*

The requirement in Definition 1 that a motion planner will return a path, if a feasible path exists, or will return failure otherwise, is commonly known as *completeness*. There are some well known strategies for implementing complete planning algorithms, however, it is well documented in planning literature [LaV06, HA92, TSK07] that complete planning algorithms tend to be computationally expensive in practice, and can be difficult to implement.

A popular approach for motion planning is to use an approximately complete algorithm. These algorithms trade completeness for performance. In some cases the planner may fail to identify that a solution exists. The benefit of these strategies is that they tend to find solutions more quickly than complete planning algorithms in practice. This work makes extensive use of motion planning algorithms that make use of these weaker forms of completeness.

Basic motion planning includes the assumption that the magnitude of the path tracking error is very close to zero. Under such conditions, even minor deviations have the potential to make a path unsafe. One approach to mitigate this risk is to enlarge the geometric size of the obstacles in the workspace by the magnitude of path error. This is sometimes termed obstacle dilation, or gross motion planning [HA92].

Later parts of this chapter will consider practical approaches that can be used to solve motion planning problems.

Constrained Motion Planning

One of the most important variations of motion planning occurs in cases where a vehicle's motion can be expressed as a dynamic system. In its most general form, such a system can be expressed in the form of equation 2.1.

$$\dot{x} = f(x, u) \tag{2.1}$$

where $x \in \mathcal{X}$ is the state of the vehicle (following the definition of state from section 2.1), and $u \in \mathcal{U}$ is the control input, drawn from the set of all possible control inputs. It is assumed that the sets $\mathcal{X}$ and $\mathcal{U}$ are bounded manifolds of dimension n and m, which can be treated of subsets of $\mathcal{R}^n$ and $\mathcal{R}^m$ by defining appropriate charts on the manifolds [CMH$^+$05]. From a practical standpoint, the set of control inputs can be thought of as a vector of inputs to actuators or a controller.

An open-loop trajectory can be constructed by simulating the response of the dynamic system 2.1 to a function of (open-loop) control inputs that vary over time.

$$\tau(t) = \int_0^t f(x, u(t))dt \tag{2.2}$$

This representation of a trajectory for a dynamic system is parameterised in terms of time, t. As such, the problem of constrained motion planning is to find a time variant set of control inputs $u(t)$, which result in the vehicle going from x_{init} at $t = 0$, to x_{goal} at some future time, $t = t_f$, while ensuring that $\tau(t) \in \mathcal{F}$ for all times, $t \in [0, t_f]$.

In order to model physical systems, the structure of equation 2.1 is potentially subject to a number of constraints[GP01]. The two common types of constraints that arise are non-holonomic constraints and differential constraints.

These constraints become more apparent when considering the simple unicycle model, a wheel that rolls, without slipping, on a horizontal plane. This model can be described by Equation 2.3, where a configuration is $[x, y, \theta]$, a position and heading, and the control input space is $[v, \omega]^T$, a forward and rotational velocity.

$$\begin{aligned} \dot{x} &= v \cos\theta \\ \dot{y} &= v \sin\theta \\ \dot{\theta} &= \omega \end{aligned} \tag{2.3}$$

A system is said to be under-actuated if there are fewer controls than degrees of freedom, and so the unicycle model is under-actuated. In the case of the unicycle model, it is not directly possibly to translate in a direction perpendicular to the line of motion of the wheel, but it is possible to achieve the same result by composing a number of actions. This is an example of a non-holonomic constraint. From a planning perspective, these constraints further complicate the task of finding paths linking configurations, even without obstacles. Significant detail on non-holonomic and under-actuated systems and their impact on planning is given in [LaV06, CMH$^+$05].

The problem of differential motion planning comes about by imposing constraints on derivatives of configuration. These constraints make it possible to express limits on the velocity and acceleration of a vehicle. Kinematic and dynamic differential constraints can be written in the form of Equations 2.4 and 2.5 respectively.

$$F(q, \dot{q}) \leq 0 \tag{2.4}$$

$$G(q, \dot{q}, \ddot{q}) \leq 0 \tag{2.5}$$

In the case of planning for autonomous vehicles, differential constraints can be used to model physical phenomena. For example, flight vehicles that stall at low velocity can be modelled by imposing a constraint on minimum forward velocity.

2.1.2 Dynamic Motion Planning

The standard formulation for motion planning is based upon the assumption that the locations of all obstacles are known prior to planning. If there is incomplete knowledge of the locations of obstacles, or if obstacles are free to move, motion planning is not sufficient to prevent collisions[1]. Dynamic motion planning is an analogue of motion planning suitable for configuration-spaces that change over time.

Time variant models of a workspace can be characterised by how predictably they evolve over time. When perfect knowledge of the paths of all obstacles is known in advance, the configuration-space is said to be *deterministic*. *Non-deterministic* models are those which include an estimate of uncertainty. In cases where a workspace changes non-deterministically, it is possible to use sensors to observe changes and reduce uncertainty.

Observability in the context of a planning problem refers to whether the knowledge of the locations of obstacles is complete. The basic formulation of motion planning makes the assumption that the workspace is *fully observable* - the locations of all obstacles are known

[1]it is possible to solve a special case of fully observable deterministic configuration spaces, using a motion planner in a configuration-time space [vdB07]

a-priori. In practice it is more likely that an configuration-space is *partially observable*, and that new obstacles must be detected using sensors.

The primary difference between motion planning and dynamic motion planning is that for the latter, the locations of the set of obstacles change over time. To this end, the obstacles are modelled as a time variant set $\mathcal{B}(t)$, which represents the best available estimate of the obstacle geometries and locations. Where necessary, the paths of obstacle can be modelled as a function of time, and uncertainty can be accounted for by enlarging boundaries.

Definition 2. *Given the geometry of the vehicle, $\mathcal{A}$, and the time evolutions of the geometry of all obstacles, $\mathcal{B}(t)$, dynamic motion planning is the problem of finding a path, $\tau(s)$, from an initial configuration, $\tau(0) = q_{init}$, to a final configuration, $\tau(1) = q_{goal}$, such that all configurations are on the path are not in a collision state $\tau(s) \in \mathcal{F}$ for all $s \in [0, 1]$, or correctly reports that no such path exists.*

Definition 2 ensures that if a path is found at a time t, that path will be safe. It does not account for obstacles that were unobservable at time t, nor does it provide a mechanism for reducing uncertainty as sensors provide more information about the environment. In order to reduce uncertainty, dynamic motion planning involves replanning when the environment changes.

The simplest approach to replanning involves periodically producing a new plan, assuming that these problems are uncorrelated. Effectively, at each time-step a new motion planning problem is solved using current localization and mapping information. A number of more sophisticated replanning algorithms have been proposed that exploit *temporal coherence*, the expectation that over short intervals, the state of the environment and vehicle will have only changed by a small amount, which allows results to be shared between successive iterations of the planner.

2.1.3 Graph Theory

One possible approach for solving motion planning problems involves finding a discrete approximation of the configuration space. Typically, such an approach involves constructing a graph, sometimes referred to as a *roadmap*, $G = (V, E)$, where the set of vertices is $V = \{v_0, v_1, ..., v_n\}$, and the set of edges is $E = \{e_0, e_1, ..., e_m\}$, where each edge is a pair $e = (v_a, v_b)$. A reduction of a motion planning to a discrete formulation on a graph allows the problem to be simplified back to finding a shortest path in the graph.

In addition to the mathematical description of the graph, there is also an underlying physical and geometric meaning. Each vertex in the graph is associated with a configuration that is in a collision free subset of configuration space. Furthermore, an edge (v_a, v_b) may only exist if there is an admissible collision free path between the configurations of v_a and v_b.

By adopting a graph-based representation of configuration space, it becomes possible to succinctly capture physical relationships. For example, by requiring edges in the graph to be directed, it is possible to denote that some paths in the configuration space can be traversed in one direction. Similarly, variations in the capability of a vehicle to traverse an edge can be captured by describing G as a weighted graph.

Given that the search graph G is a discrete representation of collision free portions of Configuration Space it is important to capture the critical information which will allow shortest path question to be answered. Following the definitions of [GO05], such a graph will achieve good *coverage* of $\mathcal{F}$ and capture its *connectivity*.

Definition 3. *G covers $\mathcal{F}$ when any configuration in the free space, $q \in \mathcal{F}$ could be connected by a simple (straight line) collision free path to a vertex $v \in V$.*

Definition 4. *G is maximally connected when for all vertices $v_a, v_b \in E$, if there exists a path in $\mathcal{F}$ between the configurations associated with v_a and v_b, then there exists a path in the graph between v_a and v_b.*

Coverage is a measure of how well distributed the milestones are over the free space. Coverage implies that the configurations in G are located at points which maximise the amount of configuration space that is reachable.

Connectivity is a measure of how well the roadmap reflects the connectivity between components in the workspace. Connectivity in this sense does not refer to connected-ness, whether the graph has disjoint sub-graphs, although this is geometrically related issue. In the case of constrained motion planning tasks, connectivity is divided into accessibility and departability, [LaV06].

2.1.4 Complexity Theory

One issue of concern in working on planning algorithms is to gain an understanding of the factors which influence how long it takes to solve a query. This period of time is of particular importance in being able to determine whether a particular query is solvable within a timing deadline.

The best known method to analyse the running time of an algorithm is to consider the asymptotic performance of the algorithm as the size of the input approaches infinity [CLRS01]. Time complexities are typically expressed in Big-O notation. Algorithms with constant, linear and quadratic running times dependant on a size N, would be respectively denoted $O(1)$, $O(N)$ and $O(N^2)$.

If an algorithm has worst-case time complexity that is dependent on the size of the input, there will exist some input that will result in a given deadline being exceeded. The running time of planning algorithms can be influenced by a number of problem specific factors including the number of obstacles, the geometric representations of the vehicle and obstacles and the dimensionality and size of the subset of configuration space used for planning. Constraining all of these parameters is a necessary step to ensure that the problem is solvable in finite time.

A key research area in the field of motion planning has been the study of the computational complexity of classes of motion planning problems, as distinct to algorithms. Of particular interest are lower bounds on complexity. A number of well known bounds, have been established for particular formulations of path and motion planning problems. For example, general motion planning problem is PSPACE-hard [Rei79]. Such high lower bounds mean that complete motion planning algorithms are likely to be impractical when subject to (tight) timing deadlines [LaV06].

2.1.5 Reachability

A fundamental concept which underlies proofs of safety for dynamic motion planning problems is associated with the capability of a vehicle to reach various parts of the configuration space. In the case where the open-loop motion of a vehicle can be predicted exactly, it is possible to use set theory to describe all of the possible configurations that are *reachable* from an initial configuration. The description of reachability presented here is based heavily on the definitions from [LaV06] and [Hsu00].

More formally, a state x_f is reachable from x_i, if there exists a control function $u(t)$ that produces an admissible control input at each time. Applying these inputs results in an (admissible) trajectory from x_i to x_f. The union of all states reachable from x_i are described as the *forward reachable set* and is given by equation 2.6.

$$R(x_i,\mathcal{U}) = \{x_f \in \mathcal{X} \mid \exists\, u \in \mathcal{U} \text{ and } \exists\, t \in [0,\infty) \text{ such that } x(t) = x_f\} \tag{2.6}$$

A similar principle can also be used to develop a *time limited forward reachable set* (eq. 2.7), which is the equivalent of the forward reachable set over a finite time horizon.

$$R(x_i,\mathcal{U},t) = \{x_f \in \mathcal{X} \mid \exists\, u \in \mathcal{U} \text{ and } \exists\, t \in [0,\tau) \text{ such that } x(t) = x_f\} \tag{2.7}$$

The primary reason for considering forward reachable sets is as a tool for demonstrating the conditions under which a state can be considered *safe*. In configuration space, it is sufficient to state that a configuration q is safe provided that it is not in collision, $q \in \mathcal{F}$. In state space, kinematics or dynamics may result in a state that is collision free, but regardless of path followed, a collision is at some time in the future. Such states are said to lie in $\mathcal{X}_{ric}$, the *region of inevitable collision*, if a collision will occur at some time in the future regardless of the trajectory followed. This can occur because a vehicle becomes trapped in a dead-end, or because of the effect of momentum.

For non-trivial state spaces, there is no way to decide whether a given state lies is in $\mathcal{X}_{ric}$ [LaV06], however, there is research into weaker bounds on safety [Fra01, PF05b]. In a workspace where the locations of obstacles can be predicted, a simple approximation to $\mathcal{X}_{ric}$, is to prove that a collision is not inevitable within a finite time horizon of T. For an initial state, x_i, any path in $R(x_i,\mathcal{U}, t + T)$, can be followed for a time of t, and be guaranteed that a time of T remains before any collision will occur. Frequently T is made sufficiently large so as to facilitate replanning.

2.2 Motion Planning Algorithms

This section surveys some of the most important planning techniques that have influenced the development of planners deployed on complex autonomous vehicles, and algorithms which are known to be capable of planning in dynamic environments.

2.2.1 Reactive Planning

Historically, the capability of employing motion planning algorithms on-board real vehicles or robotic systems has been hampered by having insufficient computational resources to employ the sense-plan-act paradigm in a dynamic workspace. The compromise of *reactive* planning involves restricting attention to the subset of the workspace near the vehicle. This allows the planner to very rapidly select paths or actuator commands that will prevent a collision, with the side effect that they are incomplete, that is, they may fail to reach a target configuration.

Artificial Potential Fields

Artificial Potential fields [Kha85] are an approach to solving the motion planning problem, based on the analogy of the motion of a particle in an electro-magnetic field. Potential fields were originally applied to finding paths for manipulators in robotic work-cells, but have since been applied to a wide range of problems, particularly navigation of mobile robots

A potential vector field associates a *potential* with each configuration. A potential field is a differentiable real valued function which measures how desirable it is to be in a given configuration. Planning in such a domain can be performed by using a gradient descent approach. One possible formulation of a potential field is to combine an attractive field, U_{att}, with a global minimum at the goal and repulsive fields, U_{rep}, for each of the obstacles. Summation of these fields results in a total potential field.

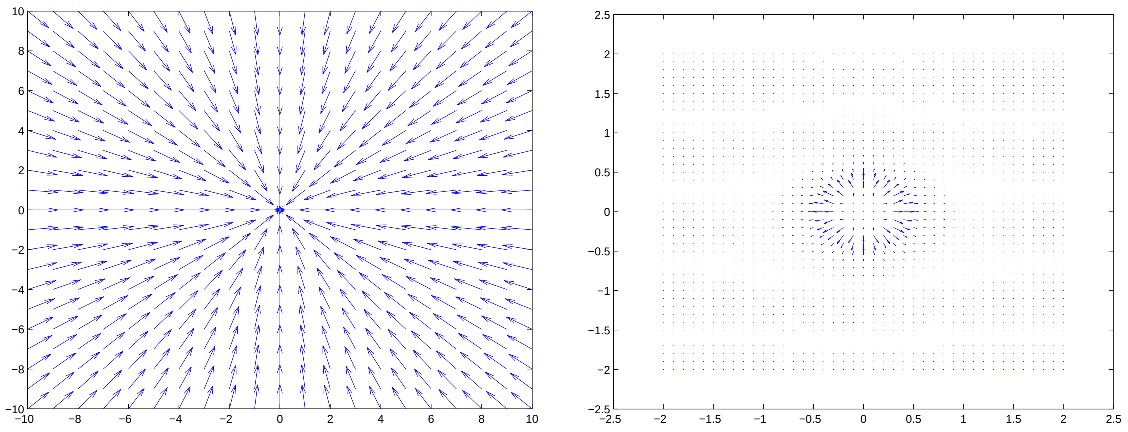

Figure 2.1: The gradient of potential fields, visualised as vector fields. Left: the potential induced by a goal; Right: the potential field induced by a single obstacle.

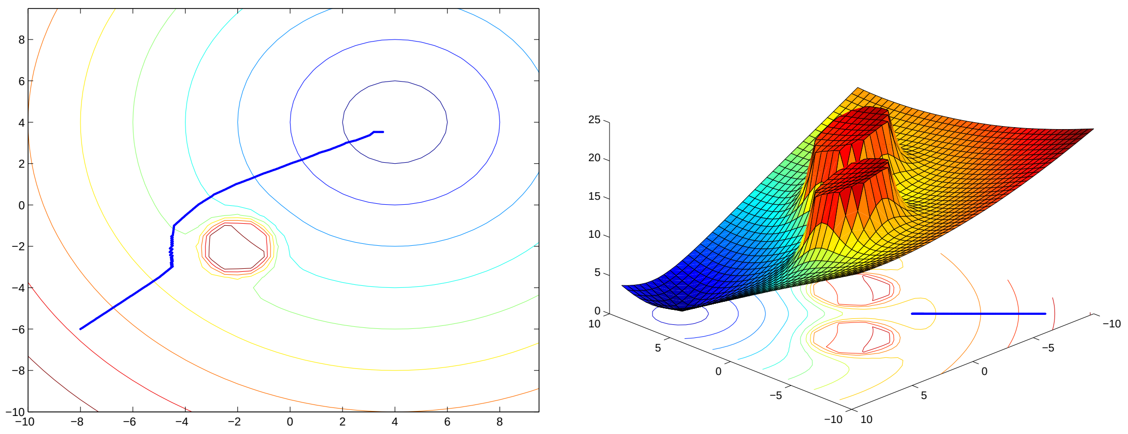

Figure 2.2: Potential field planning, with potential visualised using contour plots, and the path followed by the vehicle in blue. Left: a path to the goal. Right: a local minimum in potential prevents a goal being reached.

Although potential fields have been shown to work for some types of planning, the challenges in adapting them for use in realistic and complex problem domains are well known [KB91].

Perhaps the most important shortcoming of potential fields is that they only consider a local subset of the configuration-space, they can fail to make progress toward the goal if the vehicles state enters a *local minima* in the potential field, that is if $|\nabla U(x)| < \epsilon$, but x is not the goal state. Becoming trapped in a local minima in this fashion is a limitation of gradient descent search techniques [RN03]. This phenomena is shown graphically in Figure 2.2.

Another variation on the potential field class of potential field-like planners are Motor Schemas [Ark87]. Motor schemas generalise the concept of a potential field for obstacle avoidance and reaching the goal to allow any behaviour to be encoded into the planning system. This allows path-tracking and search type behaviours to be considered at planning time. In the context of more complex vehicles, these capabilities can be more easily managed by control or mission planning software.

Borenstein and Koren [BK91] demonstrated the Vector Field Histogram (VFH) planner, which is a practical implementation of a potential field planner using a real robotic vehicle. The limitation of this approach is intended for use on-board vehicles with particular arrangements of sensors and configuration space representations. The aim of this work is to facilitate planning for a broad class of systems, and as such, this technique is ignored from further consideration.

One of the most significant extensions to potential fields is the randomised potential field [BL91]. Rather than relying purely upon a local information to choose a control action, the randomised potential field planner can generate a series of random walks if it detects that it has become trapped in a local minimum. The completeness of this planner is still limited by the length of the random walk.

Subsumption

One of the earliest sensor based planning techniques is the subsumption architecture [Bro86]. This approach was developed to show that it was possible to develop a robust and flexible architecture for vehicle control, based on inspiration from the way biological systems respond to environmental stimuli. In practice the subsumption architecture was one of the first experiments that demonstrated the capability of low cost mobile robotic platforms to avoid obstacles in complex dynamic environments [Bro89].

The underlying goal of the subsumption architecture is to provide a robust control strategy. This is achieved by dividing the overall problem of robot control down into a number of simpler independent tasks. Each task is then addressed by a behaviour, which is effectively a function which maps sensor inputs to control outputs. Those behaviours are then executed concurrently, and the one that is active with the highest priority influences the vehicles actuators. For example, the lowest level (zero-th level) behaviour tends to be a random search (commonly termed a *wander*) through the environment. This behaviour is suppressed in cases when a higher level behaviour activates. This can occur in cases where a collision avoidance behaviour activates to change the direction of the vehicle when sensors detect an imminent collision.

In this way, the subsumption architecture provides a convenient mechanism for composing and prioritising behaviours. One of the effects of this composition is that it allows developers to construct higher level behaviours without explicitly having to consider low-level details, effectively, a control strategy *emerges* from the time- and sensor-based interactions with the world.

The subsumption architecture is an attractive approach for vehicle control in that it is robust

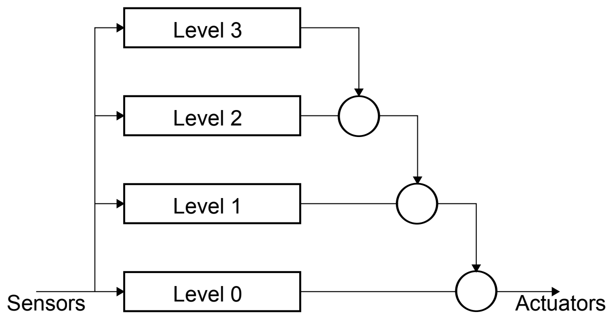

Figure 2.3: An example of the layered hierarchy of behaviours used for the basic subsumption architecture. Each of the behaviours takes inputs from the sensors and provides actuator outputs. The circular nodes denote points at which a higher level behaviour can subsume (or suppress) lower level behaviours.

and allows for rapid responses to environmental changes and has been adapted for use onboard real autonomous vehicles such as Sea Squirt [BCBH90], URIS [Car03] and Oberon [WNR$^+$01].

The most significant limitation of the subsumption architecture, and behaviour based control and planning techniques is due to issues of scaling. The effort required for validating the emergent control strategy grows proportionally to the factorial of the number of behaviours [BL00], which can make testing infeasible.

2.2.2 Deliberative Planning

Deliberative planning is an approach to planning that focuses on finding a complete path to the goal, rather than reactive planners which only consider the local subset of configuration space near the vehicle. Typically, deliberative planning involves generating a search graph or tree, using the concepts described in section 2.1.3. Once this representation is available, the planning problem can be solved using a shortest path algorithm.

Common techniques for constructing a search graph include roadmap based techniques, cell decompositions, and sampling based algorithms [Lat91, LaV06, CMH$^+$05]. In practice, the selection of a technique for constructing the graph will heavily influence the time required for planning, and subsequent replanning.

The identification of the shortest path on a search graph is well understood problem, and optimal solutions can be found using the A* algorithm [HNR68]. The A* algorithm has been described and analysed in a number of works, including [RN03, CLRS01]. The fundamental difference between graph based planning techniques is shown to be related most strongly to the approach used to construct the search graph.

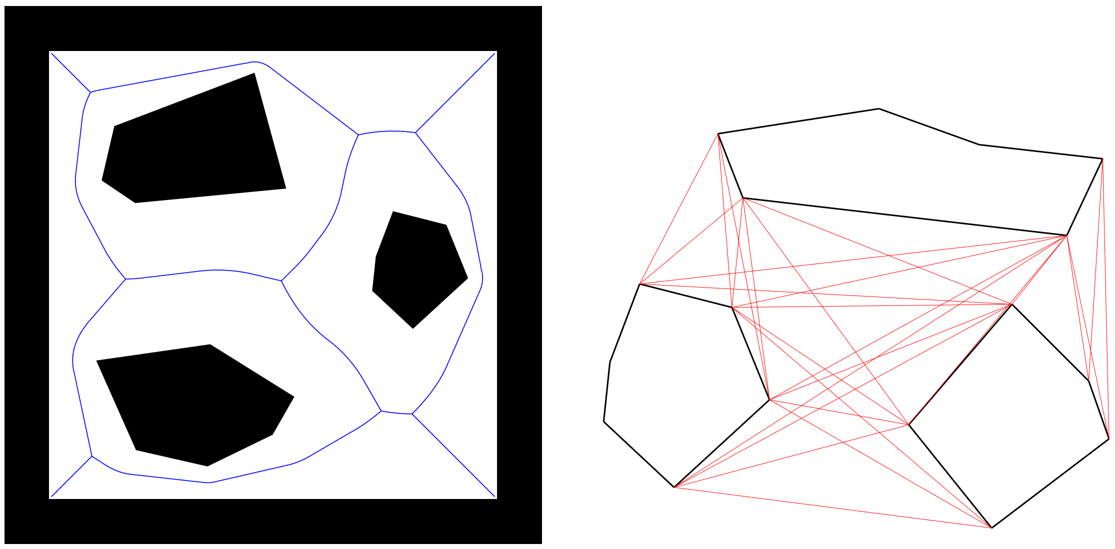

Figure 2.4: An example of the outputs of the roadmaps produced by two types of combinatorial algorithms. Left: the roadmap produced by using a Generalised Voronoi Decomposition. Right: a visibility roadmap.

Combinatorial Algorithms

Combinatorial motion planning algorithms [LaV06, Lat91], are a general class of algorithms which can be employed to construct a roadmap that will result in the completeness guarantee being satisfied. That is, provided that a solution exists, these algorithms will find it.

This section is included because combinatorial motion planning algorithms provide a clear method for visualising key concepts, particularly with respect to ideal formulation of a roadmap, and the inter-related concepts of coverage and connectivity.

A visibility roadmap, Figure 2.4, is a combinatorial algorithm that can be used to construct a roadmap in a configuration space which can by described by a series of polygonal segments. The resultant graph G can be formed where each vertex $v \in V$ also exists in the (polygonal) boundary of $\mathcal{F}$. Any pair of vertices in G for which there exists an admissible path, is added to the set of edges. This results in the roadmap providing both coverage and maximal connectivity.

The visibility roadmap places G as close as possible to obstacles, the converse approach is to place the edges, and vertices, in G so as to maximise the distance from $\mathcal{C}_{obs}$. This can be achieved by constructing a Generalised Voronoi Diagram using a retraction approach [Lat91, CMH$^+$05]. The GVD roadmap is shown in Figure 2.4. This approach also ensures that coverage and maximal connectivity are achieved.

These algorithms are considered unsuitable for use in dynamic planning situations for because in many cases they have performance characteristics which makes them unsuitable for online planning, particularly in high dimensional spaces [LaV06]. This performance is in part related to the computational complexity of the problems being solved (see section 2.1.4).

Approximate Cell Decomposition

Perhaps the simplest way to construct a roadmap of the collision free portions of the configuration space is to formulate G based on a decomposition of the configuration space. One way to do this is by dividing $\mathcal{C}$ into cells. Each cell can then be tagged as being in $\mathcal{F}$, or partially or fully in $\mathcal{C}_{obs}$. The graph G can be formed by introducing a new vertex for every cell, and edges for pairs of adjacent cells that can be connected by a collision free path.

An obvious limitation of a cell decomposition is that the discretization process can enlarge obstacles only partially in $\mathcal{C}_{obs}$. The result of this is that the cell decomposition method is *approximately complete* - coarse grids may fail to identify some feasible solutions.

A second limitation of such approaches is that while planning using fine decompositions is ideal, the trade-off comes in the form of the size, and time, required to store and search the graph. The graph effectively grows exponentially in the dimension of the configuration space, that is $|V| = n^d$, where n is the number of divisions for each of d dimensions.

A final limitation of approximate cell decompositions is that they are only appropriate in cases where there are no kinodynamic constraints on the path of the vehicle. In cases where these constraints need to be implicitly considered during planning, similar results can be achieved by using a lattice. A lattice is a regular sampling of the configuration space, in which edges are feasible motions which connect states exactly. Construction of state lattices is an ongoing area of research [LBL04, PKK09].

D*

One of the limitations of the A* algorithm is that in cases where the costs of the edges in the graph change, replanning involves a new search over the entire graph. A variant of the A* designed specifically for this situation is the D* algorithm [Ste94], which retains search information between planning episodes to improve search performance.

D* can be used on any weighted graph, a simple example of planning can be demonstrated on a two dimensional 8-connected grid. In planning nomenclature, this is effectively a graph formed from an approximate cell decomposition. Edges in the graph that would pass through an obstacles have infinite weight. This formulation allows newly observed obstacles to be incorporated into the graph.

The D* algorithm and its derivatives have been used as a basis for planning on a number of important wheeled vehicles including NASA Mars rovers [SSS^{+}00, PKK09]. Additionally, the algorithm has also been demonstrated on-board the Manta AUV [MWM03].

