SAPIENZA
UNIVERSITÀ DI ROMA

Mobile robots and vehicles motion systems: a unifying framework

Daniele Calisi

a thesis submitted for the degree of
Doctor of Research (Ph.D)
in Computer Engineering

October 2009

Contents

Contents **i**

List of Figures **v**

Notation **ix**

Aknowledgements **xiii**

Introduction **xv**
  Focus and aims . . . xvi
  Main contributions . . . xvii
  Thesis outline . . . xviii

I Current approaches to robot motion 1

1 Definitions and problem formulation 3

1.1 Definitions . . . 3
  1.1.1 The Piano Movers' Problem . . . 3
  1.1.2 Time and trajectories . . . 4
  1.1.3 Velocity constraints and robot models . . . 5
  1.1.4 Dynamic constraints and the phase space $\mathcal{X}$ . . . 9
1.2 Different approaches and problem decompositions . . . 9
  1.2.1 (Re)planning and (reactive) control . . . 9
  1.2.2 Global path-planning and local methods . . . 10
  1.2.3 Decoupled trajectory planning: path and velocity . . . 11
1.3 Sampling-based techniques and probabilistic approaches . . . 12

2 Two different approaches to local robot motion **13**
2.1 Reactive algorithms . . . 13
  2.1.1 Potential Field Methods (PF) . . . 14
  2.1.2 Vector Field Histogram (VFH, VFH+) . . . 14
  2.1.3 Dynamic Window Approach (DWA) . . . 14
  2.1.4 Nearness Diagram (ND) . . . 15

2.1.5 Trajectory generation 15
2.2 Deliberative methods 17
2.2.1 Prerequisites 17
2.2.2 The Expansive Configuration Tree algorithm 18
2.2.3 The Rapid-exploring Random Trees (RRT) family of algorithms 19
2.2.4 Path smoothing and trajectory deformation 26
2.3 Discussion 29

3 World models for high-level path-planning 31
3.1 Integrating global planners and local algorithms 31
3.2 Grid (raster) maps 32
3.3 Line maps 33
3.4 Topological maps 34
3.5 Minimalistic environment models 35

II A unifying framework for robot motion systems 37

4 Tasks and goals 39
4.1 Typical goals for a motion task 39
4.2 A general definition for robot motion goals 40
4.2.1 Reference frames 41
4.2.2 The goal fitness function ϕ and the stopping criterion 42
4.3 Trajectory and task execution issues 42
4.3.1 The trajectory fitness function ψ 42
4.3.2 The trajectory constraints P^τ 42
4.4 Redefinition of the robot motion problem 43
4.4.1 Motion system as a component 44

5 Integrating deliberative and reactive approaches 47
5.1 Data structures: the trajectory tree and the trajectory arc 48
5.2 The Dynamic Trajectory Tree (DTT) algorithm 49
5.2.1 Interleaved planning and execution 52
5.2.2 Feedback control 52
5.2.3 On-line pruning 52
5.2.4 Trajectory deformation 53
5.2.5 DTT in dynamic environments 55
5.3 The Dynamic Behavior Tree (DBT) algorithm: integrating sensor-based behaviors into the planner 56
5.3.1 Behaviors as advices for the `extendNode` function 56
5.3.2 The variable horizon: bridging pure-reactive methods and planners 57
5.3.3 Feedback behaviors as trajectory arcs 58
5.4 Automatic parameter tuning by means of machine learning techniques 60

5.4.1 Reinforcement Learning (RL) and Policy Gradient (PG) methods . . . 60
5.4.2 Applying PG methods to tune DTT and DBT parameters . . . 61

6 A compact topological representation for autonomous navigation 63
6.1 The problem of the local goal . . . 64
6.1.1 Selection of the local goal . . . 64
6.1.2 Inherent problems of the global/local decomposition . . . 65
6.2 A simple roadmap built using virtual frontier-based exploration . . . 67
6.2.1 Building the roadmap . . . 68
6.2.2 Computing the local goal . . . 71
6.3 From the roadmap to a hybrid topological/geometric representation 72
6.3.1 Local map localization . . . 72
6.3.2 Considerations about the global goal . . . 73
6.3.3 Adding labels, local goal hints and other information to the representation . . . 74

7 High-level reasoning and context-based adaptation 75
7.1 Context-based robotics . . . 75
7.2 Contextual design of the motion system . . . 78
7.2.1 System architecture and contextual knowledge . . . 78
7.2.2 Experimental results . . . 80
7.3 Context variables determination and topological map annotations . . . 82

III Evaluation and conclusions 85

8 The MoVeME evaluation framework 87
8.1 Related work in evaluating motion algorithms . . . 87
8.2 The evaluation framework . . . 88
8.2.1 Task metrics . . . 89
8.2.2 Trajectory metrics . . . 90
8.2.3 Physics-based metrics . . . 92
8.3 Comparison with other performance metrics . . . 93
8.4 Benchmark problems . . . 93

9 Experiments 97
9.1 Systems being evaluated . . . 98
9.2 Simulation experiments . . . 100
9.2.1 A typical environment: the “hospital” map . . . 100
9.2.2 A critical environment: the “zig-zag” map . . . 101
9.2.3 A critical task: parallel parking . . . 104
9.2.4 A critical task: constrained motion . . . 106
9.3 Real robot experiments . . . 107

9.3.1 A typical environment: corridor experiment 107
9.3.2 A critical environment: moving obstacles, slalom and parallel parking 109
9.4 Experiments with the whole set of MoVeME benchmarks 112
9.5 Summary 114

Conclusions and future work **115**
Conclusions 115
Future work 116

IV Appendices **119**

A The OpenRDK framework **121**
A.1 The OpenRDK architecture 121
A.2 OpenRDK applications 125

B An application: exploration and search missions **127**
B.1 Introduction 127
B.2 Outline of the method 128
B.3 Exploiting the motion system in exploration tasks 130

Bibliography **133**

List of Figures

1 Some different robots in the class of interest for this thesis. xvi

1.1 The unicycle robot model and the three variables that describe its configuration: (x, y) are the coordinates of the center of the robot, θ is its orientation. 6
1.2 A differential drive robot features two wheels mounted on an axle of length L. The robot is maneuvered by rotating the wheels at different velocities. 7
1.3 A car-like model. The robot moves approximately in the direction that the rear wheels are pointing. 8

2.1 Uses of clothoids curves in robotics and other fields. 16
2.2 The pseudo-code of the basic RRT algorithm 19
2.3 The `extendNode` function of the basic RRT algorithm 20
2.4 The RRT contains a Voronoi bias that causes rapid exploration 20

3.1 Three different representations used to model the same environment . . 32
3.2 Two ways to detect the direction to follow at each node of a topological graph: in the first case, the bearings of the edges are saved in the graph itself; in the second case, the target edge can be detected by counting edges starting from the reference edge 36

4.1 The definition of the local goal set $\mathcal{G}$ for the local motion algorithm. The goal set definition includes also the direction. 40
4.2 The interfaces of the motion system from a component-based viewpoint. 44
4.3 The robot driven by the motion system towards the goal region. 44

5.1 A *trajectory arc* in detail, with time step intervals and robot positions, and the connection among subsequent trajectory arcs in a trajectory tree. 48
5.2 The pseudo-code of the Dynamic Trajectory Tree algorithm 50
5.3 The two kinds of pruning that are performed in the DTT algorithm. The green circle is the current position of the robot in the tree, the red cross is a collision detected in one of the branches. 53

5.4 The behavior included as advices in the `extendNode` procedure of the tree building process. 57
5.5 An execution of the VFH* algorithm 58

6.1 The Local Spanned Area (LSA), the portion of the environment currently sensed by the robot. 64
6.2 The effect of the distance at which the local goal is given to the local subsystem. 65
6.3 Two situations in which the local goal guides the local subsystem towards the goal. 66
6.4 The polygonal representation of the local environment, given by the virtual range finder. 67
6.5 The tree of local polygonal representations, that can be used as a topological representation of the environment. 69
6.6 The algorithm used to compute a roadmap of the environment, using a virtual range finder sensor. 69
6.7 The situation in which a loop is found while virtual exploring the environment. 70
6.8 The procedure the local goal, given the roadmap 71
6.9 The result of the virtual exploration if the polygonal local maps are forced to be convex. 73

7.1 The contextual architecture. 76
7.2 System modules and contextual reasoning for the navigation and mapping experiments. The IF-THEN rules are interpreted following the specified order: rules acting on the same parameters are evaluated in the specified order and the first whose condition is true disables the remaining ones. A default value (the right part of the final "IF true" rules) is also specified in case no other rule is active. 78
7.3 Environment used for the context-driven navigation and mapping experiments. 80
7.4 Results of navigation and mapping experiments. 81

8.1 Three environments taken from the MoVeME benchmark dataset, showing also a robot starting pose (the red circle) and a goal (the green cross). 94

9.1 Performance evaluation in the "hospital" environment of the four systems described in the text, the values are averaged over five experiments in the same conditions, standard deviations are given in parenthesis. . . 101
9.2 The "hospital" environment, with the trails followed by three of the systems described in the text. The first one is System C-DWA: the oscillations described in the text are marked as 1, 2 and 3; the second is System T-DWA, while the last is System T-VFH, in which the trajectory presents more straight lines than the others 102

9.3 The “zig-zag” environment with the trails followed by two of the systems described in the text. The first is System T-DWA, in which the long and narrow corridor after each sharp turning cannot be predicted in time by the algorithm. System THint-DWA, shown below, can benefit from the direction of the local target, that makes it possible for the local algorithm to better adjust the trajectory and enter the narrow corridor with the right orientation, without the need of sharp turns. 103
9.4 Performance evaluation in the “Zig-zag” environment of the three systems described in the text (*Accur* values are not given, because the goal area is large), the values are averaged over ten experiments in the same conditions (algorithms use probabilistic techniques), standard deviations are reported in parentheses. 104
9.5 A trail of the robot performing a parallel parking maneuver, using System H-DBT. 105
9.6 The results of the experiments of parallel parking. 105
9.7 A trail of the robot in one of the simulation experiments using System H-DBT, in which the robot is constrained to move only forward and turn left. 106
9.8 Results of the constrained motion experiments described in Section 9.2.4. 106
9.9 The map of the real environment used for the experiments (on the left) and the robots used (on the right). The robot starts the task from the position marked with “start” and the goal is to reach the grey area, detected using local landmarks. 108
9.10 Performance evaluation in the real experiment of the four systems described in the text (*Accur* values are not given, because the goal area is large), the values are averaged over ten experiments in the same conditions, standard deviations are reported in parentheses. 108
9.11 A robot trail of the execution of System H-DBT, during the “slalom” phase of the experiment described in Section 9.3.2. 109
9.12 The results of the “moving obstacles” phase of the experiment described in Section 9.3.2. 109
9.13 The results of the “slalom” phase of the experiment described in Section 9.3.2. 110
9.14 The results of the “parallel parking” phase of the experiment described in Section 9.3.2. 110
9.15 The overall results of the experiment described in Section 9.3.2. 110
9.16 The parallel parking with the real robot and the System H-DBT. 112
9.17 Results of the experiments with the whole set of the MoVeME benchmarks. 113

A.1 An example of four modules in the OpenRDK framework. 122
A.2 The RConsole GUI, the OpenRDK tool for remote inspection. 124

B.1 The real robot that we use for exploration and search missions and an example of a simulated environment. 128

B.2 An instance of an exploration plan described using the PNP formalism. 129
B.3 A possible situation during the exploration mission, in which the system must choose among a set of frontiers (denoted with the letter "F") and a set of interesting places (denoted with the letter "I"). 129
B.4 Two maps that have been built by our autonomous exploration and search system. 130

Notation

Symbols for spaces

$\mathcal{W}$	the workspace in which the robot operates, it can be 2D or 3D
$\mathcal{C}$	the configuration space, i.e., the space of all possible configurations of the robot; its elements are denoted by q
$\mathcal{C}_{obst}$	the representation of the obstacles and other forbidden configurations in $\mathcal{C}$
$\mathcal{C}_{free}$	the subset of $\mathcal{C}$ that is not colliding with obstacles
$\mathcal{X}$	the phase space, i.e., the configuration space augmented with the derivatives of the configuration (velocities); its elements are denoted by x
$\mathcal{X}_{obst}$ and $\mathcal{X}_{free}$	the representation of the forbidden states in $\mathcal{X}$ and its complement in $\mathcal{X}$
$U, U(q), U(x)$	the action space, i.e., the set of all possible control actions that can be chosen (in a particular configuration q or state x); its elements are denoted by u
$\mathcal{T}$	the set of all possible *trajectories* in a space; depending on which space we are referring to, it can be explicited as $\mathcal{T}^{\mathcal{C}}$ or $\mathcal{T}^{\mathcal{X}}$

Symbols for variables

q, $q(t)$	a configuration of the robot (at time t)
x, $x(t)$	a state of the robot in the phase space, that includes its configuration and its velocity (at time t)
u, $u(t)$	a control input (at time t)
v, $v(t)$	the linear speed of the robot (at time t)
ω, $\omega(t)$	the angular speed of the robot (at time t)

Curves

γ, $\gamma(s)$ or $\tilde{q}(s)$	a path, i.e., a geometrical curve in configuration space $\mathcal{C}$, parametrized by the curve length s; if L is the total curve length, $\gamma : [0, L] \to \mathcal{C}$; alternatively, any other proportional parametrization can be used, e.g., $\gamma : [0, 1] \to \mathcal{C}$
τ, $\tau(t)$	a trajectory, i.e., a curve parametrized by time t, the space in which the curve lies depends on the context (see below)
$\tilde{q}(t)$, $\tilde{x}(t)$ and $\tilde{u}(t)$	a trajectory, in which the space in which it lies is explicit: $\tilde{q}(t)$ is in the configuration space $\mathcal{C}$, $\tilde{x}(t)$ is in the phase space $\mathcal{X}$, $\tilde{u}(t)$ is a trajectory in the control action space U, i.e., if the trajectory duration is denoted by T, $\tilde{u} : [0, T] \to U$

Other notation

$\mathcal{E}$	the description of the environment, usually including the description of the obstacles

Functions

`clearance(q)` or `clearance(x)`	a function that, given the configuration q or the state x, computes the distance to the nearest obstacle in the environment
`collide(q)` or `collide(x)`	this is a utility function that returns $false$ if `collide(q)` returns a distance greater than 0, and returns $true$ otherwise
$\rho(q_1, q_2)$ or $\rho(x_1, x_2)$	the function ρ is a metric for the specified space, i.e., $\rho : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$, in which $\mathcal{S}$ can be the configuration space $\mathcal{C}$ or the state space $\mathcal{X}$
$\sigma(x, \pi, \Delta t)$	a function that computes (or gives an estimate of) the state of the robot if it follows the behavior/policy/feedback law π from the state x for the specified duration Δt

Aknowledgements

It is difficult to give a summary of the many people I would like to thank for their advices, their support, or just the time they spent with me, discussing research topics or just sharing their ideas. I wish to express my gratitude to all those who have helped me to reach this final goal.

In particular, I would like to thank my advisor, Prof. Daniele Nardi, who follows my work from the beginning, granting freedom and trust to my research and giving to me many important opportunities during these years. In addition, I am also grateful to Dr. Luca Iocchi, for his ideas, that often guided my work in unexpected directions. Many other people, that I met in my laboratory and are now partially spread all over the world, deserve my gratitude, unfortunately there is not enough space here to list them all, however I cannot avoid mentioning some: Andrea Censi, Alessandro Farinelli, Giorgio Grisetti, Matteo Leonetti, Luca Marchetti, Stefano Pellegrini, Gabriele Randelli, Gian Diego Tipaldi, Alberto Valero Gomez, Vittorio Amos Ziparo, and many others.

During my doctorate, I had the opportunity to spend a fruitful stay at Tadokoro Laboratory, in Sendai, Japan. Therefore, I want to express my gratitude to Prof. Satoshi Tadokoro and Dr. Kazunori Ohno, for their help and the possibility to use their interesting robots.

Moreover, I am particularly grateful to Prof. Martin Riedmiller and his group of Osnabrueck University, Germany, who dedicated time to teach me their knowledge about their successful machine learning techniques applied to robotics.

I cannot avoid to mention a specific thanks to my parents, because, without them, many of the achievements in my life simply would not have been possible.

And, finally (latin people said "*dulcis in fundo*"), a special thanks go to Francesca, my "little witch", for just being here by my side and sharing her life with mine, as well as supporting me with her advices or simply with her presence.

Introduction

Robots are becoming part of human everyday life and people begin to think that soon we will have robots performing tasks side-by-side with humans and in their environment. Robots are mechanical devices whose physical form can vary and include a manipulator arm, a wheeled vehicle, a legged animal-like form, a flying platform, or a combination of these.

Robots perform many different activities in order to accomplish their tasks. The robot motion capability is one of the most important ones for an autonomous behavior in a typical indoor-outdoor mission (without it other tasks can not be done), since it drastically determines the global success of a robotic mission. As stated in Latombe (1991), robot motion is “eminently necessary since, by definition, a robot accomplishes tasks by moving in the real world”.

The development of a robust autonomous navigation system that can adapt to everyday situations and different non-engineered environments populated by humans, is still an open research area in the field of mobile robotics, although it has been one investigated since the beginning of robotics. Indeed, motion planning and obstacle avoidance have been studied for years, but research is still needed to achieve general and robust techniques that can cope with uncertainties and dynamics of operational environments, that become more and more demanding as far as navigation capabilities are concerned. Specifically, motion planning algorithms need to take into account the uncertainty in both sensor readings and world modeling, as well as in action outcomes. Moreover, human environments are often dynamic, i.e., can differ from the internal world model of the robot, and contain moving obstacles. Obviously, navigation should also take into account constraints to the motion of the robot and be accomplished in real-time.

The proliferation of different approaches for robot motion has many advantages: as a complete and satisfying solution does (still) not exist, many research lines can be explored in order to achieve a better understanding of the problem and its possible solutions, depending on the application at hand. Nonetheless, many methods have been used only to solve specific problems and their strengths and generality sometimes is far from being completely understood. As a matter of fact, a general framework for robot motion systems is still missing, where different methods and their variations can be arranged, composed and compared systematically.

In this thesis, we focus on the main methods for mobile robot and vehicle mo-

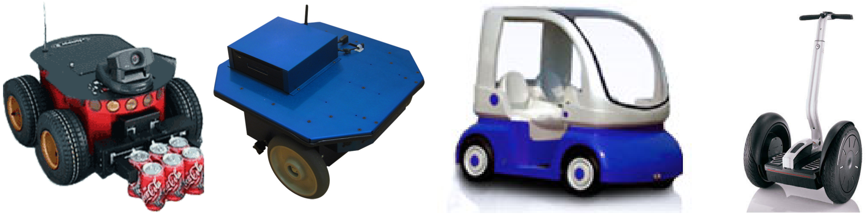


Figure 1: Some different robots in the class of interest for this thesis.

tion systems and we build a common framework, where similar components can be interchanged or even used together in order to increase the whole system performance.

Focus and aims

Robot motion is a wide area of research and it is applied to diverse kinds of robots: from industrial manipulators to autonomous cars, aerial robots, service robots, etc. Moreover, the application domains of robot motion goes beyond robotics and reaches the entertainment industry with virtual actor animation and biology with protein folding analysis. In this thesis we address a subset of these topics, in particular:

- we consider **wheeled mobile robots and vehicles**: wheeled robot bases, autonomous cars, rovers for planetary exploration, wheeled human transporters (see Figure 1) are included in this class;
- we focus on **"pure-motion tasks"**, i.e., those tasks in which the goal is to move the robot to a particular configuration, pose or area in the environment, either being a complete task by itself (e.g., when an autonomous car is asked to reach a location), or as a building block for a more complex plan (e.g., when a service robot has to go in the kitchen in order to prepare the dinner): we do not explicitly consider other motion-related tasks such as coverage or feature search, although they often require to accomplish pure-motion sub-tasks;
- we aim at designing robot motion systems that are able to **perform on-line on real robots**, as well as to react to unexpected events and moving obstacles in non-engineered environments;
- we search for algorithms and methods that are able to account, implicitly or explicitly, for **kinematic and dynamic constraints** to the robot motion: systems that do not cope with such constraints are doomed to rely on unacceptable approximations when used on real vehicles;
- we analyze algorithms and methods that can cope, implicitly or explicitly, with **inaccurate models of robots and environments** and **uncertainty in sens-**

ing and actuation: it is not realistic to assume precise models and prediction of the physical behavior of the robot, when dealing with real-world scenarios.

Nonetheless, a few topics tackled in this thesis relax some of the above restrictions. In particular, the assumptions about the kind of robots being used (e.g., the global environment representations introduced in Chapter 3 and 6). Moreover, we often describe related work that does not exactly match the above assumptions, but that can be easily extended or integrated to those settings.

This thesis aims at providing a unifying framework for robot motion systems, that deal with the robots, the scenarios and the above-described issues. The main features of this framework are the possibility to include the main algorithms and methods for obstacle avoidance and motion planning that have been developed in the past, in such a way that they can be easily interchanged or used in conjunction, to increase the effectiveness of the motion system. Currently, the general robot motion problem does not present a single solution: depending on the situation, the context and the mission-specific goals, different methods should be used. The framework should allow the motion system to adapt to the situation, and possibly allow for a smooth transitioning among different planning/reactive methods. Finally, the framework should account for high-level and large-scale representations.

Main contributions

The main contributions of this thesis are given in the following.

- We **define a general framework** for robot motion systems that adopts the well-known decoupled approach in which a low-level motion subsystem is guided by a high-level global planner. The generality of this approach allows to include in the framework almost all reactive local methods and obstacle avoidance techniques, as well as the path-planners developed so far as global subsystems.

- We give a **characterization of the low-level subsystem** that allows for the **seamless integration of pure-reactive behaviors and local planners**, taking advantage of both techniques. While the former are more suitable for dynamic environments and high speed motions, the latter are usually more accurate and can cope with a larger set of situations and tasks: we think that future motion systems should autonomously trade speed for safety or accuracy, depending upon the context and the specific task to accomplish, adapting or choosing among different methodologies. Part of the material on this topic has been published as author's original work: the tree-based planner that is the basis of the framework has been proposed in Calisi et al (2005a) and in Calisi et al (2005b), different reactive techniques have been presented in Calisi et al (2007d) and in Takeuchi et al (2008) and a study of explicit uncertainty management in motion planners has been published as Censi et al (2008).

- We address the **requirements for the high-level representation** and the **relationship with the low-level subsystem**, and we compare different environment models and the effect of their use with respect to the performance of the motion system. Moreover, we propose a topological representation of the environment, where high-level information about the context and some heuristics for the local subsystem can be included. A previous work about the use of context-based architectures for motion tasks can be found in Calisi et al (2007a), in Calisi et al (2008e) and in Calisi et al (2008d). The comparison of the performance with respect to the world models has been presented in Calisi and Nardi (2009), together with the evaluation framework described below.

- We introduce a **characterization of the tasks and the goals** for a robotic motion system, that takes into account the possibility of "fuzzy" or symbolic global tasks (e.g., "go in the kitchen") and gives some degree of freedom to the sub-goals, that can increase the performance of the low-level subsystem.

- We present an **evaluation framework** for robot motion systems, called MoVeME, that features a set of objective and quantitative performance measures and a set of benchmark problems (presenting both typical tasks and situations, as well as critical problems) in which the measures can be taken. The MoVeME framework has been presented and described as author's original work in Calisi et al (2008c) (and refined in the above-mentioned Calisi and Nardi (2009)).

Other contributions related to the development of this thesis are presented in the Appendices. In particular, the robot motion framework has been developed and tested both in simulated environments and on real robotic platforms using the OpenRDK software framework for robotics[1], that has been developed primarily by the author and currently actively used in the RoCoCo Laboratory of "Sapienza" University of Rome and in the Intelligent Control Group of University of Madrid. Details on the OpenRDK can be found in Calisi et al (2008b) and in Calisi et al (2008a).

Finally, we present a more complex robot mission, in which the motion system is embedded as a component: the exploration and search task for a rescue mission. Exploration and search tasks have been deeply analyzed by the author and the results are presented in Calisi et al (2007b), and in Calisi et al (2007c).

Thesis outline

For better clarity, this thesis is divided into three parts: **Part I** introduces the problem and describe the current algorithms and methods for robot motion. In **Part II**, the main components of a novel unifying framework for robot motion system are introduced. **Part III** is about experiments on real and simulated robots. Finally, the **Appendices** present other work of the author that are related to the development of the thesis, but are not strictly related with pure robot motion issues.

---

[1] `http://openrdk.sf.net`

Part I

Chapter 1 introduces the main definitions and the motion planning problem in its classic formalization, the *Piano Movers' Problem*. Typical approaches and problem decompositions are presented, one of these address the problem separately: (i) from a global level, i.e., considering global information about the environment, and (ii) from a local level, where low-level issues about robot movements can be addressed: this decomposition is followed also in the thesis. Subsequently, the formal framework is extended to deal with velocity and acceleration constraints that a motion system should address, in order to be effective in real scenarios. A description of sampling-based techniques and probabilistic approaches, that will be used extensively through the thesis, concludes the chapter.

Chapter 2 presents the most common algorithms that are used to solve the robot motion problem at the local level. It considers both reactive obstacle avoidance techniques and deliberative methods that are able to plan a set of control actions up to a given horizon in time or until the goal is reached. Basic methods and their common extension are described.

Chapter 3 describes the most commonly used world models for global-level path-planners. It describes grid maps, line maps, topological maps and more minimalistic representations. Advantages and drawbacks are discussed for each model and the problem of the relation with the local motion subsystem is tackled, giving different methods for each model.

Part II

Chapter 4, after presenting the tasks of a typical robot motion system, gives a novel formulation of the goal of a motion task. It introduces two fitness functions for evaluating and guiding the behavior of a motion system: one is concerned with the behavior *during the execution of the task* and is called *trajectory fitness function* ψ, while the other, called *goal fitness function* ϕ, evaluates the achievement of a given goal. Finally, a stopping criterion is defined over the function ϕ and a set of trajectory constraints are specified, that determines what a robot is not allowed to do during the execution of the task (e.g., colliding with obstacles). With these new definitions and the extensions introduced in the previous chapter, the problem of interest for this work is redefined at the end of the chapter.

Chapter 5 describes a novel approach that allows for the use of obstacle avoidance techniques in conjunction with motion planners, in a fully integrated perspective. The boundaries between the two classes are thus fuzzified and more than one method can be used in the same task, in order to increase the effectiveness of the system. Moreover, a technique for interleaving planning and execution and dynamic partial re-planning is presented. Uncertainty in both sensor readings and action results is addressed. In particular, we show how to include relevant information about uncertainty into the framework. This allows the planner to work with uncertainty information embedded in an extended space, that yield a more robust and reliable motion system. Finally, some machine learning techniques are introduced, that can be used to optimize the behavior of the different components of a motion system. In particular, reinforcement learning methods, such as policy gradient algorithms, are used to tune the parameters and optimize the trajectory generation process.

Chapter 6 introduces a topological representation of the environment that allows for the inclusion of symbolic labels and hints for the local motion subsystem. The representation does not rely on any distance-related information, although this can be included in some situations, in order to increase the model reliability and effectiveness.

Chapter 7 shows the benefits of introducing high-level reasoning to adapt the motion system to different situations. In particular, the context-based architecture is exploited, that allows for a clear decoupling between the functional part of the motion system and a set of components that are responsible to determine the current situation (context) and modify the behavior of the functional part of the system accordingly.