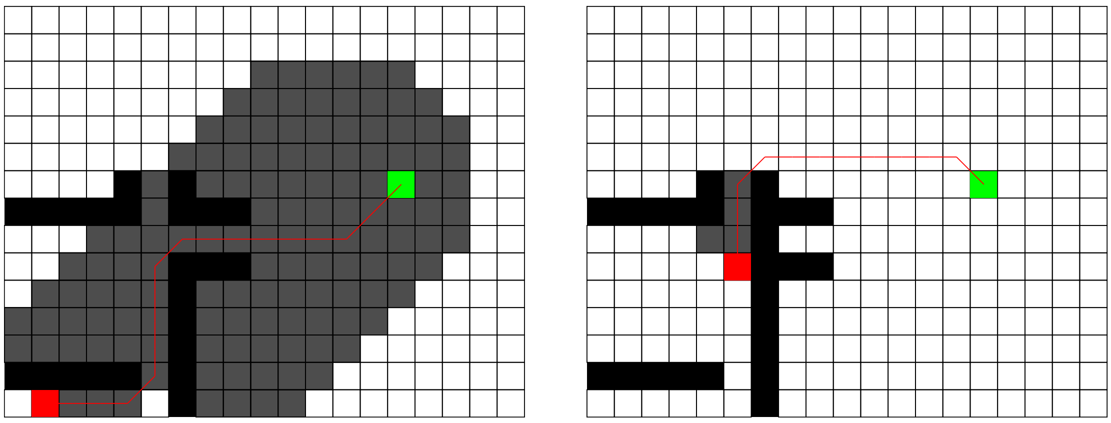

Figure 2.5: Paths found during two separate planning episodes. The black cells are obstacles, and the grey cells in both images illustrate which states were expanded during the search. Left: the initial path. Right: replanning after a newly observed obstacle obstructs the path.

From the perspective of this thesis, the most important contribution extending the D* algorithm has been the development of D*-lite [LK02, KL02, KL05]. D*-lite exhibits the same exploration characteristics as D*, and has the advantage of being simpler to analyse and extend. The key finding of this work has been to highlight that the worst case performance of D* involves two expansions of each state in G [KL02]. Practical experiments have shown that in cases where there are significant changes to the search graph it may be more efficient to plan from scratch [Lik05].

Real-Time Search

Real-time search, also described in some work as agent centred search [Koe01], is the task of planning when the time available for finding a solution is limited, possibly subject to partial knowledge of the search space. Such problems arise when failure to find a solution within a time-window limits the effectiveness of that solution. For example, certain types of chess tournaments are played under time constraints, under such conditions, it is desirable to choose the best known move, rather than having to concede by failing to meet a time limit.

Much of the work on the field of real-time search has origins in Korf's Learning Real-Time A* [Kor90] (LRTA*). This work highlighted that A* must search all the way to a solution before commitment to a first movement, and proposed an alternative method for circumventing this limitation. The algorithm performs partial searches and transitions to new states in an iterative fashion. The search component of LRTA* works by searching a finite number of nodes using the same priority based system as A*. Rather than planning from scratch, the costs of states are retained, effectively learning cost-to-goal from each state. Selection of an action, or partial path, can then be chosen so as to minimise the cost-to-goal. This ensures that the planner will backtrack, rather than becoming trapped in a local minima.

In order for LRTA* to be complete, the problem space is required to be made up of a finite number of states, that is $|V|$ is bounded. Furthermore, there must exist a path from every state in G to the goal state. This second requirement implies that G must not contain any one-way edges with dead ends. Significantly in the case of constrained motion planning - if a vehicle can enter a subset of $\mathcal{F}$ from which it can't leave (a dead-end), then it is no longer possible to reach the goal.

A second key result of this work, building on Pearl's earlier work on Iterative Deepening A*[Pea84], was to show that the running time for LRTA* can be controlled by restricting the algorithms *search horizon*. Effectively, this means that LRTA* shares much in common with Anytime algorithms (Section 3.3.2).

2.3 Sampling Based Motion Planning

One of the key limitations of planning over lattices and cell decompositions relates to the *curse of dimensionality* [Bel57]. As the dimensionality of the configuration space increases, exponentially more space is required for storage of the graph. One method for addressing the curse of dimensionality is to identify a representation of collision free portions of configuration space with a graph that is sparse (relative to a dense grid or lattice). Effectively this means finding a graph G with a small number of vertices and edges, that encodes all of the significant topology of $\mathcal{F}$.

As is the case with many problems subject to the curse of dimensionality, practical solutions can be found by introducing some degree of randomness. In the case of motion planning, such algorithms are termed *sampling based motion planning algorithms*. Reviews of the sampling based planning algorithms are included in [TSK07, LaV06, CMH$^+$05].

The general approach used by sampling based motion planning algorithms is to identify a number of *milestones* in $\mathcal{F}$. This can be achieved by randomly, or pseudo-randomly, selecting configurations in $\mathcal{C}$, and then using the information about the robot and obstacle geometries to identify whether the configuration is in collision. Acceptable sampling strategies include any approach that results in all of $\mathcal{F}$ eventually being adequately covered [LL03]. The simplest sampling strategy is uniform sampling, more sophisticated techniques include biasing sampling towards parts of the configuration space.

Once milestones have been identified, the search graph can be constructed by finding feasible paths that link milestones. One common example of an approach used involves attempting to connect each milestone to its k-nearest neighbours. Further examples of connection strategies are discussed in the context of specific planners below.

By modifying the approach used for selection and connection of milestones it is possible to produce planners appropriate for solving a number of problems. It is worth highlighting that these methods are only *approximately complete*, or more specifically *probabilistically complete*. This means that as the number of milestones and edges in the search graph increase, the probability of finding a solution, if one exists, approaches one.

2.3.1 Probabilistic Roadmaps

Probabilistic Roadmaps (PRM) [KSLO96, Ove92] are perhaps the best known form of sampling based motion planning. The most basic implementation of the algorithm follows from the description in the previous section. A graph G is constructed by randomly sampling $\mathcal{C}$, until n milestones have been identified in $\mathcal{F}$. An attempt is then made to connect each of the n milestones to their k-nearest neighbours, where *nearness* is assessed using a metric function ρ. This process is summarised in Figure 2.6.

```
1:  procedure BUILD-PRM($n, k$)
2:      $V \leftarrow \emptyset, E \leftarrow \emptyset$
3:      for $i \in [1, n)$ do
4:          $q \leftarrow$ SAMPLE-CONFIGURATION
5:          while $q \notin \mathcal{F}$ do
6:              $q \leftarrow$ SAMPLE-CONFIGURATION
7:          $V \leftarrow V \cup q$
8:      for $q_0 \in V$ do
9:          for $q_1 \in$ KNEARESTNEIGHBOURS($q_0, \rho$) do
10:             if COLLISIONFREE($q_0, q_1$) then
11:                 $E \leftarrow E \cup (q_0, q_1)$
```

Figure 2.6: Pseudo-code for the Probabilistic Roadmap algorithm

An example of a characteristic roadmap produced by PRM is shown in Figure 2.7. The green points in the image denote the milestone configurations, while the grey circles are obstacles.

Once the search graph has been generated by the PRM algorithm, motion planning queries can be solved quite rapidly. The initial and goal nodes of the query, q_i and q_g are added to V, and connected to their k-nearest neighbours. At that point it is possible to find a shortest path between q_i and q_g using a graph search algorithm. Effectively PRM solves the *all-pairs* motion planning algorithm, in that once the roadmap has been generated it can be used to solve planning queries between any pair of configurations.

Much of the research related to PRM relates to development of practical methods for reducing the time taken to solve particular types of planning queries. One example of this is the Lazy PRM [Boh99, BK00]. It is well known that a large proportion of the time required to generate a roadmap is time spent in collision tests. In cases where only a single motion planning query needs to be solved (the *single-pair* problem), only edges which are potentially

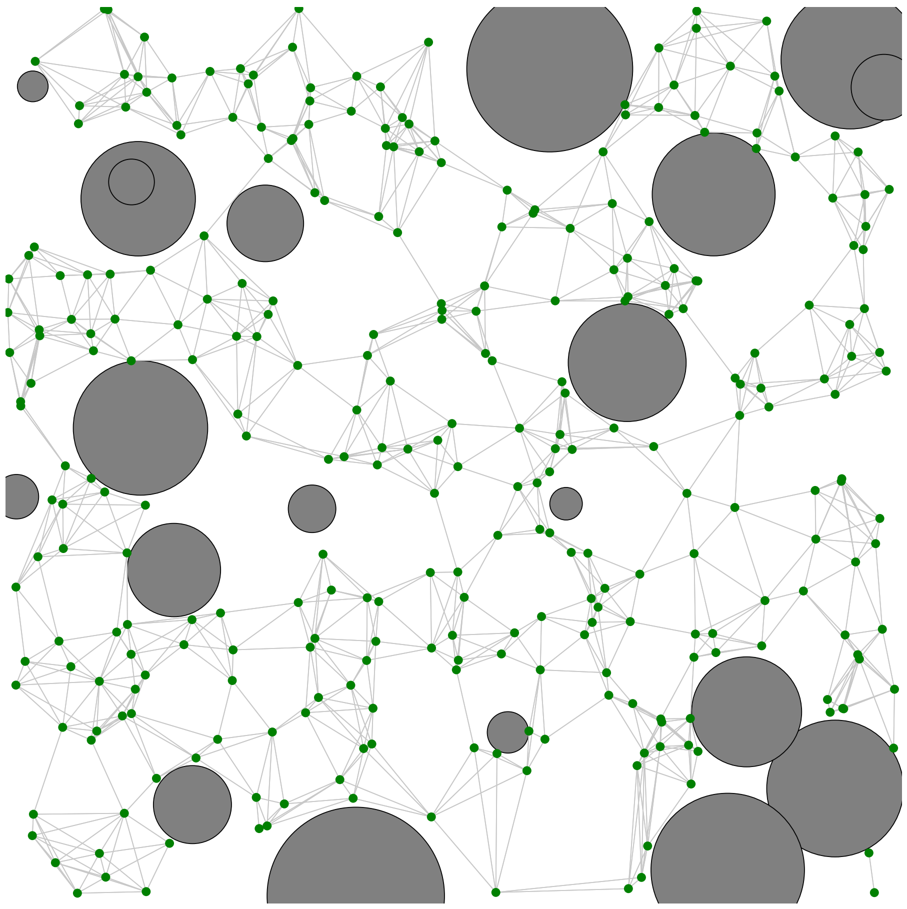

Figure 2.7: An example of a Probabilistic Roadmap formed in a two dimensional configuration space with randomly placed obstacles.

on the shortest path need to be tested for collision. Lazy PRM improves on the performance of PRM for the solution of single queries by delaying collision detection until the shortest path search.

Another method for improving on the performance of PRM, that is also applicable to all sampling based motion planners in general, relates to optimising the search for nearest neighbours. The naive approach for identifying the nearest neighbours within a roadmap is an $O(n^2)$ process. Use of appropriate data structures, such as KD-Trees, have been shown to provide a mechanism for significantly reducing the overhead of this operation [BV02, YL02, YL07].

Much of the literature on sampling-based motion planning focuses on the impact of various sampling techniques. Sampling influences the connectivity of the roadmap, which is particularly noticeable in configuration spaces with narrow passages. Significant amongst this work is the review [LB02], which concludes that low dispersion of samples throughout $\mathcal{C}_{free}$ is the critical factor in determining how effectively a roadmap can be constructed.

One of the challenges for applying PRMs to planning for an autonomous vehicle is related to the method used for linking configurations. To be applicable, there must be some way to identify a feasible, but not necessarily collision free, path. For unconstrained planning problems this is possible, however the introduction of kinodynamic or non-holonomic constraints may result in this being a non-trivial task. Finding a feasible path linking a pair of configurations for a constrained vehicle model requires the solution of a two point boundary value problem [LaV06].

Another factor which limits the usefulness of the PRM algorithm in dynamic environments is that full knowledge of the locations of the obstacles are required before planning commences. In the case of dynamic environments, the lack of this information requires that roadmaps are either periodically regenerated or edges be re-tested for collision. Approaches like Lazy-PRM mitigate this overhead, tree based planning strategies have come to dominate dynamic planning [LaV98].

2.3.2 Other Methods

In this section a range of other sampling based motion planning methods of relevance are considered [2].

A significant challenge for approximately complete sampling based motion planning algorithms is that the time required to converge to completeness varies greatly. In practice, it has been observed that the rate at which these algorithms converge depends on features of the configuration space, and the local planning method employed. For example, features such as narrow passages tend to make finding solutions more difficult, the extension of a search graph in large steps makes planning easier. These concepts are encapsulated in the work of Hsu et al. [Hsu00], who prove that by restricting attention to planning problems in *expansive spaces*, the probability of failing to find a solution decreases exponentially in the number of milestones sampled. Work on these proofs have also be backed up by experimental work with the Expansive Space Tree motion planner [HKLR02]. The authors state in that work that there are a number of challenges in practical implementations.

Work by Petti and Fraichard on Partial Motion Planning (PMP) [PF05b, PF05a] demonstrated the requirements for safe replanning in dynamic environments. In cases where time limited approximately complete planners are used, it is possible that a path to the goal may not be found. It is possible to provide a degree of safety by showing that even a partial path can be proven to be safe if it does not end in the region of inevitable collision. This is achieved by identifying the reachable set of the vehicle and proving that sufficient time remains for the vehicle to select an alternative plan which preserves safety. Similar approaches have been described by Hsu et al. [HKLR02], Frazzoli [Fra01], and have been extended to be applicable in domains where the vehicles path is kinodynamically constrained [Sch06]

[2]with the exception of the RRT method which is considered in detail in chapter 5

One of the most important pieces of work in integrating real vehicle dynamics with motion planning is the Manoeuvre Automaton [FDF00b, FDF00a, Fra01]. One of the difficulties of dealing with realistic vehicle models relates to the size of the state and control spaces. One method of reducing this space is to characterise a set of possible trim trajectories, with constant velocities and control inputs and a set of manoeuvres of finite duration between trims. The result of this formulation is that the open-loop motion of the vehicle can be reduced to a sequence of trims and manoeuvres which are amenable to use in the task of learning (near-) optimal local planning methods. Frazzoli also demonstrated that the availability of a optimal local planning method can be exploited to improve the cost of a plans, once an initial estimate on the cost upper bound is identified. In this thesis, the more fundamental task of finding an initial path, and hence an initial upper bound on cost, is considered both with and without optimal local planning methods.

CHAPTER 3

Real-Time Systems

In their simplest form, real-time systems are simply systems which have some dependence on the time at which results are produced. This chapter is intended to give a brief overview of real-time systems, as pertinent to the design of hard real-time motion planners. There has been extensive research in the field of hard real-time systems, and the notation in this chapter is based on material from Burns and Wellings [BW01], with additional notation from Puschner and Schdel [PS97].

In computer science, real-time computing has come to be associated with the task of generating results as quickly as possible. In planning literature, real-time search algorithms are associated with a limited amount of deliberation being performed between actions [Kor90]. In this work, as in the field of safety critical systems, real-time systems are defined in a fundamentally different manner [Sta88, Shi94].

Definition 5. *A real time system is any system in which successful operation depends not only on producing correct results, but also on the time required to produce those results.*

A real-time system can be categorized according to the consequences of failing to meet timing requirements. Failures can occur because results are not produced within a deadline, or, results are delivered late.

- A *hard real-time* system is one in which failure to meet deadlines can result in catastrophic failure. The automotive, avionics, medical and nuclear industries are obvious examples of fields in which hard real-time systems are present. Failure to satisfy deadline can result in catastrophic equipment failure or loss of life.

- A *firm real-time* system will accept the results of some tasks being late. The consequence of late delivery is that results may no longer be meaningful.

- A *soft real-time* system can tolerate missing of deadlines, resulting in a degradation of performance. Multimedia systems, such as those used for streaming video are frequently cited as examples of systems which can tolerate some late arrival of results.

In cases where motion planning is considered a safety or mission critical piece of software, that is *hard real-time*, timing requirements can be used to guarantee that safe and feasible plans can be produced sufficiently quickly to avoid collisions. Furthermore, a hard real-time navigation system can be used to show that time dependent inputs from sensor and outputs to the actuators can be considered sufficiently frequently to provide guarantees of both safety and effectiveness, even in dynamic environments.

3.1 Definitions

Although at any given time, only a single task can be running on a processor, it is possible for the execution of all tasks to be interleaved (and thus appear to execute concurrently), by employing pre-emptive multi-tasking. In such a context, a common formulation of a real-time system, S, is as a fixed set of n independent *tasks*, each of which is responsible for completing a time dependent behaviour. In practice, each of the tasks is associated with either an operating system thread or process.

$$S = \{\omega_1, ..., \omega_n\} \tag{3.1}$$

In order to describe the system in more detail, it is necessary to define the temporal characteristics of each of the tasks.

$$\omega_i = (T_i, D_i, C_i) \tag{3.2}$$

where T_i is the minimum time between subsequent releases, C_i is the worst-case execution time (WCET) and D_i is the deadline of the task.

Within the context of a real-time system, tasks tend to be long-running. New partial computations are started by releasing a task. The minimum interval between releases T_i is a measure of how frequently the task is required to execute. For the case in which a task executes periodically, T_i is the *period* between releases. For an aperiodic task, T_i is the maximum frequency that a triggering event can cause a release.

The *deadline* of a task is the amount of time after a task release by which all results must be available. Being able to provide results on or before a deadline is the fundamental objective of designing reliable real-time systems.

Execution time is a measure of how long a particular task has been executing for in a particular period. The *Worst-Case Execution Time*, C_i, is the maximum amount of time that will ever be required by a task to produce a result.

In the case of a fixed set of periodically scheduled tasks, the periods, deadlines and periods for a set of tasks is constrained by $C_i \leq D_i \leq T_i$. In particular, in order for each task to satisfy a deadline, $C_i \leq D_i$. Under some conditions it is possible for sets of aperiodic tasks to meet deadlines even if $T_i \geq D_i$ [BW01], but such cases are ignored in the remainder of this work.

The values of the period and deadline for each task are chosen so as to satisfy the requirements of the system. The Worst-Case Execution Time of a task must however be measured or predicted. A failure occurs within a hard real-time system in cases where the result of a task is produced after a deadline. Validation of the correctness of hard real-time system can be addressed using *schedulability analysis*, discussed in more detail in Section 3.1.3.

3.1.1 Task State

In the context of individual tasks, and real-time systems, it is useful to consider the factors which contribute to execution time. Following the work in [BW01], it is possible to describe each task in a hard real-time system as being in one of three possible states: ready, running or waiting[1]. Critically, at any one time, only a single task can be running. The waiting state is entered if the task is waiting for a shared resource. A task in the ready state is waiting to be restarted.

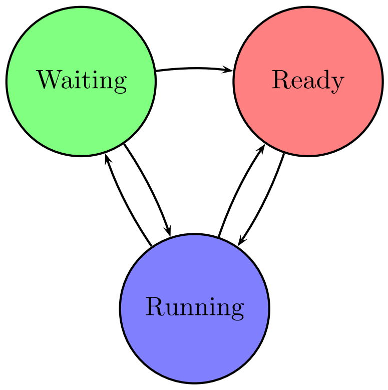

Figure 3.1: The possible transitions between states of a task

The state of a task can change for a number of reasons, a simplified transition diagram is shown in Figure 3.1. Transitions between states can occur for a number of reasons:

- The running task calls a function that *blocks*. This includes using devices for synchronization and mutual exclusion (semaphores and mutexes) and inter-process communication. In the case of the simple model, such calls are disallowed.
- The running task *yields*. If the task completes execution for a given period, control can be yielded back to the scheduler to allow another task to run.

[1]In practice, there are more than three states, but these three states are of fundamental importance while the task is operating normally

- If the running task is *pre-empted*. This occurs if the scheduler determines that another more suitable thread is ready to run.
- If a waiting thread is *released*. This occurs when the resource that a task was waiting on becomes available.

Only the task which is running is consuming execution time. Other factors such as load from other tasks, yielding, or blocking on shared data will, however have an impact on the capability of the system to meet deadlines.

3.1.2 Task Types

A sporadic task is a task which occurs when an external event triggers a release. Such events include interrupts and message handlers. Sporadic tasks typically have $D << T$. In some formulations T does not represent the period, but rather the maximum rate at which triggering events can release the task.

A cyclic task runs at a regular interval, typically with $D = T$. A pseudo-code example of the body of a cyclic task is shown in Figure 3.2. The task performs a once-off initialisation (Line 2), during which all resources are allocated. The task then enters the main loop, which introduces periodicity (Line 4). The last two elements of the loop body yield control back to the operating system, until the next activation of the task.

```
1: procedure CYCLIC-TASK
2:     INITIALIZE-TASK
3:     $t_{next} \leftarrow$ CURRENT-TIME
4:     loop
5:         $t_{next} \leftarrow t_{next} + T$
6:         TASK-BODY
7:         YIELD-UNTIL($t_{next}$)
```

Figure 3.2: Pseudo-code for the structure of a cyclic real-time task

3.1.3 Scheduling

A scheduler is a piece of software that selects which of the tasks should be executed. Commonly, a scheduler will employ the notion of *priority* - high priority threads should be executed in preference to low priority threads. The priority of a thread is dependent on the type of scheduler used. In a *Fixed Priority Scheduler*, all task priorities are assigned prior to execution. In a *Dynamic Priority Scheduler*, priorities can be modified at runtime. A priority based ordering of tasks is used to ensure that deadlines can be met by subset of tasks, or to maximise processor utilisation. Here two types of scheduling algorithms suitable for hard real-time systems are described.

The basis for hard real-time analysis is the Rate Monotonic (RM) scheduling algorithm, proposed by Liu and Layland [LL73]. The priority of tasks under RM is derived from the period of the tasks, where short periods lead to high priorities. Additionally, the temporal behaviour of tasks is assumed to be independent, that is, tasks do not become blocked or participate in inter-task communication, and that the period of all tasks is equal to their deadline $D_i = T_i$.

Rate Monotonic scheduling ensures that if a task misses a deadline, the only tasks effected will be those with a lower priority. Furthermore, the scheduler is optimal in the sense that if any static priority scheduling algorithm can meet all deadlines, then RM can too. Together, these properties make rate monotonic scheduling robust and effective in the scheduling of hard real-time systems.

A simple test of whether the computational resources are sufficient for all tasks to meet deadlines can be derived from utilisation. A tasks utilisation is the ratio of execution time to the deadline. In order for the system to be schedulable, the total utilisation, U, for all tasks must be less than one (eq. 3.3).

$$U = \sum_{i=0}^{N} (\frac{C_i}{D_i}) \tag{3.3}$$

For systems scheduling using the Rate Monotonic priority system the equation 3.4 is a sufficient condition to show whether a system is schedulable. This guarantees that all tasks will have response times less than their deadlines.

$$U \leq N(2^{\frac{1}{N}} - 1) \tag{3.4}$$

Deadline Monotonic scheduling was proposed by Audsley [Aud90] to relax some of the requirements of Rate Monotonic Scheduling. Deadline monotonic only requires that all tasks deadlines are less than their periods $D_i \leq T_i$. The priority of a tasks under deadline monotonic scheduling is based on the deadline, rather than period. Deadline Monotonic scheduling exhibits similar behaviour to RM with respect to failure tolerance.

A more sophisticated approach for schedulability analysis is based on response time analysis [BW01]. This allows the limitation on blocking introduced by RM scheduling to be lifted. The response time of a task is given by equation 3.5.

$$R_i = C_i + B_i + I_i \tag{3.5}$$

where for a task ω_i, R_i is the response time, or maximum time that it is possible for a response to be produced. B_i is the time introduced by blocking, and I_i is the interference, caused by having to wait for other higher priority tasks to finish. This equation can be rewritten to show the effect of interference from higher priority tasks.

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3.6}$$

where $hp(i)$ is the set of tasks with higher priority than i. The interference is a function of the maximum time that may be spent in higher priority tasks. Schedulability can then be resolved by recursive solution of this equation, and explicitly testing that all n tasks have response times less than their deadlines $R_i < D_i$. Extensions to the response time formula have been described that can also take into account clock granularity, hardware optimisations such as cache effects and non-zero context switching times [BW01].

Using response time analysis, it is possible to show that the only tasks which can be effected in such cases are those with lower priority. The interference caused by a task i can only influence tasks with lower priorities. If a task of priority i overruns a deadline, tasks with priority lower than i will potentially fail to meet their deadlines. This case is termed *transient overload*. In cases where there are no tasks released it is acceptable for the scheduler to not select a task to run. Such time is known as *slack-time*.

A third scheduling algorithms is Earliest Deadline First [LL73], a dynamic scheduling algorithm which adjusts the priority of tasks according to the next scheduled deadline. EDF can be used to achieve higher CPU utilisation than fixed priority scheduling systems in some cases, but some fault conditions can be unrecoverable, and as such, no further consideration will be made to this approach.

3.2 Execution Time Modelling

The most difficult aspect of embedding a task into a hard real-time task involves measuring or predicting execution time. This value will vary according to the chosen hardware platform and the implementation of the software. This is a particularly challenging research problem, and is a central issue for developing hard real-time systems.

Ideally a method for modelling execution time should enable estimates of the WCET that are greater (*safe* or *pessimistic*), but only slightly (*tight*), than the true value. In cases where the WCET estimate is *optimistic*, or below the actual WCET, a task could miss a deadline.

Calculating the true WCET of a program would involve measuring (or predicting), the execution time of all possible combinations of paths through the software, subject to all possible initial conditions. For all but the simplest types of software, the number of combinations of branches make this approach impractical.

The key concepts relevant to execution time modelling are illustrated in Figure 3.3. For a given piece of software and inputs, a range of execution times may be observed. The shape of the probability distribution of execution times will depend on hardware and software specific parameters, but will typically tend to be long tailed. Such distributions will also have a distinct upper and lower bound, the best and worst-case execution times respectively. The intention of estimating WCET is to identify a time as close to the true WCET as possible. A naive method of execution time estimation involves sampling execution times for various execution paths. As more samples are taken, the probability of finding the worst-case approaches one. Such approaches can, in the general case, only find a near worst-case execution time.

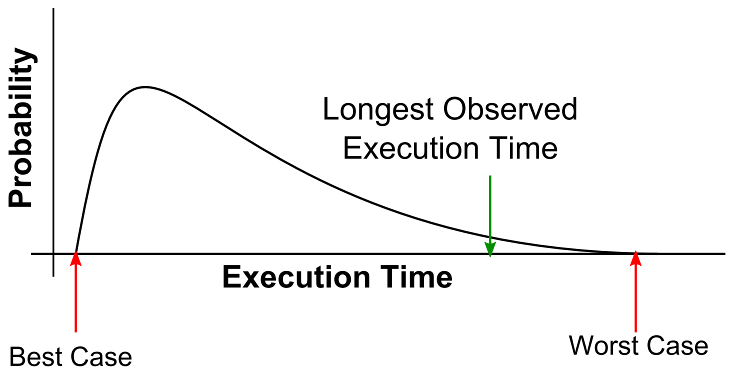

Figure 3.3: Execution time probability distribution.

One technique for modelling the execution time of a piece of software as a Control Flow Graph (CFG), where paths through the graph represent possible sequences of instructions that could be executed. A CFG is a directed graph $G = (V, E)$ where the vertices represent the individual instructions, $V = \{v_i | 1 \leq i \leq |V|\}$. The edges in the graph $E = \{e_i | 1 \leq i \leq |E|\}$, can be written as ordered pairs (v_i, v_j), and they represent the flow of the problem between instructions. The Control Flow Graph is required to be connected and have a single source instruction, v_s, with no predecessors, and a single terminal instruction, v_t, with no successors.

Most instructions in the CFG, with the exception of jump instructions, are executed sequentially. Consequently, most vertices in the CFG have an in-degree and out-degree of one. By collapsing all subsets of instructions that are executed sequentially (a *basic block*) to a single vertex, the control flow graph can be simplified.

One aspect which effects the time required to execute a given program is determined by the sequence of instructions that are executed, which is known as an *execution path*. The execution path of non-trivial programs is determined by a combination of the instructions in the program, and of the input data. An execution path can be described as a sequence of edges $P = (e_{s,1}, e_{1,2}, ..., e_{n,t})$.

If there exists a function $t(e) \mapsto \mathbb{N}$, which associates an execution time with each edge in the program, it is possible to predict the execution time of a given execution path, as given by Equation 3.7.

$$C = \sum_{e \in P} t(e) \tag{3.7}$$

By considering all possible paths through the CFG, Π, a WCET estimate can be calculated by finding the path which maximizes execution time (eq. 3.8).

$$C_{wcet}(\Pi) = \max_{P_i \in \Pi} C(P_i) \tag{3.8}$$

There are some difficulties in applying this technique in practice. Calculating the WCET using Equation 3.8 relies upon having a method of enumerating all possible valid paths through the program. An obvious example of such a case are loops within a program. Although the CFG encodes the branches, it does not include sufficient information regarding the maximum number of iterations. Such *flow facts* are either derived directly from the source code, or must be provided as additional data.

```
#define N (100)

void test-function() {
  for(int i = 0; i < N; i++) {
    if( f() ) {
      g();
    }
    else {
      h();
    }
  }
}
```

Figure 3.4: Code Snippet

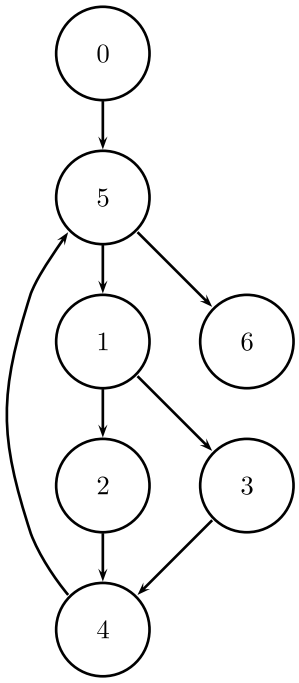

Figure 3.5: The associated CFG

The CFG in Figure 3.4 clearly contains basic blocks for the calls to the functions *f*, *g* and *h*, the body of the loop up to and including the condition, as well as the other component elements of the loop. In order to simplify the remainder of the analysis, Figure 3.5 is a simplified Control Flow Graph, with only identifiers of the basic blocks shown.

3.2.1 Modelling Limitations

One of the key sources of error in applying either modelling or measurement based techniques to the derivation of a WCET estimate is related to errors in the assumptions presented in Equations 3.7 and 3.8. Primarily, the simple execution model fails to take into account a model of the underlying hardware platform. Significantly, the assumption that the execution time of a basic block is constant is unlikely to hold on modern processors. In this section, a number of the hardware optimisations which influence execution time are considered, and the steps which can be taken to minimise their influence on measurements and calibration are proposed.

A *cache* is a store of a portion of memory which is being manipulated. In order to improve performance, a processor will load regions of memory into cache. The performance of a basic block will vary depending on whether the data being operated upon is in the cache. Frequently used data is more likely to end up in the cache. For algorithms which access data in an out-of-order fashion, the processor is less likely to be able to effectively use the cache.

Modern CPUs tend to be equipped with an *instruction pipeline* which is effectively a queue of instructions to be executed in the future. The time required to complete an instruction varies according to the design of the hardware, and the complexity of the instruction. Consequently some instructions in the pipeline can be evaluated in parallel (out-of-order), if the results of doing this will not effect the final result of those instructions.

Some sequences of instructions require the results of previous instructions, which results in the pipeline becoming *stalled*. Similarly, the pipeline uses *branch predictors* which are used to predict which sequence of instructions should be loaded into the pipeline, even in the presence of jump instructions. Some types of code can result in poor branch prediction, and consequently cause pipeline stalls.

Worst-case execution times can be heavily affected by the number and time required to perform *context switches* which occur. During each context switch the data in the pipelines and cache can potentially be affected. Additionally, the cost of making a context switch is small, but non-zero.

The consequence of each of these factors is that the execution time of each basic block is not constant, and is affected by the behaviour of all other executing tasks, and their effect on the processor. Additionally, the execution times of basic blocks are not independent - that is, the execution time of a block depends on which blocks were executed previously. Failure to consider these factors can introduce errors into the estimates of WCET.

3.3 Scheduling Complex Real-Time Tasks

One of the limitations of the basic formulation of real-time tasks is that they are not well suited for incorporating tasks whose execution time is variable (imprecise) and consequently difficult to predict, unbounded, or in cases where the average execution time is much less than the WCET.

The task of real-time scheduling is further complicated when some of the tasks involved have execution times which vary significantly. This tends to occur when computational requirements vary according to the problem. Such tasks can have a WCET that is difficult to estimate, or is unbounded. Under-estimating the execution time of such tasks can lead to scheduling failures, while over-estimation leads to increased slack time, or makes the system unschedulable. These types of tasks are referred to as *imprecise tasks* in this section.

Reasonable approaches to the scheduling of imprecise tasks require that the system remain both schedulable, and makes good use of the available resources, by maximising utilisation.

3.3.1 Soft Real-Time Tasks

Incorporating an arbitrarily complex algorithm into a hard real-time system can be resolved trivially if meeting a deadline is not critical. In the case of a Deadline Monotonic scheduler this can be done by ensuring that the planner has the lowest priority, effectively the largest deadline, in the real-time system. The result of this approach is that deadline overruns only affects the complex algorithms ability to deliver results on time, rather than any other tasks. The disadvantage of this approach is that low priority tasks are subject to more interference than high priority tasks.

A technique for generating this type of task is to divide the implementation of the task into several pieces, and interleaving their execution with tests to ensure that the deadline has not been exceeded. This approach is of particular importance to this thesis, because it forms the basis for the cited real-time guarantees of several motion planning algorithms [Hsu00, Fra01, FS06]. Algorithm 3.6 demonstrates the basic body for such a divided soft real-time task.

```
1: procedure SOFT-DIVIDED-RT-CYCLIC-TASK
2:     INITIALIZE-TASK
3:     $t_{next} \leftarrow$ CURRENT-TIME + $T$
4:     loop
5:         while TASK-BODY-INCOMPLETE and $t_{next} >$ CURRENT-TIME do
6:             PARTIAL-TASK-BODY
7:         YIELD-UNTIL($t_{next}$)
8:         $t_{next} \leftarrow t_{next} + T$
```

Figure 3.6: A framework for a divided soft real-time task

Assuming that the response time of the soft real-time task is approximately D_i, and that tasks with priority higher than i are schedulable, the maximum time spent in a soft divided real-time task is given by equation 3.9, where I_i is the interference introduced by the higher priority tasks. Note that this assumption does not affect the schedulability of the rest of the system, only that of the soft real-time task.

$$C_i \approx D_i - I_i \tag{3.9}$$

Critically, this analysis shows that the execution time available for the task is limited largely by the interference caused by other tasks.

3.3.2 Anytime Algorithms

In cases where the utility of a solution to a problem is dependent on the amount of time required to find a solution, the problem can be described as *time dependent*. It is possible to consider the work performed by a complex real-time task, as a time dependent problem. The deadline and execution time parameters of the task can be varied to achieve an acceptable degree of utility under all possible conditions. Dean and Boddy [BD89] [Bod91b] proposed the anytime algorithms, which is more correctly a framework for adapting algorithms for use in solving time-dependent problems.

Unlike traditional algorithms, which must run to completion before the results are valid, anytime algorithms are designed to produce intermediate results. Consequently, at any time, the algorithm can be interrupted, and the highest utility result is taken to be the solution. The relationship between utility and time is known as a performance profile, and a valid anytime algorithm will ensure that utility increases monotonically with time.

The basic approach for constructing an anytime algorithm is to generate an initial solution to the problem, which is potentially highly suboptimal. Additional solutions are then tested, and the best known solution at any time is retained.

This gives some insight into the classes of problems for which poor results will be achieved. If the complexity of the problem is such that in some cases it is not possible to find an initial solution, the performance profile may not be acceptable. For example, if there exists no feasible solution to a problem, the utility of the resulting empty plan can be zero.

In the case of motion planning, there are a number of results demonstrating how to adapt existing strategies into such a framework including Anytime Dynamic programming [Bod91a], Anytime-D* [LFG$^+$05], Anytime-DRRT [FS06], Anytime-PRM [vdBFJ06].

3.3.3 Imprecise Computations

Standard real-time tasks require that all deadlines are met. In a system that has imprecise computations, some tasks are required to meet deadlines, while the completion of less important tasks is optional. If sufficient time is available, optional tasks are executed, to improve the utility of the system. Imprecise computations can be described by some combination of mandatory and optional sub-tasks.

Any imprecise computations can be described as a combination of input (I), output (O), mandatory computation (C), and optional computation (X). A typical sequence of computations will be given by:

ICXCO

More accurately, this task can be described as three sub-tasks, a prologue (which performs input and mandatory initialization), an epilogue (which performs mandatory finalization and output), and an optional part. Each of the three tasks has an independent period, execution times and deadline. A valid schedule will ensure that for a given period, the optional and epilogue will only start after the prologue is finished, and that the combination of all tasks will meet their deadline. One approach to scheduling such systems is with a deadline monotonic scheduler, where the optional and epilogue tasks periods have offsets from the task release.

A wide range of algorithms have been described which can be expressed as an imprecise calculation. Examples of such approaches are described in Audsley et al [ABD+96]. They include:

- Multiple methods - in which multiple alternative approaches to finding a solution can each be attempted in turn. Each successive method has a different execution time requirements, some of which are bounded, and others of which are unbounded. The method with the best cost that is completed at the deadline is returned.
- Milestone or Approximate methods - in which an initial solution is found quickly, and is improved upon as time permits. When the available deadline is exceeded, the best current solution is returned.

In hard real-time systems, it is common to phrase anytime algorithms in terms of imprecise computations.

3.4 Execution Time Estimation

Execution Time estimation is the task of either measuring or calculating the execution required by a piece of software. In cases where the basic execution time model, introduced in section

3.2, is insufficient to predict execution time as occurs in the presence of hardware optimisations on modern processors, more sophisticated approaches must be employed.

The most straightforward approach to estimating WCET is to observe the maximum time required to execute all of the execution paths within a program. Such an approach is known as *explicit path enumeration*, and it is obvious that the impact of branches and loops results in a combinatorial explosion of possible paths. For non-trivial programs, this approach is intractable.

An alternative to explicit path enumeration is to sample the execution time for a subset of all possible execution paths. One approach for achieving this is to select representative test data. An estimate of execution time can then be derived from the maximum observed execution time. Sampling based approaches for the estimation of execution time rely upon achieving good coverage of all execution paths within a program, and can underestimate WCET when this does not occur. This technique is significant in that it has previously been applied to measuring the execution times of a real time motion planning algorithm [BV02].

Non-trivial methods for WCET estimation tend to fall into one of three classes. Static analysis methods, which examine the object code of a program, and make predictions based on a given model of how various instructions interact with hardware. Measurement or trace based approaches, which reconstruct a model of program flow and associated execution times. Finally, hybrid approaches which combine elements of both static and measurement based techniques.

3.4.1 Timing Schemas

A timing schema [PS91] can be used to extract an estimate of worst case execution time directly from a high level language. A timing schema is effectively a set of rules which can be used to derive execution time directly from program such as statements, compound statements, loops and conditionals. If the ET of a expression A is $W(A)$, the basic schema that for combining blocks is:

- $W(A) = k$ - the worst case for a particular statement is constant time.
- $W(A; B) = W(A) + W(B)$ - the worst case for a compound statement is the sum of the execution times for those statements.
- $W(\text{ if } E \text{ then } A \text{ else } B) = W(E) + max(W(A), W(B))$ - the worst case for a conditional statement can be found by combining the execution time required to evaluate the condition, and the longest time required to evaluate either branch.

- $W($ for E loop A end $) = W(E) + n(W(E) + W(A))$ - the worst case for loops is related to the maximum number of iterations that the exit condition and loop body can be evaluated.

Of these rules, only the final one requires information about the structure of the program. The maximum number of iterations for all of the loops within the program must be known in advance.

The limitation of timing schemas is that although sequential programming constructs can be represented in this way, there are some constructs for which the estimate will be pessimistic. For example, triangular nested loops like those in Figure 3.7 will overestimate execution time by a factor of two.

```
#define N (100)

void f() {
  for(int i = 0; i < N; i++) {
    for(int j = i; i < N; j++) {
      g();
    }
  }
}
```

Figure 3.7: An example of a program fragment with a WCET that is pessimistic when estimated with timing schemas

3.4.2 Probabilistic WCET

Edgar [Edg02] proposed a measurement based technique for of a estimation WCET by modelling execution time as an extremal value probability distribution function. By experimentally measuring the execution times of a program, it is possible to identify a tight bound on WCET. As is the case with explicit path enumeration, this approach is only applicable in cases with a relatively small number of branches.

Enhanced versions of Edgar's work included probabilistic versions of timing schemas, where execution time of edges in the CFG are modelled as having a probability distribution [GAP03]. By identifying correlations between the execution times of blocks, it is possible to produce tighter estimates of WCET than timing schemas or probabilistic methods alone. Further enhancements to this work were conducted by Petters who implemented tools to improve the granularity of observed execution time, by enforcing execution of specific paths in a program [Pet02].

Petters has published a review on the possible techniques which can be used to identify WCET through measurement [Pet03]. Measurement can involve either modifications to software, the use of external hardware to measure processor events, or a combination of the two approaches. In cases where a program is modified to identify timing information, this process will have an impact on the timing results.

3.4.3 Implicit Path Enumeration

An alternative to explicitly calculating the WCET of all paths through the CFG is to solve an equivalent maximum network flow problem - where the network is the CFG, and the value to be maximised is execution time [LM95]. This approach is convenient in that it can be solved using standard mathematical tools, such as Integer Linear Programming. This approach can address many problems that would otherwise be intractable to solve using explicit path enumeration.

Puschner and Schel [PS97] codified the requirements on the Control Flow Graph necessary to make the ensure that finding the WCET is decidable. These requirements include the need to provide critical information about control flow, such as constraints on the maximum number of iterations for any loops. Additionally, because the relationships between branches are modelled as linear constraints, it is possible to add extra constraints which allow tighter bounds than the simple timing schema, when there are non-trivial relationships between branches in the CFG.

Extensions to the basic implicit path enumeration technique have been proposed for cases where the execution time of basic blocks varies because of hardware effects such as data and instruction caches and pipelines [LMW95].

3.4.4 Advanced Techniques

In cases where an accurate model of the cache and pipeline hardware are significant, naive calculations of WCET can be inaccurate. This can have the effect of limiting schedulability and the effectiveness of a real time system.

Early research into the effects of caches was conducted by Mueller [Mue94] who proposed *static cache simulation*, a technique to model the interference effects of sequences of instructions on cache state. Critically, this work demonstrated that cache performance can be predicted through the use of static analysis techniques. This allows an understanding of the cache state to be combined with timing analysis to allow tighter and safer bound on WCET.

A range of research and commercial off the shelf tools have been used for WCET calculations. An overview of the current state of such tools is presented in Wilhelm et al [WEE$^+$08]. Some of these tools are based on the notion of program analysis by Abstract Interpretation, a technique from compiler design used to determine runtime properties of a program without actually executing it. This allows both optimizations caused by cache and data and instruction pipelines to be accounted for in WCET calculations.

CHAPTER 4

Hard Real-Time Dynamic Motion Planning

This chapter introduces the Hard Real-Time Dynamic Motion Planning (HRT-DMP) algorithm, an extension to the dynamic motion planning problem, which imposes a hard constraint on the time allowable for planning and replanning. The need for hard real-time dynamic motion planning arises in situations where a motion planning algorithm must be incorporated into a hard real-time system.

The basic strategy for addressing dynamic planning problems described in Section 2.1.2 is to produce an initial plan, and to replan as obstacles are detected, or observed to be moving. This means that the capability to generate safe plans is linked to the information on the workspace available when planning commences. A plan is only valid only so long as all configurations along the path are collision free, the rate at which new the environment changes, and the rate at which those changes can be observed and incorporated into planning influence safety.

Motion planning can then be viewed as a planning strategy that relies upon a static environment, an assumption that is unlikely to be valid under real-world conditions. Dynamic motion planning strategies are founded on the principle that while the environment does change, it does so slowly, and that periodic replanning will occur sufficiently often to produce a new plan before following an existing plan leads to a collision [1]. In such cases, errors in the mapping estimate can be controlled by imposing a deadline on the amount of time available for producing plans.

The naive strategy for producing a periodic planner would be to sequentially execute sensing, planning and control in turn. The challenge of such an approach is that it can be difficult to ensure that the deadlines and periods of these tasks can be satisfied, furthermore, this approach in known to be brittle and difficult to change from a software perspective [BW01]. Instead, the approach considered here is to decouple sensing, planning and control as concurrently executing tasks.

Effectively each of the planning, sensing and control tasks must satisfy independent dead-

[1]The period at which planning occurs is dependent on knowledge of the vehicle and environment, some guidance on selecting appropriate parameters is given in Section 4.2

lines. One method for doing this rigorously is to adopt the hard real-time scheduling analysis techniques from Section 3.1.3. For a planning algorithm, this highlights a critical issue - excessive execution time can detrimentally effect the capability of other tasks to meet deadlines, while failure to make use of allocated execution results in less time spent planning.

A common assumption adopted in dynamic motion planning literature is that a safe path to the goal will be found every time the planner runs. While this assumption simplifies the structure of these planners, it is easy to show that it does not hold in all cases. This can occur when the goal is unreachable, for example if $q_{goal} \notin \mathcal{F}$. This prevents the direct use of several well known dynamic motion planning algorithms including Anytime PRM [vdBFJ06], Dynamic RRT [FKS06], Anytime RRT [FKS06], Anytime DRRT [FS07] and D* [Ste94].

One method for ensuring that a plan will always be available is to provide a contingency plan that can be proven to be safe over a finite time window, as described in Section 2.1.5. Several existing techniques to construct safe partial plans exist [PF05b, Sch06], however, with a few exceptions [HKLR02, Fra01] none of them are suitable for non-linear systems subject to kinodynamic constraints. Such contingency plans are necessary to ensure the vehicle remains in a collision free state until a mission planning software makes a decision to either continue searching, waiting for a change in the configuration space, or to select a new target configuration.

The limitations of using the dynamic motion planning approaches most commonly proposed in the literature can be stated as follows. Firstly, replanning must be able to run at rates sufficiently high to be able to incorporate new observations of the environment. Secondly, available computational resources, particularly processor time, must be shared between the planner and other inter-related tasks, such as sensing and control, such that all tasks do not interfere with each other. Finally, such deadlines offer a concrete method to show that a planner will always have sufficient time available to attempt to create a new plan, in changing environments.

In order to address the problem of HRT-DMP, a novel framework for a constructing hard real-time tasks is proposed that can be used to incorporate motion planning algorithms into hard real-time systems. The intention of developing a framework for generating such tasks is to identify requirements on the execution time available for planning, taking into consideration interactions with the rest of the real-time system. This framework makes it possible to measure and evaluate the effect of changing planning algorithms, with the final intention of being able to maximise the quality of search that can be conducted within a deadline.

4.1 Problem Definition

The basic context of a HRT-DMP is that of a hard real-time system. Following Section 3.1, all tasks to be executed as part of the system, including sensing, planning and control, are embedded in a hard real-time system, S, where task i has an independent period, T_i, deadline, D_i and worst-case execution time, C_i.

Hard Real-Time Dynamic Motion Planning is the task of periodically (with period T_p), finding a new trajectory, $\tau(t)$, within a deadline D_p. As is the case with dynamic motion planning, the path is required to be continuous from the current configuration, q, to a goal configuration, q_goal and all intermediate configurations along the trajectory are collision free. The path is also required to satisfy all kinodynamic constraint introduced by the vehicle model.

A single iteration of a dynamic motion planner can be thought of as being equivalent to the task of motion planning with a static environment, subject to a hard constraint on execution time, $\hat{C}_p$. This problem is termed *Hard Real-Time Motion Planning* (HRT-MP).

Definition 6. *Hard Real-Time Motion Planning is the problem of finding a trajectory $\tau(t)$ such that $\tau(t_i) = q_{init}$ and $\tau(t_f) = q_{goal}$, and for all t, $\tau(t) \in \mathcal{F}$, or correctly reporting that no such path exists, in an execution time of no more than $\hat{C}_p$.*

The capability to produce such a planner is impractical, due to the computational complexity of motion planning, described in Section 2.1.4. For example, there will always exists some planning problem that will take a complete planner longer than $\hat{C}_p$ to solve. Instead, a weaker form of completeness is required to produce a practical planner.

Furthermore, while the total time for planning for HRT-MP is required to be less than $\hat{C}_p$, the solution to a full HRT-DMP requires additional processing to read the most current mapping and localisation data, and to write the produced plans to the controller task. The total execution time for planning is therefore:

$$C_p = \hat{C}_p + C_{IO} \tag{4.1}$$

This leads to the final definition of *Hard Real-Time Dynamic Motion Planning*:

Definition 7. *Hard Real-Time Dynamic Motion Planning is the problem of periodically finding a trajectory $\tau(t)$ such that $\tau(t_i) = q_i$ and $\tau(t_f) = q_{goal}$, for all t, $\tau(t) \in \mathcal{F}$, or correctly reporting that no such path exists, in an execution time of no more than C_p. Furthermore, the period of the task is required to be T_p, and all results must be produced within a deadline of D_p from the start of each period. The temporal parameters of the planner task (T_p, D_p, C_p) are required to be chosen so that the real-time system is schedulable.*

The process for verification of possible solutions to Hard Real-Time Dynamic Motion Planning problems involve three steps. The first is to identify the periods and deadlines for all tasks; the second is to find the worst-case execution times of all tasks, and the final step is to prove that the system is schedulable.

This approach to hard real-time motion planning allows a distinct separation of the planning and schedulability analysis. Once an upper bound on execution time is known, it is possible to confirm whether the combination of tasks in the real-time system will satisfy the schedulability requirements.

4.2 Selecting Task Parameters

Selection of parameters for the period, deadline and worst-case execution time of a hard real-time motion planning task will be heavily influenced by application specific concerns, and the capabilities and computational requirements of other tasks.

4.2.1 Basic Principles

On board an autonomous vehicle, the rate at which a control task runs is dependent on the dynamics and kinematics of the vehicle plant, the magnitude of disturbances in the environment and the capabilities of the actuators. For underwater vehicles, which tend to move at low speed, control loops typically vary from 10Hz [ASA^{+}97] - 25Hz [MWM03]. For aircraft, the rates at which controllers run can be significantly higher, around 100Hz [SL03].

The rate at which sensor observations are received varies significantly according to the types of sensors are used. Typically, on-board AUVs localization estimates are required to be available at similar rates to the controller to facilitate closed loop control, typically in the 3-30Hz range [SK08]. More computationally expensive mapping estimates are produced at much lower rates.

In situations where it is unlikely that a closed loop controller will fail to accurately track a path (gross motion planning), the need for producing new plans is limited to the rate at which new map observations can be produced. For a mapping task with a period of T_m, the planner will be using information that is at most $\lceil \frac{T_p}{T_s} \rceil + 1$ map updates out of date. The significance of this error depends on the kinematic and dynamic responses of the vehicle and the obstacles.

Selection of the deadline and minimum time between subsequent releases for a planning task could potentially be chosen to be any values $D_p \leq T_p$, such that the real-time system is schedulable. In this work, attention is focused on the case where $D_p = T_p$, and the planning task is cyclic. This formulation is significant in that it enables a simple and clear presentation of the method for replanning, while imposing only minor limitations on scheduling. The principles of analysis remain valid even if a sporadic task is employed or if the deadline is less than the period.

In cases where transient overload occurs because a task exceeds its allotted shared of execution time, some tasks in a hard real-time system may miss deadlines. Under Deadline Monotonic scheduling, the deadline of a planning task relative to other tasks will determine whether a safety critical failure occurs.

4.2.2 Latency

Producing collision free paths is dependent on being able to accurately predict the location of the vehicle and obstacles. Given that inputs are only read at the start of a period, these values will have changed by the time planning has finished, an effect best described as a *latency*. In cases where latency is high, the resultant plans may be infeasible or unsafe.

One way to avoid the problems of latency is to predict where the vehicle will be at the time when planning finishes. A simple method for doing this is to use the full model of the system, including the effect of the plant and controller, current state and path to generate an open loop prediction of where the vehicle will be at the deadline. This approach is shown in equation 4.2

$$x = \int_{t}^{t+D} f(x, g(x, \tau))dt, \tag{4.2}$$

where f and g are the plant and controller model respectively, τ is the current path, x is the last known state at time t, and the time at which the prediction is needed depends only on the deadline D.

A simpler technique, given that the controller will minimise path tracking error over short time periods, is to make the assumption that the vehicle will remain on the path (eq. 4.3). As with all forms of gross motion planning, this assumption will break down in cases where there are modelling errors, significant disturbances, and as the deadline increases.

$$\int_{t}^{t+D} f(x, g(x, \tau), \omega)dt \approxeq \tau(D) \tag{4.3}$$

Given bounds on the acceleration or velocity of obstacles, a similar approach to that of 4.2 can be used to predict the locations of obstacles over time. The errors introduced by latency can be limited by both using an appropriate prediction mechanism, or by keeping the deadline for planning as small as possible.

4.3 Proposed Framework

The intention of the proposed framework is to provide a method to incorporate a range of motion planning algorithms into a hard real-time system. The basic approach is to provide an algorithmic starting point, and to correlate that work back to the definition of HRT-DMP. The pseudo code in Figure 4.1 gives an example of periodic motion planning implemented.

```
1: procedure HRT-PLANNER
2:     INITIALIZE-PLANNER
3:     $t_{next} \leftarrow$ CURRENT-TIME
4:     loop
5:         $t_{next} \leftarrow t_{next} + T_p$
6:         $\mathcal{B} \leftarrow$ updated map
7:         $x_g \leftarrow$ current goal states
8:         $x_i \leftarrow$ state of the vehicle
9:         $\tau \leftarrow$ PLAN($x_i, x_g$)
10:        publish $\tau$
11:        YIELD-UNTIL($t_{next}$)
```

Figure 4.1: Pseudo-code for a cyclic real-time planning algorithm

As is the case with a standard cyclic task, Figure 4.1 highlights that non-time critical initialisation is performed once off, before the main loop starts to run. Given this structure, a single iteration of the main loop constitutes one period. This means that the task is released as soon as the yield statement (Line 11) is completed, which results in the deadline being equal to the period of the planner, $D_p = T_p$. One iteration of the main loop is required to take no longer than C_p to complete. The main loop (lines 6 - 10), is made up three distinct phases: input, planning and output.

The input phase deals with reading the necessary information that has potentially changed since the last planning cycle. The locations of the obstacles are updated, as are the current and goal states. The output phase involves writing the plan once it becomes available. The upper bound on the execution time for input and output is modelled as a single constant C_{IO}. Calculation of this parameter is in itself a potentially difficult problem in worst-case execution time estimation that would be difficult to resolve without additional domain specific knowledge, for example, the number of obstacles.

Critically, because both input and output involve operations on data with shared state, they will also potentially block. If the call to the planner is known not to block, then B_p, is simply the sum of the worst possible blocking times for input and output.

For each task, except for the planner, that are to make up the hard real-time system, the periods, deadlines and worst case execution time must be determined. A period and deadline for the planner can be selected, using the guidelines in Section 4.2. In order to verify that all tasks can satisfy their deadlines, an estimate for the execution time allowable for planning can be produced, and Rate Monotonic Scheduling analysis can be conducted. This process can be repeated to find a tight upper bound on planning execution time.

Once a bound on the execution time for planning is known, the planner can be measured in isolation from other tasks to ensure that it has an estimated WCET less than the maximum allowable execution time. In the case of approximate motion planning techniques, execution time is typically controlled by the selection of a number of parameters which control search quality. This makes it possible to find a combination of parameters that maximise search quality while still satisfying the execution time constraint.