Part III

Chapter 8 introduces the evaluation framework MoVeME, for mobile robot and autonomous vehicle motion systems. The framework features a set of metrics to be measured during experiments and a collection of problems of interest (benchmarks). The metrics reflect the important aspects that a robotic motion system should address, including the behavior during the execution, the safety and the accuracy of reaching the goal. The benchmarks include both *typical* situations and tasks, as well as *critical* problems, that can be unusual in practice, but play an important role in the evaluation of the given motion system.

Chapter 9 presents an evaluation of a set of motion systems in different situations, through a set of experiments. The evaluated motion systems include well-known algorithms, that are described in Chapter 2 and Chapter 3, as well as the novel algorithms introduced in Chapter 5 and in Chapter 6. The experiments have been designed following the definition of the goals given in Chapter 4, make use of the

MoVeME benchmarks and are evaluated using the performance metrics introduced in Chapter 8. Both simulated and real-robot experiments are performed.

Part IV (Appendices)

Appendix A provides an overview of the OpenRDK software framework for robotics, that has been designed and developed by the author and that is the basis for the actual software implementation of the modules described in this thesis.

Appendix B shows an example application for robot motion systems: the exploration and search task. In particular, we adopt a frontier based approach for navigation goal detection and a high-level graph-based language to formalize the high-level exploration plan.

Part I

Current approaches to robot motion

1

Definitions and problem formulation

In this chapter, we give the common definitions and concepts, that are used to address motion planning and control problems. These definitions will be used in the rest of the thesis and they will be extended in Chapter 4. Moreover, we describe the different approaches that are currently exploited to build complete robot motion systems as well as the common decompositions of the problem, that are often needed to obtain suitable solutions in appropriate time.

1.1 Definitions

In the following, we introduce some definitions that will be used in this thesis. In particular, we are defining the problem that a robot motion system is required to solve. Most of the definitions given in this section can be found in Latombe (1991), in LaValle (2006), in Choset et al (2005) or in any basic robotics book.

1.1.1 The Piano Movers' Problem

A **robot** $\mathcal{R}$ is embedded in a 2D or 3D **workspace**, denoted with $\mathcal{W}$. The robot is geometrically described as the set of points in $\mathbb{R}^2$ or $\mathbb{R}^3$ (in $\mathcal{W}$ that are *occupied* by the robot. We consider robots that are either single rigid bodies (i.e., all pairs of points of the robot maintain their mutual distance and the orientation of the whole robot cannot change) or a collection of rigid bodies, with precise constraints on how they can move with respect to each other.

The workspace contains **obstacles**, i.e., rigid bodies that must be avoided by the robot while it moves. The set of all points occupied by the obstacles is denoted by $\mathcal{W}_{obst}$. In addition, the free workspace is denoted by $\mathcal{W}_{free} = \mathcal{W} \setminus \mathcal{W}_{obst}$.

A **configuration** q of a robot is a parametric description of its pose (and relative poses of its parts, in the case of a collection of rigid bodies) in the workspace. The set of all possible configurations for a robot is the **configuration space**, that we denote with $\mathcal{C}$. We also define the set $\mathcal{C}_{obst}$ as the set of all configurations of the robot that collide with the obstacles (i.e., in which at least a point in $\mathcal{W}$ is occupied by both the robot and an obstacle). Furthermore, the set $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obst}$ is the set of all allowed configuration for the robot $\mathcal{R}$.

The basic motion planning problem, also known as the "Piano Movers' Problem", can be defined as follows: given an initial configuration q_I and a goal configuration q_G, the objective is to compute a continuous path, $\gamma : [0, 1] \rightarrow \mathcal{C}_{free}$ such that $\gamma(0) = q_I$ and $\gamma(1) = q_G$. The curve γ can be parametrized in the interval $[0, 1]$, but we will often use the curve length as the parameter: if s_F is the curve length, $\gamma : [0, s_F] \rightarrow \mathcal{C}_{free}$). It has been shown that, even in the case in which the description of $\mathcal{C}_{free}$ does not change, the problem is PSPACE-hard with respect to the degrees of freedom of the configuration space (Reif, 1979).

Informally, we can say that a **motion system** is the robot component that is responsible to move the robot from the initial configuration to the goal configuration. We assume that the motion system can interact with the robot configuration (or, more generally, with its state) using a set of **control actions**. The set of all control actions that the robot can take is denoted by U. For each state $q \in \mathcal{C}$, an action space $U(q) \subseteq U$ is defined as all the possible control actions that a robot can take in q. The law that correlate each control action with the corresponding change in the robot state depends on the particular robot and world dynamics and can contain some degree of uncertainty.

1.1.2 Time and trajectories

Both the robot and the obstacles are allowed to move in the environment: for this reason, we introduce the concept of **time** in our notation. This means that all above definitions can be time-dependent. For example, the robot configuration is actually a function of time $q(t)$, and also the obstacle description in both the workspace and the configuration space can be time-dependent and becomes $\mathcal{W}_{obst}(t)$ and $\mathcal{C}_{obst}(t)$. In order to simplify the notation, we will write explicitly this time dependence only when it is required.

Moreover, we notice that a path, as defined above, only describes the geometrical properties of the motion of the robot, and does not codify anything about its evolution with respect to the time. The time is thus another important parametrization of the curve followed by the robot in the configuration space. In this case, we talk about the **trajectory** of the robot, i.e., a path is the curve parametrized by the curve length (or any other proportional parametrization), while the trajectory is the curve parametrized by time, and we denote it by $\tau(t)$.

We denote with T the time interval over which the trajectory $\tau(t)$ is defined, i.e., $\tau : [0, T] \rightarrow \mathcal{C}$. Since we want to deal with systems that can be implemented on real computers, we often consider that the motion system is allowed to take decisions only at a discretized sequence of time steps $T_d = \{t_0, t_1, \ldots, t_F\}$.

The correlation between control actions and the transformation induced on the robot configuration, can thus be explicited by a differential law (uncertainty is not taken into account here):

$$\dot{q} = f(q, u), \tag{1.1}$$

in which $\dot{q}$ is the derivative of the configuration with respect to time (i.e., $\dot{q} = \frac{dq}{dt}$). Given the environment and the goal configuration, a motion system will produce a continuous sequence of control actions $u(t)$, such that the robot, following a trajectory $\tau(t)$ in the allowed configuration space $\mathcal{C}_{free}$, reaches the specified goal.

1.1.3 Velocity constraints and robot models

Differential constraints on the motion arise often when dealing with mobile robots and vehicles, and motion system should exploit them to increase accuracy and precision. Given the configuration space $\mathcal{C}$, **velocity constraints**, also called kinematic constraints, can be expressed as the set of the allowed velocities at each point in $\mathcal{C}$. This results in first-order differential equations:

$$g(\dot{q}, q) \bowtie 0, \tag{1.2}$$

in which $\bowtie$ can be $=$, $<$, $>$, $\leq$ or $\geq$.

Velocity constraints restrict the set of possible velocities at each configuration q, for example, they can impose maximum and minimum velocities or constrain the direction of the robot motion (e.g., a wheeled robot can move only in the direction of the wheels).

When a differential constraint can be integrated to a lower order constraint (e.g., a velocity constraint can be integrated to a configuration constraint), it is called a **holonomic constraint**. Such an integrable constraint reduces the dimension of the actual configuration space of the robot. For example, the planar motion of a point on a circle centered at the origin can be represented using the coordinates (x, y) and the velocity vector $(\dot{x}, \dot{y})$ (i.e., the direction of the motion) can be constrained on the tangent of the circle, i.e., $x\dot{x} + y\dot{y} = 0$. This velocity constraint can be integrated to the configuration constraint $x^2 + y^2 = R$, where R is defined by the initial position of the point. In this case, the constraint on the velocities can be integrated to the constraint on the configurations; in this case, an unconstrained description of the configuration is possible using a single angle coordinate θ.

In general, it is not easy to determine if a differential constraint can be integrated to yield an equivalent lower-order constraint. However, constraints that cannot be fully integrated are called **non-holonomic constraints**. Such a kind of constraint does not necessarily reduce the dimension of the space of configurations attainable by the robot, but reduces the dimension of the space of possible differential motions

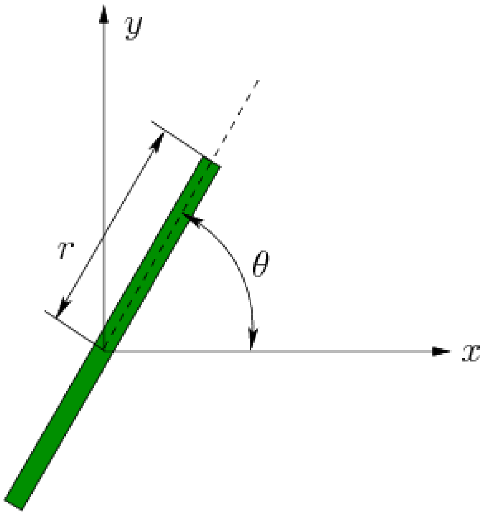


Figure 1.1: The unicycle robot model and the three variables that describe its configuration: (x, y) are the coordinates of the center of the robot, θ is its orientation.

(e.g., the space of the velocity directions) at any given configuration. For example, the motion of a car-like robot is constrained in the direction of the wheels (it cannot move sideways), but in empty space, through maneuvering, the car can reach every configuration.

In the following, we describe some of the robot models that are considered in this thesis, giving details about the constraints involved.

1.1.3.1 A simple unicycle

The unicycle is a wheel that rolls upright on a horizontal plane. Figure 1.1 shows a projection on the plane of this kind of robot. The configuration q of the robot has three dimensions, that describe the position (x, y) of the contact point of the robot and the ground (i.e., the projection of the center of the wheel on the plane) and the orientation θ. This robot can be controlled by the rolling speed v of the wheel (also called linear velocity) and the rate of change ω of the steering angle (also called angular velocity). Sideways translation of the wheel is prevented by the no-slip non-holonomic constraint imposed by the wheel:

$$\dot{x}\sin(\theta) - \dot{y}\cos(\theta) = 0. \tag{1.3}$$

The configuration transition equation is:

$$\begin{array}{rcl} \dot{x} & = & v\cos(\theta) \\ \dot{y} & = & v\sin(\theta) \\ \dot{\theta} & = & \omega. \end{array} \tag{1.4}$$

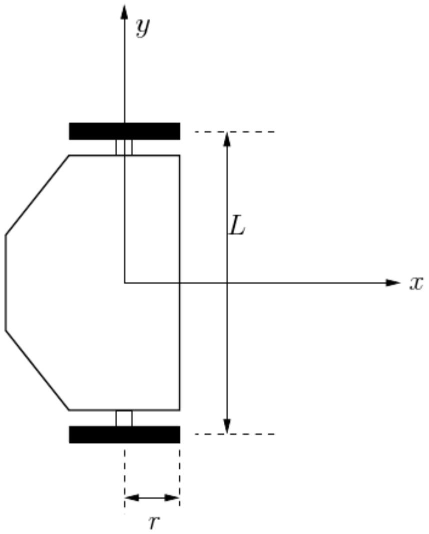


Figure 1.2: A differential drive robot features two wheels mounted on an axle of length L. The robot is maneuvered by rotating the wheels at different velocities.

The unicycle can be seen as the simplest robot model that involves wheels and the non-holonomic constraints that prevent sideways motion. The following models are very similar to the unicycle.

1.1.3.2 Differential drive robots

Most indoor robots can be modeled as differential drive. The configuration of the robot can be described using the same variables as the unicycle: $q = (x, y, \theta)$ (see Figure 1.2). There are two main wheels, that can rotate at different speeds: these are actually the two variables of the action commands, u_l and u_r. If $u_r = u_l$, the robot moves straight (forwards or backwards, depending on the sign of the speed), while if $u_r = -u_l$, the robot turns in place. The configuration transition equation is:

$$\begin{array}{rcl} \dot{x} & = & \frac{r}{2}(u_l + u_r)\cos(\theta) \\ \dot{y} & = & \frac{r}{2}(u_l + u_r)\sin(\theta) \\ \dot{\theta} & = & \frac{r}{L}(u_r - u_l). \end{array} \tag{1.5}$$

By setting $\frac{r}{2}(u_l + u_r) = v$ and $\frac{r}{L}(u_r - u_l) = \omega$, this model can be converted into the unicycle model.

1.1.3.3 Car-like vehicles

We model a car as a rectangular object moving on a plane. As usual, the robot configuration has three dimensions $q = (x, y, \theta)$, where x and y are the coordinates of the center of the car and θ is its orientation. The model is shown in Figure 1.3.

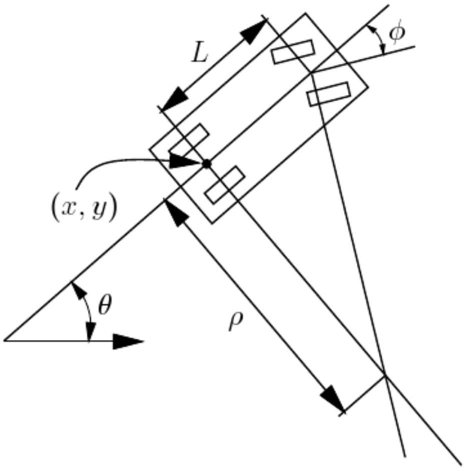


Figure 1.3: A car-like model. The robot moves approximately in the direction that the rear wheels are pointing.

Due to the usual non-holonomic constraint induced by the wheels, the robot moves approximately in the direction that the rear wheels are pointing. This means that the state transition equations for $\dot{x}$ and $\dot{y}$ are the same as in the unicycle model; this also implies that one of the control variables is the linear speed v. The second control that is possible in this kind of model is the steering angle ϕ. The state transition equation for $\dot{\theta}$ is obtained through some geometric calculus and is shown in the following, together with the other differential equations:

$$\begin{array}{rcl} \dot{x} & = & v\cos(\theta) \\ \dot{y} & = & v\sin(\theta) \\ \dot{\theta} & = & \frac{v}{L}\tan(\phi). \end{array} \tag{1.6}$$

Depending on other constraints on control actions, we can define variations of this model:

- **Tricycle**: the linear speed v is constrained to some interval $[-v_1, v_2]$ in which $v_1 > 0$ and $v_2 > 0$, while the control input ϕ is constrained to $[-\pi/2, \pi/2]$. The vehicle can turn in place if $\phi = \pm\pi/2$.

- **Simple Car** (J. P. Laumond and Lamiraux, 1998): $v \in [-v_1, v_2]$ as in the previous model, but ϕ is constrained to some interval $[-\phi_{max}, \phi_{max}]$ in which $|\phi| \leq \phi_{max} < \pi/2$. The vehicle has a minimum turning radius of $\rho_{min} = L/\tan(\phi_{max})$.

- **Reeds-Shepp Car** (Reeds and Shepp, 1990): the linear speed v is restricted to the discrete set $\{-1, 0, 1\}$.

- **Dubins Car** (Dubins, 1957): this model is obtained from the Reeds-Shepp model by removing the reverse speed $v = -1$ from the set of possible speeds.

Reeds-Shepp and Dubins Cars have been often used to demonstrate some theoretical result about shortest paths between two configurations in the absence of obstacles. Moreover, controllability theory states that these models can reach every configuration in $\mathcal{C}$, in absence of obstacles.

1.1.4 Dynamic constraints and the phase space $\mathcal{X}$

Dynamic constraints, i.e., those that allow to account of momentum and other aspects of dynamics, require higher order differential constraints. A simple example of a dynamic constraint is to bound the allowed acceleration (and deceleration) of a mobile vehicle (e.g., due to momentum, a mechanical system cannot instantaneously change its velocity). This is a fundamental issue to consider in order to design systems that are able to build physically feasible plans.

An interesting device that is introduced in order to manage these higher order constraints is the **phase space**. A phase space is obtained by augmenting the configuration space by explicitly introducing the derivatives of the configuration as new dimensions. There is obviously a trade-off between increasing the dimensions of the state space and getting lower order derivatives, but it is widely accepted that the use of the phase space is convenient with respect to the management of higher order derivatives. Given a configuration space $\mathcal{C}$, whose elements are denoted with q, we will denote the phase space as $\mathcal{X}$ and its elements as $x = (q, \dot{q})$.

1.2 Different approaches and problem decompositions

Due to the complexity of the problem and the strict time constraints, the problem of robot motion has been decoupled, in order to obtain smaller problems that are easier to solve (*divide et impera*). There are many decomposition schemes that have been applied at the general problem level, as well as at the sub-problems.

1.2.1 (Re)planning and (reactive) control

Planning a continuous path from the initial configuration to the goal configuration is enough to solve the problem in scenarios such as computer-generated animations and virtual prototyping. However, planning is often considered an off-line act, and when the application involves the interaction with the physical world and the motion in human environments, inaccurate localization, incomplete knowledge of the world and the robot model, and unexpected (moving) obstacles made it clear that there is a gap between planning a motion and executing it (Minguez et al, 2008), i.e., these issues may invalidate the plan before the robot is able to reach the goal. These problems have been historically tackled from two different points of view, that led to two parallel research lines: (partial) re-planning methods and obstacle avoidance techniques.

The former begin with computing a complete and exact plan to steer the robot from its initial configuration to a final goal. These methods rely upon a global and accurate knowledge of the environment, usually not taking into account the uncertainty in the modeling (apart from some simplified case, e.g., Censi et al, 2008). If the plan becomes invalid, due to some unexpected event or uncertainty, is must be re-built. Fast **re-planning techniques** (see, e.g., Bruce and Veloso, 2002; Khatib and Brock, 1999) and methods for interleaving planning and execution (see, e.g., Calisi et al, 2005a) have been developed, that allow for reusing part of the previous plan or to slightly change the whole plan in order to keep it feasible, avoiding the time-expensive action of rebuilding the plan from scratch.

The **obstacle avoidance techniques**, also known as local, sensor-based or reactive techniques, give up the planning step completely and exploit heuristics to move the robot using often only a partial and local knowledge of the environment. Moreover, they are able to deal with the physical, kinematic and dynamic constraints of the robot, allowing to navigate to a local goal in cluttered and dynamic environments.

In this thesis, we would like to smooth the boundary between motion planners and reactive methods. From our point of view, the only difference is the amount of look-ahead that the algorithm exploits: a reactive method *plans* only the next action, while a complete planner builds a sequence of actions that reach the global goal. However, other degrees of planning are possible: if building a complete plan to the global goal requires an excessive amount of time, a partial plan can be built using some heuristics to advance to the goal.

1.2.2 Global path-planning and local methods

Some robot motion methods exploit the whole global knowledge of the environment and they are able to compute solutions using complete algorithms. Unfortunately, due to the complexity of the problem and the strict time constraints, they cannot be used on-line in a real motion system. They can solve in a reasonable time only simplified versions of the real problems, usually relaxing some constraints or reducing the dimensionality of the state space.

For this reason, many approaches compute their motion choices by exploiting only the local portion of the environment model that surrounds the robot (and that is, for this reason, the most critical). Such motion methods are often referred as “local methods” and use some sort of heuristics in order to drive the robot to the goal. The major drawback is that they are incomplete, i.e., they may fail to solve the problem, even if a solution exists. Nevertheless, they are particularly fast in a wide range of typical situations and allows for the construction of motion systems that are both quite efficient and reasonably reliable.

In order to improve the efficacy of local methods, some kind of global information can be introduced (Latombe, 1991). Local minima and oscillations are the typical problems that can be easily overcome by the use of such a global information.

A common way to exploit both the global and the local information is to work in two stages. In the first, the motion system relaxes some constraints and/or reduce the degrees-of-freedom of the configuration space; a path is thus found in this simplified problem and used as a "guide" for the second stage, in which the constraints and the remaining degrees-of-freedom are re-introduced and a local method is used to follow the path.

Often the output of the global stage is a geometrical or topological path, describing roughly the route that the robot should follow in the second stage. This "global path" can be given at a very high level of description, such as a sequence of way-points or even a sequence of topological areas to be traversed before reaching the final goal. However, a tight coupling between the two stages is needed in order to avoid situations in which the plan computed in the first stage cannot be followed by the local method at the second stage.

In this thesis we use the global/local paradigm, in particular we concentrate separately on global-level issues, such as environment topology, and local-level issues related to steering the robot, account for kinematic and dynamic constraints, etc. Moreover, we address the connection between the two stages, the possible problems that can arise and how it is possible to solve or avoid them.

1.2.3 Decoupled trajectory planning: path and velocity

Given the dynamic model of the robot, the motion planning problem can be described as finding a control function $u(t)$ yielding a trajectory $\tau(t)$ that avoids obstacles, takes the system to the desired goal state, and perhaps optimizes some objective function while doing so (Choset et al, 2005). The decoupled approach involves first searching for a path $\tau(s)$ in the configuration space and then finding a time-optimal time scaling for the path subject to the actuator limits.

There are important distinctions between the path/velocity decoupling and the global/local approach that we described in the previous subsection. First, path/velocity decoupling can be used completely at the local level, ignoring any global information on the environment: indeed, some of the local algorithms that we describe in Chapter 2 make use of this technique. Second, the curve computed at the first stage of the path/velocity approach is assumed to be a feasible path for the robot, considering its kinematic constraints: only the timing law remains to be determined. On the contrary, the global path computed at the first stage of the global/local approach can contain cusps or other features that are difficult to be followed by a real robot. Moreover, the global planner can completely avoid the computation of a path in the configuration space, providing only a sequence of way-points or topological places to be traversed.

This path/velocity decoupling approach has been used in many different robot motion methods, for example in local obstacle avoidance techniques (Simmons, 1996), or in planning methods to tune the velocity in order to avoid collisions with moving obstacles, or to coordinate multiple robots (Kant and Zucker, 1986). Moreover, the decomposition can be pushed further, by adding a first stage in which a path is computed without considering neither kinematic nor dynamic constraints; this path is then transformed to satisfy the kinematic constraints and finally, at the third stage, the timing function is computed, that satisfies both kinematic and dynamic constraints. The path/velocity decoupling is a broad topic in robot motion (further details can be found in LaValle (2006, Chapter 14) or Choset et al (2005, Chapter 11)) and is used by some of the algorithms and optimization techniques described in the following of this thesis.

1.3 Sampling-based techniques and probabilistic approaches

One of the biggest challenges for robot motion systems is that they must perform on-line. Although pure-planning approaches often work off-line, building a plan that then should be followed, the unpredictability of the environment, the presence of dynamic obstacles or the uncertainty in actuation require often re-planning phases during the execution of the plan. The challenge is thus not only in the complexity of the general problem, but also in the strict time constraints a motion system must satisfy at each iteration.

While complete motion planning algorithms do exist, they are rarely used in practice since they are computationally infeasible in all but the simplest cases. For this reason, attention has focused on probabilistic methods, which sacrifice completeness in favor of computationally feasibility and applicability.

Sampling-based approaches are the current methods of choice for complex robot motion problems. The broad class of probabilistic approaches make use of a random generator in order to explore the state space. In robot motion planning, probabilistic approaches are used in order to explore the infinite-dimensional continuous space of the trajectories. Probabilistic approaches in robot motion planning has two main advantages:

- they allow to avoid the explicit computation of the configuration (or phase) space: they only need a simple collision detection primitive that reports if a point in the space is in collision or not;
- they are able to explore difficult state spaces (e.g., continuous spaces, with high dimensionality, etc.) in a reasonable time.

The main drawback is that, in general, although randomized planners have demonstrated good performance empirically, the are not complete: probabilistic algorithms provide only a weaker form of completeness: the *probabilistic completeness*. This means that if a solution exists, the algorithm will *eventually* find it.

In this thesis we exploit randomized approaches to solve many problems related to robot motion, from trajectory planning to optimization.

2

Two different approaches to local robot motion

Following the global/local decomposition described in Chapter 1, in this chapter, we survey a selection of algorithms and methods used at the local level. There are two conceptually different points of view, from which the local motion problem has been been historically tackled: heuristic reactive control and local deliberative planning. In this chapter, we follow this classification, describing the most common reactive methods in Section 2.1, while the following Section 2.2 addresses the most common algorithms for (local) motion planning and their variants.

2.1 Reactive algorithms

Local or reactive approaches use only a small fraction of the world model to generate robot control commands. This comes at the obvious disadvantage that they cannot produce optimal solutions. Local approaches are easily trapped in local minima such as U-shaped obstacle configurations. These disadvantages can be overcome using a global planner in a two-level approach, as described above. The key advantage of local techniques over global ones lies in the fact that they have only to deal with a subset of the global environment, thus leading to the possibility to have quicker (that can work on-line and in presence of changes in the environment, e.g., moving obstacles) and/or more complex algorithms (in order to take into account kinematic and dynamic constraints, exact robot shape in order to tackle narrow passages, or shape-changing robots, such as those formed by multiple links). These methods are also known as "reactive methods", because they usually use only the current sensor reading or a very short history and decide only the current motion commands (i.e., they do not actually "plan").

In this subsection, we analyze the most widely used algorithms for local and reactive navigation, underlining strong points, weaknesses and cases in which these algorithms cannot be used.

2.1.1 Potential Field Methods (PF)

Potential field methods (**PF**) determine the steering direction by hypothetically assuming that obstacles assert negative forces on the robot, and that the target location asserts a positive force. These methods are extremely fast, and they typically consider only the small subset of obstacles close to the robot, see Borenstein and Koren (1989) and Khatib (1991) for some example. Borenstein and Koren (Koren and Borenstein, 1991) identified that such methods have inherent limitations (e.g., often fail to find trajectories between closely spaced obstacles and they also can produce oscillatory behavior in narrow corridors).

2.1.2 Vector Field Histogram (VFH, VFH+)

The **Vector Field Histogram** (Borenstein, 1991, **VFH**) uses a two-dimensional Cartesian histogram grid as a world model. This world model is updated continuously with range data sampled by on-board range sensors. The VFH method subsequently employs a two-stage data-reduction process in order to compute the desired control commands for the vehicle. In the first stage the histogram grid is reduced to a one-dimensional polar histogram that is constructed around the robot's momentary location. Each sector in the polar histogram contains a value representing the polar obstacle density in that direction. In the second stage, the algorithm selects the most suitable sector from among all polar histogram sectors with a low polar obstacle density and the steering of the robot is aligned with that direction. Two further improvements of this method has been proposed. The **VFH+** (Ulrich and Borenstein, 1998) presents smoother trajectories, more reliability and a reduced parameter set, explicitly compensating for robot width.

2.1.3 Dynamic Window Approach (DWA)

There is a set of methods that deals directly with motion commands, this makes it possible to take into account kinetic and dynamic constraints straightforwardly. Some examples are **SAFA** (Feiten et al, 1994) and **Curvature-Velocity Method** (**CVM**) (Simmons, 1996), but it has been the **Dynamic Window Approach** (**DWA**) that won more popularity in the scientific community.

The Dynamic Window Approach (**DWA**, Fox et al, 1997) deals directly with the motion dynamics of the robot and is therefore particularly well-suited for robots operating at high speeds. The search of control commands (translational and rotational velocities) is carried out directly in the space of velocities. These commands are searched in a *dynamic window*, i.e., only among those velocities that are reachable within a short time interval (due, for example, to acceleration constraints). The choice of the velocity commands for each algorithm step is chosen using an objective function.

The Dynamic Window Approach has been used successfully in many scenarios. Recently, some algorithms extended the dynamic window concept in order to address other issues such as vehicle stability and roll-over avoidance (see, e.g., Spenko et al, 2004).

2.1.4 Nearness Diagram (ND)

The **Nearness Diagram** (**ND**, Minguez and Montano, 2004) is a reactive scheme that performs a high-level information extraction and interpretation of the environment. Subsequently, this information is used to generate motion commands.

In the first step, ND uses a range finder sensor as the main source of information about the environment surrounding the robot. In particular, by using range information, it constructs two diagrams about the nearness of the obstacles. These diagrams are analyzed in order to extract "regions', i.e., portions of the diagram between two discontinuities in the obstacle nearness measurement. Subsequently, given a (local) goal, one of these regions is selected.

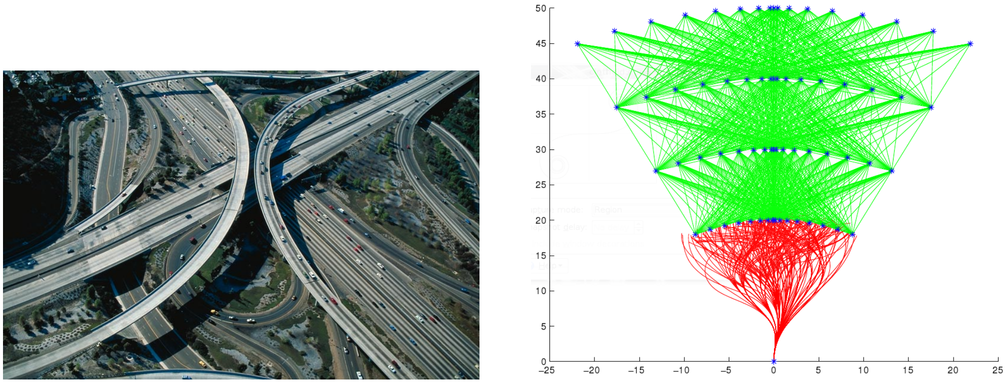
In the second step, by using a set of conditions, the algorithm selects one of five situations, taking into account the robot safety (i.e., the distance to the nearest obstacle) and other issues such as the width of the selected region. Each of the five situations has a related control law that computes the control commands to be sent to the robot base.

The Nearness Diagram has been successfully used to drive robots and autonomous wheelchairs in cluttered environments, its main advantage is that its navigation strategy is well-defined and implemented in a geometrical way, i.e., it does not rely on critical parameters to be tuned. The basic version of the ND only has two parameters, that are used only in two of the five situations described above.

Some variants of the ND have been proposed so far: the **ND+** (Minguez et al, 2004) algorithm adds a sixth situation to the five mentioned above, in order to address a specific case that is not properly handled by the original algorithm; the **Smooth Nearness Diagram** (**SND**, Durham and Bullo, 2008) is proposed as an evolution of the ND+ and make use of only *one* situation and related control law, that can address all situations considered in the ND+ algorithm, and considers all the obstacles surrounding the robot, rather than only the closest two.

2.1.5 Trajectory generation

Another method, that has been used in the last years mainly for autonomous vehicles guidance, is strictly related to the Dynamic Window Approach. Rather than discretizing the allowed velocity space, these methods generate a set of pre-defined



(a) Clothoids curves are often used in highways design

(b) NIST's HMMWV uses a set of pre-generated clothoid curves for local trajectory planning

Figure 2.1: Uses of clothoids curves in robotics and other fields.

(often pre-computed) trajectories and choose one of them through the use of a utility function (as in the basic DWA).

Some of these methods generate trajectories with a well-known geometrical shape: the clothoid curve. Clothoids are planar geometrical curves that have been used in the last years by the motion planning community, because they provide a good model of the curve that a wheeled robot can follow. A curve $\gamma(s)$, parametrized using the arc length, is called a clothoid if and only if its curvature $\kappa(s)$ (defined as the inverse of the current curvature radius) is changing linearly with the parameter s:

$$\kappa(l) = k_0 + k_1 \cdot s. \tag{2.1}$$

Clothoids are used, for example, in highway design (see Figure 2.1(a)), because the linearly changing curvature translates in a linear change of the centrifugal force (and a linear change in the driving wheel rotation translates exactly in a linear change in the curvature of the trajectory followed by a car). For the same reasons, clothoids have been used also in many autonomous driving algorithms. For instance, the NIST's HMMWV vehicle navigation system (Coombs et al, 2000), makes use of pre-generated clothoids for locally computing a set of feasible trajectories, given the current vehicle state (see Figure 2.1(b)). A similar approach is also used in Schröeter et al (2007), where a set of clothoids curves are randomly generated.

The main drawback in using clothoid curves is that they cannot be described by a closed form equation in state coordinates: this means that an incremental numerical integration is needed to compute the curve, e.g., for collision detection purposes. Approximations of clothoids, with a bounded error, have been studied in the past and can be used to perform the integration (see, e.g. Meek, 2004; Walton and Meek, 2005, , that address an approximation equivalent to Euler method).

Other methods make use of even more complex (pre-computed) curves, in order to address complex issues such as rough terrains, vehicle dynamics, models of wheel-terrain interaction, etc. (see, e.g., Ferguson et al, 2008; Howard and Kelly, 2007).

2.2 Deliberative methods

Deliberative methods are allowed to build complete plans to better consider the consequences of their current actions. Since we are dealing with continuous spaces and strict time requirements should be enforced, we consider randomized planners. Moreover, the kinematic and dynamic constraints on the robot motion causes a strict directionality in the actions and the plans that can be generated: for this reason we describe single-query tree-structure-based algorithms.

2.2.1 Prerequisites

In this section, we assume the following definitions of the search space and that the following procedures and functions are available (see LaValle and Kuffner, 2000):

- **Configuration space**: given a parametrization of the configuration of the robot, the configuration space $\mathcal{C}$ is the topological space of all possible configurations of the robot;

- **Metric**: the metric in the configuration space is a real-valued function $\rho : \mathcal{C} \times \mathcal{C} \to [0, \infty)$, which specifies the distance between pairs of points in $\mathcal{C}$;

- **Collision detector or clearance function**: we assume that a clearance function `clearance(q)` is available, that computes the distance between a given robot configuration q and the nearest obstacle in the environment; moreover, for simplicity, we also define another function `collide(q)` that returns 0 if $\texttt{clearance}(q) \geq 0$, and 1 otherwise;

- **Control input set**: the set U is the set of all possible control inputs that can affect the robot state; we use the notation $U(t)$ when the set depends on time (i.e., $U(t)$ is the set of all possible control input that is available at time t);

- **Incremental simulator**: given the current state $q(t)$ and the control input $u(t)$ applied over a time interval Δt, the incremental simulator $\sigma(q(t), u(t), \Delta t)$ computes $q(t + \Delta t)$; the function σ makes use of the robot motion equations, given in the form:
$$\dot{q} = f(q, u). \tag{2.2}$$

Given an initial configuration q_I and a set of goal configurations $\mathcal{G}$, the methods in this section search for a continuous path (or trajectory) in $\mathcal{C}_{free}$ from q_I to some goal configuration $q_G \in \mathcal{G}$.

The following algorithms address this problem, by incrementally building a tree of configurations from q_I towards the goal set $\mathcal{G}$ (or from $\mathcal{G}$ to q_I, or both). When the goal set (or the initial configuration) has been reached, a trajectory exists from q_I to $\mathcal{G}$ and can be executed.

2.2.2 The Expansive Configuration Tree algorithm

In Hsu et al (1999), the **Expansive Configuration Tree** (**EST**) is introduced. The basic form of this algorithm works by building two trees of configurations, one rooted at q_I and the other at q_G. Random configurations are sampled in $\mathcal{C}_{free}$ in the neighbourhood of the nodes of the trees and connected to them using simple local planners. At each iteration, a possible connection between the two trees is searched: if this succeed, the path is found.

Formally, the algorithm iteratively executes two basic steps: *expansion* and *connection*, until a path is found or a stopping criterion is reached. In the expansion step, the two trees are grown by randomly selecting one node q of the tree with probability $1/w(q)$, where $w(q)$ is the weight of the node. Some random configurations are sampled in the neighbourhood of q and connected with q using a local planner (often, a straight-line local planner is used).

The weight $w(q)$ is used to avoid oversampling of any region of the state space, especially around q_I. In particular, $w(q)$ contains the number of configurations already in the tree, within a predefined distance: at each iteration, a configuration q in the tree is selected with a probability that is inversely proportional to $w(q)$. The connection step tries to link the two trees built from q_I and q_G with a local path. If this path is found, a plan is available and the planner can stop its iterations.

2.2.2.1 Randomized Kinodynamic Planning with EST

An extension of the basic EST algorithm allows to build a plan considering kinodynamic constraints (Hsu, 2000; Hsu et al, 2002). This planner builds a tree rooted at q_I, in the collision-free subset of the configuration space. Specifically, this extension of the EST algorithm considers an augmented space $\mathcal{C} \times T$, in which T is the time.

Unlike the basic EST algorithm, after selecting a node q in the tree (given some weighting function $w(q)$), a *control input* u is sampled from the set of all possible control inputs U. Exploiting the forward simulation function σ, we can integrate this control input over a small interval Δt and retrieve the new state q_{new}; if this state is not colliding with the obstacles in the environment, it can be inserted in the tree. Kinodynamic constraints in the local trajectory between q to q_{new} are automatically satisfied by construction.

This procedure ends when one of the new configurations falls in a so-called "end-game" region, i.e., an area in $\mathcal{C}_{free}$ from which it is known how to reach the goal. It is also possible to build a second tree rooted at q_G and integrating the equation of motion backwards in time, thus achieving the same bidirectional search of the basic EST algorithm.

```
function basicRRT($q_I, q_G, \mathcal{E}$) → tree {
    // $\mathcal{E}$ is a description of the environment
    // $q_I$ is the initial configuration
    // $q_G$ is the goal configuration
    tree.init($q_I$)
    while (not targetReached(tree, $q_G$)) {
        $q_{rand}$ = generateRandomState()
        $q_{near}$ = selectNearNode(tree, $q_{rand}$)
        $q_{new}$ = extendNode($q_{near}, q_{rand}, \mathcal{E}$)
        if ($q_{new}$) {
            tree.insert($q_{new}$, parent: $q_{near}$)
        }
    }
}
```

Figure 2.2: The pseudo-code of the basic RRT algorithm

2.2.3 The Rapid-exploring Random Trees (RRT) family of algorithms

The goal of a single-query path planning method is to compute a trajectory from an initial configuration q_I to a goal configuration q_G, without performing any preprocessing. In LaValle (1998) and Kuffner and LaValle (2000), the **Rapid-exploring Random Tree** (**RRT**) algorithm is introduced. The algorithm uses a sampling based technique to quickly search in high-dimensional spaces that have both algebraic constraints (obstacles) and differential constraints (non-holonomy and dynamics). The main idea behind this algorithm is to bias the exploration towards unexplored portions of the search space.

The basic RRT algorithm is shown in Figure 2.2. Similarly to the EST algorithm, the RRT grows a tree data structure starting from the configuration q_I, usually the current robot position.

The difference between EST and RRT is in the procedure to extend the tree. In RRT, at each iteration, a random configuration q_{rand} in $\mathcal{C}$ is sampled. The function `selectNearNode` selects the node q_{near} already in the tree that is nearest to q_{rand}, given a metric ρ in the configuration space. The function `extendNode` makes a motion from q_{near} to q_{rand} with some fixed distance ε and tests for collisions with obstacles in $\mathcal{E}$. This procedure is also depicted in Figure 2.3.

Figure 2.4 shows that the probability to select a node in the tree is proportional to the area of its Voronoi region. This biases the RRT to rapidly explore the search space. Many variants of the basic RRT algorithm have been developed. In the following we will describe some of them and the possible choices for the main functions involved in the algorithm (i.e., `generateRandomState`, `selectNearNode`, `extendNode`).

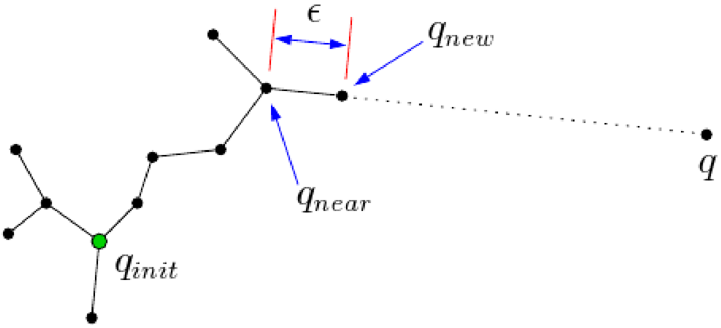


Figure 2.3: The `extendNode` function of the basic RRT algorithm

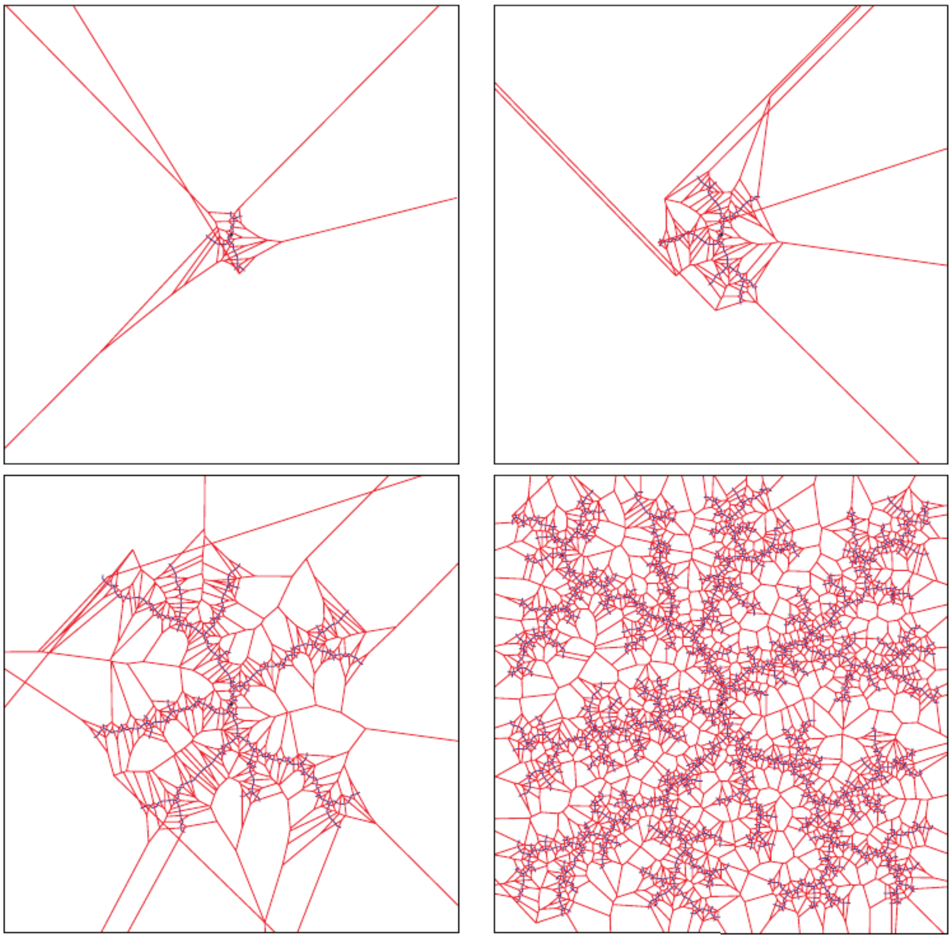


Figure 2.4: The RRT contains a Voronoi bias that causes rapid exploration

2.2.3.1 `generateRandomState`: biases and heuristics for the random configuration

In principle, the basic RRT algorithm can be used as a path planner by itself, because its vertexes will eventually cover $\mathcal{C}_{free}$, and thus reaching the goal. However, without any bias, this convergence might be very slow.

In LaValle and Kuffner (2000), two improved planners are obtained from the basic RRT, by replacing the basic `generateRandomState` function, that generates completely random configurations in $\mathcal{C}$ or $\mathcal{C}_{free}$, with a biased generator.

The first improvement leads to a variation of the basic RRT algorithm, called **RRT-GoalBias**, which uses a function that tosses a biased coin to determine what should be returned. With a given probability p it returns the goal configuration q_G, and with probability $1 - p$ it returns a random configuration. Even with a very small p (e.g., 0.05), this improved RRT planner usually converges to the goal much faster than the basic RRT. A further improvement of the above heuristic, that yields another variation of the basic RRT, called **RRT-GoalZoom**, can be obtained by modifying the `generateRandomState` function with a function that chooses a random sample either from a region around the goal or from the whole $\mathcal{C}$. The size of the region around the goal is controlled by the closest RRT vertex to the goal at any iteration. The effect is that the focus of samples gradually increases around the goal as the RRT draws nearer.

2.2.3.2 Bidirectional search

Taking inspiration from the classical bidirectional search techniques, the basic RRT algorithm has been modified in (Kuffner and LaValle, 2000) by growing two RRTs, one from q_I and the other from q_G: a solution is found if the two trees meet. A naive bidirectional search algorithm with RRTs can be obtained by generating random configurations and extending both trees to that. Another bidirectional search based on RRT is called **RRT-Connect**: at each iteration, one of the trees is extended with a new node and an attempt is made to connect the nearest node of the other tree to this new node. The RRT-Connect algorithm also uses a modified version of the `extendNode` function, called `connectNode`, that repeatedly extends the nearest neighbour until the goal or an obstacle is reached. This results in a very greedy algorithm.

2.2.3.3 Partial RRT planners

In this chapter, we are assuming to deal with local algorithms, i.e., they rely upon a local environment model surrounding the robot. This assumption is motivated by the fact that the global information of the environment that is not in the area currently spanned by the sensors can be inaccurate and obsolete. Moreover, the uncertainty in actuation yields uncertainty on the planned states that increases with the depth of the nodes in the tree: the evaluation of future states can become too vague to prevent a satisfying planning. Finally, it can be impossible to generate a complete plan within the constraint on the time allowed for each single iteration.

For these reasons, some works explores the possibility of growing RRTs that do not reach the goal state, i.e., in which the trajectories are only partial plans. For example, in Urmson (2002), the randomized growth of the RRT is allowed only for nodes within a fixed time horizon (i.e., nodes that are beyond this horizon cannot be extended). After the time slot that is allowed for each iteration, the algorithm chooses in the tree the trajectory that minimizes some utility function that considers a combination of local issues and global information provided by a global planner. This fixed time horizon issue, is further addressed in Chapter 5, when the reactive methods and local planners are merged in the same comprehensive algorithm.

In Petti and Fraichard (2005), an RRT-based partial planner, called **Partial Motion Planner** (**PMP**), is presented, that explicitly exploits a state-time space, in order to be more effective in dynamic environments. In particular, due to the fact that only partial plans can be found in the time slot allowed for each iteration, this work aims at handling the safety issues raised by partial planning, i.e., it considers the behavior of the robot at the end of the partial trajectory ("what if a car ends its trajectory in front of a wall at high speed?").

2.2.3.4 The `selectNearNode` function: metric and nearest neighbour search

One of the main issues of the RRT-based algorithms is that their performance is very sensitive to the choice of the metric that is used to find in the RRT the nearest node to q_{rand}. The simplest choice is to use a weighted Euclidean metric, but it can perform poorly in the presence of kinodynamic constraints on the robot motion. The perfect metric is the optimal cost (for any criterion, such as time, energy, etc.) to get from one state to another. Unfortunately, computing this ideal metric is as difficult as the original planning problem.

In Petti and Fraichard (2005), a metric based on the continuous curvature (CC) is exploited. The continuous curvature metric has been presented in Scheuer and Fraichard (1996), and exploits the non-holonomic constraints on the motion, and thus is more accurate (experiments in the above-cited paper shows that the exploration is more efficient, with respect to the use of a Euclidean metric).

An extension of the RRT algorithm, called Execution Extended RRT (see Section 2.2.3.6 below) considers the metric ρ described so far as a *cost-to-go* (from any q in the tree to the q_{rand}) and introduces also a *cost-to-come*, i.e., the distance between the root and the node in the tree. The actual function used to choose the q_{near} node is a linear combination between the cost-to-go and the cost-to-come. Biasing the linear combination towards the cost-to-come results in shorter paths from the root to the leaves, but also decreases the amount of exploration. On the other side, biasing the linear combination to the cost-to-go make it more similar to the basic RRT algorithm. The authors state that the parameter that tunes the linear combination is domain-dependent and thus an adaptive mechanism is needed.

In Frazzoli et al (2002), the metric is an optimal cost function in the obstacle free case, i.e., the time needed by an optimal controller to move the robot to q_{rand}, in absence of obstacles. The value of this metric is determined incrementally, by updating lower and upper bounds during the exploration. A similar approach is presented in Urmson and Simmons (2003), where the **heuristically-guided RRT** (**hRRT**) is presented. Through a rejection sampling technique, the node q_{near} in the tree is chosen with a probability that is dependent on both the size of its Voronoi region (a bias towards exploration) and the quality (i.e., the inverse of the cost-to-come) of the path to that node (a bias towards exploiting known good parts of the search space).

Furthermore, as the RRT grows, an efficient nearest neighbour search algorithm is required. The naive approach of comparing the distance from q_{rand} to every node in the RRT can be inefficient. In recent years, many efficient data structures and algorithms have been developed, that make it possible to compute the nearest neighbour in near-logarithmic time (Arya et al, 1998; Indyk and Motwani, 1998).

2.2.3.5 The `clearance` and `collide` functions

Since RRTs are based on incremental motions, the performance of the collision detection function (and the distance computation) can be dramatically improved. Many methods have been proposed (e.g., Lin and Canny, 1991; Quinlan, 1994; Mirtich, 1998; Guibas et al, 2000), some of which can compute the distance between closest pairs of points in the world in “almost constant time”. Moreover, almost any collision detection method is able to compute, as a side effect, the distance to the closest obstacle, needed by the `clearance` function.

2.2.3.6 Dynamic environments: (partial) re-planning

In Bruce and Veloso (2002), an RRT-based path planner, called **Execution Extended RRT** (**ERRT**) is developed, which interleave planning and execution. The two extensions introduced by this algorithm are

- a *way-point cache*, that improves re-planning efficiency by keeping old tree nodes between iterations;
- an *adaptive cost penalty search*, which improves the quality of the generated path.

The ERRT algorithm encodes the tree nodes in an efficient spatial data structure, the KD-tree, in order to speed up the nearest neighbour look-up. Moreover, the bi-directional search of the RRT-connect algorithm is not used here, because it decrease the generality of the goal state specification.

The main idea behind the ERRT algorithm is that if a plan was found in a previous iteration, it is likely to yield insights into how a plan might be found at a later time when planning again: the history from previous plans can be a guide. This is exploited by keeping a constant size cache of nodes, filled by plan states when a new plan is found. The target state for the RRT expand step is then chosen stochastically among a random state, the final goal or a random element in the way-point cache.

Another approach is to reuse whole trees from previous iterations, instead of a node cache. This category includes the **Extended RKP** (**ERKP**) (Calisi et al, 2005a) and the **Dynamic RRT** (**DRRT**) (Ferguson et al, 2006), that are essentially the same algorithm discovered by two independent research groups. These algorithms check for the validity of the current tree and removes those branches that begin with a collision. The main difference between the two algorithms is that in the interleaved plan-and-execution version of the algorithms, the ERKP builds the tree from the robot towards the goal, while the DRRT builds the tree in the opposite direction. This, if an obstacle make some branches invalid, forces the DRRT to always re-grow the tree until it reaches the current robot pose, in order to have the current action to take, while in the ERKP the collisions prunes branches towards the goal, thus making partial plans to the goal, but not requiring a full re-growing before having the current action to take. The ERKP algorithm also removes those branches that belongs to plans that have been discarded by current choices at junctions. On the other hand, if the plans are long (i.e., the goal is far from the robot), the DRRT technique allows for keeping a large part of the tree (since obstacles are found near the robot, i.e., in the space currently spanned by the sensors).

Finally, *forest-based RRT planners* go a step beyond and maintain multiple trees instead of just a single tree: they removes from the tree only the edges that are in collision, thus splitting the initial tree in multiple trees, in addition, they plant several tree roots in the configuration space and grow an RRT from each of them, thus generalizing the bi-directional search RRT-Connect algorithm. Tree merging and pruning are often included in the algorithms in order to keep the forest management tractable.

Among the forest-based RRT planners, the **Reconfigurable Random Forest** (**RRF**) (Li and Shie, 2002) algorithm, and the **Multipartite RRT** (**MP-RRT**) (Zucker et al, 2007), extend the basic RRT algorithm and can account for environmental changes while keeping the size of the tree small. They can be used either for single-query or for multiple-query problems. Keeping previously computed trees saves a lot of computation in successive queries. These trees, after a disconnection due to a (moving) obstacle in the environment, are grown towards each other in order to reconnect them. Certain portions of the forest can be invalidated due to moving obstacles: if a collision is found in any of the nodes of the forest, the sub-trees rooted at its children are trimmed off and becomes new trees in the forest.

Finally, the **Lazy Reconfigurable Forest** (**LRF**) (Gayle et al, 2007) extends the RRT algorithm by using a forest of RRTs. The tree that contains the current robot state is called *inhabited tree*. The method maintains a set of *task paths*, assuming that the robot has multiple task to be executed, when the robot chooses a task, the planner directs the robot towards the corresponding task path, and while the robot is moving or a change in the environment is detected, the LRF reconfigures the task paths, the inhabited tree and the whole forest if necessary. The inhabited tree's root is shifted as the robot moves, in order to keep it associated with the robot current state. The algorithm is said to be “lazy” because it grows the trees only when task paths are needed and because it checks for collisions only along the current paths, instead of the whole forest.

The LRF algorithm combines ideas from both the DRRT/ERKP and RRF/MP-RRT algorithms, in particular, it improves the efficiency of the RRF/MP-RRT algorithms by the use of lazy evaluation. The LRF only checks links along the task paths, similarly to how ERKP focus on links in the path to the goal, unlike the RRF/MP-RRT that checks for invalidated nodes among all trees in the forest. However, removing links in the LRF spawns new trees, as RRF/MP-RRT does, while the DRRT and ERKP destroy entire sub-trees.

In Li and Shie (2002), a `pruneRRT` procedure works as follows: each tree in the forest is visited in post-order (each node is examined after its sub-trees are traversed), and a node is removed if it is considered redundant. A node q is considered redundant if one of the following conditions is true:

- the distance between one of its child nodes (q_c) and its parent node (q_p) is less than some fixed limit and a collision free local path exists between q_p and q_c: in this condition, we perform a *vertical merge*, i.e., we connect q_p directly to q_c; moreover if q remains without children, we remove it from the tree;

- the distance between a pair of its child nodes (q_{c1} and q_{c2}) is less than some fixed amount and all q_{c1} child nodes can be linked to q_{c2} with collision free local paths: in this condition, we perform an *horizontal merge*, deleting q_{c2} and linking all q_{c2} children to q_{c1}.

2.2.3.7 Randomized Kinodynamic Planning with RRTs

The inclusion of kinodynamic and non-holonomic constraints can be accomplished by modifying the `extendNode` function (LaValle and Kuffner, 2001). A *state transition equation* $\dot{x} = f(x, u)$ is defined to express the non-holonomic constraints. By integrating f over a finite fixed time interval Δt, the next state q_{new} can be computed from a given initial state q and a control input u using numerical integration.