Critical to this formulation is the assumption that the planner is capable of always producing a safe path, or where no such safe path can be found, an alternative path that can be followed safely until the next planning period. This ensures that the algorithm is independent of the size, or complexity, of the planning problem.

The proposed framework prescribes a structure to the real-time planning task, and the following procedure verifies the planner within a hard real-time system.

1. Identify the deadline and period based on system requirements for all tasks. Measure the worst-case execution time of all tasks, except for the planner.

2. Use a rate monotonic schedulability analysis to identify the maximum execution time allowable for planning, incorporating measures of execution time for input, output and blocking and any safety margins for transient overload.

3. Verify that, for a selected set of parameters, the planner has a worst case execution less than the maximum allowed.

4. The previous step should be repeated, varying the selected set of parameters used for planning in order to maximise the quality of search that can be conducted per period.

4.3.1 Demonstration

To demonstrate the basic process required to use the framework, the performance characteristics of a standard planning algorithm are considered. It is instructive to consider an algorithm for which there is an obvious trade-off between search quality and the time required to produce a solution.

A well known example of a discrete search algorithm is the depth-limited graph search [LaV06, RN03]. This algorithm can be used to find paths between states in a discrete graph. Such an approach can be applied to motion planning, if the configuration space can be approximated by an implicit lattice (Section 2.1.3). Local connections in the lattice can be constructed by using a finite subset all possible short feasible paths which can be found by exploiting symmetry and optimal control techniques [Fra01], or regular decompositions in some special cases [LaV06], or by using splines [CMH$^+$05].

A simple method to find a path is to explore such branches to a fixed depth, returning a solution if one is found. An example of this strategy is shown in Figure 4.2. The search continues to a fixed depth. On a lattice where each state has no more than b successors (the branching factor), and the search is limited to a depth d branches from root to leaves, the algorithmic worst case of the depth limited search visits b^d states.

Motion planning based on depth-limited search is incomplete. Failure to use an adequately deep search or dense lattice can result in paths to goal not being identified, even if they exist. For a dense lattice, in a finite subset of configuration space, as the branching factor and search depth increase, the likelihood of finding the search graph coming arbitrarily close to any configuration increases.

```
1: procedure DEPTH-LIMITED-SEARCH($x, x_g, depth$)
2:     for $x_s \in$ SUCCESSORS($x$) do
3:         if COLLISION-FREE($x, x_s$) then
4:             if $x_s \in x_g$ then
5:                 return $x_s$
6:             if $depth < d$ then
7:                 $p \leftarrow$ DEPTH-LIMITED-SEARCH($x_s, x_g, depth + 1$)
8:                 if $p \neq \emptyset$ then
9:                     return $p \cup x_s$
10:    return $\emptyset$
```

Figure 4.2: Pseudo-code for depth-limited search

The worst-case execution time for a depth limited search will be influenced by a number of factors, including the time complexity of generation of successor states and collision testing of paths. Regardless, the algorithmic worst-case performance gives a strong indication of which parameters can be used to manage the trade off between search quality and execution time.

This point highlights that WCET is a function of branching factor and search depth, with values that can be evaluated for a particular implementation of the algorithm. In practice, any of the techniques described in Section 3.4 could be used to solve this problem, with varying degrees of difficulty. Maximising search quality would then involve finding the maximum values of b and d for which the execution time constraint is satisfied.

This strategy is also significant in that the search graph generated by the planner can easily be adapted to find an approximately safe partial path in cases where no path to the goal can be found. By randomly selecting any node in the search graph that requires longer than T_p to reach, the vehicle is guaranteed to have sufficient time to attempt to produce a new path.

4.4 Performance Evaluation

This section considers the performance measures that could be optimised for while satisfying the hard limit on execution time.

4.4.1 Completeness and Feasibility

Ideally, it would be possible to structure a planner so that it would always be possible to find an admissible path to the goal from the current configuration if one exists. The computational complexity of complete planning algorithms highlight that some problems exist that cannot be solved within a finite deadline (Section 2.1.4).

In situations where the time available for planning is constrained, a suitable approach is to employ an approximately complete planner. This type of planner is characterised by the probability of finding a solution increasing as the time spent planning increases. An important aspect of the quality of such algorithms is the rate at which they converge to completeness. Rapid convergence is of particular importance when the planning problem is very difficult, or the deadline is very short.

Accurately measuring the rate of convergence for a planner is impractical as it would require considering how the algorithm behaves under all possible conditions. Some classes of probabilistically complete motion planning algorithms, given mild constraints on the environment, can be shown to converge to completeness in exponential time [HKLR02, Fra01].

Failure to employ a planner that has a high probability of being complete, considering the current deadline and problem, will make finding paths to the goal difficult. In the case of dynamic motion planning, the need for approximate completeness is just as important. For example, if a planner fails to find a solution in one planning period, the vehicle must follow a safe partial path. A more systematic problem can occur if the vehicle cannot be shown to eventually reach a target state.

4.4.2 Path Cost

One of the most significant objectives for this thesis was to develop planners that were both safe and efficient. There are several approaches in motion planning that can be used to assess the efficiency of a path. The cost of a path typically involves a measure of path length, time required to traverse the path, or the energy expended to track the path. Where possible, the cost of the path produced by a planner should be minimised.

In conditions where a non-optimal planning strategy is used, it is possible that naively generating a new plan in each period may not converge to the goal state. This can occur if it is possible for the cost of following a new path to the goal can be greater than the cost of the current path to the goal.

If this requirement is not satisfied it is possible that the paths could result in no progress being made toward a goal. This mean that the naive replanning process can introduce local minima in the search.

Such minima can be avoided by ensuring that repeated calls to the planner retain sufficient information between calls to ensure a new plan will only be produced, in cases where an old path is now invalid, or when the cost of reaching a goal is lower than on the current path.

4.5 Summary

This chapter has argued that the effectiveness of dynamic motion planning is closely linked to the time required for planning, and the rate at which that planning occurs relative to other related processes, in particular control and sensing, and the rate at which the environment changes. In cases where such interactions are time critical, the task of planning cannot be addressed with the existing approaches for dynamic motion planning. This work highlights that a more rigorous problem definition is required so that temporal affects can be taken into consideration.

The definition of a new class of planning problem, the Hard Real-Time Dynamic Motion Planning (HRT-DMP) was presented, which is useful in described time critical planning operations. This definition imposes additional constraints on the time available for planning that are necessary to describe the role of the planner within a hard real-time system.

A novel framework for addressing HRT-DMP problems was proposed, which allows planners to be composed as a cyclic task in a hard real-time system. This structure for a hard real-time planner ensures that sensor inputs will be read and new plans produced, oncer per period, prior to a specified deadline. By selecting appropriate values of deadline and period for the planner it is possible to prove that all tasks in the hard real-time system will satisfy their deadlines. Regular replanning also ensures that in cases where environmental changes are observed that will result in a collision, sufficient time is available to attempt to find a new plan.

By describing the process of hard real-time planning in terms of a general framework, it becomes clear that the execution time required for planning and interactions with other tasks can be modelled as independent values. The task of HRT-DMP is shown to be capable of being solved using any planner for which a worst case execution time can be calculated. This means that the approach is applicable in configuration spaces with a varying dimensionality and topological complexity.

Finally, this work has described that approximate motion planning algorithms are well suited as a basis for designing planners to address the HRT-DMP problem, due to their predictable algorithmic worst-case performance characteristics. Furthermore, an indication of methods of evaluating and comparing different types of planners has been described.

CHAPTER 5

Hard Real-Time Rapidly-exploring Random Trees

Having proposed a framework for addressing hard real-time dynamic motion planning problems in Chapter 4, this chapter develops a novel planner, the Hard Real-Time Rapidly-exploring Random Tree (HRT-RRT), and demonstrates the search and execution time characteristics of that planner.

The conceptual starting point for this work is the Rapidly-exploring Random Tree (RRT) algorithm, first presented by LaValle [LaV98]. RRT is a planning algorithm that incrementally generates a search tree until either a solution is found or a fixed amount of search has been conducted. RRT is an ideal starting point for research into hard real time planners for autonomous vehicles because it is probabilistically complete, directly considers kinodynamic constraints during planning, and employs heuristics to efficiently explore a configuration space [LK99].

The novel aspect of the HRT-RRT, is to limit the amount of search performed, based on the predicted worst case execution required for a particular set of parameters. By calculating the WCET of the planner for a number of parameters, it is possible to identify those parameters which maximise search quality while satisfying hard real-time constraints.

5.1 The RRT Algorithm

This section presents a review of the fundamental building blocks of the Rapidly-exploring Random Tree algorithm. This work is an essential part of building an understanding of the algorithmic complexity of RRT, which is necessary for finding a relationship between search parameters and WCET.

While there are a number of variants of RRT, the version considered in this chapter has been chosen to mirror early formulations presented in [LaV98, LK99]. It is worth noting that there are a number of more efficient extensions and variants of RRT[1], this formulation is a well known example of a sampling based motion planning, useful for examining worst-case execution time performance characteristics. Where there were multiple possible implementations

[1]a variant will be considered in Chapter 6

for part of the RRT could have been employed, the approach with the most obvious worst-case performance was selected.

RRT is a search that propagates outward from a single initial configuration. The search tree is grown through an incremental process of sampling of the configuration space, and by extending the nearest neighbour in the search tree toward that sampled configuration. Initially, the search tree only contains the initial configuration of the system, and the search terminates when the tree reaches a goal configuration.

A pseudo-code implementation of RRT is shown in Figure 5.1, where $\mathcal{T}$ is the search tree (a directed acyclic graph), and the aim is to find a path from the initial configuration x_{init}, to the goal configuration x_{goal}. The search algorithm attempts to add up to n new configurations to the search tree. The complexity of BUILD-RRT is $O(n)$, in the number of nodes to be added to the search tree.

```
1: procedure BUILD-RRT($x_{init}, x_{goal}$)
2:     $\mathcal{T}$. INIT($x_{init}$)
3:     for $i \in [1, n)$ do
4:         $x_{samp} \leftarrow$ SAMPLE-CONFIGURATION
5:         $x_{near} \leftarrow$ NEAREST-NEIGHBOUR($x_{samp}, \mathcal{T}$)
6:         $u_{new} \leftarrow$ SELECT-INPUT($x_{near}, x_{samp}$)
7:         $x_{new} \leftarrow$ EXTEND($x_{near}, u_{new}$)
8:         if $x_{new}$ not nil then
9:             $\mathcal{T}$. ADD-EDGE($x_{near}, x_{new}, u_{new}$)
10:            if $x_{new} \in x_{goal}$ then
11:                break
12:    return $\mathcal{T}$
```

Figure 5.1: Pseudo-code for the RRT algorithm

The growth of the search tree is guided by the sampling process. The simplest type of sampling strategy is to randomly draw samples from a subset of configurations using a uniform distribution, rejecting configurations that are in collision. The proofs for completeness of RRT are based on the principle that the spatial distribution of nodes that appear in the search tree will end up following the same distribution as that used for sampling. In the case of uniform sampling, this means that given sufficient time, RRT will be able to extend the search tree from the initial configuration to the goal.

The down-side of using uniform sampling is that it results in a high level of exploration, which can result in search being focused in parts of the configuration space unlikely to assist in finding a solution. A common alternative is to use a biased sampling algorithm [LL04] (Figure 5.2). This approach has the advantage of greedily biasing the search tree to grow toward the goal configuration, while running in constant time, $O(1)$. Even a small bias (5%) can be sufficient to result in improved rate of convergence for a planner [LK00].

```
1: procedure SAMPLE-CONFIGURATION
2:     $p \leftarrow$ UNIFORM-RANDOM
3:     if $p \leq bias$ then
4:         return $x_{goal}$
5:     else
6:         return COLLISION-FREE-UNIFORM-RANDOM-CONFIGURATION
```

Figure 5.2: Pseudo-code for biased random sampling algorithm

Having identified a sampled configuration, the choice remains open about which node in the search tree to extend toward this target. The use of a nearest-neighbour search ensures that the node in the search tree closest to the sampled state is used in the extension process. Distance, or an estimate of distance, between configurations can be found by using an appropriate metric function.

A valid metric function, $\rho : X \times X \mapsto \mathbb{R}$, produces an estimate of the cost of the path between a pair of configurations, or states. In the context of this work, the metric is required to satisfy a number of constraints including:

- non-negativity: $\rho(x, y) \geq 0$
- reflexivity: $\rho(x, y) = 0 \iff x = y$
- triangle inequality: $\rho(x, y) + \rho(y, z) \geq \rho(x, z)$

An ideal measure of nearness would be to calculate the optimal cost of moving between configurations. In many cases, the computational cost of finding an optimal path that the optimal costs can be approximated with a metric or pseudo-metric [LaV98, LaV06].

The significance of the time required to identify nearest neighbours occurs because for a search tree with n configurations, a naive implementation of the nearest neighbour search, such as the pseudo-code in Figure 5.3 requires $O(n)$ evaluations of the distance metric.[2].

Having identified the node in the search tree closet to the sampled configuration, the next step is to identify a sequence of control inputs which, when applied from a starting state will result in the tree growing toward the sampled configuration. To simplify the process of selecting inputs, in this work, the full set of input functions to the vehicle controller are chosen from a discrete subset of all possible inputs $\mathcal{U}_d \subseteq \mathcal{U}$. The input functions to the RRT algorithm are taken to be a single constant input from $\mathcal{U}_d$ applied over a fixed time-step.

The approach demonstrated in Figure 5.4 is to apply each of the possible control inputs in turn. The input which minimises the metric distance between the new configuration and the sampled configuration is returned. The running time for this algorithm is $O(|\mathcal{U}_d|)$, where $|\mathcal{U}_d|$ is the total number of discrete control inputs inputs.

[2]in special cases, the complexity for the nearest neighbour search can be reduced to $O(\log n)$ by using appropriate spatial data structure [BV02, YL02, YL07]

```
1: procedure NEAREST-NEIGHBOUR($x_t$)
2:     $c_{best} \leftarrow$ nil
3:     $x_{best} \leftarrow$ nil
4:     for $x \in \mathcal{T}$ do
5:         $c \leftarrow \rho(x_t, v)$
6:         if $c < c_{best}$ or $c_{best} =$ nil then
7:             $c_{best} \leftarrow cost$
8:             $x_{best} \leftarrow x$
9:     return $best_x$
```

Figure 5.3: Pseudo-code for a linear time nearest neighbour algorithm

```
1: procedure SELECT-INPUT($x_{cur}, x_{targ}$)
2:     $cost_{best} \leftarrow$ nil
3:     $u_{best} \leftarrow$ nil
4:     for $u \in \mathcal{U}_d$ do
5:         $x_{new} \leftarrow f(x_{cur}, u, dt)$
6:         $cost \leftarrow \rho(x_{new}, x_{targ})$
7:         if $cost < cost_{best}$ or $cost_{best} =$ nil then
8:             $cost_{best} \leftarrow cost$
9:             $u_{best} \leftarrow u$
10:    return $u_{best}$
```

Figure 5.4: Pseudo-code for the select-input algorithm

The EXTEND function (Line 7 in Figure 5.1), performs an open loop simulation to predict the path followed by the vehicle using Equation 2.2. During the extension process, the path segment is tested for collisions, if it can be shown to be collision free, a new node is added to the search tree. In order to reconstruct a path without having to search the entire tree, each node contains a reference to its parent. If a state is sufficiently close to a goal, a safe solution path can be extracted by walking up the tree.

An example of Build-RRT in practice is shown in Figure 5.5. The blue nodes show the configurations which have already been added to the tree. The solid lines denote the branches within the tree and the dotted line to the red node represents the nearest neighbour to the sampled node, prior to connection.

Taking into account the complexity of the subroutines BUILD-RRT, NEAREST-NEIGHBOUR and SELECT-INPUT, which have complexities of $O(n)$, $O(n)$ and $O(|\mathcal{U}_d|)$ respectively, the complexity of the RRT algorithm of a whole will be $O(n^2)$ provided that $n >> |\mathcal{U}_d|$.

Having considered the various components of the RRT algorithm, the next step is to consider the set of parameters which need to be controlled so as to ensure that the WCET can be bounded. The two critical factors which can be controlled are the size of the search tree, n and the number of discrete control inputs, $|\mathcal{U}_d|$. For a given discretization of the control space,

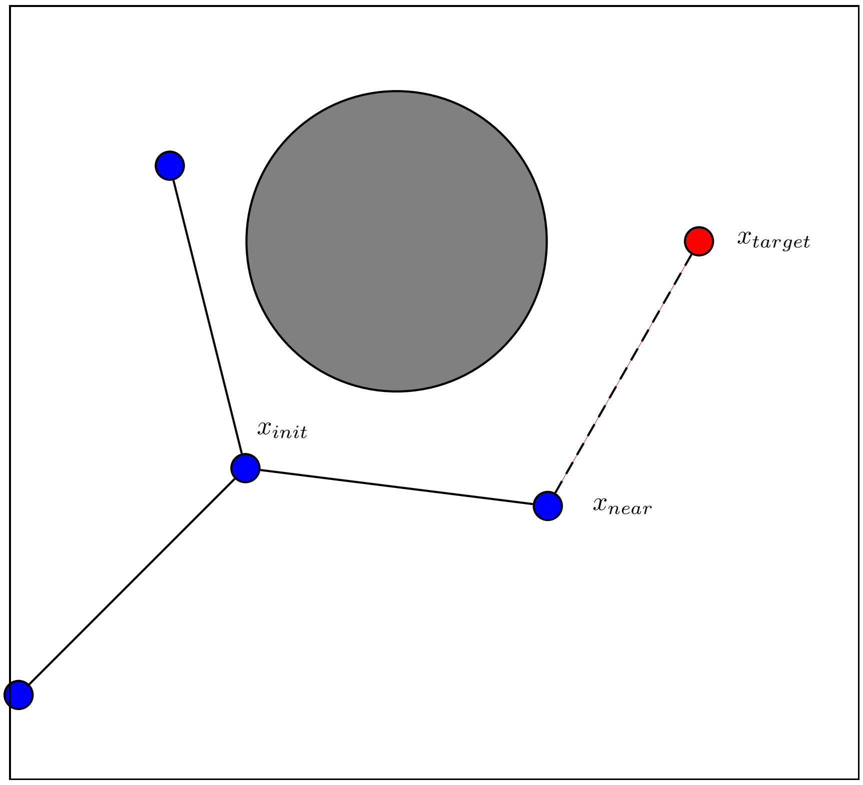

Figure 5.5: A demonstration of the approach used for extending the RRT search trees

the size of the search tree is the factor which will have the most significant effect on execution time.

This formulation makes a number of simplifying assumptions. Primarily, the time required to extend the search tree must be constant. This in turn requires that only a constant number of collision detection tests. For sampling based collision detection strategies this can be achieved by limiting the time step used for each tree extension.

The number of nodes added to the search tree RRT is a measure of search effort. If the search is terminated before a solution is found, the problem is considered to be infeasible. Consequently, the task of finding an efficient hard real-time equivalent to RRT can be addressed by finding the relationship between n and the WCET.

5.2 Execution Time Estimation

Selection of an appropriate technique for execution time measurement for hard real-time planners depends on a number factors including the required accuracy and the target hardware. Many of the techniques presented in the catalogue of execution time estimation methods in Section 3.4 could be used to solve this problem. For the purposes of verification of the RT-

RRT algorithm, the intention was to demonstrate a technique that could be adapted for use on a range of autonomous systems with varying levels of computational resources and minimal effort.

Low cost modern processors tend to be equipped with a number of hardware optimisations, such as caches and pipelines for data and instructions, that make naive cycle counting inaccurate. More rigorous methods for calculating execution time require accurate models of hardware, which can be difficult to acquire and validate. Without access to processor models for each hardware target, a reasonable alternative is to rely upon the principles of probabilistic worst-case execution time measurement.

In general, probabilistic WCET estimation typically relies upon inserting instrumentation into object code. This process relies upon having an extensive knowledge of the underlying format of the object code, and often requires development of a customized compiler tool-chain. A coarse approximation can be produced by instrumenting the source code of the planner. This has the advantage of being both easy to implement, and prevents the need to modify the compiler tool-chain.

The execution time estimation strategy employed in this chapter is based on a combination of a single probabilistic estimate of the execution time of each edge in the Control Flow Graph, and prediction of WCET of the planner using Integer Linear Programming techniques using the approach of Puschner and Schdel [PS97]. Figure 5.6, shows the steps involved in this process.

The first step is to write and test an implementation of the planner that functions without timing concerns. The source code of the algorithm can then be modified to include instrumentation at locations which are likely to be associated with jumps or jump labels in the final assembly code. In an imperative language, such as C [KR88], this includes conditional statements and loops. The purpose of the instrumentation is to record the point in the program and time-stamp for later use.

When the instrumented program is executed, the instrumentation code writes data to an in-memory buffer. At the termination of the program the execution trace, a combination of the execution path and associated timings can be saved. One or more execution traces can then be post-processed to produce the CFG and estimates of execution times of each basic block.

Identification of the CFG from an execution trace involves finding all of the successive pairs of identifiers in the trace. These pairs then form the edges of the CFG. The execution times of each edge in the CFG can be calculated as the difference in absolute measured time.

Even though manual instrumentation of source code is conceptually a simple process, it has a number of inherent flaws. Firstly, it is possible that some locations within the program are not sufficiently instrumented, that is, the observed Control-Flow-Graph does not match the real CFG. This is particularly challenging problem given that modern compilers can perform

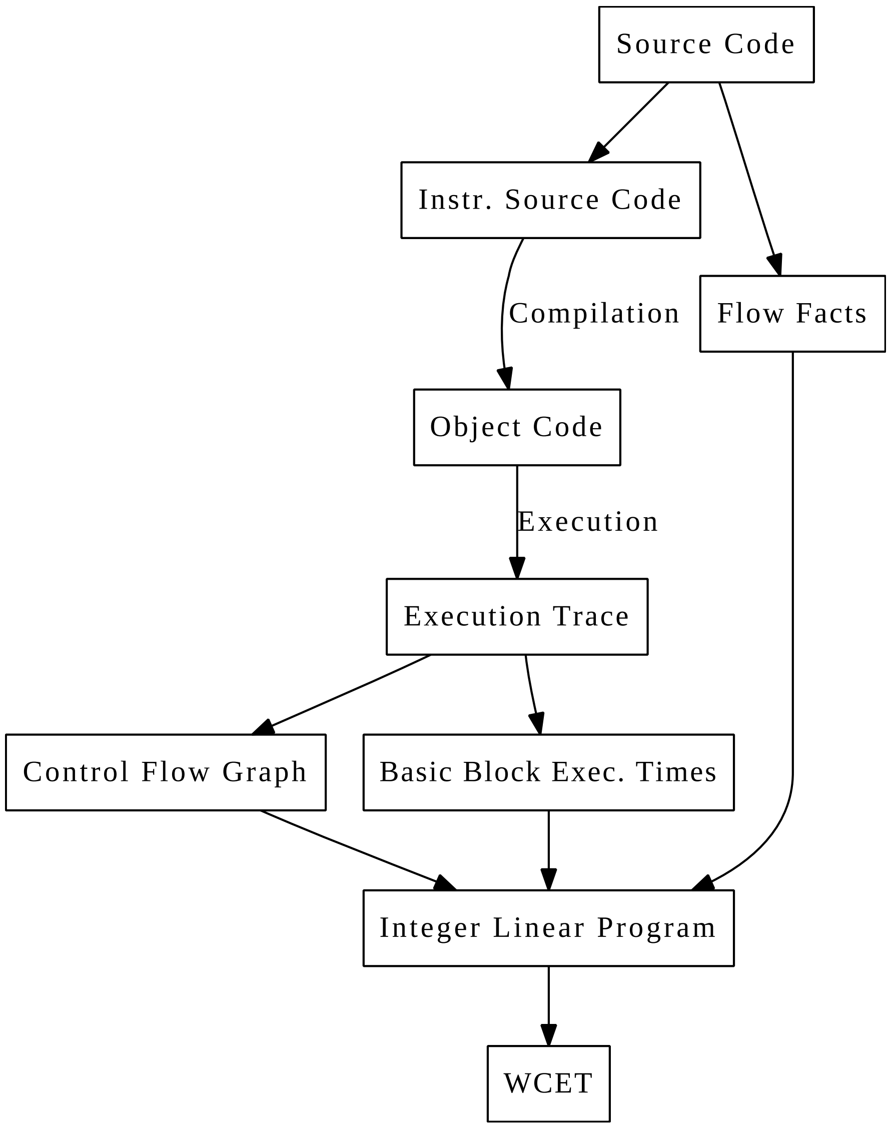

Figure 5.6: The tool-chain used for WCET Estimation

optimisations which reorder basic blocks.

A critical aspect of the validation of the execution time estimation process is to ensure that all branches in the program have been adequately measured. This can be achieved by performing either coverage testing, or by comparing a representation of the observed CFG with one produced directly from the disassembled object code of the planner. These processes can then be used to guide the introduction of additional instrumentation statements, or to selectively re-execute the planner in different initial conditions so as to improve the coverage of execution traces.

5.2.1 CFG Edge Execution Time

The execution trace analysis approach presented above is a useful method for extracting all execution time data for each visit to each edge in the CFG. In order to apply Integer Linear Programming techniques, it is necessary to have a single estimate of the execution time of each edge in the CFG.

The simplest approach for calculating a per block execution time is to simply select the maximum observed execution time. Unfortunately, the accuracy of single measurements can be severely affected by task switches, cache misses and pipeline stalls. This can result in the execution time of some blocks being grossly overestimated. A more refined approach is to find a probability distribution function that models the execution time of each edge in the CFG, and to then select a value that is statistically unlikely to be exceeded. One suitable family of distributions are those commonly used in extremal value statistics [Edg02].

In cases where the measured execution times of the edges in the CFG exhibit wide variation, or in cases where edges are executed with high frequency, the expected value of the total time spent executing code on that edge could be significantly overestimated. This factor could potentially result in an pessimistic estimate of execution time for the entire algorithm.

The approach adopted in this work is to calculate the mean of the observed execution time of each edge in the Control Flow Graph. The advantage of this approach is that it is simple to compute, and that it reflects a best estimate of the expected value of the total execution time spent in each edge. The results presented below demonstrate that even this optimistic measure of block execution time can be sufficient to allow a coarse estimate of WCET.

This approach to estimating execution times in the CFG is susceptible to errors introduced by insufficient sampling, and if the execution time measurements are not IID (Independent and Identically Distributed). Features such as caches and branch predictors can result in the execution times of basic blocks being correlated which influences the accuracy of such a model. The limitation of this approach is that it can underestimate the execution time of small numbers of executions of a particular edges, leading to underestimation of WCET.

One of the challenges of using a probabilistic execution time modelling method is that the execution time estimates for each edge are assumed to be uncorrelated to CPU state. In particular, the RRT was tested running at maximum priority, and with only a minimal set of operating system tasks active.

A critical aspect of measuring execution time is the capability to measure time quickly, accurately and with minimal latency. While operating system services do offer timers, the values of such clocks have are updated with an interrupt that is triggered in the order of 1 ms. While on QNX it is possible to decrease the clock sampling period, this can introduce a significant overhead on time available for all scheduled tasks [QNX07].

A more common approach in the real-time literature is to make use of a hardware time-stamp counter [Edg02, Pet02]. For example, modern x86 architecture, machines are equipped with a *TSC* Time-Stamp Counter register, that can be accessed within a *RDTSC* Read Time-Stamp Counter assembly code instruction [Int07]. The rate at which this counter is updated depends on the hardware model and configuration, but tends to be updated at sub-nanosecond rates. This allows finer granularity of timing measurements, and restricts the impact of clock reads to a single process.

5.2.2 Worst Case Execution Time

The actual process to derive the Worst-Case Execution Time can be found by converting the Control Flow Graph, and the associated execution times to a maximum network flow problem [PS97]. In order to apply ILP to finding an estimate of WCET, the CFG is required to have one source and one terminal basic block. Furthermore, at least one path from the source to the terminal block exists. In practice, these requirements are met for most common types of software.