The `extendNode` function can be implemented as follows. Given the set of all possible control inputs U, a finite subset $\bar{U} \subseteq U$ is chosen (either randomly or using some heuristic criteria). Applying these control inputs over a fixed (or randomly generated) interval, the control input u that yields a new state as close as possible to q_{rand} is chosen. The `extendNode` function implicitly uses also the `collide` function, in order to determine if the generated q_{new} is in $\mathcal{C}_{free}$.

2.2.3.8 Decrease the number of generated trajectories

Some authors try to prune large sets of generated trajectories, since many of them are very similar to each other and their number can be reduced without a loss of the overall problem of path-finding. In Branicky et al (2008), the concept of *path* and *trajectory diversity* is used: the best pruning for a given pre-computed trajectory set is the one that maximizes the probability of the survival of paths, averaged over all possible obstacle environments. This result can be generalized to any set of trajectory, not only for those that can be found on a RRT.

2.2.4 Path smoothing and trajectory deformation

The output of a path planner is a continuous path along which the robot will not collide with obstacles. However, any model of the real world will be incomplete and inaccurate, thus collisions may still occur if the robot moves blindly along such a path. Control theory and obstacle avoidance techniques enable the robot to use sensing to close a feedback loop and interact with the environment in real-time. Another technique is to deform the initial path in such a way that it remains free of collisions. Moreover, if the planner uses randomized techniques, it is customary to perform path smoothing to partially optimize the solution paths. Many algorithms exist that are able to deform the initial path/trajectory in order to optimize it following some criteria such as increasing the distance from obstacles, reducing the path length, smoothing the path, etc.

For holonomic planning, simple and efficient techniques can be employed, while in the presence of differential constraints, the problem becomes slightly more complicated. The variational techniques from classical optimal control theory can be used: they work by iteratively making small perturbations to the trajectory by slightly varying the inputs and verifying that they remain free from collisions. For many problems, this approach produces a trajectory that is optimal over the homotopy class that contains the original trajectory.

All the methods described in this section deforms the trajectory in the set of homotopic trajectories to which the initial trajectory belongs. This means that the trajectory cannot change class of homotopy during the deformation. It is thus important that the initial trajectory lies in the homotopy class of the optimal trajectory.

A simple basic approach can be found in Zucker et al (2007): at the end of the tree building process, the smoothing algorithm iterates back from the goal state to try to find a “shortcut” that connects directly to the initial state. This smoothing method tends to produce paths that pass closer to obstacles (it optimize the path length). However, this method does not enforce directly kinodynamic and non-holonomic constraints.

In the following of this section, we present some algorithms that can be used to optimize the trajectory that is found on a RRT, and/or to continuously deform it in order to keep it collision-free in the presence of moving obstacles, or in order to account for inaccuracy in robot motion execution.

2.2.4.1 Exploiting Lie group symmetries

In Cheng et al (2003), a gap-reduction technique is presented to deform the initial trajectory in order to bring the final state nearer to the goal state. The main motivation for this algorithm is that quantization is performed in most planning algorithms, which leads to approximate the goal satisfaction for some specified precision $\epsilon > 0$. The algorithm described in this subsection allows for increasing the accuracy after the planning step has finished. In most cases, the running time of the planning algorithm increases dramatically as ϵ is decreased. A planning algorithm can generate a solution quickly for large ϵ tolerance and then improve the accuracy in a second stage. This method can be used, for example, to reduce the gap between the two RRTs of the basic bidirectional RRT-Connect algorithm.

The trajectory $\xi(t) \in \mathcal{X} \times T$ is discretized into H steps: in this way, the control input function is discretized in a vector $u_{[0:H]} = \{u_0, u_1, \ldots, u_H\}$ of control inputs. The final state of the trajectory is denoted by x_F as usual. The proposed method perturbs the control inputs $u_{[0:H]}$ in such a way that the final x_F is nearer to the goal region. In particular, it perturbs a subset of the components of the control vector $u_{[0:H]}$, actually solving a nonlinear program in which the objective function is the distance between the final x_F and the goal region (or some fitting function related to the goal).

The main feature of the method is that by exploiting the group symmetries in the system dynamics, it can avoid most of the (numerical) integrations needed after each perturbation: if the perturbed state transition differs from the original state transition by an action of the symmetry group, this allows the remaining part of the trajectory to be rigidly translated without the need of any re-computation (integration).

2.2.4.2 The Elastic Strips Framework

The Elastic Strips Framework (Khatib and Brock, 1999), developed from the previous Elastic Bands Framework (Khatib and Quinlan, 1993), is another approach to real-time path deformation. In the Elastic *Band* Framework, a previously planned path is modeled as elastic material. Obstacles exerts a repulsive force on the trajectory, this can be seen as a moving obstacle pushing and deforming a rubber band. When the obstacle is removed, the trajectory will return to its initial configuration, just as a rubber band would. The elastic *band* is a one dimensional curve in the configuration space $\mathcal{C}$.

Since the configuration spaces for many degrees of freedom lead to a high computational complexity and since tasks are specified in the workspace, the Elastic *Strips* Framework operates entirely in workspace, even for many degrees-of-freedom robots, such as mobile manipulators. In this case, the trajectory and the task are both described in the workspace; the trajectory can be seen as elastic material filling the volume swept by the robot along the trajectory. This strip of elastic material deforms when obstacle approach and regains its shape as they retract.

Along the trajectory, a discretization of configurations $q_0, q_1, \ldots, q_H$ is used. Collision-checking along the trajectory is achieved using a *protective hull* around each configuration in the workspace. The union of such protective hulls forms an *elastic tunnel* along the trajectory.

The path is improved using a pair of virtual forces. A contraction force (called *internal force*) simulates the tension of a stretched elastic band and removes any slack in the path. A second force (called *external force*) repels the band from the obstacles. The two forces deform the elastic until equilibrium is reached.

The framework also enforces the on-line trajectory execution and uses cubic splines to connect the configurations that result from the discretization of the trajectory (Brock and Khatib, 2002).

2.2.4.3 Non-holonomic trajectory deformation

In Lamiraux and Bonnafous (2002); F. Lamiraux and Lefebvre (2004), an approach to path/trajectory deformation is developed that, unlike the Elastic Bands/Strips Framework, explicitly accounts for non-holonomic constraints. The method assumes that an initial collision-free path is available from some path-planner, and deforms this initial path in order to keep it in the $\mathcal{C}_{free}$, also in the presence of unexpected and/or moving obstacles. The path deformation process is modeled as a dynamic control system.

A trajectory $\tau(t)$ is uniquely defined by an initial configuration $q(0)$ and a control input function $u(t)$ defined over an interval $[0, t_F]$. To deform the initial trajectory, we thus need to perturb the input function of the trajectory. A (vector-valued) perturbation function $v(t)$ is defined over the same interval $[0, t_F]$ and a set of perturbed trajectories $\tau(t, \beta)$ is defined. The set $\tau(t, \beta)$ is obtained by perturbing the control inputs of the initial trajectory $\tau(t, 0)$ with the function $v(t)$ scaled by the parameter β. That is, for each $t \in [0, t_F]$, the control input becomes $u(t) + \beta v(t)$.

Using some utility function that is related to obstacle distance and other features, a potential field $V(\tau)$ is defined over the set of trajectories. In this way, the trajectory can be optimized following the gradient of this potential field with respect to the parameter β.

In Lamiraux et al (2004), this non-holonomic trajectory deformation method is used in conjunction with a variant of the RRT-Connect algorithm to improve efficiency. In particular, a tolerance in the distance between the two trees is required, during the connection step of the algorithm. On the one side, if this tolerance is small, the RRT algorithm would create huge (inefficient) trees before the stopping criterion is met. On the other side, if the tolerance is big, either the trajectory ends far from the goal or there is an important discontinuity in the solution trajectory. Using the above trajectory deformation, a large tolerance is allowed, and the planned trajectories can be deformed afterwards in order to reduce discontinuities and/or make their final configuration nearer to the goal.

2.2.4.4 Trajectory deformation using the “Teddy” Trajectory Deformer

Another approach to *trajectory* deformation is given in Fraichard and Delsart (2008). In this paper, the **“Teddy” Trajectory Deformer** is introduced. The trajectory is discretized into a sequence of nodes. Each node is state-time pair. Teddy operates periodically with a fixed time period. At each iteration, it deforms the part of the trajectory that remains to be executed.

The deformation step, i.e., the displacement of the nodes, work as in Elastic Strips Framework: an external force push the nodes away from obstacles and an internal force is aimed at maintaining the feasibility and the connectivity of the trajectory, i.e., to ensure that the trajectory satisfies the dynamics of the robot. The main difference with the Elastic Strips Framework is that Teddy works in the state-time space $\mathcal{X} \times T$ (T is the time dimension), rather than in the workspace $\mathcal{W}$: the trajectory is represented by a finite sequence of nodes $n_i = (x_i, t_i)$, in which $\forall i, x_i \in \mathcal{X}, t_i \in T$.

External forces are repulsive forces exerted by the obstacles of the environment, their purpose is to deform the trajectory in order to keep it collision-free. Internal forces on the other hand are aimed at maintaining the feasibility and the connectivity of the trajectory. Finally, for the sake of collision-checking and connectivity evaluation, it is desirable to maintain a regular sampling level along the trajectory, for this reason, depending on the situation, nodes are added or removed accordingly.

The external forces are determined using the distance from the obstacles in $\mathcal{W} \times T$. They are translated in forces in $\mathcal{C} \times T$ using the Jacobian (i.e., the derivatives of the configurations $\in \mathcal{C}$ with respect to the positions $\in \mathcal{W}$). For the internal forces, the trajectory connectivity is related to the concepts of forward reachability for a state x, denoted with $\mathcal{R}(x)$, that is defined as the set of states that is reachable from the state x using some control law $u(t)$ for a given interval Δt, and backward reachability $\mathcal{R}^{-1}(x)$, that is defined as the set of states from which it is possible to reach a given state x. For each three consecutive nodes n_{-}, n and n_{+}, they are connected if and only if $n \in \mathcal{R}(n_{-}) \cap \mathcal{R}^{-1}(n_{+})$. The internal forces act in order to keep the nodes inside the set $\mathcal{R}(n_{-}) \cap \mathcal{R}^{-1}(n_{+})$.

2.3 Discussion

The reactive methods and the deliberative methods, that have been presented in this Chapter, are seldom used in conjunction. Actually, they form two different research lines: from the one hand, obstacle avoidance techniques focus on reactivity and dynamic environments, trading off the accuracy and the optimality of the maneuvers with a fast response to environment changes and a low computation overhead; on the other hand, (partial) planning methods address more complex scenarios and robot models, and are often used in those cases in which the heuristics of the reactive methods fail in providing the proper motion that reaches the goal.

In Chapter 5, a unifying framework is presented, that allows to include both reactive methods and (partial) planners in the same algorithm, by smoothing the boundaries between the two classes and seamlessly choose among them, depending on the situations.

3

World models for high-level path-planning

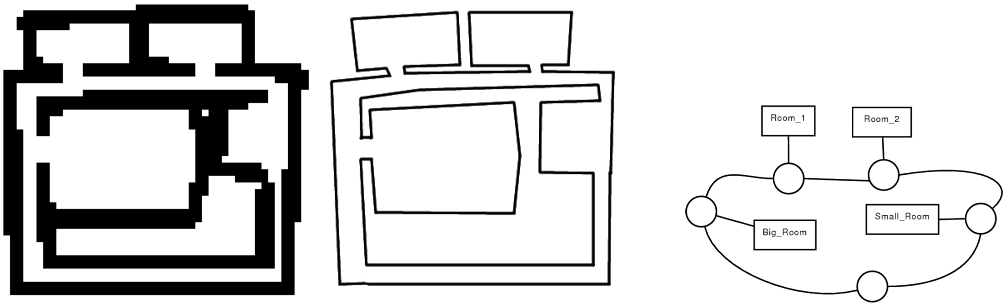
The use of pure-local methods can lead to problems, such as oscillatory behaviors and trap situations, due to the limited model of the world that is exploited. The usual solution is to incorporate a global knowledge into the system. In this chapter, we detail this integration and we show the characteristics of the most known world representations that are used at the global level.

3.1 Integrating global planners and local algorithms

As reported in Minguez et al (2001), local methods developed in the past, eventually evolved to deal with their lack of global reasoning. The main problem to overcome is the presence of local minima in the pure-reactive methods, that, in some situations, prevent the system to reach the global goal. The common solution to this problem, as we already mentioned in Section 1.2.2, is to introduce a global path-planner that is able to “guide” the local system. For example, the DWA and the ND approaches described in Section 2.1.3 and Section 2.1.4 evolved into GDWA (Brock and Khatib, 1999) and GND (Minguez et al, 2001), by adding a global planner that computes the global path on a 2D grid-based representation of the environment.

Moreover, the integration of these two modules often lead to the definition of more complex architectures, that include also modeling modules. For example, Minguez (2005) and Montesano et al (2006) describe a three-component architecture, comprising a modeling component, a global planner and a local obstacle avoidance controller. Well-defined interfaces allow for easy component substitution: in particular, the usual interface between the global planner and the local obstacle avoidance method is a subgoal location (as in the system architecture defined in this thesis or in Stachniss and Burgard, 2002), or a local direction (as in Minguez et al, 2001).

In following of this chapter, we show the most common environment representations that a global-level path-planner can use in order to drive the local-level motion subsystem in a local-minima free execution. For each of the described representations, one or more methods to compute the relevant information for the local-level algorithm (i.e., the local goal or the local direction) is given. Moreover, some details regarding the space occupied by the representations and, therefore, their scalability, are included in the descriptions.



(a) grid map (b) line map (c) topological map

Figure 3.1: Three different representations used to model the same environment

3.2 Grid (raster) maps

The scientific community began to use *grid-based representations* (see Figure 3.1a) from the very beginning (Nilsson, 1969). In recent years, grid-based representations became more and more fine-grained and began to include uncertainty (e.g., occupancy grids). These models are currently the most used ones, because they are easy to build and to update, and they present a straightforward support for probabilistic measures. Using a grid map, the global goal is defined using a global reference frame and the robot is assumed to know its position with respect to this reference frame.

Three main shortcomings affect grid-based representations. The first is that the resolution must be fine enough in order to reflect the connectivity of the environment, i.e., it cannot allow a narrow passage in the real world to be lost in the map (i.e., grid cells should have a size compatible with the robot size): since in this chapter we are dealing with global maps, that can represent very large areas, the resulting size of the representation can be huge. The second drawback is directly connected with the first one, and regards efficiency, with respect to the computation needed by motion planning algorithms, such as for collision detection or for path-planning. In order to alleviate these drawbacks, hierarchical and/or maps comprising cells of different sizes can be used. Finally, pure grid-based representations are also the most deeply affected by errors deriving by global localization and are the most difficult to be "corrected" when topological errors are detected (many current Simultaneous Localization and Mapping algorithms use hybrid representations consisting of small raster maps connected by topological relations, see, e.g., Bosse et al (2003); Grisetti et al (2007, 2008)).

Given a grid map, finding a global path is usually achieved with algorithms such as A* or D*. The local goal can be computed as the farthest position on the computed global path that is inside the environment currently spanned by the sensors, while the direction can also be computed using the direction of the path.

3.3 Line maps

The representation of the environment by *line models* (see Figure 3.1b) is an instance of the more general geometric modeling, and is a popular alternative to the grid-based approximation. Also in this direction, the research has begun very early (Giralt et al, 1979; Thompson, 1977). Line models require less memory than grids and they scale better with the size of the environment. They are also more accurate, since they not suffer from discretization problems, although the transformation process that converts sensor readings to lines is an approximation and, depending on the method used, can lead to different line maps (see, e.g., Sack and Burgard, 2004; Amigoni and Gasparini, 2008, for a survey on line model approximation of an environment). Given a global line model, a graph called *roadmap* is computed, in a way that it represents the connectivity of the environment. Cell decomposition and visibility graphs are possible choices (Latombe, 1991), as well as the computation of the Voronoi graph (as in, e.g., Choset and Nagatani, 2001; Bhattacharya and Gavrilova, 2008). The local target can be computed on the roadmap using the same method described for the grid maps. Finally, also in this case, the global goal is given using a global reference frame, in which the robot is assumed to be localized. In this case, obstacles are modeled as line segments and the methods to extract information for the obstacle avoidance algorithm are based on geometric cell decomposition, on visibility graphs (Latombe, 1991), or on the computation of a Voronoi graph (e.g., Bhattacharya and Gavrilova, 2008). In both cases, the result is a graph called *roadmap*, that contains metric information about vertexes and edges, and is localized on the map itself. A path is then computed using well-known graph-search algorithms (e.g., Dijkstra or A*) and the local goal is computed, as for the grid maps, as the oriented pose on the path that is at the limit of the area spanned by the robot sensors.

3.4 Topological maps

Geometric models still include metric information. A step further towards minimalistic world models is represented by *topological maps* (see Figure 3.1c), where only the relations between places are represented. Topology is an area of mathematics that studies the characteristics of a space, regardless of metric information. The motivating insight behind topology is that some geometric problems do not depend on the exact shape of the objects involved, but on the way they are combined together. Topological navigation is a behavior that is used by a variety of different animal species, including humans (Lynch, 1960). Topological representations in robotics discretize the continuous world into a finite set of *places* connected by *paths*. This facilitates large-scale spatial reasoning, mainly because of the compactness of the representation. Moreover, this abstraction helps the communication among robots and with humans. The crucial issue in topological mapping is the ability to recognize places, in particular to recognize those places that have been already visited ("close the loops").

An example of the autonomous construction of a topological representation is given in Kuipers and Byun (1991), that relies on the possibility to identify distinctive places, that are connected using arcs that correspond to local control strategies that describe how a robot can follow the link connecting two distinctive places. In Filliat (2008), topology is implicitly inferred only from images taken by the robot camera while navigating in the environment. The representation built has no metric information at all and topological localization is performed thanks to an active perception strategy. This topology is used to perform tasks such as visual homing. In Beeson et al (2005), a topological representation of the environment is build upon a local Voronoi graph, in which the places are the junctions of the graph and the paths are represented by its edges.

In the absence of an underlying metric model, some sensor measurements need to be collected in the places and matched against the current place, in order to recognize an already-visited place and thus "close a loop". Moreover, the topological model of the environment can help detecting sensor-wise identically places that are actually distinct. This topological reasoning can be performed either actively (e.g., the "rehearsal procedure" in Kuipers, 1985) or reasoning about the already visited places (e.g., Werner et al, 2008). A robust algorithm to detect places on the Voronoi graph, as well as already visited places, are the most critical parts of these works: despite of their inherent advantages over grid-based and geometric maps, the construction of a topological description of the environment is more difficult. The topological representation is built on top of a local metric model: this makes the place recognition more stable and reliable and, since it relies only to local metrical information to build the Voronoi graph, it is much less sensitive to global localization errors, that can be found in pure grid-based metric maps. Following this idea of using local metric maps, some works exploit the use of topological maps in order to improve the consistency of the global metric map being built (for example to improve the loop closing robustness) or make it computationally tractable (e.g., Bosse et al (2003); Grisetti et al (2007, 2008)).

Topological maps are usually represented as graphs, in which vertexes are landmarks or places and edges indicates that it is possible to travel directly between them. Exact metric measurements about the position of places and about the relations between places are not stored in the representation. When a goal is given, it is required to localize both the current robot position and the goal in the topological representation, in order to find a topological path, and the ability to compute the current local target pose or direction, given the instructions from the topological path. We will assume that the robot is always aware of its own position in the topological representation and that the goal is given using a topological description (e.g., “go to the docking station that is in the kitchen”, in which “kitchen” is a topological place and the docking station has known local coordinates with respect to a landmark in the kitchen). Moreover, a pure topological approach can be used to compute the current local target direction, as shown, e.g., in Rawlinson and Jarvis (2008): since in pure-topological navigation, decisions have to be taken at junctions, these are treated as landmarks and a simple method can be used to detect the target direction at each junction, for example using angles or counting the edges starting from the one from which the robot is approaching the junction.

Given a topological representation of the environment, both the current position of the robot and the position of the global goal have to be localized with respect to this representation. Moreover, a method is needed that translates pure topological instructions (e.g., the edge to follow at each junction) into the target for the local algorithm. Although the metric information is lost in a topological representation, it can be recomputed using the local sensor readings. For example, in Rawlinson and Jarvis (2008), the places of the topology are the junctions that are found in a local metric Voronoi graph. At each junction, the edge of the topological graph from which the robot comes to the junction is called *reference edge*; there are two ways to detect the direction to follow: the exact orientation can be saved on the graph itself, and thus can be detected using the reference edge, otherwise, it can be identified by counting the edges starting from the reference edge. These techniques are depicted in Figure 3.2.

3.5 Minimalistic environment models

In this group we include those methods that make use of a very limited model of the environment, such as the “bug algorithms” (Kamon and Rivlin, 1997; Lumelsky and Stepanov, 1987) and the **Gap Navigation Tree** (**GNT**, Tovar et al, 2004, 2007). Making use of capability-limited sensors (respectively touch or short-range distance sensors, and gap sensors), these methods are able to build a very simple world model (or not build it at all) from which it is still possible to detect the target direction with respect to the current sensor readings and perform tasks such as environment exploration, surveillance, etc. Classical approaches to motion planning often lack of reliability when applied in practice, due to problems such as map-

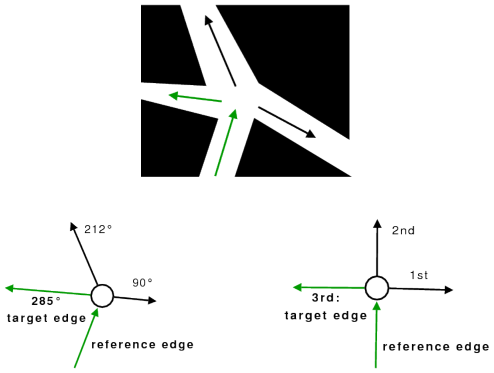


Figure 3.2: Two ways to detect the direction to follow at each node of a topological graph: in the first case, the bearings of the edges are saved in the graph itself; in the second case, the target edge can be detected by counting edges starting from the reference edge

ping uncertainty, localization errors and unpredictable control errors. The primary cause to this is that classical motion planning algorithms, as we said above, rely on a complete and perfect knowledge of the environment, that is impossible to obtain in the real world. For this reason, some works investigate the use of *minimalistic environment representations* and develop algorithms that minimize the information requirements. The famous "bug algorithms" (see, e.g., Kamon and Rivlin, 1997; Lumelsky and Stepanov, 1987) are a typical example of this research line, in which there are also some recent developments (Tovar et al, 2004, 2007), concerning tasks such as unknown environment exploration, surveillance, etc. Compact representations are attractive for two reasons. From one hand, they are more efficient to be built and used, and allow for the modeling of very large environments; moreover, the complexity of the representation is a function of the complexity of the environment, rather than of hand-tuned parameters such as the resolution in metric grid-based modeling. On the other side, these representations are less sensitive to measurement errors and uncertainty and this affects both the environment model and the robot localization. For example, a precise metric localization is often not needed when performing pure-motion tasks (unless the task is to reach a precise goal pose).

Part II

A unifying framework for robot motion systems

4

Tasks and goals

In this chapter, we analyze a set of possible motion tasks that can be requested to a motion system, and extend the goal definition and the problem definition given in Chapter 1. We show the benefits of using this extended definition, both for the local goal and for the global goal. Moreover, we describe some important issues regarding the *task execution* (with respect to its final accomplishment) that should be considered in computing the trajectory. These issues will be considered when we define the evaluation framework in Chapter 8.

4.1 Typical goals for a motion task

The classical motion planning problem definition represents the goal of the motion system as a configuration q_G. However, reaching an exact configuration in a global reference system is not the only task that can be requested to a robot motion system. In fact, it is seldom the *actual* task that we want the robot to accomplish. This definition is thus often expanded by considering the goal as a *set* of configurations $\mathcal{G}$: the objective of a motion system becomes to move the robot from an initial configuration q_I to a configuration q_F such that $q_F \in \mathcal{G}$.

In the following, we give some real examples of robot motion goals and of the corrensponding set $\mathcal{G}$.

- *Move the robot to a specific configuration* as accurately as possible. This is the most known goal for a robot motion algorithm; it is the typical task for a fixed robot manipulator, but is seldom what is really needed to be accomplished. In this case, $\mathcal{G}$ contains a single configuration: $\mathcal{G} = \{q_G\}$.

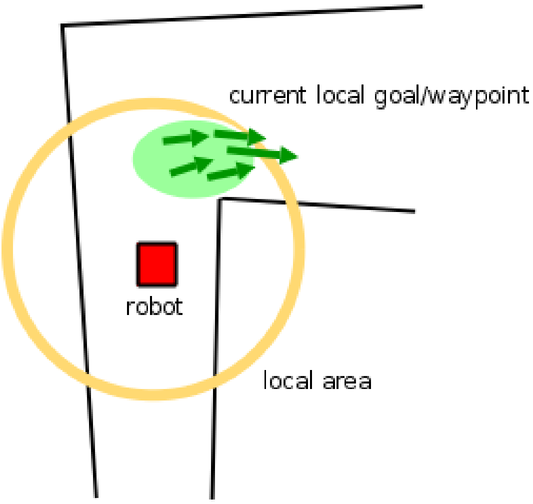


Figure 4.1: The definition of the local goal set $\mathcal{G}$ for the local motion algorithm. The goal set definition includes also the direction.

- *Reach a place*. This is actually the most common goal that arises from a human user when trying to command a mobile robot. The robot is typically required to go "in the kitchen", "outside a building", "through a corridor" and so on: it is not crucial which precise pose or link configuration must be reached. Moreover, complex plans that require a precise pose at the end (e.g., autonomously reach the battery charger), can be decomposed in such a way that only the last action requires a precise final pose (e.g., if the battery charger is in the kitchen, the task can be decomposed in first going to the kitchen and then reaching the battery charger). The description of the target pose set usually includes predicates concerning only the cartesian position of the robot and can contain infinite configurations.

- *Move close to an object*. There are many applications that require to achieve this goal; for example, when the robot has to take some picture of interesting objects (e.g., in rescue missions), or in some position where a manipulator, that is mounted on the robot, can reach an object. The robot goal is not to go to the object, but to go at a specific distance (e.g., to take a picture or in such a way that the manipulator can accomplish the task). The description of the target pose set, in this case, can be very complex and usually includes also orientation and/or the configurations of the robot links.

4.2 A general definition for robot motion goals

In this section we aim at defining a general formalism for a robot motion goal, that generalizes the concepts of the previous section. Although the generalization is trivial, it increases the effectiveness of the motion system and, moreover, is more suitable to deal with high-level goal descriptions. Consequently, all algorithms and methods described in this thesis are compatible with this definition.

Finally, the goals described in this section are generic goals for any component of a motion system. Following the global/local decomposition of Section 1.2.2, this definition applies both to local goals and waypoints for local methods, and to the global task goal. For example, in Figure 4.1, the definition of a local goal set is depicted. Giving a more vague definition of the local goal gives the local algorithm more freedom, that can be exploited, for example, to optimize other trajectory-related criterion, such as safety or speed.

4.2.1 Reference frames

The goal set can be defined using different reference frames, as explained below.

- In the classical definition of the motion planning problem, the robot and the goal configurations are assumed to be given in a **global reference frame**. For an off-line planner, this reference frame should be available during the planning phase, while for an on-line (re)planner, it should be always available to the motion system, in order to compute the plan to the goal, to make the required corrections, and to handle the uncertainty of sensor readings and control action execution. Although such a reference frame can be provided by a reliable GPS or an accurate SLAM algorithm, it is often impossible to obtain sensors and algorithms that are immune to accumulated errors, especially when navigating in very large environments (if GPS is not available, due to environmental reasons, e.g., in indoor scenarios).

- Another way to specify a goal is **with respect to landmarks**, that are assumed to be reliably identifiable. For example, grasping an object is by definition a motion specified with respect to the position of the object. The idea is related to the so-called *landmark navigation*, i.e., navigating always with respect to visible and identifiable environment features, and define the final goal using them. Landmarks are the key feature for well-known SLAM algorithms (e.g., Montemerlo et al, 2002) and can be used to correct the robot motion in the presence of localization and modeling errors (see, e.g., Lazanas and Latombe, 1995; Taïx et al, 2008).

The above reasons justify a bias towards the definition of a goal with respect to reliably identifiable landmarks in the environment. “Reliably” has here a meaning of preference, in order to include the classical global-frame definition of the goal as a special case, in which the landmark(s) with respect to which the goal is defined, is the origin of the reference frame itself. However, definitions given with respect to local (i.e., in the area spanned by robot sensors) landmarks are more reliable and, moreover, are the most common robotic task definitions (e.g., “go near the window”, “go to grasp that object”, “follow that person”, etc.).

4.2.2 The goal fitness function ϕ and the stopping criterion

We can extend the goal set $\mathcal{G}$ definition in the following way. A **goal fitness function** $\phi : \mathcal{X} \rightarrow \mathbb{R}$ is defined, that describe how "good" is an achieved state with respect to the overall task. Given a threshold $\bar{\phi}$, the goal set $\mathcal{G}$ is defined as the set of all the states where the goal fitness function is greater than $\bar{\phi}$:

$$\mathcal{G} = \{x \in \mathcal{X} | \phi(x) > \bar{\phi}\} \tag{4.1}$$

Given the function $\phi(x)$, this definition make it straightforward to tune the accuracy of the motion system, by modifying only the threshold $\bar{\phi}$, possibly during the motion execution.

The threshold $\bar{\phi}$ actually defines the **stopping criterion** of the motion system, i.e., as we formalize below, a motion system is said to accomplish a task if its state x reaches some state such that $\phi(x) > \bar{\phi}$.

4.3 Trajectory and task execution issues

There are two elements in describing and evaluating a motion task: how the robot reaches the goal for the specified task and how it behaves during the execution. For example, in evaluating the performance of a motion system, as we describe in Chapter 8, we consider the final result of the system execution, as well as the behavior during the execution. In this section we describe the issues regarding the task execution.

4.3.1 The trajectory fitness function ψ

As in the case of the goals, we can define a fitness function over the set of the possible trajectories. The **trajectory fitness function** ψ is defined as $\psi : \mathcal{T} \rightarrow \mathbb{R}$. The definition of this function should account for issues such as safety, duration, etc. Choosing among the trajectories that are solution for the problem (i.e., that end in some state $\in \mathcal{G}$), a motion system can optimize the ψ value of the task execution. In particular, given two different ways to accomplish the task (i.e., two different trajectories during the execution and the achievement of the task goal), a behavior is considered better than another if its fitness function is higher. Depending on the application requirements, the robot model and the environment, a trajectory fitness function can be related to reducing oscillations, decreasing the effects of the centrifugal force, increasing the safety by keeping the robot distant from obstacles, etc.

4.3.2 The trajectory constraints P^{τ}

The final concept that we define in this chapter is the **trajectory constraints**, that is the counterpart of the stopping criterion for the goals. However, we define the trajectory constraints in a more general form. The trajectory constraints P^{τ} is a set of predicates that state if a given trajectory is feasible or not: $P^{\tau} : \tau \rightarrow \{true, false\}$. The trajectory constraint predicates generalizes the concept of collision-free motions: some execution can be discarded for reasons that are different from collisions (e.g., kinematic/dynamic feasibility).

Therefore, a successful task execution is an execution that reaches a particular goal (i.e., a situation that meet the requirements defined by the goal definition) in such a way that the task execution never fail to satisfy the constraint predicates on the trajectory. The motion system can try to optimize the planned trajectory (with respect to the trajectory fitness function for the trajectory), while keeping it in the set of feasible trajectories.

4.4 Redefinition of the robot motion problem

We can redefine the robot motion problem, given in Section 1.1.1, using the concepts introduced so far in this chapter. In particular, the problem is defined in terms of the control actions of the motion system and to the generalized concepts of goal fitness, stopping criterion, trajectory fitness and constraints.

- A state space (phase space) $\mathcal{X}$ is defined that can represent all states (configurations and velocities) of the robot.
- For each state $x \in \mathcal{X}_{free}$, an action set $U(x)$ is defined as all the possible control actions that can be taken in x, the union of all possible actions is called $U = \bigcup_{\mathcal{X}} U(x)$.
- A robot model is specified using a differential state transition equation in the form $\dot{x} = f(x, u)$.
- A *trajectory fitness function* is defined as a function $\psi : \mathcal{T}^{\mathcal{X}} \rightarrow \mathbb{R}$, it describes the goodness of a trajectory, regardless the final goal achievement.
- A set of *trajectory constraint predicates* P^{τ} is defined on the trajectory space and determines the set of *feasible* trajectories.
- A *goal fitness function* is defined as a function $\phi : \mathcal{X} \rightarrow \mathbb{R}$, it describes how good is a state with respect to the task goal.
- A *stopping criterion* is defined over the function $\phi(x)$ and defines the goal region $\mathcal{G} \subseteq \mathcal{X}$.

A motion system must compute an action trajectory $\tilde{u} : T \rightarrow U$ such that the cornersponding state trajectory $\tilde{x}$ satisfies the trajectory constraint predicates and there exists some t_F such that $\tilde{x}(t_F) \in \mathcal{G}$. Optionally, the motion system can choose a better trajectory to optimize the trajectory fitness function ψ. However, a motion system is not required to compute a complete action trajectory *before* the task execution: this robot motion problem definition refers to the action trajectory and state trajectory that are the *results* of the motion system on-line behavior.

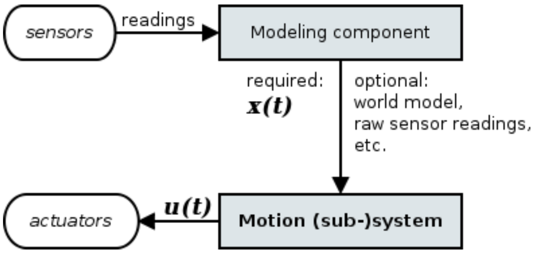

sensors
readings
Modeling component
required:
x(t)
optional:
world model,
raw sensor readings,
etc.
actuators
u(t)
Motion (sub-)system

Figure 4.2: The interfaces of the motion system from a component-based viewpoint. The motion system retrieves inputs from a generic modeling component (possibly a set of sub-components): the modeling component is required to provide the current robot state $x(t)$ and optionally can provide also a world model, the direct sensor readings, etc. At each iteration, the motion system generates a control action $u(t)$. The goal description and the trajectory predicates are assumed to be given.

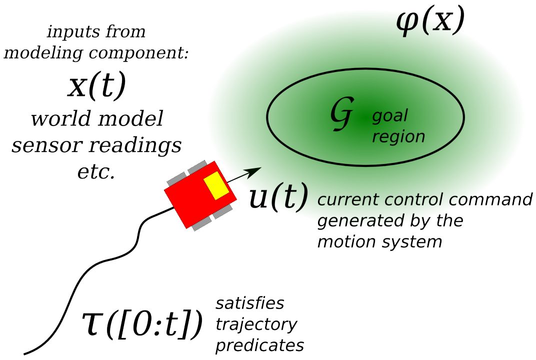

inputs from
modeling component:
x(t)
world model
sensor readings
etc.
φ(x)
G goal region
u(t) current control command generated by the motion system
τ([0:t])
satisfies trajectory predicates

Figure 4.3: The robot driven by the motion system towards the goal region. Given the inputs provided by the modeling component, the motion system generates the current control command $u(t)$, in such a way that the trajectory followed so far ($\tau([0:t])$) satisfies the trajectory predicates and that the robot eventually reaches the goal region ($\mathcal{G}$), i.e., the region of states in which the goal fitness function $\phi(x)$ is greater than a given threshold (stopping criterion).

4.4.1 Motion system as a component

From the overall system point of view, the motion (sub-)system can be seen as a component that interacts with other capabilities of the robot, such as modeling and high-level reasoning. In particular, the interfaces between the motion (sub-)system and other components are depicted in Figure 4.2. Given a goal definition, consisting of a goal fitness function $\phi(x)$ and a stopping criterion, and given the set of trajectory predicates (that include those regarding obstacles), the motion component retrieve inputs from a generic modeling component and generates control (action) commands.

The modeling component is an abstraction that may consist of a set of modules (e.g., they can include the direct sensor reading for obstacle avoidance methods, a map of obstacles for a path-planner, etc.), and is required to provide the current state $x(t)$ of the robot. At each iteration, given the current state $x(t)$, the motion system should provide a control command $u(t)$, such that $u(t) \in U(x(t))$, $\tilde{x}([0 : t])$ (the state trajectory followed so far) satisfies the trajectory predicates, and in such a way that eventually the robot reaches a state $x(t_F) \in \mathcal{G}$ (see Figure 4.3). Optionally, the motion system can choose a better trajectory with respect to the trajectory fitness function $\psi(\tau)$. However, since $\psi(\tau)$ refers to the *final* trajectory, the system can choose a better action $u^*(t)$ only by estimating its effects on the whole final trajectory.

In the following of this thesis, we will make use of this extended problem formalization and the related component-based point of view.

5

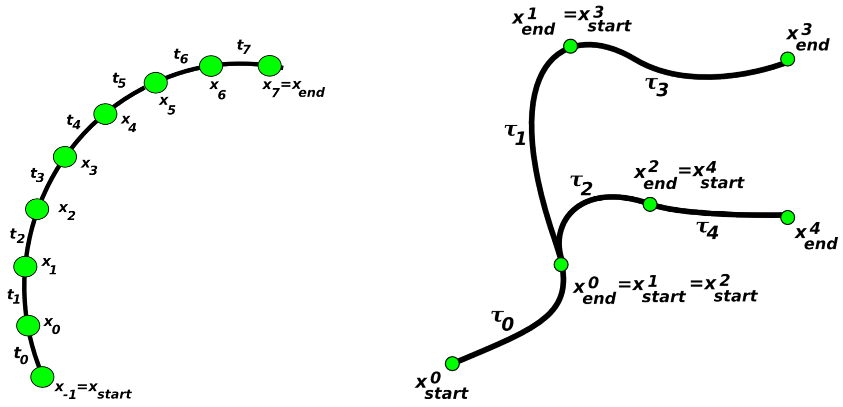
Integrating deliberative and reactive approaches

This chapter introduces two strictly related novel algorithms, developed by the author: the Dynamic Trajectory Tree (DTT) and the Dynamic Behavior Tree (DBT). The basis of the two algorithms is the randomized tree planning paradigm (i.e., RRT and EST) described in Chapter 2. Moreover, DTT and DBT form a sort of framework where different variations of the above-mentioned algorithm can be exploited in order to increase the effectiveness and the efficiency of the motion system.

These algorithms are suitable for on-line use, as they account for feedback control and interleaved execution and planning, to provide robustness to external disturbances, uncertainty and modeling errors, as well as incrementally constructing and correcting the trajectory as new information is available from the environment.

Finally, the Dynamic *Behavior* Tree algorithm introduces the integration with feedback reactive behaviors, such as obstacle avoidance techniques, in the tree construction process, thus reducing the need of randomized search in the case in which well-known reactive methods are able to construct a good solution.

In Section 5.1, the main data structures that are exploited by both algorithms are presented. The DTT algorithm is then detailed in Section 5.2 and Section 5.2.4. The DBT algorithm is described in Section 5.3. Finally, a description of machine learning techniques that are used to tune the parameters of the algorithms is presented in Section 5.4.



(a) a single trajectory arc

(b) the connection among different trajectory arcs

Figure 5.1: A *trajectory arc* in detail, with time step intervals and robot positions, and the connection among subsequent trajectory arcs in a trajectory tree.

5.1 Data structures: the trajectory tree and the trajectory arc

The basic data structure used by the DTT and the DBT algorithms is a tree of trajectories, that is built using (variants of) the RRT algorithm. This **trajectory tree** is defined as a set of linked nodes (vertexes) that has an acyclic structure where each node has a set of zero or more *children* and at most one *parent* node. Each node in the trajectory tree is associated with a structure called **trajectory arc**, that contains relevant information with a specified meaning, as detailed in the following.

A trajectory arc represents a movement of the robot in the state space, following a given time-dependent control action $u(t)$, i.e., a trajectory $\tilde{u}$ in the control space. Each trajectory arc is defined in a time interval $[t_{start}, t_{end}]$ and starting from a state $x_{start} = x(t_{start})$ in the phase space that encodes the robot position q and its velocities $\dot{q}$. A trajectory arc is thus defined as a tuple:

$$< t_{start}, t_{end}, x_{start}, \tilde{u} > \tag{5.1}$$

An incremental simulator σ is assumed to be available that is able to compute the resulting state x_{end} of the robot if it *executes* the trajectory arc.

In the following, we assume that the robot can be controlled by velocities and we consider control actions that have a fixed acceleration along each trajectory arc. Moreover, we assume that the low-level component of our system is able to run its iterations at fixed time intervals. The interval between two subsequent iterations is a fixed (tunable) constant δ_s. The component is able to send commands to the robot at the corresponding rate and the commands are assumed to be kept constant between two subsequent iterations.

An example of a trajectory arc is shown in Figure 5.1(a), where the starting state of the robot is denoted by x_{start}, while the ending state is denoted by x_{end}. The trajectory $\tilde{x}(t)$ represented in a trajectory arc resides in the phase space, i.e., it encodes also the evolution of velocities. The total duration of the trajectory arc is chosen as a multiple of δ_s; in this case, its execution lasts $7 \cdot \delta_s$. During the execution of a trajectory arc, the motion system sends control commands (velocities) to the robot actuators. Since the acceleration along a trajectory arc is constant, the velocities are incremented or decremented at each schedule interval by a fixed step acceleration. Informally, the meaning of a trajectory arc is: starting from state x_{start} and executing the control $u(t)$ (that is allowed to change only at fixed intervals δ_s) for a time $t = 7 \cdot \delta_s$, the robot state will be (approximately) x_{end}. Velocities are uniform inside each interval $t_i, i \in \{0, \ldots, 7\}$. In Figure 5.1(a) is also possible to see the robot states at the end of each schedule interval: this means that starting from the state x_{-1}, that is the ending state of a previous trajectory arc, the system applies a constant velocity during the first interval t_0, resulting in the state x_0 and so on until the final $x_7 = q_{end}$ state is achieved.

Trajectory arcs are linked together in the trajectory tree, in such a way that the ending state x^i_{end} of a trajectory i is equal to the starting state x^j_{start} of the subsequent (children) trajectories j. Figure 5.1(b) shows a trajectory tree where the starting and ending states of the trajectory arcs are highlighted.

5.2 The Dynamic Trajectory Tree (DTT) algorithm

In this section we describe the **Dynamic Trajectory Tree** (**DTT**) algorithm, that is an improvement with respect to our previous Extended Randomized Kinodynamic Planning (ERKP) described in Section 2.2.3.6 and in Calisi et al (2005a). The DTT algorithm is the basis for further improvements introduced in the following of this chapter (i.e., the DBT algorithm), its main improvements with respect to our previous work are: (i) the precise formalization of the meaning of the trajectory arcs, as described in Section 5.1, as well as the introduction of constant accelerating trajectories; (ii) the trajectory deformation process, that allows for a better precision at the global goal, as well as an increase of performance with respect to efficiency; (iii) the introduction of many different variants of the basic RRT algorithm, that have been listed and detailed in Section 2.2.3, in such a way that it is possible to use different components or techniques of different variants of RRT at the same time.

The outline of the Dynamic Trajectory Tree algorithm is given in Figure 5.2. The different variants of the method, that is possible to obtain by this algorithm, are hidden behind the functions, as we detail in the following.

`pruneCollidingArcs` This function performs a collision check along the current trajectory being followed by the system. If some collisions are detected, the corresponding trajectory arcs should be pruned. There are two possible choices on how to prune the tree:

- we can remove the whole sub-tree starting from the colliding arc, as in the

```
currentTrajectory = { }
tree = { }
function dynamicTrajectoryTree {
    do {
        pruneCollidingArcs(currentTrajectory)
        $x_{rand}$ = generateRandomState()
        do {
            $a_{near}$ = selectNearArc($x_{rand}$)
            $A_{cand}$ = generateCandidateArcs()
            $A_{new}$ = selectNewArcs($A_{cand}$, $a_{near}$)
            tree.insert($A_{new}$, parent: $a_{near}$)
        } while (params.connectNode and
            (not stateReached(tree, $x_{rand}$) or A_{new} = \emptyset)
        currentTrajectory = selectBestTrajectory()
        deformTrajectory(currentTrajectory)
        executeTrajectory(currentTrajectory)
    } while (not taskAccomplished())
}
```

Figure 5.2: The pseudo-code of the Dynamic Trajectory Tree algorithm

Extended Randomized Kinodynamic Trees and Execution Extended RRT algorithms (see Section 2.2.3.6);

- we can remove only the colliding arc and split the tree, in such a way that we obtain a forest, as in Reconfigurable Random Forest, in the Multipartite RRT or in the Lazy Reconfigurable Forest (see Section 2.2.3.6).

`generateRandomState` There are three sets from which we can choose the random state x_{rand}:

- the whole state space;
- the goal set $\mathcal{G}$, as in the RRT-GoalBias described in Section 2.2.3.1;
- a subset of the state space, e.g., around the $\mathcal{G}$ set, as in the RRT-GoalZoom described in the above-mentioned section;
- a state in other trees in the forest, in the cases in which the `pruneCollidingArcs` generates a forest;
- a state in the previously pruned part of the trajectory, as in the ERRT algorithm.

The exact behavior of this function is ruled by a set of parameters that define the probability to choose the random state in one of the sets specified above.

`selectNearArc` This function, given the random state x_{rand}, chooses a trajectory arc from the tree. The following criteria can be used:

- a random trajectory arc in the tree is chosen (this does not make use of x_{rand}): this is the only choice of the Expansive Space Tree, described in Section 2.2.2;
- given a metric ρ in the state space, the trajectory arc that is nearest to x_{rand} is chosen: this is the only choice of the basic RRT algorithm;
- as above, but joining the metric with a *cost-to-go* on the trajectory to reach the node in the tree being evaluated, as explained in Section 2.2.3.4, describing the Heuristic RRT algorithm.

The metric ρ has a large set of choices, in addition to the straightforward use of a weighted Euclidean metric. An example is the Continuous Curvature metric described in Section 2.2.3.4. An optional parameter can be used to limit the planning horizon: it is important to notice that, by setting this horizon to 0, only the root of the tree can be selected as a_{near}, thus yielding a pure-reactive motion system.

`generateCandidateArcs` A set of candidate arcs are generated by this function. In particular, in DTT, trajectory arcs are determined by constant accelerating control velocities. The actual candidate arcs generated at each iteration by this procedure can be either a fixed set (e.g., by discretizing the allowed accelerations, as in the Dynamic Window Approach) or generated randomly (e.g., random clothoid arcs as in Schröeter et al (2007).

`selectNewArcs` This function selects, among the candidate arcs generated by the previous function `generateCandidateArcs`, those to be added to the tree. The selection is performed by using some heuristic that can include the metric ρ, so that the arcs that move the robot closer to the x_{rand} state are chosen. The exact amount of new arcs that can be selected to be inserted into the tree, depends on a parameter: the arcs are sorted using the above-mentioned heuristic and then a specific number of them are selected for addition. If the value of this parameter is 1, only the first arc is added to the tree, as in the original RRT algorithm, otherwise, a set of arcs are added to the tree, as in Kalisiak and van de Panne (2006).

`selectBestTrajectory` Given the trajectory fitness function $\psi(\tau)$ that computes the cost-to-come, and a heuristic that evaluates the final state of the trajectory with respect to the goal set $\mathcal{G}$, for example using the metric ρ, the best trajectory is chosen to be followed by the system. The open choice for this function is which metric ρ to use to evaluate the final state of the trajectory.

`deformTrajectory` Any of the methods described in Section 2.2.4 or in Section 5.2.4 can be used to refine the trajectory being followed and increase the accuracy at the target while decreasing the cost of the trajectory. The same pair formed by the trajectory fitness function $\psi(\tau)$ and the heuristic cost-to-go, used for the previous `selectBestTrajectory` should be used also for this function.

`executeTrajectory` The current trajectory (actually, the first *trajectory arc* in the chosen trajectory) is executed by the system. As we detail below, the trajectory following is integrated by a feedback controller, in order to deal with external disturbances, uncertainties and modeling errors.

`stateReached` This function checks if the tree has reached the goal, it is used to implement the RRT-Connect behavior, i.e., if `params.connectNode` is true, the tree is extended to x_{rand} until it is reached or it is not possible to do any further extension towards it.

5.2.1 Interleaved planning and execution

In their basic form, RRT-based algorithms do not take into account the issues that can arise during the execution phase. Planning is thus an off-line phase the plan execution is a blind process that does not consider possible changes in the environment or uncertainty in the effects of the control commands. Moreover, relevant information for building the trajectory tree can be discovered while the robot moves in the environment, i.e., executes a (partial) plan.

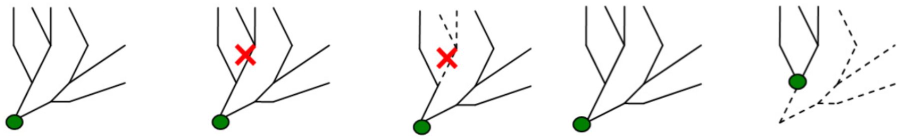
In our method, we interleave tree building and execution of one trajectory in the tree. Once the trajectory execution is started, the algorithm keeps on growing the tree, thus allowing for incrementally building the tree when new information about the environment is discovered.

5.2.2 Feedback control

Actuation uncertainty reduces the reliability of the function σ, that is used to predict the state of the system along the trajectory execution. This means that the open-loop control encoded in the trajectory arc could move the robot in a state that is different from the predicted state. In order to overcome this problem, we correct the control action of the trajectory arcs, using a feedback controller. The specific controller used by our DTT algorithm makes use of the *dynamic feedback linearization* (Luca et al, 2001).

5.2.3 On-line pruning

The plan being executed can become invalid if a collision is found along its path. Inaccurate steering and dynamic environments can invalidate the pre-computed collision-free path on the tree. Since the plans whose trajectories collide with some obstacles cannot be executed, the trajectory arc that results in a collision is pruned from the tree, together with all its children.



(a) Pruning colliding branches

(b) Pruning discarded branches

Figure 5.3: The two kinds of pruning that are performed in the DTT algorithm. The green circle is the current position of the robot in the tree, the red cross is a collision detected in one of the branches.

Since the robot is executing a plan on the tree, we can also prune all those branches that have not been taken in the execution (i.e., after each choice, we can delete the discarded branches). Figure 5.3 shows the two types of pruning.

5.2.4 Trajectory deformation

Despite of the rapid exploration of the state space in RRT-based approaches, its randomized nature and the lack of relevant distance functions require the use of some tolerance distance for the stopping criterion. A small tolerance is needed in those tasks where an accurate positioning of the robot is requested, but a small tolerance distance yields the creation of huge trees before reaching the goal. A solution to this problem is to work in two phases: in the first, a large tolerance distance is used, that reduces the size of the tree needed to reach the proximity of the goal; in the second, the solution trajectory computed on the tree is iteratively deformed in order to increase the accuracy at the goal.

As we described in Section 5.1, in the DTT, a **trajectory** is an ordered list of trajectory arcs, each of which encodes a control action $\tilde{u}(t)$ over a specified time interval and the state in the phase space (i.e., robot pose and velocities) that is the predicted result of this control action. Many trajectory deformation methods (for example those seen in Section 2.2.4) work in the workspace or in the configuration space, and correct unfeasible motions (e.g., that do not enforce kinematic or dynamic constraints) in a second step. On the contrary, we apply perturbations directly on the control actions u: in particular, since we are assuming only a constant-acceleration motion on a single trajectory arc, we apply the perturbations on the accelerations. The benefit of this method is that even large perturbations produce feasible trajectories by construction, but at the cost of re-integrating the controls along all the trajectory at each deformation step.

Furthermore, the exact trajectory deformation method can be chosen among those that allow to apply perturbations directly over the control inputs, for example those in Cheng et al (2003) or in F. Lamiraux and Lefebvre (2004), that we already detailed in Section 2.2.4. In the following of this section we propose two alternative methods for trajectory deformation and we discuss the differences with those presented earlier.

The first algorithm for trajectory optimization exploits the *hill-climbing* method for non-linear optimization. In particular, at each iteration, we generate a set of alternative trajectories by randomly perturbing the control inputs along the trajectory arcs of the chosen trajectory. For each of these candidate trajectories, we compute its utility value (or cost), given by a combination of the trajectory fitness function $\psi(\tau)$ and the goal fitness function $\phi(x)$, as defined in Chapter 4. In particular, if $\tau(x)$ is the candidate trajectory, and

$$x_G = \operatorname*{argmin}_{x(t) \in \tau \cap \mathcal{G}} t, \tag{5.2}$$

i.e., the first state x in the trajectory (given an time-based order in the set of the states in the trajectory τ) that belongs to the goal set $\mathcal{G}$, the utility v is defined as:

$$v = \alpha_\phi \phi(x_G) + \alpha_\psi \psi(\tau(x)). \tag{5.3}$$

The candidate trajectory with the highest utility value is thus selected as the new trajectory.

The hill-climbing method has the following features:

- it relies solely on randomization, i.e., no model knowledge and no information about the utility function (e.g., the gradient) is taken into account: in some cases this requires the computation of a large number of trajectories;
- the new trajectory is guaranteed to have a higher value of utility: the algorithm always improves the trajectory fitness;
- there is one parameter to be tuned: the amount of random perturbation applied to each input (i.e., the size of the interval for a uniform noise, the variance for a Gaussian noise, etc.).

The second method used for trajectory optimization is based on the *gradient ascent*, i.e., the deformation of the parametrized curve (the parameters are the control perturbations along the trajectory) follows the direction of the gradient of the utility in the parameter space. Different methods can be employed to compute the gradient of the utility function with respect to the control perturbations. For example, a well known method is that of the *finite differences*, that is one of the oldest approaches, originated from the stochastic simulation community and quite straightforward to implement. Given the (discretized) controls along the trajectory $\tilde{u} = \{u_0, u_1, \ldots, u_T\}$, a set of K control variations $\Delta\tilde{u}_i, i \in \{0, \ldots, K\}$ is generated. For each variation $\Delta\tilde{u}_i = \{\delta_0, \delta_1, \ldots, \delta_T\}$, the trajectory is integrated and the utility is computed. The gradient of the utility function can be thus obtained by regression.