Furthermore, while the CFG describes flow, it does not impose any limits on the number of visits to each part of the program. Thus, to find a WCET, it is necessary to impose additional constraints to limit the number of times backward edges in the CFG can be executed. In particular, this requires the number of iterations for all loops to be bounded. In the case of the RRT algorithm, the planning parameters $(n, |\mathcal{U}_d|)$ can be used to impose these limits. Additional constraints can also be imposed as required to tighten the execution time estimate.

Research has been conducted into automatic identification of the parameters which influence the execution time of a program, such as loop bounds (sometimes described as flow facts [GESL06]). In this work the set of parameters which influence execution time were derived by manual inspection of the original source code. A combination of the flow facts, the CFG and the basic block execution times can then be used to produce an Integer Linear Program, that can be solved using standard tools. The solution of such a problem identifies an estimate of the worst case execution time. By varying parameters (in particular loop bounds), it is possible to identify the maximum planning effort which will result in a safe, but tight, WCET.

5.3 Dynamic Planning with RRT

Having shown that a method exists to measure the execution time of an arbitrary motion planning algorithm, that approach can then be applied to the RRT algorithm. The RRT can then be inserted into a cyclic hard real-time task, subject to the assumption that a new plan is generated in each time-step, ignoring any previous plans. The HRT-RRT, which combines dynamic motion planning with hard real-time deadlines is shown in Figure 5.7.

```
1: procedure HRT-PLANNER
2:     INITIALIZE-PLANNER
3:     $t_{next} \leftarrow$ CURRENT-TIME
4:     loop
5:         $t_{next} \leftarrow t_{next} + T_p$
6:         $\mathcal{B} \leftarrow$ updated map
7:         $x_g \leftarrow$ current goal states
8:         $x_i \leftarrow$ state of the vehicle
9:         $\mathcal{T} \leftarrow$ BUILD-RRT($x_i, x_g$)
10:        $\tau \leftarrow$ EXTRACT-PATH($\mathcal{T}$)
11:        publish $\tau$
12:        YIELD-UNTIL($t_{next}$)
```

Figure 5.7: Pseudo-code of the RT-RRT algorithm

As is the case with the basic implementation in Figure 4.1 this algorithm consists of three distinct stages, sensing and updates to the model, planning, and publication of the path to execute. The sensor updates should be designed to compensate for latency, as per Section 4.2.2. This means that the mapping and localization estimates are based on predictions of where the vehicle will be when the plan is ready.

The primary addition in this version of the RT-RRT pseudo-code is the addition of the function to extract a path from the search tree generated by the RRT. In cases where a feasible path to the goal can be found, the path is returned. If no feasible path to the goal is found, an (approximately) safe partial path can be produced by selecting a path in the tree that would take longer than the planners period to execute. This ensures that sufficient time remains to make at least one more attempt to find a safe plan.

One of the limitations of the RRT algorithm is that different solutions will potentially be found each time the algorithm is run. This variation in paths is introduced by the randomness in the sampling process. The result of this variation can be considered in terms of an overall effect on the cost of reaching a goal. In order to ensure that the RT-RRT algorithm will converge to the goal state, an additional criteria is enforced, that a path is only followed if it can be shown to have lower cost than that of the cost of following the previous path. Pseudo code for this strategy is shown in Figure 5.8.

5.4 Experiments and Results

This section draws together the material from this chapter, and presents an analysis of the HRT-RRT algorithm.

```
1: procedure EXTRACT-PATH($T$)
2:     $\tau_{k+1} \leftarrow$ last valid path to goal
3:     $\tau_k \leftarrow$ current path from the search tree $T$
4:     if $\gamma(\tau_{k+1}) < \gamma(\tau_k - L(\tau_k))$ then
5:         return $\tau_{k+1}$
6:     else
7:         return $\tau_k$
```

Figure 5.8: Pseudo-code for incremental cost minimisation

5.4.1 Models

Up to this point, it has been possible to provide a general strategy that is independent of hardware platform, vehicle and sensor models and representations of geometries. This work focuses on a single example, in order to demonstrate the end-to-end process involved in producing a hard real-time motion planner.

Vehicle Model

The selection of models, and the development of the simulation environment were aimed at producing a motion planner that could eventually target the Mullaya class AUV [VWD05, VN07] produced by DSTO's Maritime Platforms Division. Rather than attempting to produce plans for the complete vehicle model, which takes system dynamics, hydrodynamics and propulsion into consideration, this work employs a constrained kinematic approximation to the true vehicle.

The Dubin's car [Dub57] model is a well known non-holonomic kinematic vehicle model, with a restricted turning radius, and constant speed. The Dubin's car can be thought of as an approximation to the standard operating conditions for Mullaya manoeuvring at a fixed depth, with constant speed and assuming minimal effects introduced by dynamics. The basic equations of motion for this model are:

$$\begin{aligned} \dot{x} &= v\cos\theta \\ \dot{y} &= v\sin\theta \\ \dot{\theta} &= \omega \end{aligned} \tag{5.1}$$

where the position of the vehicle is (x, y), and the current heading is θ. The inputs to the model are the forward velocity v, and the input heading rate command ω. The forward velocity is set to be 1.5 m/s, which is the trim velocity for Mullaya in normal operating conditions. The constant forward velocity term can be thought of as a condition that ensures it is always in motion, which prevents hydrodynamic stall from becoming an issue. A 10 meter turning radius, which equates to a maximum heading rate of $|\omega| \leq 0.125$ radians per second is also enforced.

Dubin's original work on this model showed that the optimal set of paths for this model can be constructed by composing using extremal inputs, hard left, hard right and straight, where the straight sections are singular arcs. In this work, this minimum set of discrete inputs is adopted: $\mathcal{U}_d = \{-0.125, 0, 0.125\}$. A small number of intermediate values for ω could potentially be used to improve convergence under some conditions.

A further necessary approximation is to select an appropriate integration time-step δt to convert the continuous time model to a discrete time model required for use in the RRT algorithm. This value can be hand tuned for a given tree size to achieve an acceptable compromise between coverage and connectivity.

The vehicle is assumed to be able to track the path produced by the planner perfectly. While in practice some paths cannot be followed perfectly, this assumption is equivalent to concept of gross motion planning described in Section 2.1.1, provided that the vehicle stays within a specified distance of the path, collisions can be prevented by enlarging the perceived size of obstacles in the environment.

Sensor Models

The basic structure proposed in this thesis involves having decoupled the sensors from the planner, consequently, the most important issue related to the processed observations from the sensors relates to how obstacles in the workspace are modelled. This has a significant impact on the capability to estimate the worst-case execution time, and consequently the planner.

The approach used in this chapter is to model each obstacle as circular disks. This representation has much in common with the tracking of features in Simultaneous Localization and Mapping Problems, and is frequently employed in dynamic motion planning literature [vdB07].

When the geometry of the vehicle is approximated by a circular disk, centred on at p_x, p_y and sufficiently large to encapsulate the entire vehicle. For a maximum of m obstacles, the running time of a naive collision detection test between the vehicle and each obstacle, will take $O(m)$ time. Provided that an upper bound on the number of possible obstacles that can occur is known, it is possible to estimate the WCET of the collision detection routine.

This approach to collision testing can be implemented computationally cheaply, without regard to the vehicles heading. The limitation of this approach is that the bounding approximations tend not to be tight, which can result in the planner not detecting some feasible paths. This type of motion planning is accepted as part of the trade-off for improved collision detection performance.

While this approach to collision detection has been adopted for this experiment, it is worth noting that it is not particularly efficient, nor is it the only possible implementation of a hard real-time collision detection strategy[GB96]. Other strategies, such as spatial partitioning can also significantly reduce the expected running times of collision detection.

Task Timing Parameters

In order to examine the effects of varying degrees of execution time and search quality, the period of the planner is set to 2 seconds, or an update rate of 0.5Hz, assuming that this is the fastest rate at which maps can be updated reliably. The planner is required to operate over a range of execution times, up to a maximum of 600 ms, which equates to a CPU utilisation of less than 30%.

Hardware

The target platform for the experiments was an Intel Pentium 4 Xeon. To minimise the measurement errors caused by differences in the initial state of the processor, the cache was invalidated. Serializing instructions were inserted to ensure that the instruction pipeline was empty prior to commencing measurement.

The real-time operating system used for calibration and testing was QNX 6.3 [QNX07]. During calibration, high frequency hardware interrupts were disabled, and the priority of the planner was boosted to maximum. This limits the number of tasks which can pre-empt the planner to those which are critical to the operation of the kernel.

Workspace

The operational area for the experiments was a rectangular workspace with dimension 100 meters by 200 meters. This region is equivalent to the nominal area normally used for testing the Mullaya AUV. The initial and goal configurations were selected to be at opposite ends of the configuration space.

Within the workspace, the vehicle is considered to have successfully reached the goal if it closes within a distance of 10 meters of a target point, with any heading. A large goal was selected to maximise the probability of the vehicle reaching the goal.

5.4.2 Calibration Trials

The first set of experiments were aimed at producing the correlation between the search parameters and execution time. Following the process laid out in Section 5.2, the source code was written, debugged and instrumented.

The locations of obstacles within the workspace are a factor which can have a significant influence on the execution path followed by the RRT algorithm. This observation can be exploited to ensure that the coverage of instrumentation is adequate. To achieve good measurement coverage three basic workspaces were considered: the empty environment (with no obstacles); the infeasible workspace (with a single obstacle obstructing any path to the goal); and a further obstructed workspace, designed to be feasible, but more difficult than the empty workspace.

The empty, infeasible and obstructed workspaces, along with sample search trees produced using RRT in each are shown in Figures 5.9, 5.10 and 5.11. The boundaries of these images denote the shape of the workspace, the dark grey circles denote the obstacles, the red circles are the location of the goal configurations, and black lines are indicative of the search trees produced by the RRT algorithm during the experiments.

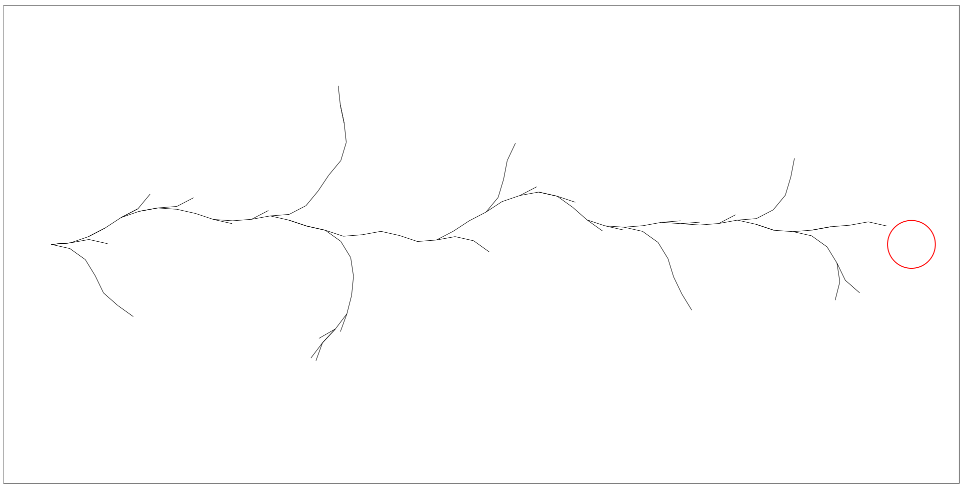
Figure 5.9: The Empty Workspace

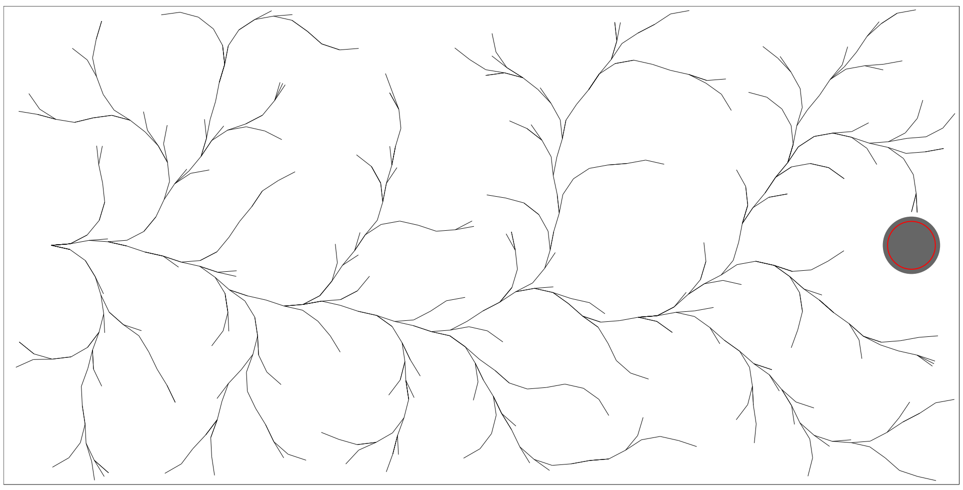
Figure 5.10: The Infeasible Workspace

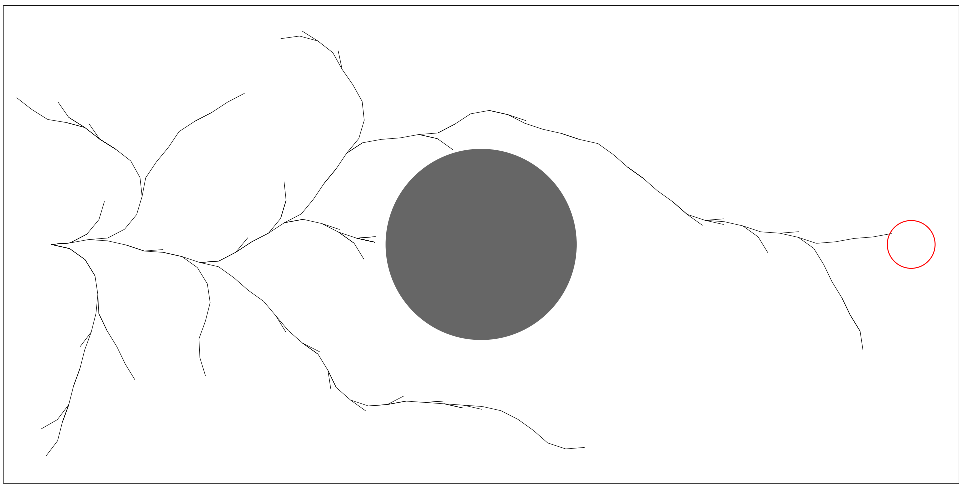

Figure 5.11: The Obstructed Workspace

The empty workspace exercises the branches in the RRT algorithm that deal with successfully finding the goal. This means that only the constraints on the motion of the vehicle can prevent a solution from being found. In a workspace of this shape, this environment is effectively the best possible case.

The infeasible workspace is an example of a problem that is close to the pathological worst-case. The presence of an obstacle preventing access to the goal results in the search tree expanding, without early termination because a goal is found. This case is expected to result in near worst-case performance.

The obstructed workspace was selected to represent a more typical case. The narrow passages around the obstacle result in an interleaving of expansions, and collisions that would indicate whether any micro-architectural effects were having a significant contribution to the final result.

Within each of the workspaces, 100 trials were conducted to produce a composite execution trace. Additional trials could be conducted to improve the quality of the execution time fits for each edge in the CFG, but these values tended to converge with many fewer than 100 trials. Similarly, a much smaller number of trials could have been used to identify all of the edges within the CFG.

An example of a subset of the Control Flow Graph produced by analysing the combined execution trace, with calls to functions omitted, is shown in Figure 5.12. This highlights that the key branches within the RRT algorithm all appear within the CFG [3]. For example, The cycle of vertices 8, 9, 3, 4 and 7 are a result of the main loop within the RRT, and the branch between 9 and 10 is caused by the early exit of the planner when a solution has been found.

In order to ensure that an edge is measuring an approximately constant time operation, it

[3]The labels on the CFG in Figure 5.12 are a function of the instrumentation process, and are only shown to assist in describing parts of the algorithm

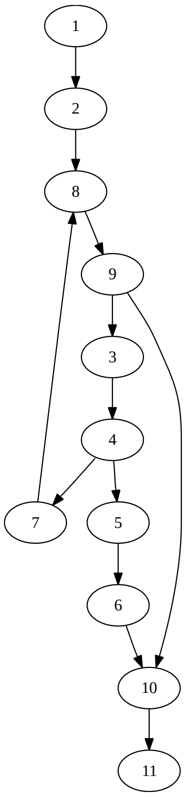

Figure 5.12: An subset of the complete Control Flow Graph for the RRT planner

is useful to plot of the execution times of one such edge. A plot of the observed execution times for a representative edge are shown in Figure 5.13. This plot also highlights that the execution time for an individual edge includes a number of spikes in execution time, outliers. The frequency of execution times (Figure 5.14) shows that while the mean execution time for this execution times of the edge is about 150 clock cycles, the maximum value is an order of magnitude larger.

These results are representative of the distribution of execution times that occur for the majority of edges in the CFG. The bulk of the execution times are distributed about a single value, with a small number of outliers that have a significantly larger execution time. The presence of such outliers demonstrates the difficulty of selecting a single value for modelling the execution time of such edges.

The CFG and the execution times of each of the basic blocks were then used to produce a series of Integer Linear Programming problems, where the value of the search tree size was varied. This allows the WCET to be identified as a function of search tree size. This forms the basis for comparison with all of the total observed execution times that result from running the planner in each of the three calibration workspaces.

The execution time profile from multiple searches in the empty workspace are shown in Figure 5.15. A high proportion of the observed execution times are distributed close to zero,

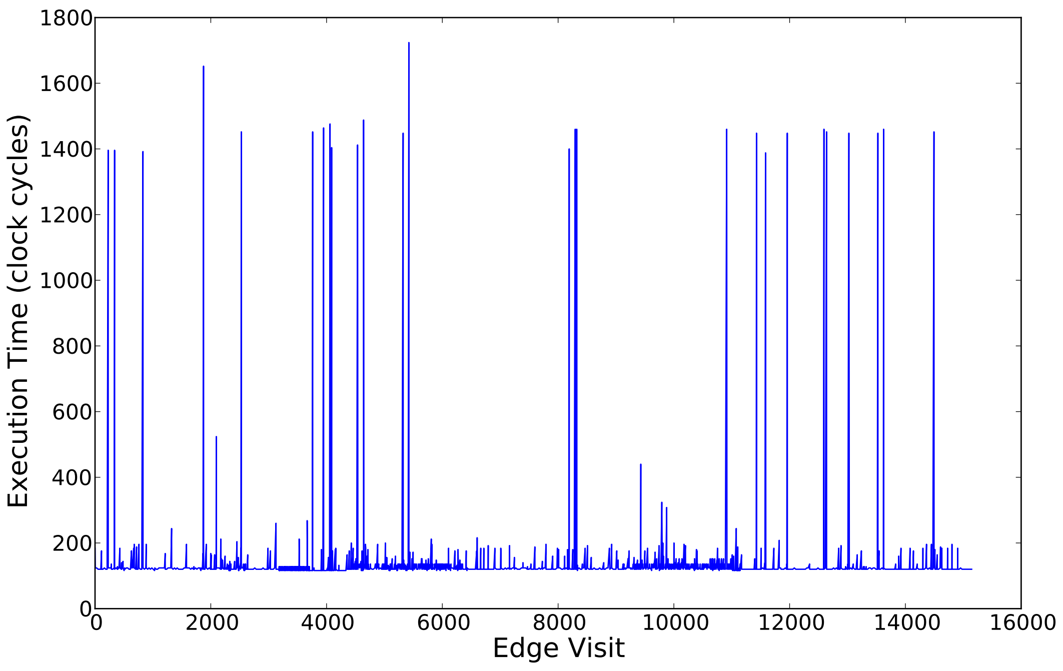

Figure 5.13: Observed execution times for an edge in the CFG

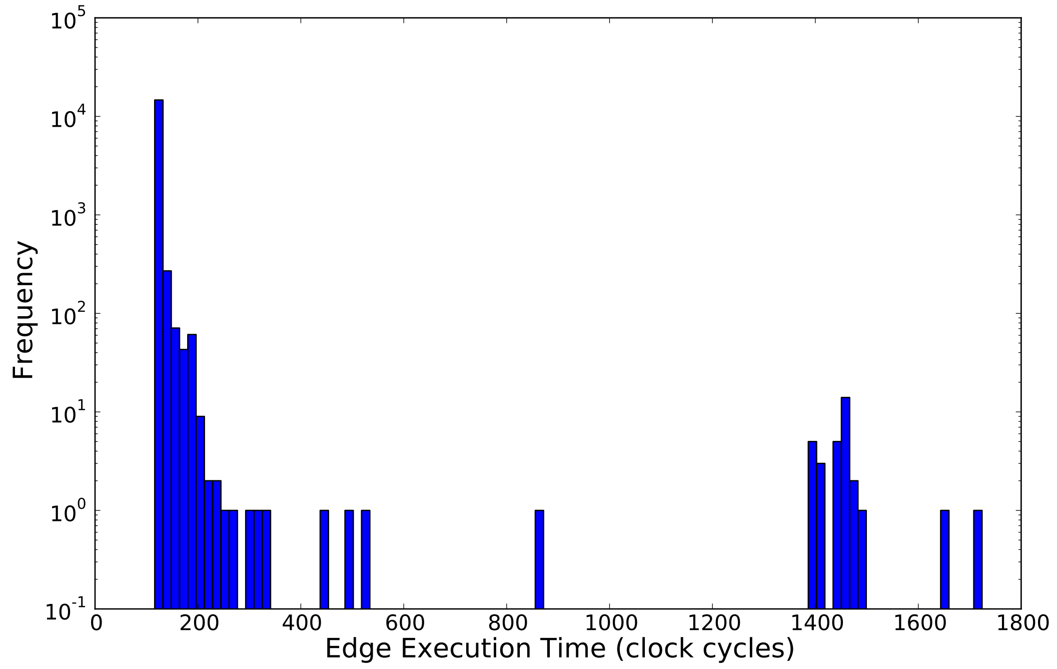

Figure 5.14: Frequency of observed execution times for an edge in the CFG

regardless of the number of nodes that can be added to the search tree. The small number of nodes close to the WCET occur when the planner fails to find a solution, which typically because of the under-sampling of the set of control actions. These results indicate that when the problem is easy, a solution tends to be found quickly, relative to the worst-case execution time.

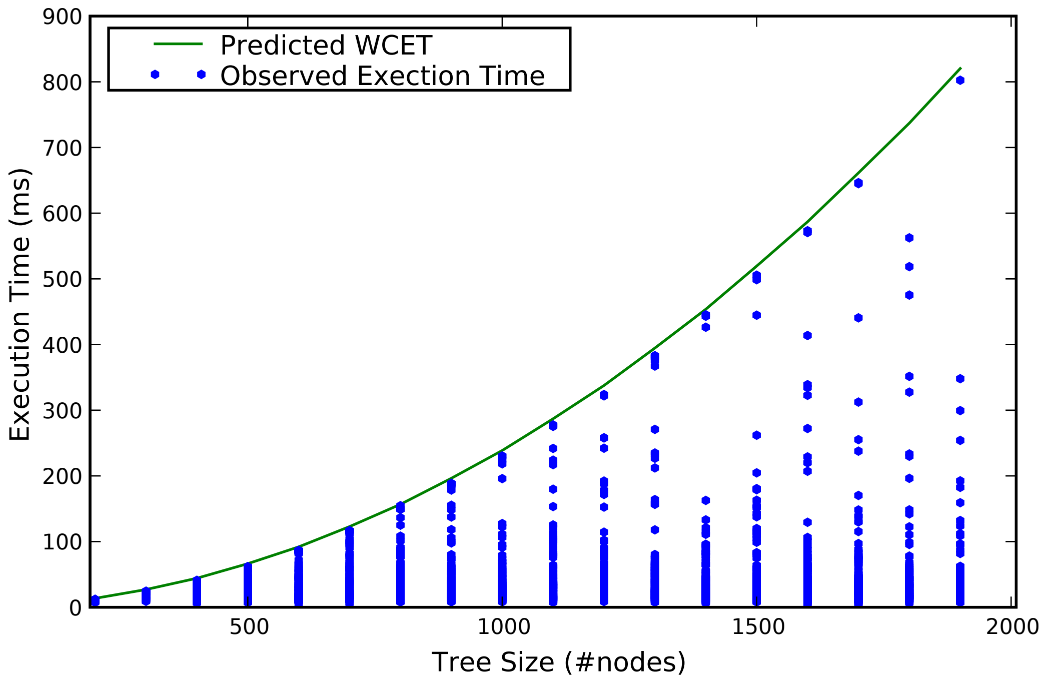

Figure 5.15: Comparison of predicted WCET and observed execution time in the empty workspace

The results of the execution time profile of the RRT planner running in the infeasible workspace are shown in Figure 5.16. This demonstrates a clear difference to the empty workspace. This occurs because it is not possible to find a path to the goal, the planner will continue searching until the available storage resources are exhausted. This results in the observed execution times being tightly clustered just below the theoretical WCET curve.

The results for the obstructed workspace are shown in Figure 5.17. They show that the distribution of observed execution times for a problem that is more difficult than the empty workspace, and easier to solve than the infeasible case. The observed execution times, regardless of search tree size, are more skewed toward the WCET curve than for the empty workspace.

Significantly, this result also highlights the clear dependency between search tree size, and the amount of execution required to find a solution in the worst case. By considering a series of planning problems with varying obstacle placement demonstrated that there is also a correlation between the difficulty of a planning problem and how much time is required to find a solution.