In gradient ascent methods, at each iteration h, the parameters (in our case, the control inputs) are updated according to the gradient update rule:

$$\tilde{u}^{(h)} = \tilde{u}^{(h-1)} + \eta^{(h)} \hat{\nabla}^{(h)} v, \tag{5.4}$$

where $\eta^{(h)} \in \mathbb{R}^+$ is a learning rate (that can change at each iteration h) and $\hat{\nabla}^{(h)} v$ is the gradient that has been estimated by regression at iteration h. The choice of the learning rate η is critical, since it can prevent the algorithm from converging or take an excessive time to finish; for this reason, we introduce a learning heuristic that has been proved to be very effective in gradient descend methods, although it has been used mostly in neural network training: Rprop (Riedmiller and Braun, 1993). To overcome the inherent disadvantages of pure gradient descent algorithms, Rprop updates are influenced only by the behavior of the sign of the gradient, rather than by its absolute value.

In detail, the Rprop method updates *each* control u_t in the trajectory by the following rule:

$$u_t^{(h)} = u_t^{(h-1)} + \Delta_t^{(h)}, \tag{5.5}$$

where the individual update value $\Delta_t^{(h)}$ is determined, at each iteration, by the rules explained below. We denote with $\hat{\nabla}_t v$ the t-th component of the gradient, i.e., the partial derivative of the utility function with respect to the t-th control input. The absolute value of the update is given by:

$$\|\Delta_t^{(h)}\| = \begin{cases} \eta^+ \cdot \|\Delta_t^{(h-1)}\| & \text{if } \hat{\nabla}_t^{(h)} v \cdot \hat{\nabla}_t^{(h-1)} v > 0 \\ \eta^- \cdot \|\Delta_t^{(h-1)}\| & \text{if } \hat{\nabla}_t^{(h)} v \cdot \hat{\nabla}_t^{(h-1)} v < 0 \\ \|\Delta_t^{(h-1)}\| & \text{otherwise,} \end{cases} \tag{5.6}$$

$$\text{where } \; 0 < \eta^- < 1 < \eta^+, \tag{5.7}$$

where η^+ and η^- are parameters, that are very easy to tune. In fact, it has been shown by empirical experiments (Riedmiller, 1994) that they can be set to 1.2 and 0.5 respectively, for all applications. The sign of the update is then determined as follows:

$$\text{sign}(\Delta_t^{(h)}) = \begin{cases} \text{positive} & \text{if } \hat{\nabla}_t^{(h)} v > 0 \\ \text{negative} & \text{if } \hat{\nabla}_t^{(h)} v < 0. \end{cases} \tag{5.8}$$

In the (very rare) case in which the component t-th of the current gradient is equal to 0, the corresponding control is not updated.

In Figure **??**, a comparison of the performance of the two methods is given.

The main advantage of the two methods presented in this section is that they can be used also when it is not possible to derive the gradient of the utility function with respect to the robot state. If the derivative $\frac{\partial v}{\partial x}$ can be computed in analytical closed form, other methods perform better, since they can exploit an exact gradient, rather than an estimation (see, e.g., the aforementioned methods in Cheng et al (2003) or in F. Lamiraux and Lefebvre (2004)). However, it is still possible to use Rprop to update the control actions along the trajectories.

5.2.5 DTT in dynamic environments

The DTT algorithm executes a trajectory deformation (in the `deformTrajectory` function), a trajectory check (in the `pruneCollidingArcs` function, that performs a collision detection check along the current trajectory) and a tree growing procedure at each iteration. In a dynamic environment, when an obstacle is moving towards the current trajectory, two situations may happen:

- the trajectory deformation process is fast enough to correct the trajectory in such a way that it does not result in a collision;
- the moving obstacles crosses the planned trajectory and the `pruneCollidingArcs` function removes at least the colliding edge; this situation can happen also if the trajectory becomes topologically invalid (e.g., when a door is closed).

The tree growing process can produce a better trajectory (first situation) or eventually produces a new trajectory to be followed (second situation).

5.3 The Dynamic Behavior Tree (DBT) algorithm: integrating sensor-based behaviors into the planner

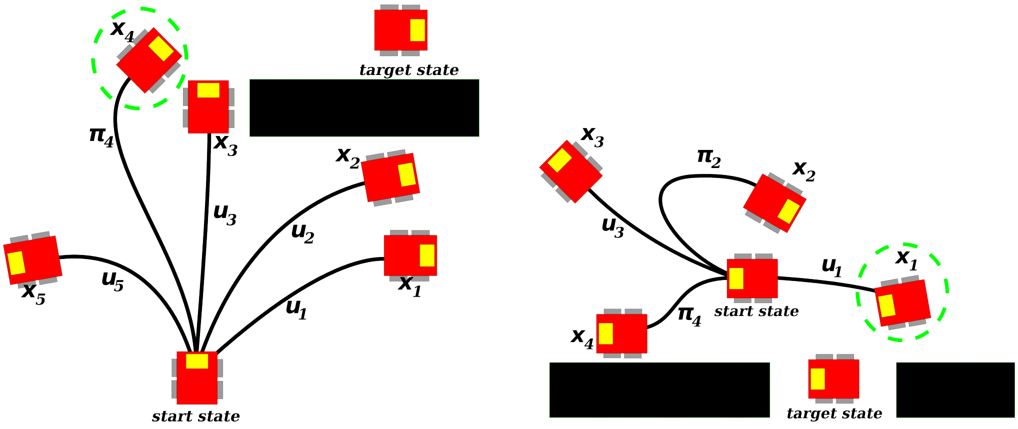
In this section, we show that it is possible to integrate a sensor-based behavior into the DTT planner, using different methods. This extended version of the algorithm is called **Dynamic Behavior Tree** (**DBT**). These methods indeed provide a bridge between the pure-reactive behaviors and the deliberative planners. The major requirement of a reactive behavior to be integrated in a planner, using the methods described in this section, is that the state of the system (i.e., pose and velocities) can be estimated up to a specified time horizon (possibly with some degree of uncertainty). Since the output of the behavior are control actions (e.g., velocities or accelerations) and a model of the robot dynamics (or an proper approximation) is usually available (in the form $\dot{x} = f(x, u)$), this requirement is not demanding.

In the following, we denote with $\pi(x; \mathcal{G}, \mathcal{E})$ or $\pi^{\mathcal{G},\mathcal{E}}(x)$ a generic (feedback) **behavior** that, given the goal set $\mathcal{G}$, the environment description $\mathcal{E}$ and a robot state x, computes a control command u that steers the robot to reach a pose in the goal set. Often (but this is not a requirement), the behavior π takes advantage of the environment description to avoid obstacles. As usual, the function $\sigma(x, \pi, \Delta t)$ computes (or estimates) the state of the robot, if it follows the behavior policy π, from a given state x and for a specified time Δt.

All reactive methods and obstacle avoidance techniques revised in Chapter 2 can be encompassed in this definition of behavior. In the following we introduce the features of the DBT algorithm, that is based on the previously described DTT algorithm and is realized by extending the `extendNode` function.

5.3.1 Behaviors as advices for the `extendNode` function

The first integration we propose is to include the output of the behavior π as one of the possible commands in the `extendNode` function (see the basic RRT algorithm shown in Figure 2.2). Since obstacle avoidance techniques are designed to solve a



(a) a situation in which a behavior, π_4 is chosen

(b) a situation in which the behaviors perform poorly

Figure 5.4: The behavior included as advices in the `extendNode` procedure of the tree building process.

large amount of common situations, it is likely that the candidate $\bar{x}$ produced by the integration $\bar{x} = \sigma(x, \pi^{\mathcal{G},\mathcal{E}}, \Delta t)$ is chosen as x_{new} and added to the tree.

Typically, obstacle avoidance techniques are suitable for a particular set of situations or introduce additional constraints on the system: for example, there can be some local minima in their behavior, or they can be designed for forward-only maneuvers.

Figure 5.4 shows the `extendNode` procedure in two particular situations, where a set of behaviors π_i has been added for the procedure itself. In the first case, the control action resulting from a behavior π_4 is chosen and its resulting trajectory arc (in which $x_4 = \sigma(x, \pi^{\mathcal{G},\mathcal{E}}, \Delta t)$) is added to the tree. In the second situation, the behaviors π_2 and π_4 perform poorly, and a random control u_1 (that, during execution, will be corrected by the feedback controller) is chosen instead.

5.3.2 The variable horizon: bridging pure-reactive methods and planners

In this subsection we discuss a possible point of view that allows to consider both pure-reactive behaviors and motion planners in the same comprehensive class. The major apparent difference between pure-reactive behaviors and deliberative methods is that the former neither explicitly account for the effects of their actions in the future, nor plan actions other than the current one. We can thus consider that their planning horizon is 1 actions in the future.

The problem that arises from this limited time horizon is that pure-reactive methods only consider the immediate effects of their selected trajectories, without verifying their later consequences. This insight led to the development of the

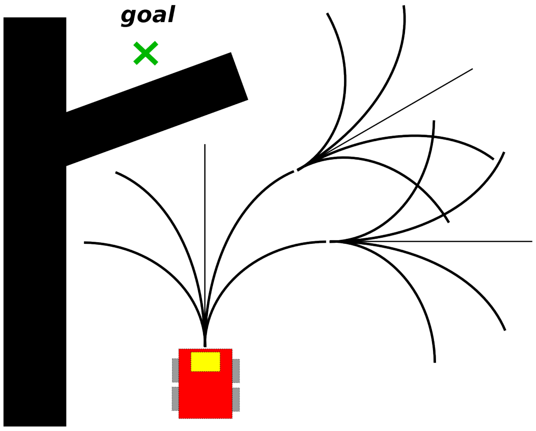


Figure 5.5: An execution of the VFH* algorithm

VFH* algorithm (Ulrich and Borenstein, 2000), as an improvement of VFH+ (that has been already described in Section 2.1.2).

Rather than introducing a global path-planner, VFH* adds some limited look-ahead to the VFH+. In particular, as shown in Figure 5.5, starting from the robot position, a set of candidate directions is computed following the canonical VFH+ algorithm. For each of these directions, the projected robot pose, at a fixed step distance d_s, is computed and the VFH+ algorithm is applied to it. The process is repeated n_g times, building a tree of trajectories of depth n_g. In fact, an A* algorithm is used to choose which node of the tree to expand, in order to reduce the computation. The VFH+ algorithm can thus be seen as a particular case of the VFH* in which the look-ahead d_s is set to 1. The direction to be followed is then chosen by using a cost function similar to that of VFH+, integrated along the branches leading back to the start node.

In next section, we show how this insight can be exploited to include any pure-reactive behavior, such as the Dynamic Window Approach and the Nearness Diagram.

5.3.3 Feedback behaviors as trajectory arcs

In Section 5.2.2, we stated that it is possible to increase the reliability of the execution, by adding a feedback controller, that corrects the errors (or some hidden variables) of the system model. The feedback controller is a function of the current expected state $\hat{x} = \sigma(x, \tilde{u}, t)$. We can couple the control trajectory and the feedback controller in a function $\zeta^{\tilde{u}}(t)$ that outputs a corrected control action u^*. The trajectory arc is thus a tuple $< t_{start}, t_{end}, x_{start}, \zeta^{\tilde{u}} >$ (compare this with the previous definition, given in Section 5.1). We can further extend this definition, by allowing any (possibly closed-loop) behavior to be introduced in the trajectory arc, in place of ζ. A trajectory arc is thus a tuple:

$$< t_{start}, t_{end}, x_{start}, \text{trajectory-behavior} >, \tag{5.9}$$

where:

$$\text{trajectory-behavior} = \begin{cases} \zeta^{\tilde{u}} & \text{i.e., a feedback corrected trajectory} \\ & \text{in the control space} \\ \pi^{\mathcal{G},\mathcal{E}} & \text{i.e., a (closed-loop) behavior} \\ & \text{that steers the robot to } \mathcal{G}\text{, possibly} \\ & \text{avoiding obstacles in } \mathcal{E} \end{cases} \tag{5.10}$$

The tree building process can be thus straightforwardly extended to include this new definition of trajectory arcs. The `extendNode` procedure can generate both (feedback) trajectories $\tilde{u}$ and behaviors $\pi^{\mathcal{G},\mathcal{E}}$ (that means to choose among a set of pre-defined behaviors π_i, with $\mathcal{G} = \{x_{rand}\}$).

This method can be also seen as a generalization of Frazzoli et al (2002), where a single optimal policy, that is defined for free-space motions, is used. The tree is thus grown using branches that encode the parameters of the policy, i.e., the goal state. Moreover, branches are extended until they reach the generated x_{rand} (as in the `connectNode` function of the RRT-Connect algorithm, i.e., by setting `params.connectNode` in the algorithm of Figure 5.2) or they result in a collision. Finally, goal states are always stable equilibrium states, although the algorithm allow to create secondary goal states along the trajectories towards primary goal states.

The main differences with the method described in this section is that we consider a set of behaviors, instead of a single policy, that we generate random states that are not forced to be equilibrium states and, finally, that the behaviors themselves are not forced to be effective only in free spaces, but can be complex obstacle avoidance techniques.

A fixed horizon is set in order to reduce the computation and allow for on-line applicability, as in Urmson (2002): trajectory arcs beyond a specified time horizon cannot be extended by the algorithm.

The main drawback of introducing this kind of complex behavior into the tree based planner is the computational overhead required by obstacle avoidance techniques with respect to open-loop control commands or optimal policies defined over free spaces. However, the benefits of the overall system, demonstrated in Chapter 9, alleviate this shortcoming: this is basically due to the fact that obstacle avoidance behaviors are able to steer the robot to the local goal, while avoiding obstacles, by themselves. Moreover, at any time it is possible to reduce the planning horizon and seamlessly transform the motion system into a pure-reactive behavior.

A further remark should be made about the difference between the time horizon (look-ahead) used by the reactive method and the planning horizon of the branches of the tree (the duration of the trajectory arcs). The first regards the estimation of the effects of a given action that is used by the reactive technique to choose only the current action (e.g., many methods compute the effects of keeping the control command for a fixed time interval): no plan is actually neither built, nor executed. The second concerns an actual plan that is built (possibly incrementally), it is kept through iterations and it is executed (with feedback control, trajectory deformation, collision checks along the trajectory, etc.).

5.4 Automatic parameter tuning by means of machine learning techniques

The algorithm introduced in this chapter includes almost all the possible variations of both motion planners and reactive techniques that have been presented in this chapter. However, it introduces a large set of parameters to be tuned, some of which are related to the choice of which behavior/control law to be used. Moreover, the parameter set of the system strongly depends on the specific situation, or context, in which the robot acts, and on the particular task it is performing.

In general, when designing autonomous multi-purpose robots, it is necessary to tackle some inherent bottlenecks (see Thrun, 1996), the most critical ones are, in our view, the knowledge needed to design a successful robot controller, that can be hard or impossible to obtain (e.g., robot dynamics, some characteristic of the robot sensors, or the configuration of the environment where the robot will perform its tasks); or the fact that as robotic hardware becomes more and more complex, and robots are to become more reactive in more complex and less predictable environments, the task of hand-coding a robot controller will become more and more a cost-dominating factor in the design of robots.

Machine learning aims to overcome these limitations, by enabling the robot to collect its knowledge on-the-fly, through real world or simulated experimentation. A specific area of machine learning applied to robotics, that is suitable to automatic parameter tuning is that of the **Reinforcement Learning** (**RL**).

5.4.1 Reinforcement Learning (RL) and Policy Gradient (PG) methods

The Reinforcement Learning framework is used to solve problems that can be described as follows: an agent (a robot in our case) is supposed to find an optimal behavioral strategy while perceiving only limited feedback from the environment. The agent receives information on the current state of the environment, can take actions, which may change the state of the environment, and receives reward or punishment signals, which reflect how appropriate the agent's behavior has been in the past. The goal of RL is to find a behavioral policy that maximizes the long-term reward (see Heinrich-Meisner et al, 2007, for an introduction to Reinforcement Learning in robotics).

RL is a very wide and active area in Machine Learning, and many different methods have been proposed in order to learn the optimal policy as defined in the previous paragraph. One of the most successful methods, especially suitable for robotic applications, is the **Policy Gradient** (**PG**) method. In PG, the form of the policy $\pi_\lambda(x)$ (where x is the current perceived state of the robot and/or the environment) is assumed to be given and a set of parameters $\lambda \in \mathbb{R}^K$, that influences its behavior, must be tuned in order to increase its performance, i.e., the expected reward J.

The PG method, assuming the reward J as a function of the policy parameters λ, updates λ, at each learning iteration, by following the gradient update rule:

$$\lambda^{(h+1)} = \lambda^{(h)} + \eta^{(h)} \cdot \nabla_\lambda J, \tag{5.11}$$

where $\lambda^{(h)}$ is the value of the parameters at learning iteration h, η is a learning rate, and $\nabla_\lambda J$ is the gradient of the function $J(\lambda)$ with respect to λ.

Policy Gradient methods are suitable for robot learning, and in particular for motion system tuning, for many reasons. For example, the straightforward specification of the policy representation can allow to incorporate previous domain knowledge, that can speed up the learning process. The main problem in Policy Gradient methods is to obtain a good estimator of the policy gradient. In the next subsection we revise one of these methods, that has been used in our system to optimize the parameters of the two algorithms that we presented in this Chapter.

5.4.2 Applying PG methods to tune DTT and DBT parameters

When the goal is to optimize both a set of different behaviors, and a composition of the same behaviors, the parameter space can be very large and learning/optimization techniques can converge too slowly. In these cases, the *layered learning* (Stone and Veloso, 2000) approach can be exploited. In the layered learning approach, the behaviors are first optimized by their own, singularly. Afterwards, their composition (i.e., the parameters that control their composition) is optimized as well. In our case, the different behaviors to be tuned are the obstacle avoidance techniques (i.e., Nearness Diagram, Dynamic Window Approach, etc.) that have been introduced in the DBT algorithm. Afterwards, the parameters that control the DBT algorithm itself (i.e., the parameters that determine the growing of the tree, such as the number of trajectory arcs to be added at each iteration, etc.) are learned as well.

In our system, in order to learn each behavior parameter, as well as the parameters that control the DBT growing procedure, we use the method presented in Kohl and Stone (2004) to estimate the gradient of the reward function with respect to the parameters, and the Rprop method (Riedmiller and Braun, 1993) to update the parameters at each learning iteration. All the systems that are tested and compared in Chapter 9 are optimized by means of this method.

The algorithms for local motion presented in this chapter have been tested and compared using a framework for motion system evaluation, that is introduced in

Chapter 8. The results, presented in Chapter 9 show that this adaptive system, making use of both reactive behaviors and randomized trajectory trees, is able to solve efficiently different problems. In particular, it selects fast reactive obstacle avoidance techniques when they are suitable for the scenario, and smoothly switch to randomized trajectory tree algorithms when the situation requires a longer look-ahead or complex maneuvers not considered in the pure-reactive techniques.

6

A compact topological representation for autonomous navigation

In this chapter we develop a global method to be included in the motion system, that drives the local-level subsystem, described in Chapter 5, through a local-minima-free course, in order to avoid the limitations of a pure-local-based approach, as described in Section 1.2.2 and in the beginning of Chapter 3.

After dealing with the relevant information to be exchanged between the two subsystems, we detail our algorithm in two phases: first, we define a procedure to construct a roadmap of the environment; afterwards, we transform the roadmap into a hybrid topological/geometrical representation, that improves the reliability of the navigation. Finally, we explain how to use this representation to define the goals for the motion system and how it can be augmented with critical information for the local subsystem, in order to increase its reliability and efficiency.

The algorithm presented in Sections 6.2 and 6.3 uses concepts from computational geometry, that have been deeply investigated in the past (see, e.g., Zwynsvorde et al, 2000, 2001)). Our algorithm does not introduce novelties in the building process, but the generated representation forms the basis for the subsequent addition of high-level information, context knowledge and the definition of the goals (Section 6.3.2, Section 6.3.3 and the following Chapter 7).

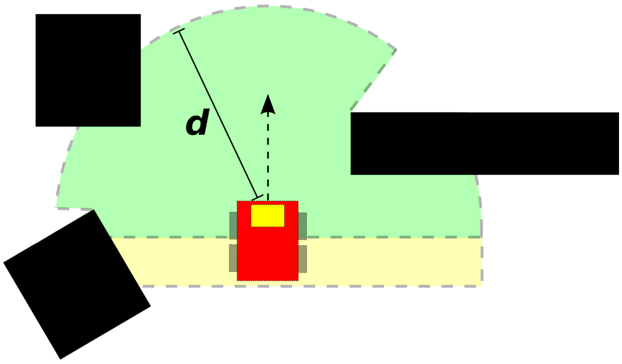


Figure 6.1: The Local Spanned Area (LSA), the portion of the environment currently sensed by the robot.

6.1 The problem of the local goal

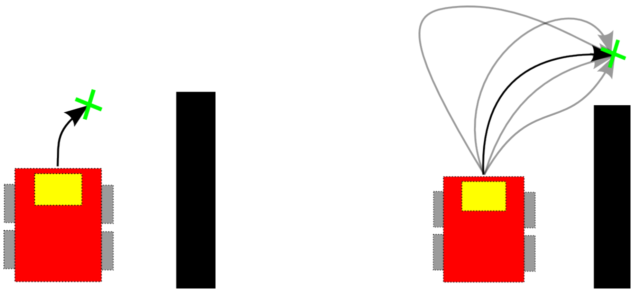
As we stated in Section 1.2.2, the key information that is exchanged between the global level planner and the local subsystem is a *local goal*. In our work, the definition of the local goal is extended using the functions and the criteria introduced in Chapter 4. However, there are some inherent problems in the global/local problem decoupling. In this section we address some issues regarding the selection of the local goal and we discuss the feasibility of this approach, as well as possible solutions to its limitations.

6.1.1 Selection of the local goal

Local algorithms mainly expect two forms of local goals: either a goal *direction* or a goal *pose*. The former is often used by pure-reactive methods. In this chapter we assume that the global subsystem is able to output a goal pose, since it contains more information than the direction (e.g., the orientation at the goal) and, moreover, because a goal pose is more general, i.e., given a local goal pose, it is always possible to compute the corresponding local direction for the local subsystem. Finally, we consider goal *states*, augmenting the pose with velocities. Finally, as described in Chapter 4, we allow the goal definition to include fuzzy values, by exploiting the goal fitness function $\phi(x)$, or unspecified values (e.g, the orientation is not important, the speed is unspecified, etc.).

The local subsystem has access to a local subset of the environment representation. This is mainly motivated by the fact that there is a high uncertainty in the world that is not currently sensed by the robot: trajectories should be generated where information is available (Urmson, 2002).

In Figure 6.1, we show an example of the **Local Sensed Area** (**LSA**), i.e., the



(a) The local goal is too close to the robot

(b) If the local goal is given at a greater distance, the local subsystem has more freedom

Figure 6.2: The effect of the distance at which the local goal is given to the local subsystem.

part of the environment that is currently sensed by the robot, where, as an example, we consider a set of sensors that are able to detect the environment up to a maximum distance d and within 180 degrees of field of view. We can slightly extend the definition of the LSA, including also the portion of the environment that has been sensed within some (short) interval of time in the past (shown in yellow in the figure). For the reasons explained above, it is preferable that the local goal is within the LSA, in order to avoid to consider trajectories where we do not have information or where the knowledge of the environment can be inaccurate or the environment can be changed since the time it was sensed.

However, if the local goal is chosen too close to the robot, this allows a very limited look-ahead of the local subsystem, and, in addition, it limits the freedom to generate better local trajectories. See Figure 6.2 for an example. The logical consequence is that the best placement for the local goal is as far as possible, but within the LSA. This definition coincides with the frontiers of the LSA. In the following of this chapter, therefore, we describe how to choose the local goal on one of the current frontiers.

6.1.2 Inherent problems of the global/local decomposition

In the following, for explanation purposes, we assume that a global optimal deterministic path planner (GODPP) is available. Given a global goal specification, the GODPP is able to compute the trajectory τ^{opt} that is optimal in a global sense, given any state x in the state space $\mathcal{X}$. Furthermore, the GODPP is able to compute, at any instant, the best current control action to be sent to the robot, i.e., it can compute a *navigation function* (see, e.g., Latombe, 1991; Rimon and Koditschek, 1992). Since the trajectory is optimal, its cost is minimum: $cost(\tau^{opt}) \leq cost(\tau), \forall \tau \neq \tau^{opt}$, i.e., any modification to the control actions from the navigation function induced

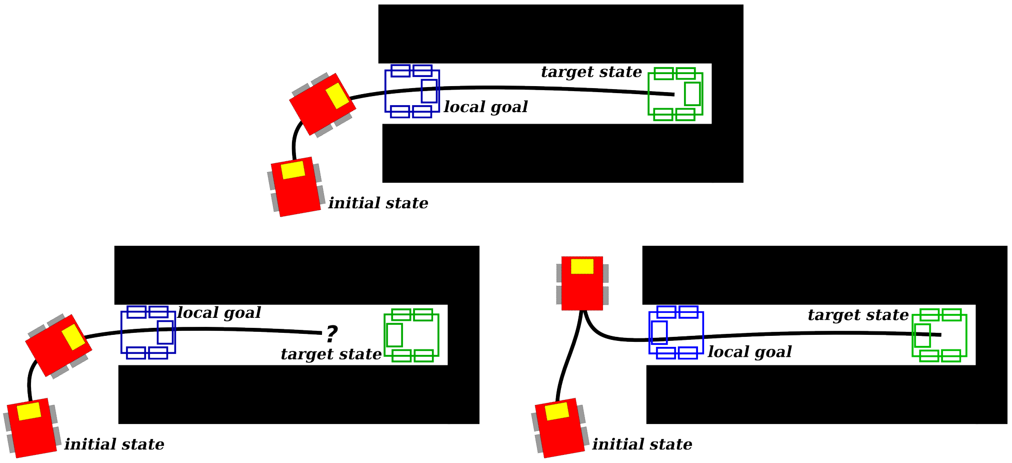


Figure 6.3: Two situations in which the local goal guides the local subsystem towards the goal. The global state is shown as an outlined green robot. The robot starts its trajectory in the position marked with "initial state" and receives the local goal, depicted as a blue outlined robot. In the first picture, the local goal is appropriate to successfully accomplish the task. In the second picture, on the contrary, the local goal is a wrong choice, because the robot, once entered the corridor, cannot turn in order to match the requested orientation. The right local goal is the one shown in the third picture.

by the GODPP, results in a higher (or equal) cost trajectory.

When a global/local decomposition is exploited, there are two issues that can cause a deviation from the optimal trajectory: the local goal and the local subsystem.

Regarding the latter, given a local goal x_g that lies on the optimal trajectory, $x_g \in \tau^{opt}$, we can assume that the local subsystem, with an appropriate look-ahead and a good feedback control, will follow the optimal trajectory, because it is provided with all the information it needs to compute the optimal trajectory up to x_g (i.e., the surrounding environment and the local goal).

The local goal is thus the most critical issue in the global/local decomposition: the global subsystem should be able to provide a local goal that lies on the global optimal trajectory τ^{opt}. Unfortunately, this is as difficult as the original problem: we cannot rely on a GODPP.

Figure 6.3 shows that a wrong local goal can affect not only the optimality of the trajectory, but also the final success of the task. The scenario requires the robot to reach a goal pose at the end of a narrow corridor, where no rotation is allowed. This means that, once the robot has entered the corridor, it can only go forward or backward, without any possibility to turn. Figure 6.3(a) shows a situation in which the method given in Section 6.1.1 yields the appropriate local goal to the local subsystem. On the contrary, if the global goal is the one shown in Figure 6.3(b), the

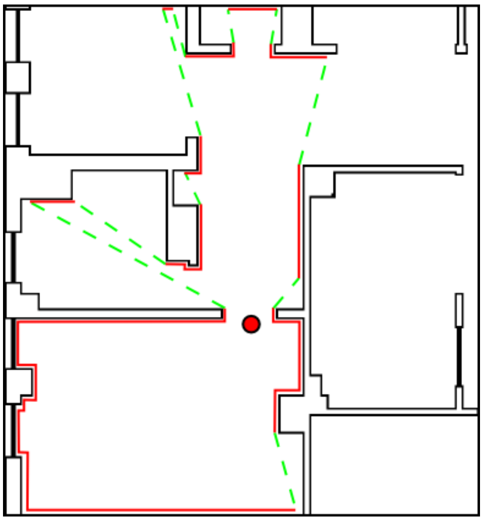


Figure 6.4: The polygonal representation of the local environment, given by the virtual range finder. Red segments are obstacle segments, while green dashed segments are frontiers. The red circle is the virtual robot position.

goal-on-frontier method gives the wrong local target. The robot has entered the corridor with the wrong orientation, thus, when the global goal enters its LSA, it is not able to turn and reach the global goal with the right orientation: this, depending on the length of the corridor and the size of the LSA, results in an oscillating behavior or a complete failure. The correct local goal should be that depicted in Figure 6.3(c).

In a global/local decomposition, in order to address this problem, one needs to allow the global subsystem to produce the "right" local goal, without the need of computing the complete trajectory towards the global goal (that, as we stated above, is as hard as the complete problem). A possible solution to this issue is given in Section 6.3.3 and is based on the possibility to attach critical information regarding the local goals directly to the global representation.

6.2 A simple roadmap built using virtual frontier-based exploration

In this section, we propose a novel algorithm to build a roadmap of the environment. A roadmap is a graph, built on a predefined map, that captures the topological connectivity of the environment, in such a way that a global subsystem can exploit this roadmap in order to compute the local goal to give to the local subsystem. The algorithms is simple to implement and allows to be straightforwardly extended to address 3D environments.

In Sections 3.3 and 3.4, we showed that a roadmap can be built upon a grid or a line map, in order to allow the global subsystem to output local goals. There are many methods to construct a roadmap that encodes the topological structure of an environment. In this section we introduce a new method that is based on the concept of virtual frontier-based exploration. In the following, we consider a 2D world, whose representation is available as a grid or a line map. The presented technique is similar to the exploration algorithm presented in Freda and Oriolo (2005), although in our work the technique is used as a virtual tool to build the roadmap. Additionally, we are assuming that a representation of the environment is already available: the construction of an environment representation that is topologically and/or metrically accurate is not among the topics of this thesis and we take it for granted; similarly, the choice of the sequence of positions that the robot should reach in order to build such a representation is not taken into account in this thesis, although some example methods for (actual) frontier-based exploration are presented in Appendix B, as an application of the motion systems.

6.2.1 Building the roadmap

A range finder sensor constructs a polygonal representation of the environment surrounding the robot. We can thus define a virtual range finder that is able to give this local representation, given the global map and a pose for the virtual range finder. Figure 6.4 shows an example of a polygonal representation of the local environment, given by the virtual sensor (in practice, a ray-tracing method can be used to simulate the virtual range finder). A line fitting method can be used to transform the virtual sensor readings (i.e., points) into a set of segments. These segments can be subdivided into two classes: obstacle segments (depicted in red in the figure) and frontier segments (depicted as dashed green segments). The latter are the boundaries between the sensed and the unsensed area. The virtual exploration given in this section is strictly related to the (actual) frontier-based exploration introduced in Yamauchi (1997).

The algorithm described in this section is shown in Figure 6.6: it virtually "explores" the environment representation $\mathcal{E}$ using a depth-first-like search. Figure 6.5 shows a possible execution instance, in which the algorithm is exploited to build the roadmap of an indoor environment.

The main data structure used, called `roadmap` in the figure, is a graph where nodes can be *places* or *frontiers*, and links are allowed only between a place and a frontier. The functions `insertPlace` and `insertLink` are used to interact with this data structure. Moreover, a stack, called `openFrontiers`, is used for the depth-first search.

The algorithm works as follows. Starting from a given position in the environment, q_{init}, marked with 1 in Figure 6.5, a first "snapshot" is taken by means of the `virtualSensor` function. The resulting spanned area is attached to q_{init} and inserted into the roadmap. Additionally, links are inserted between q_{init} and the frontiers in the current virtual sensed area. Finally, the new frontiers are pushed into the stack to be explored in next iterations.

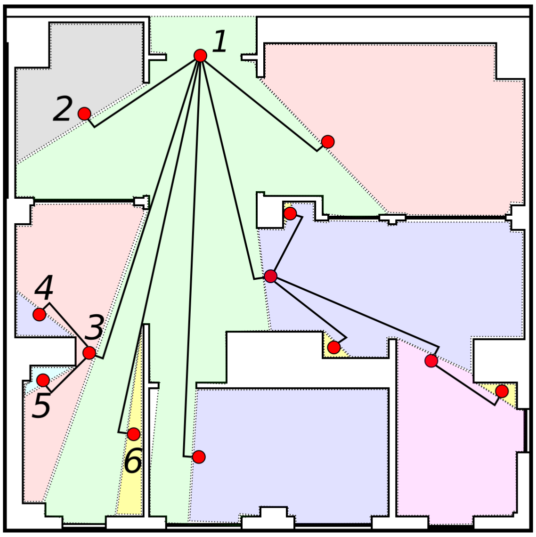


Figure 6.5: The tree of local polygonal representations, that can be used as a topological representation of the environment. The red circles are the virtual positions from which the virtual range finder is used.

```
function virtualSensorRoadmap($\mathcal{E}$, $q_{init}$) \rightarrow roadmap {
    // $\mathcal{E}$ is a grid or line map
    // $q_{init}$ is an initial pose in the environment
    roadmap.insertPlace($q_{init}$, vsa)
    localMap = virtualSensor($q_{init}$)
    foreach (f in localMap.frontiers) {
        roadmap.insertLink($q_{init}$, f)
    }
    openFrontiers.push(vsa.frontiers)

    while (not openFrontiers.empty()) {
        currentFrontier = openFrontiers.pop()
        q = placeFromFrontier(currentFrontier)
        localMap = virtualSensor(q)
        roadmap.insertPlace(q, vsa)
        roadmap.insertLink(currentFrontier, q)
        foreach (f in localMap.frontiers) {
            roadmap.insertLink(q, f)
        }
        openFrontiers.push(vsa.frontiers)
    }
}
```

Figure 6.6: The algorithm used to compute a roadmap of the environment, using a virtual range finder sensor.

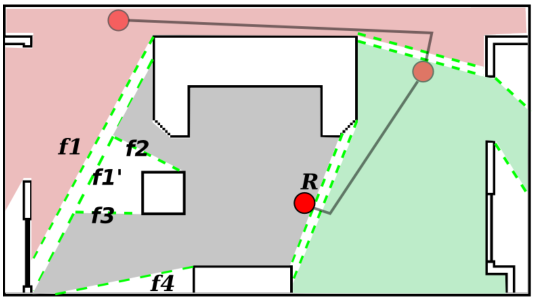


Figure 6.7: The situation in which a loop is found while virtual exploring the environment. The virtual robot is in the position denoted with the red circle, from there, it can see through the frontier denoted with "1" and thus a loop has been detected; the proper frontier denoted with "2", is treated normally.

At each subsequent iteration, one of the frontiers is popped from the stack to be processed: a place is created near the frontier (see, e.g., the place marked with 2 in Figure 6.5) and from this new place another "snapshot" of the environment is taken, pushing the resulting new frontiers into the stack. The whole procedure can be iterated until no frontier remains to be "explored".

It is possible that there is an edge between a place and a frontier, but the frontier cannot be actually reached from the place (e.g., a narrow corridor in which the robot does not fit, with a frontier at the end). In order to overcome these situations, a method such as that presented in Minguez and Montano (2004) should be used to ensure the reachability of a frontier from a place.

In the case of a loop-free environment, we obtain a tree-shaped roadmap, that can be seen as the connectivity graph of the environment, as shown in Figure 6.5: this representation can be used to determine the topological route from any starting configuration to any goal configuration (or set).

In the general case in which loops are present in the environment, we can transform the tree-based structure in a graph-based structure, using the following procedure. When a new local polygonal representation is added to the tree/graph, we check not only that the new area does not overlap with the previous (parent) node, but also that it does not "go through" any other frontier. If this happens, the two local representations are connected with an edge (thus violating the loop-free definition of a tree and making it a graph) and the corresponding frontier is removed from the stack of the open frontiers. A more complex situation is shown in Figure 6.7: in this case, an open frontier, $f1$ in the figure, can be seen only partially from the current LSA. If this happens, the open frontier $f1$ is removed from the stack, but other frontiers are added where the LSA cannot see through it (in the figure, a new frontier is added, marked as $f1'$ in the figure). Other frontiers, determined using the usual criteria, are pushed into the stack ($f2$, $f3$ and $f4$ in the figure).

Finally, it is possible to extend the method described above to a 3D environ-

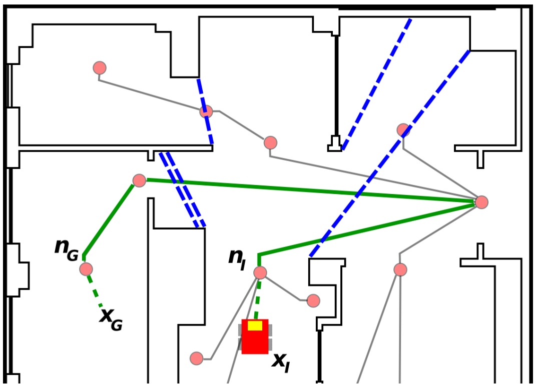


Figure 6.8: The procedure to compute the local goal, given the frontiers in the current LSA (dashed blue lines) and the roadmap (places are red circles and links are gray segments). The current robot pose x_I is connected with the closer node n_I in the roadmap. The same happens with the goal pose x_G, that is connected to its respective closer node n_G in the roadmap. Subsequently, the path is searched in the graph (green bold segments). The chosen frontier, following the criterion explained in the text, is shown as a double blue dashed segment in the figure.

ment. The main differences that must be taken into account regard the fact that detected obstacles have to be translated to polygons and that frontiers, in a 3D environment, are polygons as well. Techniques borrowed by Constructive Solid Geometry (CSG) can be used to perform an incremental virtual exploration of the 3D environment.

6.2.2 Computing the local goal

In order to compute the local goal for the local subsystem, we proceed as follows, as shown in Figure 6.8. We denote with x_I and x_G, as usual, the initial state of the robot and the goal state, respectively. We create a temporary connection between the goal state and the nearest node in the graph n_G, and between the initial state and its respective nearest node n_I. Subsequently, we can compute the path from n_I to n_G, on the graph. The result is a set of edges that encodes the path from the current robot position to the goal.

The local goal is computed as the frontier in the LSA that intersect the path segment that is nearest to the global goal (or that intersect the part of the segment that is nearest to the global state, if the same path segment intersects more than one frontier, as is shown in Figure 6.8).

6.3 From the roadmap to a hybrid topological/geometric representation

The problem with roadmaps is that they rely on an accurate global map of the environment and an accurate and a reliable global localization on that map. Although many SLAM algorithms can build metrically and topologically consistent maps of very large environments, and many localization algorithms are able to track the pose of the robot in these large representations, in this section we give an alternative method that is more reliable, also in the case in which the representation is only topologically correct and the metric accuracy is reliable only in local portions of the map. This is motivated by the fact that the error in metric accuracy increases with the distance only: in the local portion of the map that surrounds the robot, it can be considered low enough to allow for a good localization.

Using a graph of local maps is a well-known technique in localization and SLAM communities. Modern SLAM algorithms exploit this concept using both line-based or grid-based local maps (see, e.g., Grisetti et al, 2006; Burgard et al, 2009). Moreover, topological maps are often augmented with geometric information, that is useful for the generation and execution of navigation plans (see, e.g., Fabrizi and Saffiotti, 2002).

6.3.1 Local map localization

In the roadmap constructed using the algorithm presented in the previous section, we attached a local map to all places in the graph, in order to check for loop-closing events. We can use this local map to localize the robot locally, for example, using a scan-matching approach such as Iterative Closest Point (ICP) (Zhang, 1994).

While the robot is moving in one local map, it can use its features (i.e., corners, walls, etc.) to localize itself inside that local map. However, when passing from one local map to the next, in the course to the global goal, it has to switch between the features/landmarks of the first local map to those in the second local map. We call **gate** the two joint frontiers that connect two local maps. It is possible to avoid a discontinuity in the localization process, using the method described in Taïx et al (2008), that allows to weight the landmark importance (reliability), when solving the localization problem. Landmarks in the second local map are associated with lower weights with respect to those in the first (current) local map and those weights are increased as the robot approaches the gate. In this way, if the two local maps are not aligned with respect to each other, due to an inaccurate mapping process, the robot remains localized in the first local map, and seamlessly considers also the second map, while it approaches the gate.

A final remark about the frontier/gate construction during the virtual frontier-based exploration should be made. Although using directly the frontiers that are returned by the virtual sensor, reduces the number of iterations needed to explore the whole workspace, this can produce local maps with a poor number of landmarks that can be used for localization. Indeed, it is possible to consider a modified

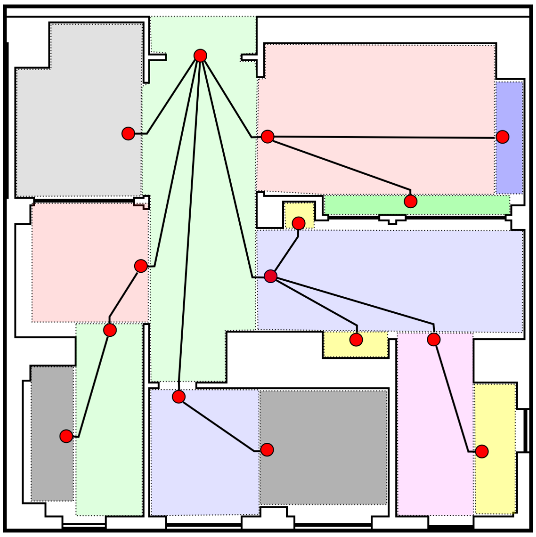


Figure 6.9: The result of the virtual exploration if the polygonal local maps are forced to be convex.

version of the virtual sensor that produces only convex local maps. The resulting representation of the same map seen in Figure 6.5 is given in Figure 6.9. This choice of "places" is also more meaningful for human beings and places can be easily labelled, as shown in Section 6.3.3.

Finally, we can consider the representation given in this section as "topological" in the sense that only *local* metric information is considered at each place, and a set of gates link each place to its neighbours. In this way, after the topological path is computed on the graph of places/links, the motion system has to move to a local feature (the gate, or the current frontier beyond a gate), without the need of any global metric information.

6.3.2 Considerations about the global goal

This graph of local maps can be also used to specify the global goal, i.e., the global task goal should be specified using the pair composed by the local map and the precise goal definition given in the reference frame of the local map. During the motion, two situations can happen, as described below.

- The global goal is *inside* the LSA: in this case, the global goal becomes also the local goal, and their definitions coincide, i.e., the global subsystem does not have to generate an extended local goal definition anymore.
- The global goal is *outside* the LSA: in this case, the global subsystem must generate a local goal (extended) description for the local subsystem. In particular, as we discussed above, this local goal should be generated in the proximity of one of the LSA frontiers.

6.3.3 Adding labels, local goal hints and other information to the representation

Given the possibility to detect topological places inside the LSA, the global subsystem can thus deal with goal descriptions that are given using semantic labels, that are meaningful for a human being, e.g., the global goal can be described as "go in the center of the kitchen" (where "kitchen" is a label attached to one of the local maps).

Moreover, the global subsystem is able to exploit additional information and give it to the local subsystem. In particular, the knowledge about the direction of the environment after a frontier can be given as a hint through the local goal description (e.g., giving a high goal fitness to that specific direction). In this way, although the local subsystem is not directly aware of the area outside the LSA, it can indirectly exploit a knowledge that is beyond its look-ahead time interval. Moreover, it is possible to include other information to avoid the situations described in Section 6.1.2, either by hand or by some supervised learning technique. This additional information forces the global subsystem to give the proper local goal, possibly activating the extra constraints on the local goal only in special cases (e.g., in the narrow corridor scenario described in Section 6.1.2, the "right" local goal at the entrance of the corridor depends on the orientation of the final goal at the end of the corridor).

This hybrid environment representation, given by local geometric maps connected by topological links, is used in the rest of this thesis and in particular in Chapter 9, where a set of experiments are performed in order to evaluate the system in a number of different scenarios. Chapter 9 presents also a set of experiments that compares the use of the hybrid representation described in this chapter to other solutions. It is important to underline that the algorithms given in this chapter could be replaced by any other approach which satisfies the requirement that, given a global goal description, is able to provide a local goal description to the local subsystem.

7

High-level reasoning and context-based adaptation

In this chapter, we introduce the context-based architecture, that, in this thesis, forms the basis for the adaptation of the motion system with respect to specific situations (contexts). The chapter is organized as follows. Section 7.1 introduces context-based robotics and how its concepts are currently exploited to solve many robotics problems. Section 7.2 describes our context-based architecture design, that collect the contextual reasoning in a specific component in a robotic architecture, decoupling it from the rest of the functional components. Some experiments that show the benefits of the use of a context-based architecture are also presented in this section. Finally, Section 7.3 shows that contextual information can be attached to the global environment representation, together with other high-level information, as we shown in Chapter 6.

7.1 Context-based robotics

In Calisi et al (2007a), the "contextual architecture" for the design of robotic applications is introduced. The notion of **context** has been deeply investigated both from a cognitive standpoint and from an AI perspective (see, e.g., McCarthy and Buvač, 1997). In the former case, the study is more focused on the principles that underlie human uses of contextual knowledge, while in the latter case, the main point is how to provide a formal account that enables the construction of actual deductive systems supporting context representation and contextual reasoning.

The interest for context in robotics development is twofold. On the one side,

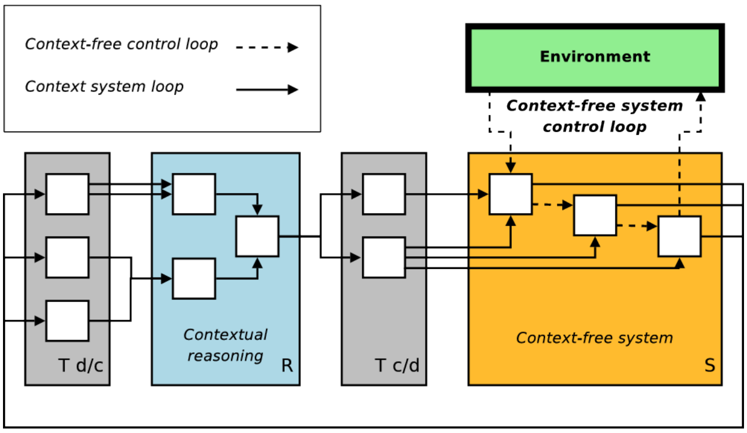


Figure 7.1: The contextual architecture: modules in S are the common functional modules in a robotic application (e.g., mapping, motion system, image processing, etc.), modules in R are responsible of reasoning about the context and tune S modules accordingly, $T_{d/c}$ and $T_{c/d}$ converts numerical data from S modules in a format that is usable by the modules in R, and vice-versa.

the design and implementation of experimental systems that are focused on cognition; on the other side, the need to improve the performance and the scope of applicability of robotic systems by providing them with high-level knowledge and capabilities. Turner (1998) specifically addresses contextual knowledge in robotic applications. He characterizes context as: “any identifiable configuration of environmental, mission-related, and agent-related features”. Such a definition, that we take as the basis of our approach, highlights a relationship with a more recent stream of work on the use of semantic knowledge in robotics (ICRA07, 2007); in particular, contextual knowledge about the environment is strictly related to semantic knowledge.

The use of context in robotics has been proposed in the literature to improve the effectiveness of many different components of a robotic architecture, related to tasks such as exploration, navigation, behaviors, mapping, localization and perception. For example, there are systems that can improve the map construction process, by knowing that the robot is currently moving in the corridor of an office building. However, contextual knowledge is typically not fully exploited, since it is built in each of the system modules. It seems therefore very appropriate, from an engineering perspective, to build and maintain a single representation of the contextual knowledge that can be used to improve many different processes.

Figure 7.1 shows a typical arrangement of a contextual-based architecture. Small squares denote modules or components of the system. The modules enclosed in S are functional modules that are not context-aware, their tasks are those related to the application (e.g., localization, mapping, navigation, etc.). Some of the S modules can produce information that can be used to infer the current context. $T_{d/c}$

modules translate data in a form that is usable by the contextual reasoning modules R, that detect the current context and output a set of behavior modification signals for the S modules, that are translated into parameters or operational modes by the $T_{c/d}$ modules, thus allowing the system to control the robot behavior.

The use of a contextual architecture allows to decouple the contextual reasoning and the functional modules. The contextual knowledge can be subdivided into three categories: mission, environmental and introspective. Mission related knowledge is often specified off-line and has been addressed by several approaches presented in the literature. Consequently, we focus on environmental and introspective knowledge. The first is closely related to semantic knowledge, since typically it aims at providing a high-level representation of the environment, as opposed to a task oriented interpretation of sensor data. For example, knowing that a robot is in front of a closed door, may be more informative for deciding the behavior of a system, than simply knowing that the laser scan returns a straight line in front of a robot. However, rather than providing new forms of exploitation of contextual knowledge about the environment, we aim at providing a systematic approach to deal with it. In fact, we show that the same kind of contextual knowledge, if suitably represented, can support several tasks that a mobile robot needs to accomplish. In particular, we suggest that, by taking a context-based approach to robotic system design, it is possible to rely on very specialized, efficient methods, leaving it up to the contextual reasoning the choice of the best suited to the current context.

Introspective knowledge refers to the ability of the system to analyze its own internal state. Contextual knowledge on robot's subsystems includes the specific approach currently used in a given subsystem (e.g., which motion planning algorithm is currently running or what is the maximum speed allowed) as well as knowledge on subsystems' behavior. Notice that this last feature is the basis for the creation of a self-diagnosis process. Finally, contextual knowledge on the controller's internal state includes knowledge on controlling actions taken by the controller in the current control cycle. This allows the controller to reason on its own state. As an example, the controller can check whether it has not enough knowledge about the environment and request an action in order to acquire knowledge about the environment.

In this chapter we mainly focus on the uses of contextual knowledge for motion systems. However, since the contextual architecture deals with the system as a whole, other components (e.g., mapping, localization) are discussed together with the motion system. A detailed description of the design of a contextual-based architecture and a discussion on its features can be found in Calisi et al (2007a), in Calisi et al (2008e) and in Calisi et al (2008d).

Module	**Parameter**	**Values**
Navigation	`MAX_SPEED`	{ low, medium, high }
	`MOTION_PLANNER`	{ RKT, DWA }
Mapping	`MAPPING_MODE`	{ static, dynamic, off }
	`SCAN_MATCH`	{ on, off }

Contextual Variable	**Meaning**
cluttered	the robot is in a cluttered area
rough	the robot is in a rough terrain
big_ramp	the robot is approaching or on a big ramp
ramp	the robot is approaching or on a small ramp
dynamic	the robot is in an area with dynamic obstacles
rotating	the robot is rotating
DWA_stalled	the robot is stalled with DWA motion planner
RKT_stalled	the robot is stalled with RKT motion planner

Contextual Rules
IF *cluttered* **OR** *ramp* **OR** *rough* **THEN** `MAX_SPEED` = low
IF *big_ramp* **THEN** `MAX_SPEED` = medium
IF *dynamic* **THEN** `MAX_SPEED` = medium, `MAPPING_MODE` = dynamic
IF *cluttered* **THEN** `MOTION_PLANNER` = RKT, `SCAN_MATCH` = off
IF *big_ramp* **AND** *rotating* **THEN** `SCAN_MATCH` = on
IF *ramp* **THEN** `MAPPING_MODE` = off
IF *ramp* **OR** *big_ramp* **THEN** `SCAN_MATCH` = off
IF *DWA_stalled* **THEN** `MOTION_PLANNER` = RKT
IF *RKT_stalled* **THEN** `MOTION_PLANNER` = DWA
IF *true* **THEN** `SCAN_MATCH` = on
IF *true* **THEN** `MAPPING_MODE` = static
IF *true* **THEN** `MOTION_PLANNER` = DWA
IF *true* **THEN** `MAX_SPEED` = high

Figure 7.2: System modules and contextual reasoning for the navigation and mapping experiments. The IF-THEN rules are interpreted following the specified order: rules acting on the same parameters are evaluated in the specified order and the first whose condition is true disables the remaining ones. A default value (the right part of the final "IF true" rules) is also specified in case no other rule is active.

7.2 Contextual design of the motion system

7.2.1 System architecture and contextual knowledge

In the following we show an example system in which we adapt the motion system behavior to the context. Two components of the system are considered in these experiments: the motion system, a localization module based on scan-matching and a mapping module based on an occupancy grid.

Two forms of contextual knowledge are considered: (i) environmental knowledge: terrain slope, clutterness, small passages, dynamic moving obstacles; (ii) introspective knowledge: the internal status of software modules (e.g., navigator and obstacle avoidance modules).

The contextual knowledge used for the motion system is related to both the characteristics of the environment and introspection about the status of the running modules. In particular, we limit the speed of the robot in presence of ramps, cluttered areas and moving obstacles. The appropriate mapping modality is chosen according to the dynamics of the environment, and scan matching is disabled when the robot faces or is on top of 3D obstacles, since this introduce a large error in the 2D map being built. Furthermore, in presence of anomalous situations, recognizable in the status of the system modules, e.g., when the robot is stalled, the switch of the motion planning mode allows to overcome the problem.

In order to asses the advantages of contextual knowledge and reasoning in the motion system, we have defined an experimental setup. The task is to navigate through a map where different situations are encountered: the objective of the context-driven system is to adapt the motion system (and the mapping module as well) in order to increase the effectiveness and the performance of the overall system.

Figure 7.2 shows the contextual knowledge and the contextual rules used during the experiments. In contextual knowledge we consider the presence of ramps (with different behaviors depending on their length) and the presence of dynamic obstacles (in fact, people moving in the environment during the experiments), as well as introspection about stalls occurred, when using each of the motion planning modes. The contextual knowledge is here represented as a simple set of IF-THEN rules (in a contextual-based architecture, any other formalism can be used), that are interpreted following the specified order: rules acting on the same parameters are evaluated in the specified order and the first whose condition is true disables the remaining ones. A default value is also specified in case no other rule is active.

The `MAX_SPEED` parameter is used to limit the maximum speed of the robot and it can assume three predefined values, labeled with { low, medium, high }. While the `MOTION_PLANNER` parameter is used to switch between two modalities in the motion planner: RKT and DWA. These are actually two different values for the planning horizon of the motion system, as described in Chapter 5, i.e., in RKT (Randomized Kinodynamic Tree) mode, the planning horizon is unlimited, and the motion system builds a tree and follows a plan on the tree, while in DWA mode, the planning horizon is set to 1, thus transforming the motion system in a simple reactive behavior that is assimilable to the Dynamic Window Approach described in Section 2.1.3. The tree-based mode is able to perform complex maneuvers and plan longer trajectories, but it is computationally intensive and sensitive to localization errors, oscillations and delays: for these reasons it is not suitable to be used at high speeds. Instead, the reactive mode is able to drive the robot at high speeds, but it is not able to perform complex maneuvers in cluttered areas.