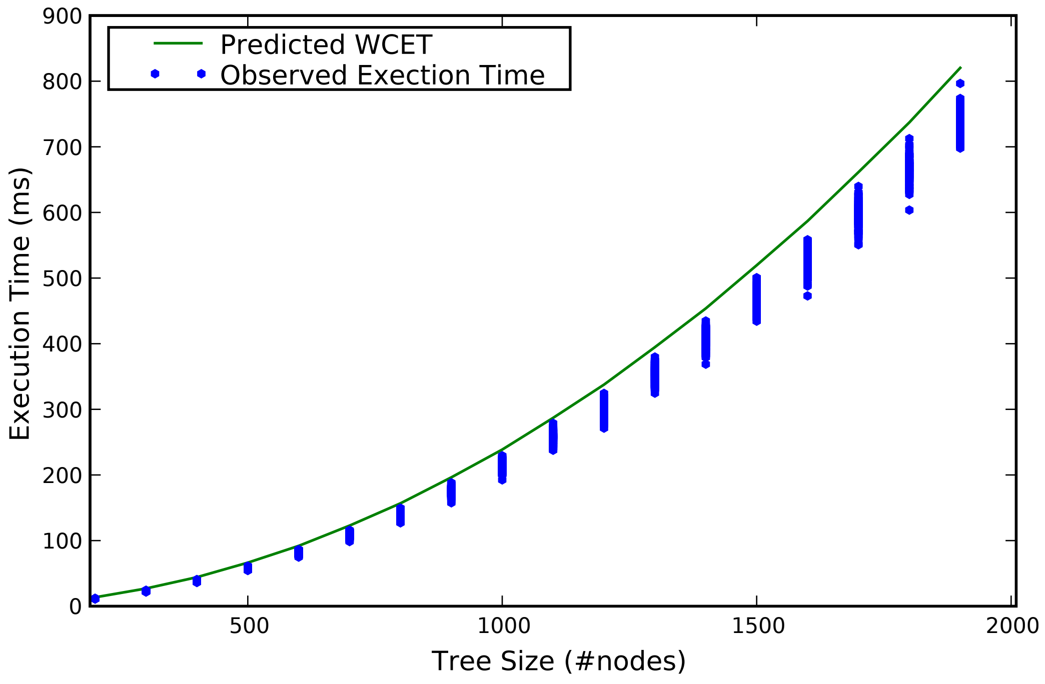

Figure 5.16: Observed execution times in the infeasible workspace

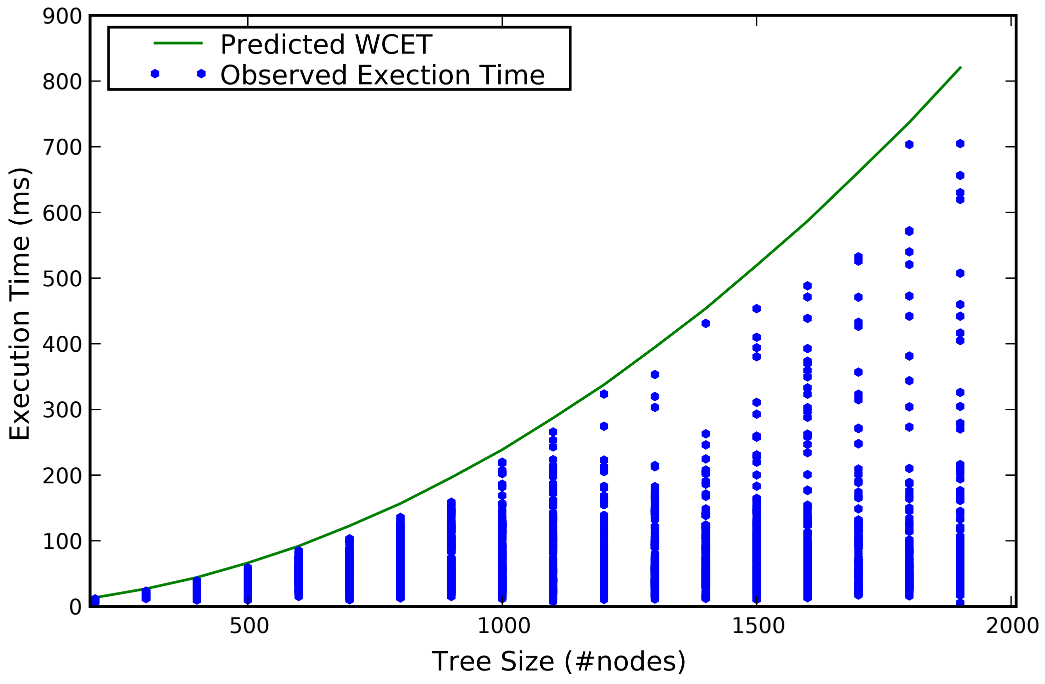

Figure 5.17: Observed execution times in the obstructed workspace

These results also show that no overruns of the predicted WCET were observed, even using an optimistic measure of execution time for edges in the CFG. While this does not preclude an overrun from occurring, it is a good indication that such an event would occur infrequently.

Using this approach for calibration, and then verification of results, it is possible to find the maximum size of the search tree which ensures that the planner can satisfy a deadline. For the examples shown, a 600ms deadline equates to a tree size of approximately 1600 nodes.

5.4.3 Measurement Error

In addition to the previous description of the execution time spent in each edge of the CFG, it is also useful to consider the measurement errors introduced over the total time spent constructing a plan. Such measurements are a useful tool for understanding how well execution time is being estimated.

One approach to estimating such errors is by calculating the ratio between estimated and observed execution times. The estimated execution time is calculated as a sum of the execution times of each edge in the CFG (using eq. 3.7). This decouples the measurement error from both the environment and planning parameters.

One such set of results are shown in Figure 5.18. For the example chosen, the relative error is normally distributed, and is centred on zero mean relative error. This error distribution indicates that the simple model of mean edge execution times will result in an underestimate of execution time in some cases.

Ideally, the results in Figure 5.18 could also be used as part of a process to compensate for measurement errors by modifying the execution time budget of a task. While an approximate normal distribution model of expected execution error could be used to reduce the probability of overrun, characterising such a distribution would depend on how both relative and absolute errors vary as a function of search tree size and planning problem. Accurately quantifying such errors would be equivalent to the task of exhaustively searching for a WCET, and is consequently impractical.

The systematic errors in expected execution time arise because of the limitations of using the simplified mean observed execution time for each of the edges in the CFG. An alternative execution time model that incorporated dependencies between edges, or hardware characteristics, could be used to minimise such errors.

5.4.4 Coverage Tests

Having calculated the maximum number of nodes that can be added to the search tree in the allowable deadline, it is possible to test the performance of the RT-RRT algorithm as a whole in an online scenario. The selected test environments are based on a randomly generated workspace, populated with m static obstacles of varying sizes.

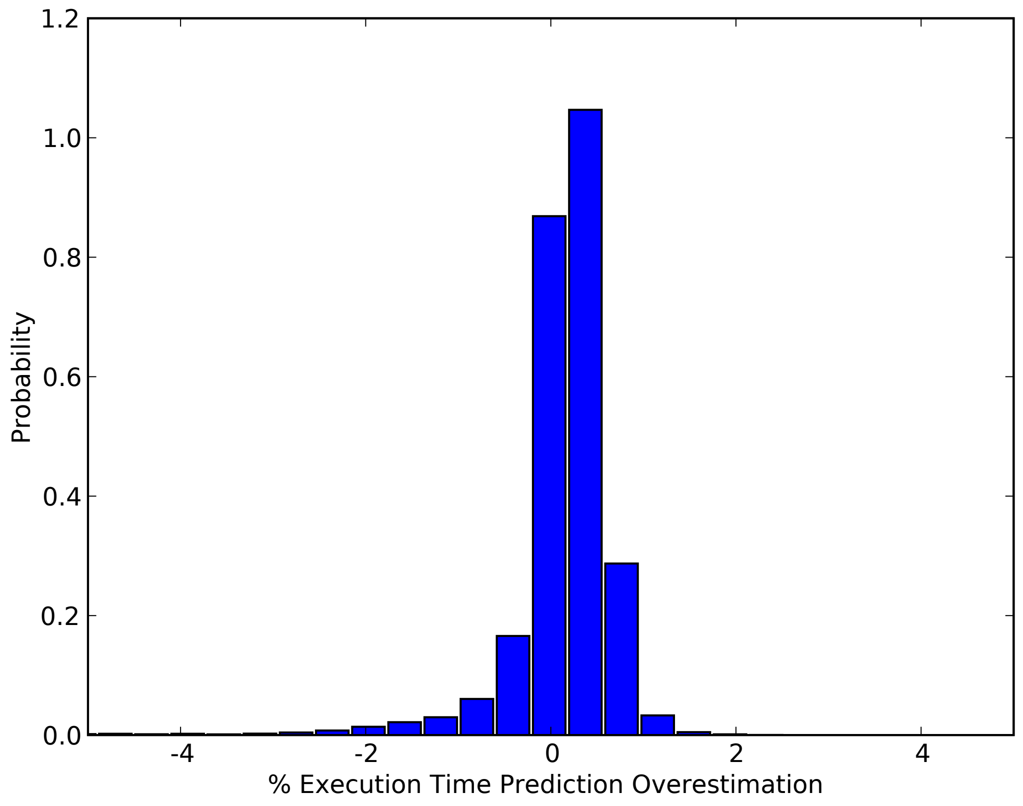

Figure 5.18: Probability distribution function showing the probability of the predicted execution time being overestimated, when compared to observations.

The experiment is conducted in a partially observable environment. The simulated vehicle is capable of sensing any obstacle within 20 meters. This ensures that the planner has a safety horizon much larger than required, and as such, should be able to avoid collisions.

Results

The results in Figure 5.19 demonstrate the evolution of the path of a vehicle through a randomly generated environment, using the constraints on time available for planning described in the previous sections. Each diagram includes shows the true locations of obstacles and the obstacles visible to the vehicle in a given time-step. The current path produced by the planner is also marked on the diagram, as is the search tree identified during that planning period.

In the first diagram, the planner identifies an initial path, through a workspace that is assumed to be initially empty, the locations of the obstacles are shown in light grey. The minor deviations in the path generated by the planner are an artefact of the random sampling employed by the RRT algorithm, and could potentially be removed in a post-processing step.

The second diagram shows the configuration of the vehicle have safely avoiding a group of obstacles, shown in known in dark grey. The planner continues to be able to identify a safe path to the goal configuration. In the third diagram, a new obstacle has been observed that invalidates the previous path. Further search was conducted, but because no path to the goal could be found during planning, a temporary safe partial plan is being followed, until a more suitable plan becomes available.

In the final diagram, the vehicle approaches the goal state, prior to the termination of the search, with the locations of most of the obstacles now known.

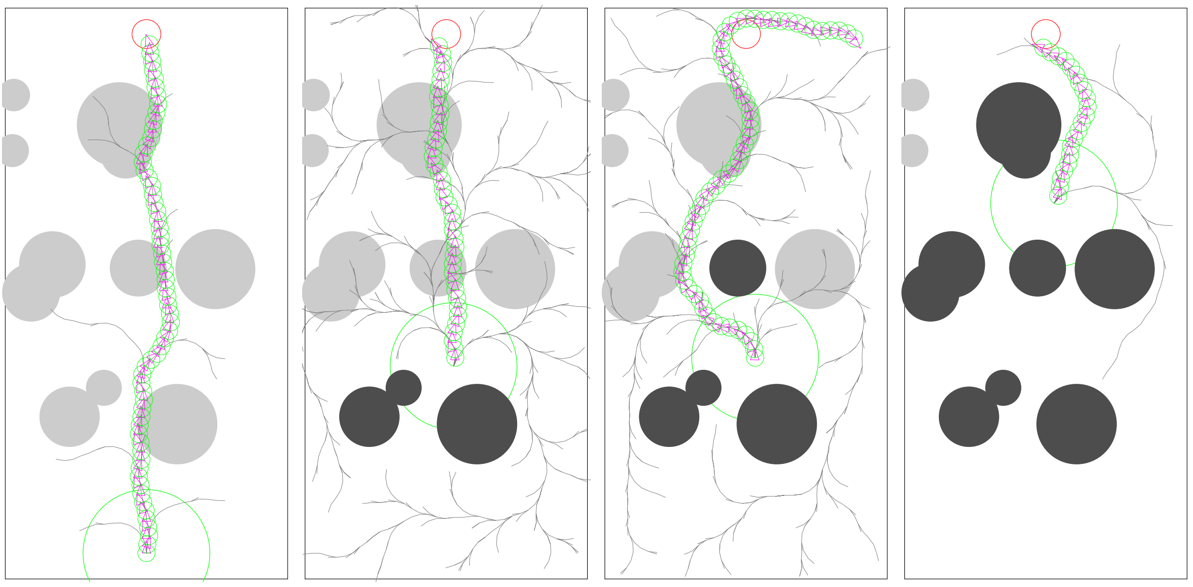
Figure 5.19: An example of RT-RRT replanning in partially observable environment with static obstacles.

5.5 Summary

This chapter has presented the Real-Time Rapidly-exploring Random Tree algorithm (RT-RRT) using the framework developed in chapter 4. This algorithm demonstrates that it is possible to solve motion planning problems, when a hard real-time constraint is imposed. By selecting an appropriate measure of search effectiveness, in this case the number of nodes to be added to the search tree, a method for managing the trade-off between execution time and search efficiency was achieved - key factors which influence the capability of a planner to interoperate with other tasks in a hard real-time system safely. Finally, the RT-RRT algorithm was demonstrated in the context of a simulation where planning, sensing and actuation where interleaved in a dynamic workspace with obstacles that were detected online.

This work also highlights some of the challenges of working with hard real-time systems in practice. Producing tight bounds on WCET is complicated by any interference that can be introduced within the kernel including interrupts and task switches, or by hardware optimisations. These factors introduce significant variation in the observed execution times for each edge which makes formulating models that are both tight and pessimistic challenging. Even in the presence of such errors, it was also demonstrated that for long running processes, mean observed execution times for each edge in the CFG can be used to develop a tight, but potentially optimistic, model of WCET.

Having documented the limitations of the approach used to measure WCET it is worth noting that the approach for selecting parameters that optimise search tree size, and consequently influence the effectiveness of the search, does not depend on a particular strategy for measuring WCET. This means that in the future this strategy could be adapted to make use of more accurate techniques for calculating WCET.

CHAPTER 6

Generalized Hard Real-Time Rapidly-exploring Random Trees

This chapter provides a further demonstration of a hard real-time dynamic motion planner, in the context of the framework proposed in Chapter 4. The aim of this chapter is twofold - to demonstrate how to incorporate a large class of existing motion planning algorithms into a hard real-time system, and also to highlight a technique for improving the efficiency of real-time search algorithms.

This work builds upon the concepts described for the RT-RRT algorithm. In that case, the efficiency of the search was maximised by imposing an upper bound on the size of the search tree which could be grown, while ensuring that constraint on available execution time was satisfied. This was achieved by finding a relationship between search-tree size and execution time.

One of the challenges of generalising the strategy developed for RT-RRT is that if the difference between observed worst-case performance and the predicted worst-case is large, then this slack time will be recovered by the scheduler. One potential approach for improving the search performance is to find a way to make use of this slack time.

The application of motion planning algorithms to a broad range of computationally difficult problems has led to a wide range of strategies to optimise search performance. In the case of probabilistically complete planning algorithms, one common optimisation involves trying to increase the rate at which the algorithm converges to finding solutions [KL00, HKLR02]. The effect of this optimisation is to reduce the average execution time required to find a solution to a planning problem.

This chapter will show the effect of using planning algorithms which converge to completeness tend to result in reducing the average execution time required to find a solution, and that this consequently results in the introduction of slack time. This situation is representative of a more general case than RRT, and by finding an approach to make use of any slack time, it becomes possible to make efficient use of a wider range of algorithms for hard real-time planning.

The remainder of this chapter is presented in two parts, the first involves further discussion of the RRT algorithm and the introduction of a variant algorithm, the RRT-LPM. The results of that work are then used as a basis for developing a strategy for developing hard real-time planners, applicable in a broader context. Finally, a concrete example of this process the HRT-RRT is presented and discussed.

6.1 Planning Optimisations

One common technique for increasing the rate at which a search algorithm will converge to a solution is to employ a greedy search technique. A greedy search is one which biases search toward parts of the search space which are most likely to assist in finding a solution. A well known extension of the RRT algorithm is Kuffner and LaValle's RRT-Connect [KL00]. RRT-Connect is greedy in the sense that it attempts to greedily reduce the Voronoi bias by growing long branches in the search tree. Put another way, the RRT-Connect algorithm tries to greedily achieve coverage of the configuration space.

The basic algorithm for RRT-Connect is very similar to that of RRT (Fig. 5.1). The difference between the two comes in the approach used to extend the existing search tree. Rather than new states being added to the search tree via a single step of forward simulation, a connection process involves repeated extension of the search tree from an existing node in the search tree.

One common generalisation of search tree growth via extension or connection is to assume that a Local Planning Method (LPM) exists [LaV06]. A LPM can be used to construct paths or trajectories that are kinodynamically feasible, but ignore the effect of obstacle constraints. Effectively, an LPM can be defined as a function that produces a path from an initial configuration toward a target configuration. $f : \mathcal{C} \times \mathcal{C} \mapsto x(s)$ where $x(s) \in \mathcal{C}$, is a path parameterised in terms of $s \in [0, 1]$, where $x(0) = q_{init}$ and $x(1) = q_{final}$.

While it is possible to apply RRT-Connect to complex motion planning problems subject to kinodynamic constraints, there have been a number of specialised alternatives that can construct trajectories in a more efficient manner. Two specific examples of planners that employ a Local Planning Method are Hsu's Expansive Space Tree [HKLR02], and Frazzoli's Manoeuvre Automaton [Fra01]. In both cases, the Local Planning Method is used to reduce the number of configurations being required to achieve coverage of the configuration space. The result of this is that these planners are shown to have excellent performance in the average case.

In cases where the motion of the vehicle is subject to constraints, tree based planners typically relax the requirement on a LPM to exactly reach a target configuration. Instead, it is sufficient to find any feasible path which has a final configuration near (according to a metric) the goal. Furthermore, while it is desirable for a LPM to be (near-) optimal, this is not a requirement on the function. These relaxations on trajectory generation imply that a LPM that can generate feasible paths between a pair of configurations can be found in a number of ways. Suitable approaches for constructing a LPM vary according to the constraints on the vehicle. A summary of some relevant techniques for path and trajectory generation that could be applied to the development of LPMs are presented in [TBF05].

6.1.1 RRT-LPM

This section proposes the Rapidly exploring Random Tree augmented with a Local Planning Method (RRT-LPM). RRT-LPM builds on the RRT and RRT-Connect algorithms, but rather than linking configurations by forward simulation of the vehicle model dynamics, it employs a Local Planning Method for connecting configurations. The major difference from the basic implementation of the RRT described in Chapter 5 is that the inter-connection of configurations and the process of collision detection need to be altered slightly.

The approach used for generating collision free trajectories connecting configurations is shown in 6.1. The basic process for connecting vertices involves using the local planning method to generate a path. This path can then be collision tested at regular intervals of δ, and when the final configuration is reached, the edge can be added to the search tree.

```
1: procedure LPM-CONNECT($x, x_{target}$)
2:     $T \leftarrow$ TRAJECTORY-GENERATION($x, x_{target}$)
3:     $s \leftarrow$ TRAJECTORY-STEPS($T, \delta$)
4:     for $i \in [1, s)$ do
5:         $T_i \leftarrow T(i)$
6:         if ISINCOLLISION($T(i).x$) then
7:             break
8:     $\mathcal{T}$. ADD-EDGE($x_{near}, T(s)$)
```

Figure 6.1: The basic connection algorithm, as commonly described

The approach presented for generating path segments has the significant limitation that the algorithmic worst case is bounded by the maximum possible length of the path, and the size of the sampling step. In order to ensure that an upper bound on the execution time for the LPM-Connect function can be identified an additional constraint must be imposed on the length of such paths. This can be achieved by imposing an upper bound on the number of samples, $s \leq s_{max}$, and consequently collision tests, that are required to show that a path is collision free.

This constraint on the maximum number of samples limits the maximum length of any edge that can be added to the search tree. This means that the set of configurations that are reachable from the search tree will grow more slowly than for a planner not subject to this constraint.

A further well-known optimisation to the connection strategy involves adding additional branches to the search tree at the intermediate sampled configurations [KL00, Fra01, Hsu00].

This has the effect of improving the quality and convergence to completeness by influencing the coverage and connectivity of the search graph. This requires some additional book-keeping to ensure that the maximum number of nodes in the search tree is not exceeded.

The final connection algorithm with the enhancements for bounding the algorithmic worst-case, and introducing intermediate nodes is shown in Figure 6.2. This variation assumes that only every M-th sampled configuration is added to the search tree, and that s_{max} is the largest number of samples to be considered.

```
1:  procedure LPM-CONNECT($x, x_{target}$)
2:      $T \leftarrow$ TRAJECTORY-GENERATION($x, x_{target}$)
3:      $s \leftarrow n - \mathcal{T}.size -$ TRAJECTORY-STEPS($T$)
4:      $s \leftarrow \min\{s, s_{max}\}$
5:      for $i \in [1, s)$ do
6:          $T_i \leftarrow T(i)$
7:          if ISINCOLLISION($T(i).x$) then
8:              break
9:          if $i$ mod $M == 0$ then
10:             $\mathcal{T}$. ADD-EDGE($x, T(i).x$)
```

Figure 6.2: The revised connection algorithm

The result of the added complexity of using the LPM for connecting configurations means that the structure of the planner needs to be altered slightly (Fig. 6.3).

```
1:  procedure BUILD-TREE($x_{init}, x_{goal}$)
2:      $\mathcal{T}$. INIT($x_{init}$)
3:      for $i \in [1, k)$ do
4:          $x_{rand} \leftarrow$ SAMPLE-STATE
5:          $x_{near} \leftarrow$ NEAREST-NEIGHBOUR($x_{rand}, \mathcal{T}$)
6:          LPM-CONNECT($x_{near}, x_{rand}$)
7:          if $soln \neq$ nil then
8:              return $\mathcal{T}$
9:          if $n = \mathcal{T}.size$ then
10:             return nil
11:     return nil
```

Figure 6.3: The RRT-LPM algorithm

In order to bound WCET, the revised algorithm introduces a new free parameter k, which controls the maximum number of nodes that can be added to the search tree. This modification is important because it is no longer appropriate for the number of iterations to be fixed based on the maximum search tree size, which can contain at most $[0, \lceil s_{max}/M \rceil)$ configurations, and is independent of k.

The RRT-LPM algorithm stops searching for a solution when any of three conditions are true. Firstly, in cases where no progress is being made by the planner, an iteration counter is used to limit the search. Secondly, if the maximum number of nodes have been added to the tree which can be equated to exhausting the available memory. Finally, the search stops when a solution is found.

Figure 6.4 is a demonstration of the search tree produced by the RRT-LPM planner solving a motion planning problem in a randomly generated environment. As with the results from Chapter 5, the paths in this figure have been generated for the Dubin's car model. These images highlight that the use of the local planning method results in paths with a small number of long steps, whereas the RRT algorithm results in many short path segments, as shown in Figures 5.9, 5.10 and 5.11.

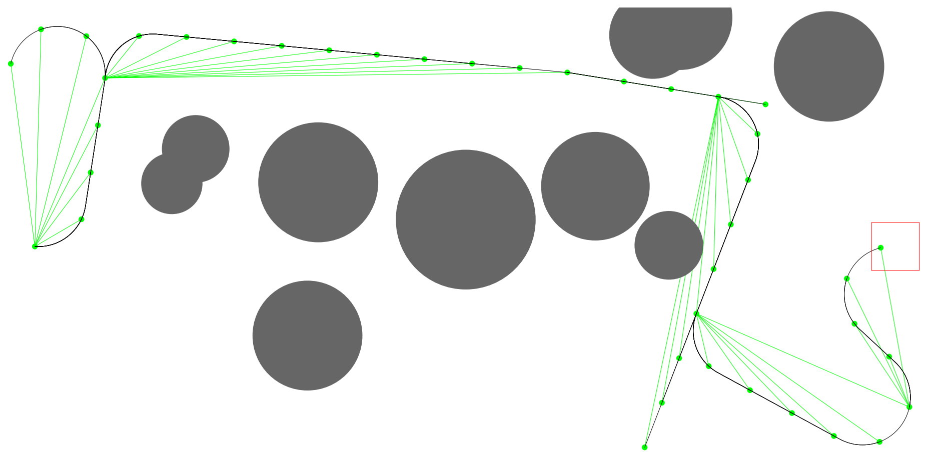

Figure 6.4: Trajectory based RRT in randomly generated environment

6.2 Comparison Metrics

Perhaps the most important aspect of developing an improvement for a motion planning algorithm relates to finding a technique for showing a measurable improvement in performance. In this section, specific performance measures of aspects of motion planning algorithms are considered in the context of their impact on hard real-time systems.

6.2.1 Representative Environments

In order to characterise the performance of a planner (with a given vehicle model) with respect to completeness, it would be necessary to examine the performance of that planner over all possible configuration spaces. This is impractical because the set of possible problems in all configuration spaces is infinite.

By selecting environments that are representative of the algorithmic best and worst cases, it is possible to investigate how the performance of the planner tends to change with regard to the difficulty of a planning problem. To this end, the empty, obstructed and infeasible environments from Chapter 5 are reused, and an additional workspace with narrow passages, shown in Figure 6.5, are used to demonstrate the RRT-LPM planner.

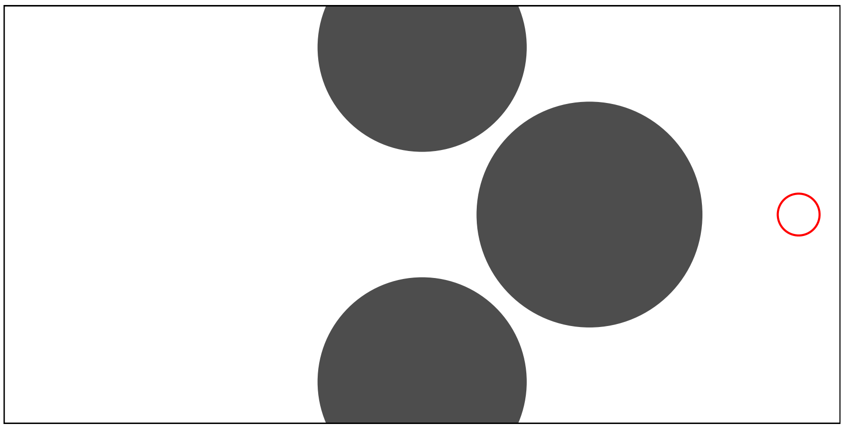
Figure 6.5: The workspace with narrow passages

It is also worth noting that the distance between configurations, the vehicle model and parameters of the planners will affect any evaluation of the performance of the planner, in this work the emphasis is on providing a mechanism that allows improved understanding of how all planners will perform with respect to execution time.

6.2.2 Completeness

Hard real-time motion planning in dynamic environments makes traditional measures of completeness both impractical, due to computational complexity (sec. 2.1.4), and infeasible, due to uncertainty in the evolution of the configuration space (sec. 2.1.2). The work of this thesis has focused on addressing this problem using approximately complete planners. Furthermore, probabilistically complete planners, such as RRT, have execution time profiles that vary smoothly with respect to search quality.

It is worth noting that the probability of finding a solution to a motion planning query using a sampling based algorithm is dependent on both the dimension and geometry of the workspace and on vehicle model, which is characterised by the forward reachable set (sec. 2.1.5). One way to draw insight into probabilistic completeness is by examining the probability of finding a solution relative to the amount of search effort expended for specific problems. The expectation when using stochastic sampling for probabilistic motion planners, the randomness can result in different search trees being produced, even when all values except the random seed are kept constant.

Here the intention is to highlight issues of completeness and convergence for the RRT and the RRT-LPM algorithms. Two specific planning problems were selected in slightly different workspaces. The vehicle model used was the Dubin's car model, and the Dubin's curves were used as an optimal local planning method [Dub57, SL01]. Within each of the environments, the RRT and RRT-LPM planner were executed with varying upper bounds on search tree size. Other parameters including the maximum numbers of samples and the maximum number of connection attempts were kept constant.

The results of these trials are presented as a plot of the probability of failing to correctly identify a path, as a function of search effort, measured in terms of the number of nodes added to the search tree.

The first results were produced for the empty workspace (fig. 5.9), and are shown in Figure 6.6. The variation observed in these curves comes about as a result of the random sampling method used for the planners. Even subject to this variation, the overall trend that can be observed is that the probability of finding a solution increases as search effort increases for both planners.

Figure 6.6 also demonstrates that RRT exhibits both a slower rate of convergence, and appears to require a number of nodes before being able to solve any queries. This can be explained in terms of the approach used for the construction of the search tree. RRT employs a fixed step-size, consequently each vertex in the search graph can only be connected to near-by configurations. Until the graph has propagated through the free space, some configurations are unreachable. In the case of RRT-LPM, the use of the local planning method means that in cases where there are few obstacles, arbitrary pairs of configurations occurs frequently, which leads to fast exploration of free space, and thereby to rapid coverage and connectivity. The results reinforce published results, that predict that connection based strategies converge quickly [KL00].

A second set of similar results generated for the obstructed workspace (fig. 5.11) and are shown in Figure 6.7. Again, as expected, the probability of finding a solution increases for both planners with increased search effort.

By comparing the Figures 6.6 and 6.7, it can be observed that even a small increase in the difficulty of the planning problem affects the rate at which the planners converge to completeness. Furthermore, the effect on the RRT planner is more noticeable than for that of the RRT-LPM. This result indicates that the RRT-LPM is less sensitive to the difficulty of search problems than RRT in some types of environments.

The work demonstrates that RRT-LPM can be used to find solutions to motion planning problems with significantly fewer vertices than RRT. Furthermore, the use of a LPM offers a parameter free technique for robustly tolerating changes to problem scale.

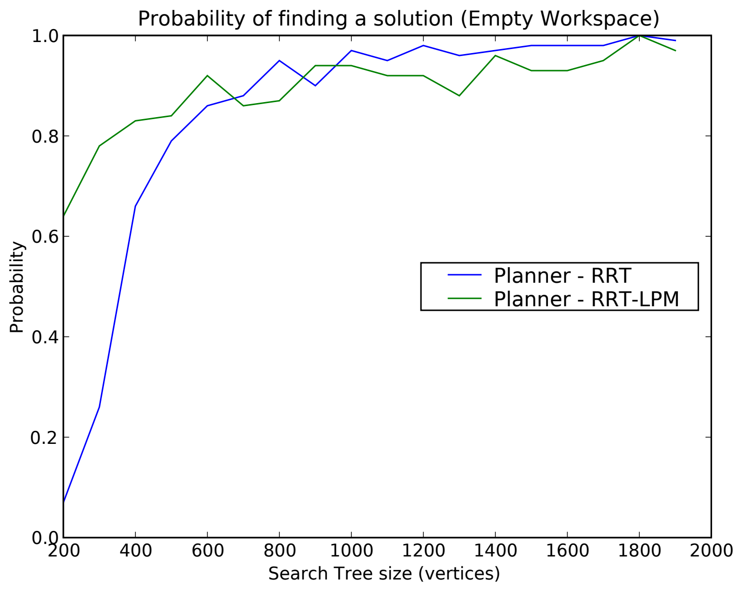

Figure 6.6: Probability of finding a solution as a function of search tree size in the empty workspace

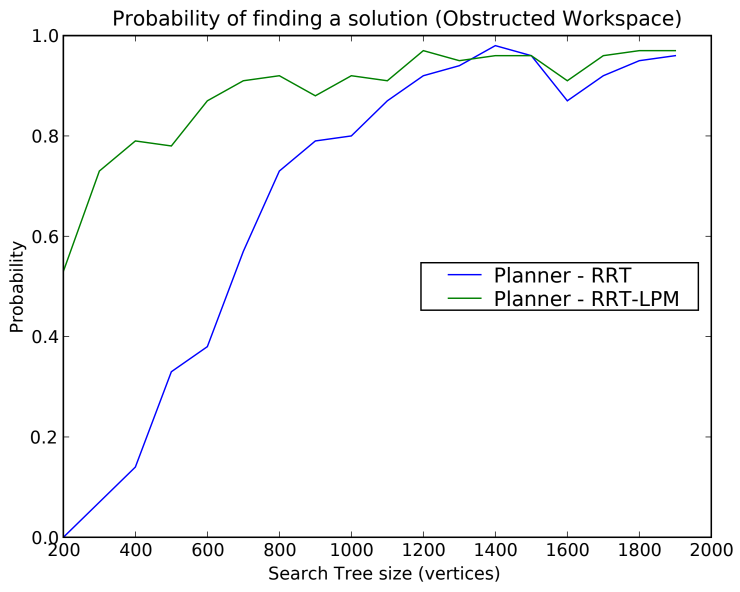

Figure 6.7: Probability of finding a solution as a function of search tree size in the obstructed workspace

6.2.3 Slack Time

The aim when designing a real-time system is typically to ensure that a particular combination of tasks are schedulable. In the case where the execution time, and quality of results, of a task can be modified by a parameters the problem of real-time scheduling becomes one of constrained optimisation. In this context, design of a hard real-time system involves selecting the parameters for a task which maximise quality of solutions, while ensuring that the system remains schedulable.

This second model of maximising search effort while ensuring a real-time system remains schedulable matches the planning algorithms proposed. The planning algorithms considered in this work have performance and worst-case execution times which vary according to a set of parameters.

One limitation of this conceptual model of maximising scheduling performance is that it is based on the assumption that the worst-case execution time is a tight bound. That is, there is little difference between observed execution times, and worst case execution time, or put another way, that there is little slack time (section 3.1.3).

Some slack time in hard real-time systems is considered acceptable. It helps in producing robust systems capable of tolerating minor variations or peak load conditions. Single tasks which consistently generate significant slack time however are indicative of poor usage of available resources.

In the case of motion planning, this can happen in two distinct cases. Firstly, the planner will terminate early in cases where a path to the goal has been successfully identified. The second case occurs when other resources required by the planner, such as memory, have been exhausted. It is to be expected that the execution time of a motion planning algorithm will approach the WCET in cases where no valid solution can be found.

In order to consider slack-time in more detail, the execution time profiles for the RRT-LPM planner are examined. An experimental approach similar to that used in 6.2.2 is adopted. The planner is run in different environments, and the execution times for each trial are recorded. Sample scatter plots of the execution time for varying tree sizes are shown in Figures 6.8 and 6.9.

Figure 6.8 presents the execution time variation that results from a near worst-case situation, where no feasible path exists because the goal is unreachable. This problem is important in that it is representative of a workspace which requires maximum search effort, and time. This figure also shows that the observed execution times, at each of the different search tree sizes are tightly clustered. Significantly, each of those clusters, which are expected to be close to the actual worst case, are less than the predicted worst-case. This occurs because the estimated WCET for the RRT-LPM is overestimated, because it frequently is not possible, or required, to perform s_{max} collision tests. In the presented examples, this occurs because many candidate paths would fall outside the domain of the considered workspace.

For the purposes of comparison, a second execution time scatter plot for the environment with a single large obstacle is presented. The intention of this comparison was to see whether the interaction with obstacles in the environment would result in different performance to the

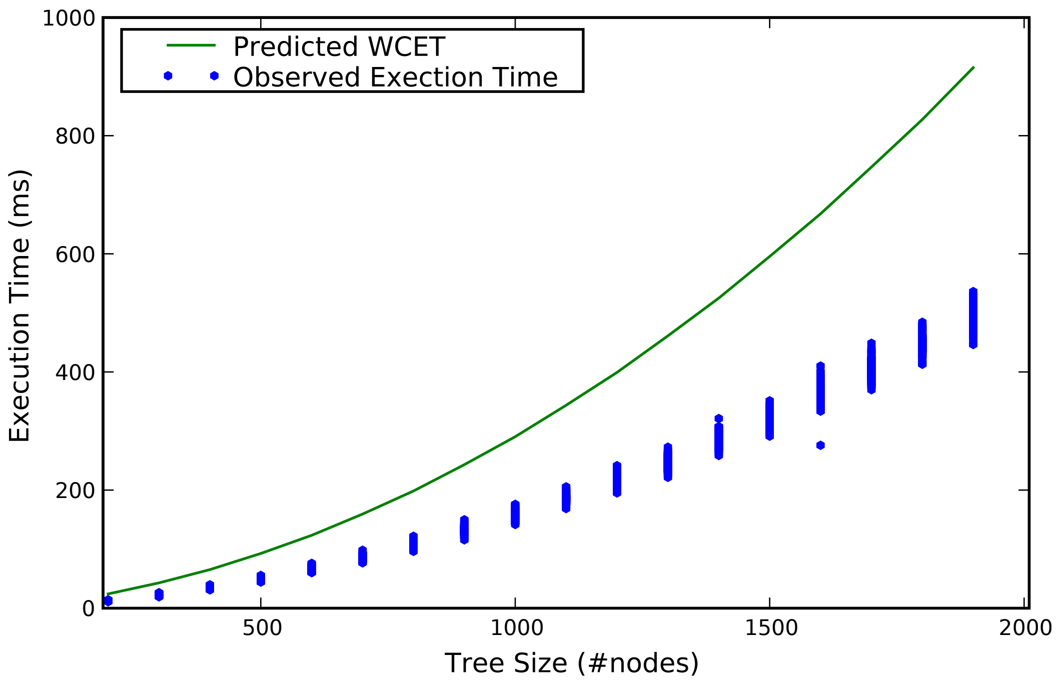

Figure 6.8: Observed execution time scatter plot for the RRT-LPM in a workspace with no feasible solution

previous result. As could perhaps be expected, Figure 6.9 shows a broader range of execution times occur for the environment.

There are two distinct cases for the termination of the RRT-LPM algorithm which are finding a solution or terminating after no solution can be found. Early termination is likely to result in lower execution times, while reaching the limit on the number of nodes in the search tree is likely to be closer to the worst case. These two clusters in execution time are visible, particularly for large maximum tree sizes.

The slack-time in RRT-LPM arises as a result of some combinations of branches being unlikely to occur. For example, the distance between configurations is likely to decrease over time as the Voronoi bias increases, and as such the lengths of branches added to the tree is likely to decrease. This in turn decreases the number of required collision tests in the LPM-CONNECT subroutine. While this effect occurs in practice, it cannot easily be quantified as a constraint for reducing the WCET estimate.

The key point illustrated by both of these plots is that for a given experiment there is an obvious difference between the observed worst-case performance, and the estimated WCET. The estimated WCET is calculated based on the Control Flow Graph and basic block execution times, thus for any given scenario, some degree of slack time will be introduced. This is illustrated in Figure 6.10.

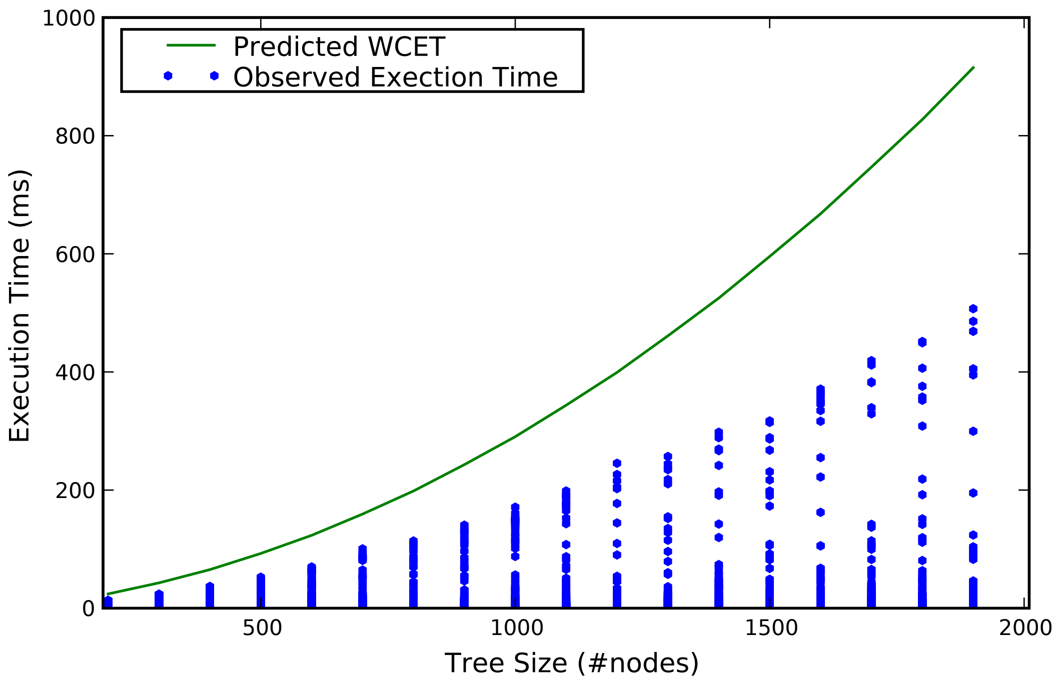

Figure 6.9: Observed execution time scatter plot for the RRT-LPM in a workspace with a single large obstacle

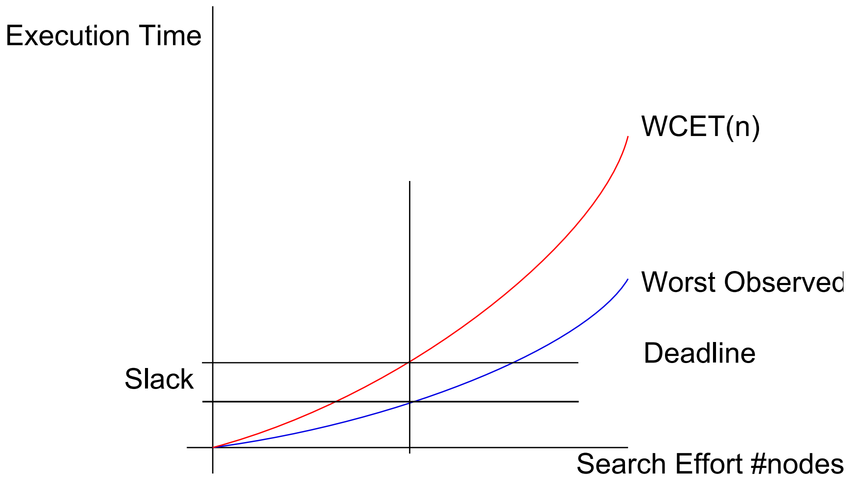

Figure 6.10: Diagram highlighting the slack time introduced because of the divergence between the theoretical and observed worst-case execution times

While the approach for bounding execution times used in chapter 5 is applicable to RRT-LPM, the result of adopting this approach is that the planner can construct smaller search graphs than it would have if it were possible to produce a tighter WCET estimate.

6.3 HRT-RRTLPM

The crux of the proposed modification is that if it were possible to observe execution directly, rather than using an indirect approach based on a single parameter, it would be possible to make use of the slack time for doing additional planning. Figure 6.11 highlights this technique - and demonstrates the improvement in search tree size that can be achieved by using this approach.

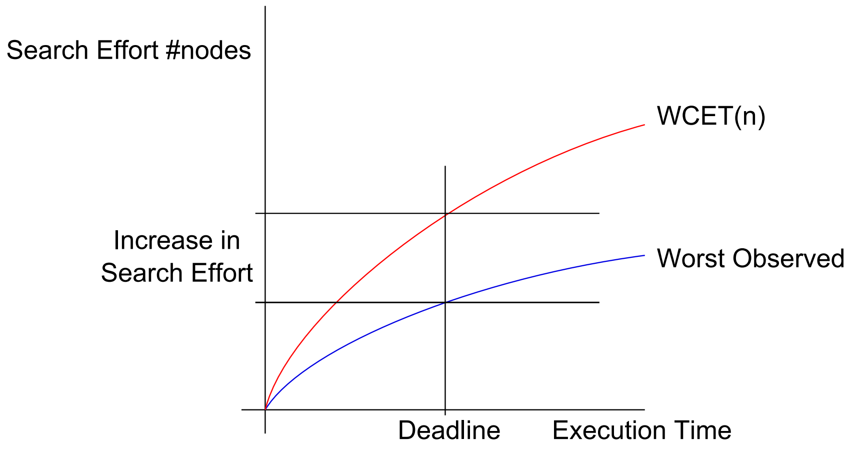

Figure 6.11: Diagram highlighting the potential improvement that can be gained by making use of closed loop estimates of execution times

Methods of achieving termination of an executing task were discussed in Chapter 3, and include using timeouts for soft-real time tasks (Section 3.3.1), and exploiting scheduler slack to perform imprecise calculations (Section 3.3.3).

Traditionally, timeouts for long running tasks involve using wall clock time which is measured by repeatedly polling a clock provided by the operating system to determine whether a deadline has elapsed. The usefulness of such measures is limited both because of the granularity and overhead of using such clocks, and because a task can only be stopped after a deadline occurs. Perhaps more importantly, such measures do not consider the impact of execution time - a factor critical in ensuring that a task will not adversely affect the schedulability of a hard real-time system.

Planning with imprecise calculations would require the division of the planner into multiple tasks. The mandatory section of the planner would then require that a safe plan could be identified, and the optional section would attempt to find a final path to the goal. While this approach allows for much flexibility, it adds significant complexity, requiring language features that implement asynchronous interruption of an executing task.

Here, a simpler mechanism is proposed, which is shown to be a good compromise between hard and soft timing requirements. Rather than relying upon polling a wall clock to implement a timeout, an alternative is to terminate a planner when a prediction of elapsed execution time in each period by the planner exceeds a limit.

6.3.1 Algorithm

The only change required to introduce an execution time limit to any algorithm involves providing a mechanism to measure elapsed execution time. Figure 6.12 shows the result of applying this concept to the RRT-LPM algorithm.

```
1: procedure BUILD-TREE($x_{init}, x_{goal}$)
2:     $\mathcal{T}$. INIT($x_{init}$)
3:     RESET-ELAPSED-EXECUTION-TIME
4:     for $i \in [1, k)$ do
5:         if ELAPSED-EXECUTION-TIME $\geq C_{max}$ then
6:             break
7:         $x_{rand} \leftarrow$ SAMPLE-STATE
8:         $x_{near} \leftarrow$ NEAREST-NEIGHBOUR($x_{rand}, \mathcal{T}$)
9:         LPM-CONNECT($x_{near}, x_{rand}$)
10:        if $soln \neq$ nil then
11:            return $\mathcal{T}$
12:        if $n = \mathcal{T}.size$ then
13:            return nil
14:    return nil
```

Figure 6.12: Pseudo-code of the RRT-LPM algorithm, with timeouts

Having proposed a basic structure, the only issue to be resolved is how to measure elapsed execution time. The strategy proposed in Chapter 5 for predicting WCET included insertion of instrumentation code into the planner to record execution time. Using a calibration process based on those results provides a convenient way to identify the execution times for each edge in the CFG.

A measure of elapsed execution time for a single run of the algorithm can then be expressed as a summation of the execution times for each of the basic blocks on the execution path (eq. 3.7). An alternative formulation of this technique is shown in Equation 6.1.

$$C = \sum_{e \in E} f(e)t(e), \tag{6.1}$$

where $f(t)$ is the number of visits to each of the edges between basic blocks, $t(e)$ is the execution time for each of the edges, and E is the set of edges in the Control Flow Graph. By modifying the instrumentation framework to also record the frequency of visits to each edge in the program, it is also possible to generate values for $f(e)$.

The upper bound on the execution time is C_{max}, a value which can be thought of as a user selected WCET. Comparing this approach to soft real-time planning techniques, C_{max}

plays the same role as the duration of the timeout. In order to satisfy the requirements of a hard real-time system, $C_{max} < D$, failure to meet this requirements means the system would be unschedulable. The updated algorithm continues until the elapsed execution time exceeds C_{max}, at which point it is terminated.

6.3.2 Execution Time

In order to demonstrate the revised performance of the RRT-LPM algorithm subject to the execution time timeout, the experiments from Section 6.2.3 can be rerun to show a concrete example of how the planner performs with respect to time. Plots of the observed execution times in environments, subject to deadlines are shown in Figures 6.13 and 6.14.

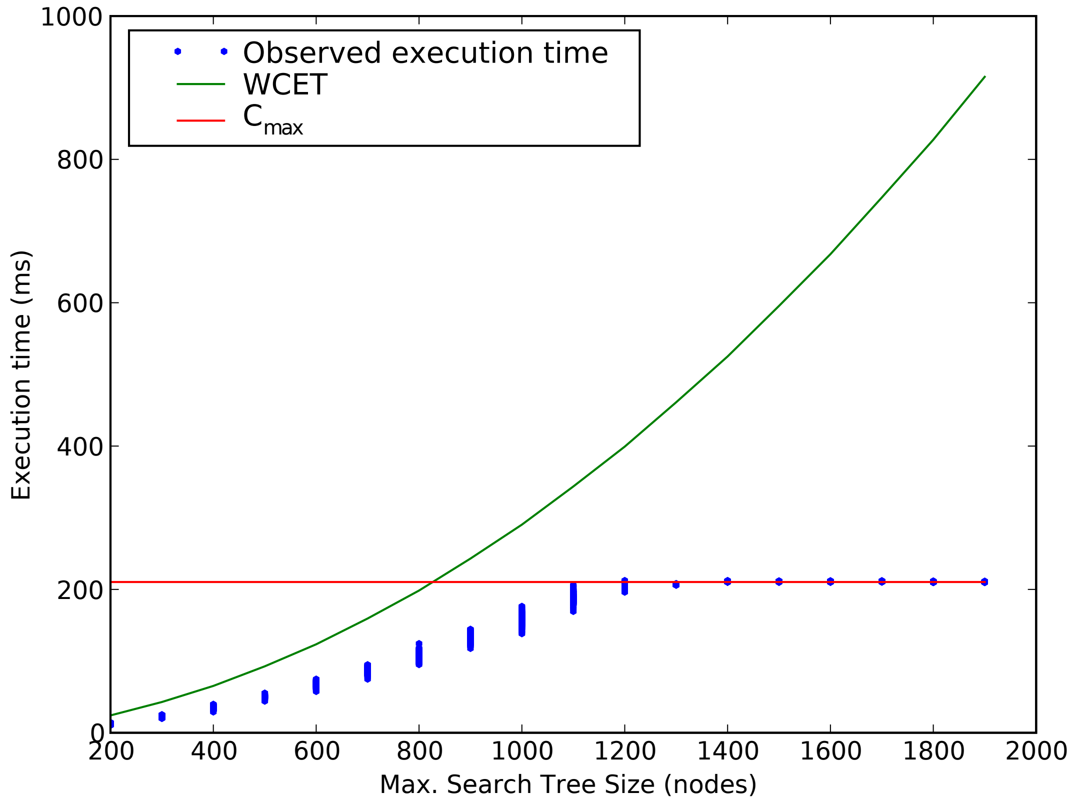

Figure 6.13: Observed execution-times for RRT-LPM with timeouts in the infeasible workspace

These plots show that when required, the execution time of the planner is limited to just more than C_{max}. The overshoot of the specified deadline comes about because the test for a timeout only occurs at a finite number of locations in the algorithm. In the case of the RRT-LPM with timeouts, the magnitude of such deadline overruns depends on the duration of a

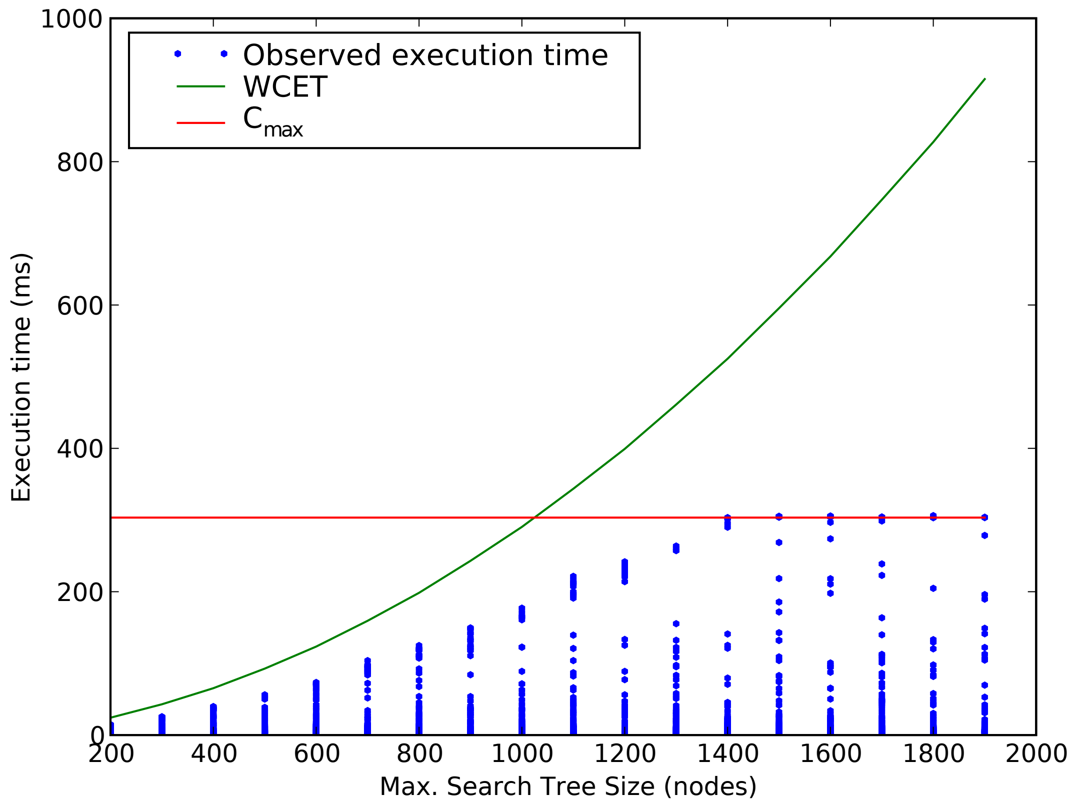

Figure 6.14: Observed execution-times for RRT-LPM with timeouts in the empty workspace

single iteration of the main loop of the BUILD-TREE subroutine (Algorithm 6.12, lines 4 - 13). It is possible to remove the effect of the overshoot by calculating the WCET between timeouts.

In order to prevent a deadline overrun, it is possible to calculate an estimate of the WCET in the main loop, and reduce C_{max} by that amount in order to guarantee that no overruns will occur. The result of such a modification is that a small amount of slack can be reintroduced.

6.3.3 Completeness

As was the case with the comparison between RRT and RRT-LPM, the focus in this section is on the differences between the probabilities of finding solutions. Empirical estimates of the probability of successfully finding a solution have been collected by running the RRT-LPM planner with and without timeouts, at various deadlines, in problems of varying difficulty.

Up until this point, comparative results have been presented in terms of search tree sizes. For this work, it is necessary to present results as a function of deadline. For RRT-LPM this can be done by finding the largest search tree size which will guarantee that a deadline can still be met.

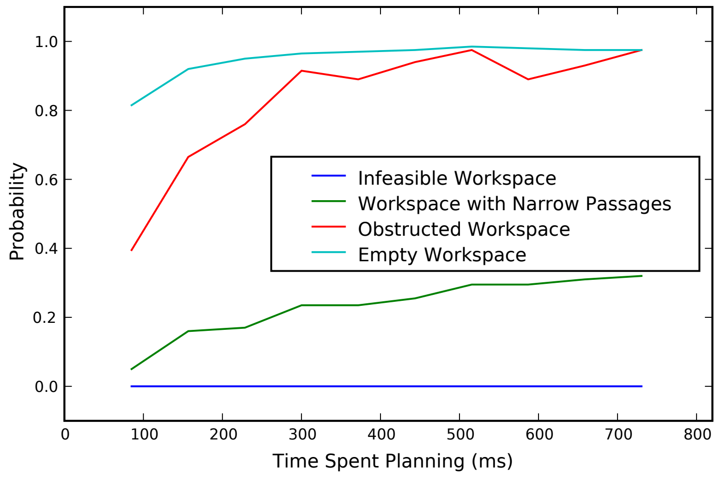

Figure 6.15: The effect of deadline on the probability of finding a solution with RRT-LPM

In order to present a baseline for comparison, Figure 6.16 shows the probability of finding a solution for the planner without timeouts. The probability of finding a solution increases as the deadline increases, except for the infeasible problem. Also, as problems becomes more difficult to solve, the rate of convergence slows noticeably.

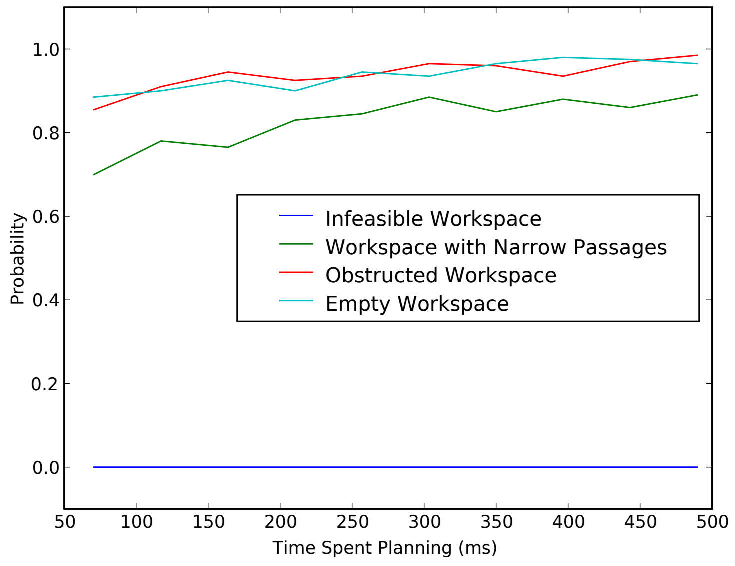

Figure 6.16: The effect of deadline on the probability of finding a solution with RRT-LPM, with timeouts

Figure 6.16 shows the probability of finding a solution using the RRT-LPM with timeouts. The most important feature of this figure is that in cases where the deadline is small, or the problem is very difficult, the planner with timeouts has a higher probability of finding a solution within the deadline.

6.3.4 Concurrent Task Execution

Thus far, the timing results presented for the planners in this thesis have been focused on bounding the execution time of the planner which is required to satisfy a necessary condition for schedulability. In contrast, the sufficient condition for all tasks in a real-time system being schedulable depends on the priorities, periods, deadlines and execution times of all tasks. In terms of the framework of Chapter 4, additional parameters which also introduce temporal effects include the time spent in inter-task communication and synchronization for the planning task. By employing a schedulability analysis as described in Section 3.1.3, it is possibly to prove whether hard real-time system is mathematically schedulable.

In practice, simple models of real-time systems may fail to identify critical factors which influence schedulability. In particular, the real-time systems models used throughout this work make a number of assumptions, including that the timing characteristics of all tasks can be modelled, and that the overhead of the scheduling is zero [BW01]. In order to show that the current real-time system model adequately captures the scheduling process, an additional experiment was run, with two instances of the RRT-LPM planner running as tasks in a hard real-time system.

The execution time traces from the planners can be used to produce a scheduling diagram that shows when each task was executed. Figure 6.17 is a representative scheduling diagram that shows a comparison of the mathematically predicted and experimentally observed execution times for two real-time planning tasks. Tasks one and two had periods (and deadlines) of 250ms and 200ms and maximum allowable execution times of 100ms and 75ms respectively. Using deadline monotonic scheduling, task two has the shorter deadline, and consequently the higher priority.

The simulated results were generated by modelling the tasks in terms of the ideal real-time system. The actual observed execution time results measured for this diagram were combined with additional information about task periods and priorities to infer additional information about which task was running at any given time.

It is worth noting that the minor differences between the observed and simulated results in Figure 6.17 could have been introduced by errors in the operating system's estimate of clock cycles per second, or by unmodeled execution times from the infrastructure of the real-time task. Such errors could be further minimised by using adding more instrumentation into the task bodies or by making use of more accurate instrumentation. This change would be more important to validate more complex scheduling scenarios.

This results shows that in each period there is strong agreement between the simplified

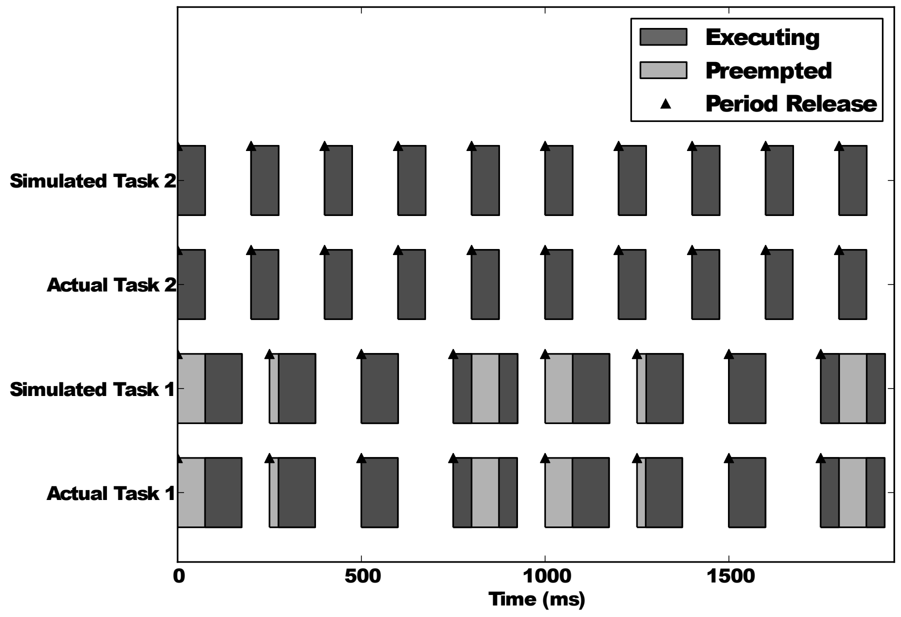

Figure 6.17: Scheduling Diagram for two RRT-LPM planners in parallel

scheduling model and the observed results. In particular, the execution times, and pre-emptions of the tasks are similar. This highlights that the technique of modelling execution time of a planning task developed in Chapter 4 is a suitable method for applying constraints that can assist system designers in verifying the schedulability of real-time systems.

6.4 Summary

Subsequent to the development of the RRT algorithm, a number of enhancements have been proposed that perform better than basic RRT. One of the best examples of such enhancements are planners that make use of local planning methods [KL00, Fra01]. This chapter described the RRT-LPM, a novel variant of RRT that makes use of a local planning method, but remains verifiable in a hard real-time sense. RRT-LPM was then demonstrated to converge to completeness more quickly than RRT. The effect of this improved convergence means that, given a limit on execution time, the planner will be more likely to be able to solve difficult planning problems than RRT.

While the RRT-LPM performs better in terms of completeness than the RRT algorithm, execution time profiling showed that the good average performance came at the price of introducing slack time. In order to make best use of that slack time, a further extension to the RRT-LPM algorithm was proposed. Rather than identifying a set of parameters that would limit execution time, the alternative is to continue planning until a limit on execution time has not been exceeded.

Evaluating the RRT-LPM with the limit on execution time demonstrated that provided sufficient memory is available for planning, it is possible to increase the number of nodes that are added to the search tree, beyond what would have been possible for a static set of parameters.

This chapter has focused on improving the efficiency of a planner by using slack to improve the probability of finding a solution, relative to RRT. This in turn means that the planner will require fewer periods to respond to observed changes in the workspace, which in turn improves the capability of avoiding collisions.

By considering how to best make use of the execution time available for planning opens further avenues for incorporating efficient dynamic planning strategies into hard real-time systems. In particular, heuristically guiding search tree growth and pruning [Fra01, FS06, FS07] in combination with limits on execution time could be used to improve the cost of the traversing paths produced by hard real-time planners.

RRT-LPM is an example of a practical compromise between attempting to maximise quality and quantity of search that can be conducted while still satisfying constraints that allow it to be a component of a hard real-time system. Furthermore, this work has demonstrated that the basic approach to formulating hard real-time planners can be generalised to apply to any algorithm that can be augmented with a timeout. The downside of this approach is related to the inaccuracies and overhead of executing the instrumentation within the finished algorithm - an issue which could be resolved by making use of more accurate techniques for estimating WCET.

CHAPTER 7

Conclusions

The focus of this thesis has been on showing that it is possible to develop efficient hard real-time motion planning algorithms which could be employed as a component in a safe and temporally verified navigation system. Development of this class of planner is a necessary first step in showing that the interactions of all components of a navigation system interoperate correctly.

The capability of an autonomous vehicle to perform safe and efficient navigation in dynamic environments is dependant on being able to perform the tasks of sensing, planning and actuation. Sensing involves incorporating observations into a representation of the state of the vehicle and workspace. Planning involves identifying a collision free path or trajectory from the current state of the vehicle to a goal state. Actuation involves specifying commands to actuators so as to follow the current plan.

The tasks of planning, sensing and actuation are coupled in two major ways. Firstly, they interchange information related to the locations of obstacles in the workspace, the current state of vehicle, and the current plan being followed. If any task fails to produce correct results, the vehicle may end up in an unsafe states, such as a collision, or perform inefficiently, consuming excess energy. Secondly, the tasks are temporally coupled, the time taken for each task to produce results the results of other tasks, both through the time lags which influence data quality, and more importantly the time available to produce results.

7.1 Summary of Findings

This dissertation considered the problem of developing safe and efficient dynamic motion planning algorithms suitable for use on board autonomous vehicles. This task is challenging because of coupling between the planner and other tasks with respect time and information. In order to verify that the system as a whole can satisfy the safety and efficiency requirements, each task must produce results that are correct, and those results must be produced within an appropriate time window. The combination of requirements on both safety and timing can be satisfied by treating the software running on board an autonomous vehicles as a hard real-time system.

Dynamic motion planning problems are rarely formulated in terms of deadlines on the time available for planning. Chapter 4 presented the problem of Hard Real-Time Dynamic Motion Planning (HRT-DMP), a new definition of a planning problem which imposes hard deadlines on the time available for (re-)planning. This definition describes the necessary timing requirements for a planner to participate as a component of hard real-time system. By combining timing information from all of the all tasks in a real-time system, a schedulability analysis is sufficient to show that all tasks will meet their deadlines.

A major contribution of this work was a novel framework that can be used to produce HRT-DMP algorithms. By considering only the necessary timing, planners can be constructed that can later be analysed in terms of a specific hard real-time system. This framework involves using a cyclic hard real-time task to ensure that inputs from sensors, replanning, and outputs to the actuators happen at a sufficiently high rate. By setting an upper bound on the execution time available for each period, a limit on the time available for planning can then be found. A planner that can be shown to have a worst-case execution time less than this limit can be used. The final aspect of the framework was to recognise that this ensures that the timing requirements are satisfied, but that an equally important task is to maximise the quality of the search that could be conducted within a time limit.

The framework for generating HRT-DMP algorithms was demonstrated by proposing the RT-RRT, a new planner that is capable of efficiently searching a workspace, while still satisfying hard time constraints. This process involves finding the worst-case execution time of the planner as a function of search effort. In the case of RT-RRT, the size of the search tree produced is the single most important parameter in determining search quality and execution time. The search for the best possible planner then becomes a constrained optimisation problem, trying to identify the largest search tree size that results in an execution time limit being satisfied.

In order to show that the framework generalises to be useful for solving problems with more efficient planners, the RRT-LPM algorithm was proposed. Unlike RRT and RT-RRT, the search trees produced by RRT-LPM are consistently produced in less time than the WCET, regardless of the complexity of the arrangement of obstacles in the workspace. This unused time means that the planner tends to not make efficient use of the allotted execution time in each period. The process used to identify the WCET of a planner was then modified to provide an online estimate of elapsed execution time. By planning until an execution time limit is exceeded, it was demonstrated that it was possible to make efficient use of the time available for planning.

These contributions highlight that dynamic motion planning without considering deadlines can result in tasks failing to meet deadlines, which can in turn result in errors in localisation, mapping, planning and actuation. Major failures in any of these systems can then result in an unsafe state being entered. This necessitates a more rigorous approach to safety of planning on board vehicles. In particular it this requires improved definitions of dynamic planning under time constraints, systematic approaches to incorporating new and existing algorithms into hard real-time systems and new methods for analysing and optimising such planners.

7.2 Future Work

The formulation of the Hard Real-Time Dynamic Motion Planning problem, and the presentation of a framework for modifying planners that can solve such problems offer fertile ground for further research.

This work has demonstrated safe and efficient navigation based on the RRT algorithm, and the more complex RRT-Connect algorithm. A significant extension of this research would be to apply a variety of algorithms to hard real-time dynamic motion planning. Such an approach would allow more informed decisions to be made about the types of motion planning algorithms that could be used in such situations.

Additional emphasis could also be used to improve the cost of the paths produced by the planner. For systems subject to kinematic, dynamic or non-holonomic constraints where simply finding a feasible path is a difficult problem, it is acceptable to consider any feasible path to be *good enough*. A range of techniques, including Anytime variants of the RRT algorithm [FS06, FS07] and the Manoeuvre-Automaton tree of Frazzoli [Fra01] can be used to reduce cost after an initial path is available. The impact of the additional complexity of these algorithms makes it unclear which is likely to result in the most improvement in path cost.

A second area of further work would be to employ improved strategies for measuring execution times, both for algorithms as a whole, and for the individual edges in the Control Flow Graph. This change would have the effect of reducing risk of the planner overrunning deadlines. Such work could be performed by exploiting hardware models and static analysis, or by employing more sophisticated measurement approaches, such as the object-code rewriting described by Petters in [Pet02].

Another method for improving the performance of hard real-time planners would involve removing, or at least minimising the amount of instrumentation that remains in the code. This has the potential to improve cache performance significantly but has the potential to complicate online estimates of elapsed execution time.

This work has focused exclusively on the environment where the cost of traversing the configuration space is homogeneous. This assumption could be addressed by employing a inhomogeneous model of costs and an appropriately modified tree [FS06, FS07] or grid based planner [KL05], with the concepts presented for hard real-time planning.

The examples presented in this work have been demonstrated in low-dimension configuration spaces, while in practice full dynamics models of flight or underwater vehicles have much higher-dimension configuration spaces [SL03, Fos94]. While the expectation is that the execution time will grow exponentially in dimension, the question of a practical upper bound on the dimensionality of the search space (subject to reasonable deadlines) remains an open-question.

7.3 Final Remarks

This work has demonstrated a clear limitation of current dynamic motion planning strategies, in that they are of limited use when results must be produced subject to time limits. Failure to take such constraints into consideration can result in deadlines being missed, tasks operating with old information, which can increase uncertainty and inefficiencies or lead to more serious failures such as collisions.

The limitations of dynamic motion planning can be overcome by using a suitable formulation for the planning subject to time constraints, such as HRT-DMP. A further challenge involves realising practical algorithms that can solve HRT-DMP problems. This work introduced a framework for solving this problem by employing planners with bounded worst-case execution times.

The other key finding of this work is in developing algorithms that can solve HRT-DMP problems efficiently. This is effectively a constrained optimisation problem, attempting to find the most effective planner in a range of workspaces that can still satisfy a hard time constraint.

By considering the specific problem of incorporating motion planning into a real-time navigation system, this work has also explored the more general concerns of scheduling algorithms with long or unpredictable execution times. This work offers practical solutions to this problem, which is a problem of interest in the broader field of real-time computing.

References

[ABD$^+$96] N.C. Audsley, A. Burns, R.I. Davis, D.J. Scholefield, and A.J. Wellings. Integrating Optional Software Components into Hard Real-time Systems. *Software Engineering Journal*, 11(3):133–140, 1996.

[Ark87] R. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *IEEE International Conference on Robotics and Automation*, volume 4, pages 264–271, 1987.

[ASA$^+$97] B. Allen, R. Stokey, T. Austin, N. A. Forrester, R. A. Goldsborough, M. A. Purcell, and C. A. von Alt. REMUS: a small, low cost AUV; system description, field trials and performance results. In R. Stokey, editor, *OCEANS '97. MTS/IEEE Conference Proceedings*, volume 2, pages 994–1000 vol.2, 1997.

[Aud90] N.C. Audsley. Deadline Monotonic Scheduling. Technical Report YCS 146, Department of Computer Science, University of York, 1990.

[BAN09] R. Button, Acquisition and Technology Policy Center, and National Defense Research Institute (U.S.). *A Survey of Missions for Unmanned Undersea Vehicles*. RAND, 2009.

[BCBH90] J. G. Bellingham, T. R. Consi, R. M. Beaton, and W. Hall. Keeping Layered Control Simple. In *Proceedings of the Symposium on Autonomous Underwater Vehicle Technology*, pages 3–8, 1990.

[BD89] M. S. Boddy and T. Dean. Solving Time-Dependent Planning Problems. In *International Joint Conferences on Artificial Intelligence*, pages 979–984, 1989.

[Bel57] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[BK91] J. Borenstein and Y. Koren. The Vector Field Histogram - Fast Obstacle Avoidance for Mobile Robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288, 1991.

[BK00] R. Bohlin and L. Kavraki. Path Planning Using Lazy PRM. In *Proceedings of the International Conference on Robotics and Automation*, volume 1, pages 521–528, 2000.

[BL91] J. Barraquand and J.-C. Latombe. Robot Motion Planning: A Distributed Representation Approach. *International Journal of Robotics Research*, 10(6):628–649, 1991.

[BL00] A. A. Bennett and J. J. Leonard. A Behavior-based Approach to Adaptive Feature Detection and Following with Autonomous Underwater Vehicles. *IEEE Journal of Oceanic Engineering*, 25(2):213 –226, 2000.

[Bla91] J. H. Blakelock. *Automatic Control of Aircraft and Missiles*. Addison-Wesley, 1991.

[Bod91a] M. S. Boddy. Anytime Problem Solving using Dynamic Programming. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 738–743, 1991.

[Bod91b] M. S. Boddy. Solving Time-Dependent Problems: A Decision-Theoretic Approach to Planning in Dynamic Environments. Technical report, Brown University, 1991.

[Boh99] R. Bohlin. *Motion Planning for Industrial Robots*. Licentiate thesis, Chalmers University of Technology, 1999.

[Bro86] R. A. Brooks. Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[Bro89] Rodney A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. *Neural Comput.*, 1:253–262, June 1989.

[BV02] J. Bruce and M. Veloso. Real-time Randomized Path Planning for Robot Navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2383–2388, 2002.

[BW01] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages. Ada 95, Real-Time Java and Real-Time POSIX*. Addison-Wesley, 2001.

[Car03] M. Carreras. *A Proposal of a Behavior-based Control Architecture with Reinforcement Learning for an Autonomous Underwater Robot*. PhD thesis, University of Girona, 2003.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[CMH$^+$05] H. Choset, Lynch K. M., S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and Thrun S. *Principles of Robot Motion*. MIT Press, 2005.

[Dep04] Department of the Navy. The Navy Unmanned Undersea Vehicle (UUV) Master Plan, 2004.

[Dub57] L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79:497–516, 1957.

[Edg02] S. F. Edgar. *Estimation of Worst-Case Execution Time Using Statistical Analysis*. PhD thesis, York University, 2002.

[FDF00a] E. Frazzoli, M. A. Dahleh, and E. Feron. Real-time motion planning for agile autonomous vehicles. *AIAA Journal of Guidance, Control and Dynamics*, 25:116–129, 2000.

[FDF00b] E. Frazzoli, M.A. Dahleh, and E. Feron. Robust Hybrid Control for Autonomous Vehicle Motion Planning. In *IEEE Conference on Decision and Control*, 2000.

[FKS06] D. Ferguson, N. Kalra, and A. Stentz. Replanning with RRTs. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1243–1248, 2006.

[Fos94] T. Fossen. *Guidance and Control of Ocean Vehicles*. John Wiley and Sons, 1994.

[Fra01] E. Frazzoli. *Robust Hybrid Control of Autonomous Vehicle Motion Planning*. PhD thesis, Massachusetts Institute of Technology, 2001.

[FS06] D. Ferguson and A. Stentz. Anytime RRTs. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5369–5375, 2006.

[FS07] D. Ferguson and A. Stentz. Anytime, Dynamic Planning in High-dimensional Search Spaces. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1310–1315, 2007.

[GAP03] B. Guillem, C. Antoine, and S. Petters. pWCET: a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems. YCS-2003-353, University of York, 2003.

[GB96] M. Greenspan and N. Burtnyk. Obstacle count independent real-time collision avoidance. In N. Burtnyk, editor, *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 2, pages 1073–1080 vol.2, 1996.

[GESL06] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 57–66, Washington, DC, USA, 2006. IEEE Computer Society.

[GO05] R. Geraerts and M. H. Overmars. Reachability Analysis of Sampling Based Planners. In *IEEE International Conference on Robotics and Automation*, pages 406–412, 2005.

[GP01] Herbert Goldstein and Charles P. Poole. *Classical Mechanics*. Addison Wesley, June 2001.

[HA92] Y. K. Hwang and N. Ahuja. Gross motion planning—a survey. *ACM Computing Surveys*, 24(3):219–291, 1992.

[HKLR02] D. Hsu, R. Kindel, J.C. Latombe, and S. Rock. Randomized kinodynamic motion planning with moving obstacles. *Interntational Journal of Robotics Research*, 21(3):233–255, 2002.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Hsu00] D. Hsu. *Randomized Single-Query Motion Planning in Expansive Spaces*. PhD thesis, Stanford University, 2000.

[Int07] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, August 2007.

[Kan08] Kansal, Aman and Zhao, Feng. Fine-grained energy profiling for power-aware application design. *SIGMETRICS Perform. Eval. Rev.*, 36:26–31, August 2008.

[KB91] Y. Koren and J. Borenstein. Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1398–1404, 1991.

[Kha85] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 500–505, 1985.

[KL00] J. J. Kuffner and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. *Proceedings IEEE International Conference on Robotics and Automation*, 2:995–1001, 2000.

[KL02] S. Koenig and M. Likhachev. Improved Fast Replanning for Robot Navigation in Unknown Terrain. Technical report, Georgia Institute of Technology, 2002.

[KL05] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, 21(3):354–363, 2005.

[Koe01] S. Koenig. Agent-centered search. *AI Magazine*, 22(4):109–131, 2001.

[Kor90] R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[KSLO96] L. E. Kavraki, P. Svestka, J. Latombe, and M. H. Overmars. Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. *IEEE Transactions on Robotics and Automation*, 12:566–580, 1996.

[Lat91] J. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[LaV98] S. M. LaValle. Rapidly-Exploring Random Trees: A New Tool for Path Planning. Technical Report 98-11, Computer Science Dept., Iowa State University, October 1998.

[LaV06] S. M. LaValle. *Planning Algorithms*. Cambridge, 2006.

[LB02] S. M. LaValle and M. Branicky. On the Relationship Between Classical Grid Search and Probabilistic Roadmaps. In *Proceedings of the Workshop on the Algorithmic Foundations of Robotics*, 2002.

[LBL04] S. M. LaValle, M. S. Branicky, and S. R. Lindemann. On the relationship between classical grid search and probabilistic roadmaps. *International Journal of Robotics Research*, 23(7/8):673–692, 2004.

[LDW91] J.J. Leonard and H.F. Durrant-Whyte. Simultaneous map building and localization for an autonomous mobile robot. *IEEE/RSJ International Workshop on Intelligent Robots and Systems*, 3:1442–1447, 1991.

[LFG$^+$05] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 2005.

[Lik05] M. Likhachev. *Search-based Planning for Large Dynamic Environments*. PhD thesis, Carnegie Mellon University, 2005.

[LK99] S.M. LaValle and J.J. Kuffner. Randomized kinodynamic planning. In *Proceedings IEEE International Conference on Robotics and Automation*, volume 1, pages 473–479, 1999.

[LK00] S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In *Proceedings Workshop on the Algorithmic Foundations of Robotics*, 2000.

[LK02] M. Likhachev and S. Koenig. Incremental Replanning for Mapping. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 667–672, 2002.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LL03] S. R. Lindemann and S. M. LaValle. Incremental low-discrepancy lattice methods for motion planning. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 2920–2927, 2003.

[LL04] S. R. Lindemann and S. M. LaValle. Current issues in sampling-based motion planning. In P. Dario and R. Chatila, editors, *Proceedings International Symposium on Robotics Research*. Springer-Verlag, Berlin, 2004.

[LM95] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, 1995.

[LMW95] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–307, 1995.

[LP83] T. Lozano-Pérez. Spatial Planning: A Configuration Space Approach. *IEEE Transactions on Computing*, C-32(2):108–120, 1983.

[Mue94] F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Florida State University, 1994.

[MWM03] P. Miotto, J. Wilde, and A. Menozzi. UUV On-board Path Planning in a Dynamic Environment for the Manta Test Vehicle. In *Proceedings OCEANS 2003*, volume 5, pages 2454–2461, 2003.

[Oga01] K. Ogata. *Modern Control Engineering*. Prentice Hall PTR, 2001.

[Ove92] M. H. Overmars. A Random Approach to Motion Planning. Technical Report RUU-CS-92-32, Department of Information and Computing Sciences, Utrecht University, 1992.

[Pea84] J. Pearl. *Heuristics*. Addison-Wesley, Reading, MA, 1984.

[Pet02] S. M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Institute for Real–Time Computer Systems, Technische Universität München, Munich, Germany, September 2002.

[Pet03] S. M. Petters. Comparison of trace generation methods for measurement based wcet analysis. In *3nd International Workshop on Worst Case Execution Time Analysis*, Porto, Portugal, 2003.

[PF05a] S. Petti and T. Fraichard. Partial motion planning framework for reactive planning within dynamic environments. In *proceedings of the IFAC/AAAI International Conference on Informatics in Control, Automation and Robotics*, 2005.

[PF05b] S. Petti and T. Fraichard. Safe Motion Planning in Dynamic Environments. In *proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2210–2215, 2005.

[PKK09] M. Pivtoraiko, R. A. Knepper, and A. Kelly. Differentially Constrained Mobile Robot Motion Planning in State Lattices. *Journal of Field Robotics*, 26(3):308–333, 2009.

[PS91] C. Y. Park and A. C. Shaw. Experiments with a Program Timing Tool Based on Source-level Timing Schema. *Computer*, 24(5):48–57, 1991.

[PS97] P. P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times - A Graph-Based Approach. *The Journal of Real-Time Systems*, 13:67–91, 1997.

[QNX07] QNX Software Systems. QNX System Architecture Documentation, 2007. http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/about.html.

[Rei79] J. H. Reif. Complexity of the mover's problem and generalizations. In *20th Annual Symposium on the Foundations of Computer Science*, pages 421–427, 1979.

[RN03] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[Sch06] T. Schouwenaars. *Safe Trajectory Planning of Autonomous Vehicles*. PhD thesis, MIT, 2006.

[Shi94] Shin, K.G. and Ramanathan, P. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6 –24, jan 1994.

[SK08] Bruno Siciliano and Oussama Khatib, editors. *Springer Handbook of Robotics*. Springer, Berlin, Heidelberg, 2008.

[SL01] A. M. Shkel and V. Lumelsky. Classification of the Dubins set. *Robotics and Autonomous Systems*, 34(4):179–202, 2001.

[SL03] B. L. Stevens and F. L. Lewis. *Aircraft control and simulation*. John Wiley, Hoboken, NJ, 2003.

[SS83a] J. T. Schwartz and M. A. Sharir. On the ‘piano movers’ problem I: A case of a two-dimensional rigid polygonal body moving amidst polygonal barriers. *Commun. Pure Appl. Math.*, 36:345–398, 1983.

[SS83b] J. T. Schwartz and M. A. Sharir. On the ‘piano movers’ problem: II. general techniques for computing topological properties of real algebraic manifolds. *Advances in applied Mathematics*, 4:298–351, 1983.

[SSS$^+$00] S. Singh, R. Simmons, T. Smith, A. Stentz, V. Verma, A. Yahja, and K. Schwehr. Recent progress in local and global traversability for planetary rovers. In *Proceedings of the IEEE International Conference on Robotics and Automation, 2000*. IEEE, April 2000.

[Sta88] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[Ste94] A. Stentz. Optimal and Efficient Path Planning for Partially-known Environments. In *proceedings IEEE International Conference on Robotics and Automation*, pages 3310–3317 vol.4, 1994.

[TBF05] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005.

[Tiw94] Tiwari, V. and Malik, S. and Wolfe, A. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* , 2(4):437 –445, dec. 1994.

[TSK07] K. I. Tsianos, I. A. Sucan, and L. E. Kavraki. Sampling-based robot motion planning: Towards realistic applications. *Computer Science Review*, 1(1):2–11, 2007.

[vdB07] J. van den Berg. *Path Planning in Dynamic Environments*. PhD thesis, Utrecht University, The Netherlands, 2007.

[vdBFJ06] J. van den Berg, D. Ferguson, and Kuffner J. Anytime path planning and re-planning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2366–2371, 2006.

[VN07] F. Valentinis and R. Neil. A UUV System for Very Shallow Water Operation. In *Proceedings Undersea Defence Technology (UDT) Europe*, 2007.

[VWD05] F. Valentinis, J. M. Wharington, and S. Dunn. An Experimentation Framework to Support UAV Design and Development. In *Proceedings Australian International Aerospace Conference*, 2005.

[WEE$^+$08] R. Wilhelm, J. Engblom, A. Ermedahl, et al. The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools. *ACM Transactions Embedded Computer Systems*, 7(3):1–53, 2008.

[WNR$^+$01] S. B. Williams, P. Newman, J. Rosenblatt, G. Dissanayake, and H. Durrant-Whyte. Autonomous Underwater Navigation and Control. *Robotica*, 19(5):481–496, 2001.

[YL02] A. Yershova and S. M. LaValle. Efficient Nearest Neighbor Searching for Motion Planning . In *IEEE International Conference on Robotics and Automation*, 2002.

[YL07] A. Yershova and S. M. LaValle. Improving Motion-Planning Algorithms by Efficient Nearest-Neighbour Searching. *IEEE Transactions on Robotics*, 2007.