The ideal behavior of the robot during the experiments is to appropriately control the maximum speed and the accuracy of the motion planner, according to the situation at hand. Therefore, in difficult areas the speed is set to a minimum value for careful maneuvers. When the robot is in a known area the speed takes a high value for fast exploration. A medium value for the maximum speed is used for default. Furthermore, DWA (i.e., fast but inaccurate) motion planner is used in free areas, while RKT (i.e., slow but accurate) is preferred in cluttered areas.

In this set of experiments, also the mapping module is affected by contextual reasoning. The mapping module uses an on-line scan matching technique to produce a map of the environment, that is suitable for navigation. The mapping module includes two parameters: `MAPPING_MODE`, that can be disabled or can enable either a static or a dynamic mode, when producing the map, `SCAN_MATCH`, that can enable or not the scan matcher during the task execution. In contextual knowledge, we distinguish here the knowledge about big and small ramps, to have different behaviors in the two cases. Moreover, we consider the presence of dynamic obstacles (in fact, people moving in the environment during the experiments), as well as introspection about stalls occurred, when using each of the motion planner modules.

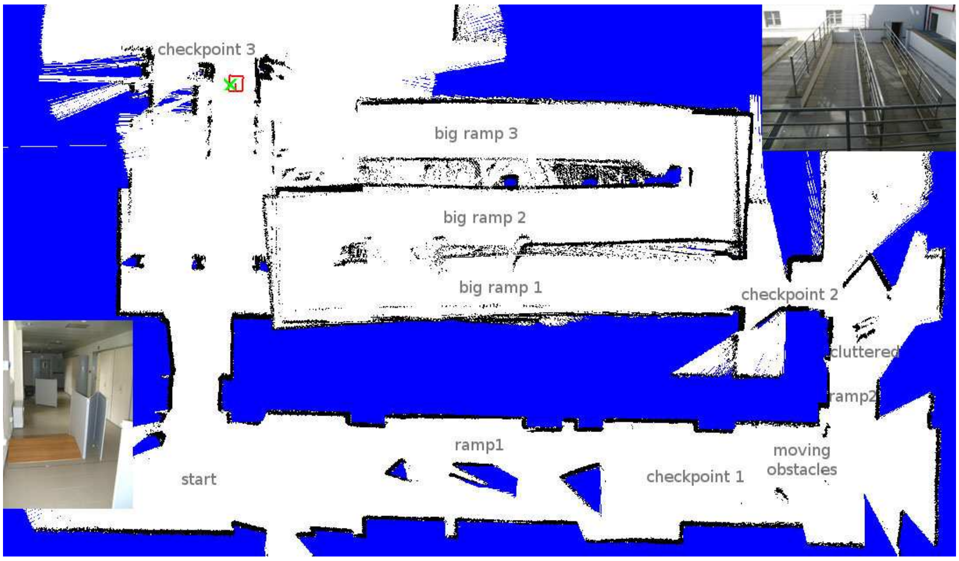


Figure 7.3: Environment used for the context-driven navigation and mapping experiments.

7.2.2 Experimental results

The objective of these tests is to show how the presented architecture can effectively be used to configure modules that can exhibit with better performance according to the characteristics of the environment. Experiments have been performed both on a real robot and on the USARSim simulator in the same kind of environment (i.e., we modelled in USARSim the same portion of the environment in which the real robot operates)[1]. Figure 7.3 shows the map of the environment built by the real robot after a successful run and some snapshots of the environment. The figure is annotated with start and check points, positions of the ramps, and areas with moving obstacles and clutter. The environment is 20 m x 10 m.

In order to evaluate the performance of the context-based architecture, we use the time needed to reach a set of predefined checkpoints in the environment as evaluation metric. Observe that, although the quality of the produced map is not directly measured, it influences the time measure. In fact, when the quality of the map is poor, the robot may face fake obstacles or gets lost, and these situations increase task completion time, or determine a task failure.

Experiment	**Check1**	**Check2**	**Check3**
C_1^R	136 s	299 s	362 s
C_2^R	67 s	214 s	367 s
C_3^R	103 s	229 s	394 s
Avg (stddev)	102 s (34 s)	247 s (45 s)	374 s (17 s)
R_1^R	FAIL (stalled)	FAIL (stalled)	FAIL (lost)
S_1^R	210 s	265 s	804 s
C_1^S	51 s	70 s	239 s
C_2^S	59 s	64 s	193 s
C_3^S	63 s	72 s	202 s
C_4^S	57 s	83 s	250 s
C_5^S	59 s	65 s	189 s
C_6^S	55 s	71 s	225 s
Avg (stddev)	57 s (4 s)	70 s (7 s)	216 s (25 s)
R_1^S	FAIL (stalled)	FAIL (time)	FAIL (lost)
R_2^S	FAIL (stalled)	FAIL (time)	383 s (almost lost)
R_3^S	52 s	FAIL (stalled)	FAIL (lost)

Figure 7.4: Results of navigation and mapping experiments.

The results of the experiments are reported in Figure 7.4, where the use of contextual knowledge is denoted with C, while two configurations without contextual knowledge are indicated with R and S; experiments labeled with superscript ^{R} refer to the real robot, while ^{S} is used to denote simulated experiments. For the runs with no contextual knowledge two configurations were tested: R is a risky and fast configuration with `SCAN_MATCH` = on, `MAPPING_MODE` = static, `MOTION_PLANNER` = DWA, `MAX_SPEED` = high, while S is a safe and slow configuration with `SCAN_MATCH` = off, `MAPPING_MODE` = dynamic, `MOTION_PLANNER` = RKT, `MAX_SPEED` = medium.

The table shows the results of 5 runs with the real robot (3 with context and 2 without) and 9 runs in the simulator (6 with context and 3 without). For each run, three checkpoints have been considered and time to reach each of them has been reported.

---

[1] Videos and further details on these experiments are available on-line at www.dis.uniroma1.it/~iocchi/RobotExperiments/CBA.

The experimental results show that, most of the times, contextual knowledge is critical to actually accomplish the mission. In other words, in such a difficult environment the navigation task cannot be accomplished with a single configuration. As expected, the risky non-contextual configuration R presents many failures, while the safe non-contextual configuration S is much slower. The major reason of failure is the noise introduced by the mapping module when the robot faces 3D structures. In many cases, without contextual knowledge about the presence of ramps, the robot sensors see fake walls, the mapper includes them in the map and the navigation module either is unable to find a path to the target or determines a longer trajectory. A similar situation happens in presence of moving obstacles, since the occupancy grid mapper is too slow to update the map and free the occupied cells. These failures are labeled in the results with 'FAIL (stalled)' and 'FAIL (time)'. Moreover, there are areas (e.g., the big ramp) in which the scan matcher fails, due to reflective materials and noise introduced by the slope. On the other hand, rotating on the ramp relying only to the odometry can give large orientation errors. For this reason, without the use of contextual knowledge the mission results in a failure, due to the robot getting completely lost, being thus unable to reach the target point (these situations are indicated with 'FAIL (lost)'). Finally, notice that there is a single run in simulation (R_3^S), where the non-contextual setting was better than the contextual one. This was due to a "fortunate" and very dangerous run, in which the robot was able to overcome all the obstacles at a great speed without major collisions.

7.3 Context variables determination and topological map annotations

In a context-based architecture, the designer is left free to choose any appropriate method to determine the values of the contextual variables, as well as the rules to infer each context. In In this thesis, we assume that system components capable of measuring relevant information are available. In fact, a gyroscope and/or an accelerometer can be used to determine the slope of a ramp. Other modules can determine the presence of dynamic obstacles and measure the clutterness of the local area, based on (possible subsequent) sensor readings.

In Calisi et al (2009), the approach is improved by introducing the possibility to represent spatial and temporal variables, as well as allowing a rule to contain variables and function symbols. Moreover, ramps are detected by exploiting an *elevation mapper* module, which builds a representation of the ground surface topography, using two differently tilted lasers.

In Section 6.3.3 we showed that it is possible to attach important information to the global representation of the environment, in particular, semantic labels or hints for the global planner to be used to compute the local goal. Contextual variables can exploit the same technique to attach spatial-related contextual information to the same global representation of the environment. This is somehow similar to the behavioral maps (see, e.g. Dornhege and Kleiner, 2007), with the important difference that, in our case, complex reasoning can be exploited in order to choose an appropriate behavior, or, when using the Dynamic Behavior Tree algorithm, to tune the parameters according to the context.

Part III

Evaluation and conclusions

8

The MoVeME evaluation framework

The scientific community is conducting an intense effort to define open and standardized benchmarks and performance metrics in order to objectively evaluate the many components of a robotic system and allow to compare different strategies and solutions. Although this issue is considered of critical importance to increase the knowledge about existing algorithms and methods, and to analyze their benefits and drawbacks, this effort is only at the beginning, and many research communities lack of common testbeds where they can compare their achievements.

In this chapter we propose a novel framework for robot motion system evaluation, called MoVeME, featuring a set of quantitative performance metrics as well as a set of benchmark environments and tasks, that include both typical and critical situations, that have been collected from previous work. The framework has been presented in Calisi et al (2008c) and Calisi and Nardi (2009), and further extended in this thesis.

8.1 Related work in evaluating motion algorithms

In the last years, some efforts have been devoted in building common testbeds for motion planning algorithms. For example, the Algorithm and Application Group at Texas A&M University has built a collection of benchmark problems[1] to be used to compare various motion planning algorithms. The MOVIE project[2], ended in

---

[1] `http://parasol.tamu.edu/groups/amatogroup/benchmarks/mp/`

[2] `http://www.give.nl/movie/index.php`

2005, had the objective of developing motion planning techniques, that could compute in real time visually-convincing motions for multiple autonomous entities, that navigate through complex virtual worlds. The outcome of this project is a repository of motion planning benchmarks, maintained by Utrecht University. Another benchmark database is maintained by University of Parma[3], with the support of the EURON community[4]. The repository is intended to serve as the basis for further discussion on the requirements and the design of benchmarks in motion planning: the project is still in progress and offers only a small set of robot models. Unfortunately, all these collections of benchmarks aim at comparing motion *planning* algorithms, without taking into account real-world issues such as uncertainty. In fact, their focus in on geometrical motion planners, robots with many degrees-of-freedom, industrial assembly and dis-assembly tasks, protein folding, computer animations, and so on.

Some steps towards the definition of common measures of the performance of a robot motion system have been done, e.g., in Muñoz et al (2007), where a set of metrics are presented that allow for an evaluation of the quality of control architectures of mobile robots and quantitative comparisons between different techniques for robot motion. This set of metrics goes beyond the typical shortest-time or shortest-distance metrics, usually used as the sole performance measure for motion planning techniques: they include, e.g., measures regarding trajectory smoothness, that are closely connected with real robot dynamic constraints. Moreover, Raño and Minguez (2006) describe a framework that allows to generate synthetic environments, in order to evaluate obstacle avoidance algorithms in many different situations. Metrics for both the environment and the trajectory followed by the robot are provided.

8.2 The evaluation framework

In this section we describe an evaluation framework for robot motion systems. This framework is then applied to some different motion system, that make use of different world models, in order to quantitatively measure their performance and show which information is crucial for motion systems and which has a negligible effect.

In Calisi et al (2008c) and Calisi and Nardi (2009), the MoVeME (Mobile robots and Vehicles Motion algorithms Evaluation) framework (formerly MoVeMA) has been introduced. This framework features a collection of performance metrics and a set of simulated benchmark experiments for robot motion algorithms evaluation. Benchmarks can be roughly divided into *typical* situations and *critical* situations. Both of them are important to understand and evaluate the behavior of a motion system. Moreover, depending on the kind of evaluation of interest, some benchmarks are more suitable to be used. In this case, we will use large environments that present local challenges for obstacle avoidance techniques.

---

[3] `http://mpb.ce.unipr.it/index.html`

[4] `http://www.euron.org`

There are many aspects of the execution of a motion system that should be taken into account and measured: the accuracy at the goal (meaning with respect to the kind of goal of interest, as described in Chapter 4), how the algorithm behaves along the trajectory, what kind of risks it faces, etc. Some of the performance metrics are in contrast with others: the "best" method has to be chosen depending on the whole task (e.g., accuracy at goal versus time to reach it, trajectory length versus risk of collisions, etc.).

In the following of this section, we give the MoVeME performance metrics, describing them in detail and discussing their goals and the possible similarities with the other similar evaluation frameworks listed in Section 8.1.

8.2.1 Task metrics

In this section we will introduce some metrics that are related to the task definition and the goals of a motion system. In particular, they reflect the needs underlined in Chapter 4.

Success ratio (*Success*). Some motion systems may fail in accomplish their task: this can be for an intrinsic unreliability of the motion system itself, or because it is not designed for the particular task being executed. In order to compute this metric, an experiment is repeated a given number of times, rep_{total}, and *Success* is defined as:

$$Success = \frac{rep_{success}}{rep_{total}}, \tag{8.1}$$

where $rep_{success}$ is the number of times the system is able to accomplish the task.

Accuracy at target (*Accur*). This is the distance between the final pose x_F and the nearest pose $\in \mathcal{G}$:

$$Accur = \underset{x_G \in \mathcal{G}}{arg\,min}\, ||x_F - x_G|| \tag{8.2}$$

This is one of the most important metrics for goals specified by a precise position, as described in Chapter 4, while it becomes much less significant for the others. This metric is not used in other evaluation frameworks, where the accuracy is considered only as a threshold to decide whether the task is finished or not. Since at high speeds it is more difficult to obtain an accurate maneuvering of the robot to a precise pose, we add explicitly this metric in order to consider the trade-off with respect to *Time*, described in the following, that rewards higher speeds.

Time to reach the goal (*Time* and *Time*%). The time to reach the goal is the most used measure when evaluating the performance of an autonomous system in general. The execution time t_F is considered. However, there is a trade-off between *Time* and *Accur*, hence it is up to the algorithm to decide when to stop (i.e., the t_F), depending on the task at hand: that is, spending more time while slowly and accurately positioning the robot in the target pose (i.e., optimize the *Accur* value), or reaching only a neighbourhood of the goal at high speeds (i.e., optimize the *Time* value). The *Time* metric is somehow related to the Control Period (LeM) in Muñoz et al (2007). In order to compare the motion systems using a more general metric, that is not strictly related with the environment, we compute the shortest path and we denote with Len_{min} its length. Given a maximum robot linear speed v_{max}, we can thus roughly calculate the minimum time required to follow the given path: $Time_{min} = \frac{Len_{min}}{v_{max}}$. Finally, the *Time*% can be computed as the ratio between the time needed to accomplish the task and the minimum theoretical time:

$$Time\% = \frac{Time}{Time_{min}}. \tag{8.3}$$

Notice that this metric has always a value greater than 1.0.

8.2.2 Trajectory metrics

The set of metrics presented in this and in the following subsection, reflecting the requirements highlighted in Section 4.3, measure the behavior of the motion system during the trajectory following. In particular, in this subsection, we present metrics that are not directly related to physics, i.e., to inertial forces acting on the robot.

Path Length (*Len* and *Len*%). This measure is needed for tasks in which the power consumed is a critical issue, it should be combined with the velocity, the acceleration and/or the friction, depending on the kind of robot used. The *Len* metric is thus defined as:

$$Len = \int_0^{t_F} ||\tau'(t)||dt \tag{8.4}$$

In which $\tau'(t)$ is the derivative of the trajectory with respect to t. For the same reasons given in the previous sub-section, we would like to compute a metric that is not strictly related to the particular map in which the experiment is performed. Given the minimum path length, Len_{min}, we define:

$$Len\% = \frac{Len}{Len_{min}}. \tag{8.5}$$

Also in this case, the *Len*% metric is always greater than 1.0. The *Len* and *Len*% metrics are considered also in other works: *Len* is the same of the Length of Trajectory Covered (P_L) of Muñoz et al (2007), and *Len*% is also related to the Optimality of Raño and Minguez (2006).

Risk with respect to obstacles (*AvgRisk* and *MaxRisk*). This is a metric that expresses the hazards along the trajectory, with respect to the obstacles. Given a function $\beta(t)$ that measures the distance to the nearest obstacle for each t, *Risk* is defined as:

$$Risk = \int_0^{t_F} \frac{1}{\beta(t)} dt. \tag{8.6}$$

We also define:

$$AvgRisk = \frac{Risk}{Time}, \tag{8.7}$$

and

$$MaxRisk = \max_{\tau} \frac{1}{\beta(t)}. \tag{8.8}$$

Safety-related metrics are, together with the time to reach the target, the most used metrics for robot motion tasks. Many metrics, indeed, have been proposed to objectively judge the risk to hit an obstacle along the trajectory. For example, in Muñoz et al (2007), three metrics are given: *SM1* measures the average of the distances to the obstacles taken by all the sensors of the robot, *SM2* measures the average of the $\beta(t)$, *Min* measures the minimum value of $\beta(t)$. Cluttered areas make *SM1* higher than *SM2*, but we want to measure the risk taken by the robot through its movement, that is related to the nearest obstacle. The *Safety* metric of Raño and Minguez (2006) measures how close the given trajectory matches the path computed on the Voronoi graph (i.e., the locus of points that have the maximum distance from the obstacles, and thus are the safest points): this metric is strictly related to our *Risk* metric, since the safest trajectory has the maximum *Risk* value.

Curvature Change (*CC* and *AvgCC*). This measure is particularly important for car-like robot models, where the curvature is determined by the direction of the wheels, that is one of the variables that can be directly controlled, for example by the driving wheel (Kostov and Degtiariova-Kostova, 1995; Tomomi et al, 2003). This measure is also useful to detect oscillations in the trajectory. We will use the geometrical concept of curvature of a curve, i.e., the reciprocal of the instantaneous curvature radius. When dealing with trajectories, that are parametrized with respect to time, given the linear speed $v(t) = \frac{ds}{dt}$ and the angular speed $\omega(t) = \frac{d\theta}{dt}$, the curvature equation becomes (see, e.g., Coolidge, 1952; Mjolsness and Swartz, 1987, for a detailed discussion about curvature and approximations):

$$\kappa(t) = \left| \frac{\omega(t)}{v(t)} \right|. \tag{8.9}$$

The *Curvature Change* metric is defined as:

$$CC = \int_0^{t_F} |\kappa'(t)| dt. \tag{8.10}$$

In Muñoz et al (2007), the Smoothness of Curvature measures the same concepts: although the value is integrated along the geometric curve, rather than with respect to time. Finally, as usual, we compute an average value, for comparison purposes:

$$AvgCC = \frac{CC}{Time}. \tag{8.11}$$

8.2.3 Physics-based metrics

With the following set of metrics, we measure how the trajectory followed by the robot affects the stability of the motion and the mechanics of the robot itself. These metrics can be considered physics-based metrics, because they are related to vehicle dynamics, and are used to choose the best trajectory or control command to be executed in those systems and applications where the dynamics of the vehicle have to be taken into account (Rosenblatt, 1997; Shmaglit et al, 2006; Ozguner et al, 2007, e.g.,). They are defined as the inertial forces acting on the robot along the trajectory followed; however, the mass of the robot is not considered, since we are not concerned with a particular robot (i.e., we consider the robot as an ideal unit-mass object).

Lateral Stress (*LS* and *AvgLS*). It is a well known issue in evaluating vehicle trajectories with respect to safety, as well as in handling vehicles at high speeds, that trajectories should be as straight as possible; one of the reasons is that this reduces the centrifugal force acting on the vehicle, that affects the vehicle stability. With the *Lateral Stress*, we measure directly this force acting on the robot, integrating it along the trajectory:

$$LS = \int_0^{t_F} \frac{v(t)^2}{r(t)} dt, \tag{8.12}$$

in which $v(t)$ is the instantaneous linear speed of the robot and $r(t)$ is the instantaneous curvature radius. The centrifugal force acting on the robot affects the turning stability of the trajectory, since it can lead to lateral wheel skidding. Moreover, we can compute the average of the *LS* metric:

$$AvgLS = \frac{LS}{Time} \tag{8.13}$$

In other works, the Bending Energy (B_E) and the Total Bending Energy (TB_E) try to consider the same concept, but only with respect to the geometrical path: this fails to account the fact that, at high speeds, the effect of the centrifugal force is higher. On the other hand, the *LS* metric measures directly this force. This metric is not directly related to the *Curvature Change*: for example, an oscillatory trajectory has a high value of *CC*, but can result to a low *LS* value, e.g., if the linear speed is low; however, a sharp turn gives a high value of *LS* (because of short curvature radius and high speed), but a null *CC* measure.

Tangential Stress (*TS* and *AvgTS*). It measures the magnitude of the inertial force acting on the robot as it is moving on a straight line, integrated along the trajectory:

$$TS = \int_0^{t_F} |a(t)| dt, \tag{8.14}$$

where $a(t)$ is the instantaneous linear acceleration. This metric is sensitive to sudden braking and sharp accelerations, that are not desirable for both the integrity of the mobile robots and the possibility of wheel skidding while turning. The usual average is computed with respect to the total duration of the trajectory:

$$AvgTS = \frac{TS}{Time} \tag{8.15}$$

This metric is not considered in any of the previous work (they consider the geometric path rather than the trajectory).

8.3 Comparison with other performance metrics

Table 8.1 shows a comparison between the MoVeME metrics and those in Muñoz et al (2007) and in Raño and Minguez (2006), as well as their classification with respect to Guo et al (2003). Some of the metrics used in these two previous work do not find a related one in the MoVeME framework. In particular, the Success metric of Raño and Minguez (2006) is not used because we assume the motion systems to be able to accomplish the task in all experiments (the Raño and Minguez (2006) is a framework for obstacle avoidance techniques that, being local methods, can be get stuck in local minima; on the other hand, MoVeME metrics are thought for complete motion systems, that are assumed to always finish the task successfully). The Mean distance to the goal (*Mgd*) of Muñoz et al (2007) is not used here, because we want to evaluate the system as a whole, and following a reference trajectory is only a building block of such a system. Moreover, an important difference between these metrics and similar ones considered in related work is that the MoVeME framework computes metrics on trajectories, velocities and forces, while the other methods concentrate only on the geometrical paths.

8.4 Benchmark problems

In order to evaluate a system, beside the metrics described above, a set of experimental scenarios must be choosen. There are two different ways that an evaluation framework can adopt for this issue: from the one hand, the framework can provide a tool to automatically generate random environments, in order to include in the experiments many different situations; on the other hand, a set of pre-defined scenarios and situations can be provided. The first approach has been chosen in Raño and Minguez (2006), where spherical worlds are built by generating random

MoVeME Metric	**Muñoz et al (2007)**	**Raño and Minguez (2006)**	**Guo et al (2003)**
Accuracy at target (*Accur*)	N.A.	N.A.	Geometry-based
Time to reach the goal (*Time*)	Control Periods (*LeM*)*	N.A.	Time-based
Path length (*Len*)	Length of trajectory (P_L)	Optimality	Geometry-based
Risk w.r.t. obstacles (*Risk*)	Security Metric 1 (*SM1*)*, Security Metric 2 (*SM2*), Minimum Distance (*Min*)*	Safety	Safety
Lateral Stress (*LS*)	(Total) Bending Energy ($(T)B_E$)*	N.A.	Physics-based
Tangential Stress (*TS*)	N.A.	N.A.	Physics-based
Curvature Change (*CC*)	Smoothness of Curvature (*Smoo*)	N.A.	Geometry-based

Table 8.1: The set of MoVeME metrics compared with Muñoz et al (2007) and with Raño and Minguez (2006) metrics, and classified with respect to Guo et al (2003). The metrics marked with an asterisk are only loosely related to those provided by the MoVeME framework, while the other are equivalent (although some differences can still be found, see the text for details).

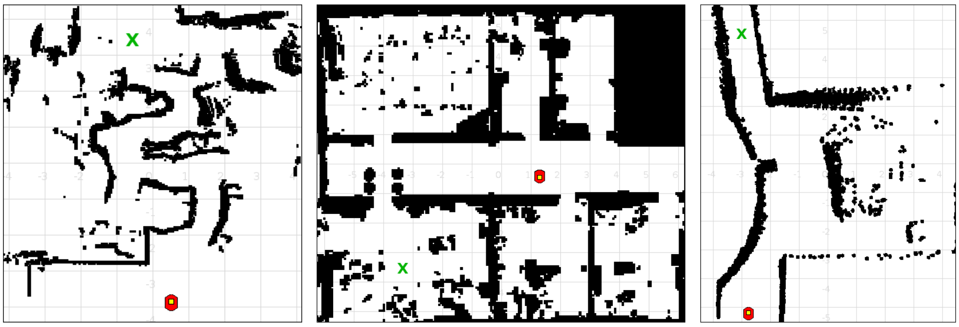


Figure 8.1: Three environments taken from the MoVeME benchmark dataset, showing also a robot starting pose (the red circle) and a goal (the green cross).

spherical obstacles: the main benefit of this method is that there is the possibility of discover some critical cases, that otherwise would have not been considered in the experiments. However, because of the randomness, many experiments are needed in order to collect measures that are statistically relevant.

In the MoVeME framework, we choose the second method, selecting well-known relevant situations to provide a set of pre-defined benchmark problems[5]: this approach reduces the number of experiments that are needed to collect performance measures. However, benchmark problems have to be selected carefully in order to include as many relevant cases as possible. These benchmark problems have been presented in Calisi et al (2008c) and can be roughly divided into *typical* and *critical* situations, and include examples given in past research papers (e.g., Borenstein, 1991; Fox et al, 1997; Minguez and Montano, 2004; Simmons, 1996; Ulrich and Borenstein, 1998), especially those that involve an analysis of one particular algorithm or a comparison between different methods (e.g., Koren and Borenstein, 1991; Fernández et al, 2004; Stachniss and Burgard, 2002). Figure 8.1 shows some of these environments. The MoVeME performance metrics reflect, with objective and quantitative measures, the statement that have been obtained in these works by qualitative evaluations and visual inspection of the trajectories.

The current set of benchmark problems included in the MoVeME framework makes use of the Player/Stage software (Collet et al, 2005), a suite of free software tools for robotic applications, that contains also the simulator (Stage) that is used for some of the experiments in next section.

Moreover, we extend this initial collection with other critical situations found through our past research and with other typical scenes (office-like maps, more unstructured environments, etc.). For each of these maps a set of pairs starting pose/goal is given. Depending on the kind of evaluation of interest, some benchmarks are more suitable to be used than others.

---

[5] the benchmark problems and a utility to measure the performance can be downloaded from `http://www.dis.uniroma1.it/~calisi/moveme`

9

Experiments

In this section, through a set of experiments, we present the evaluation of some motion systems, by relying on the metrics described in Chapter 8. In particular, we show that, often, the use of a quantitative measurement leads to a better understanding of the systems being used that cannot be achieved by a qualitative inspection. Part of these experiments have been presented in Calisi and Nardi (2009).

The experiments, that make use of the MoVeME set of benchmarks described in Chapter 8, have been designed following the goal definitions given in Chapter 4, while the motion systems to evaluate make use of different environment models as described in Chapter 3 and Chapter 6. Moreover, we use some of the local methods described in Chapter 2, as well as the DTT and DBT algorithms described in Chapter 5. A detailed description of the motion systems used in the experiments is given in Section 9.1.

Experiments are performed using different environments, depending on the particular goal of the specific experiment itself. Specifically, is Section 9.2, we use a subset of the MoVeME (simulation) benchmarks in order to derive both general assessments about the motion systems (e.g., that the global representation has a very small impact on the overall performance of a motion system), and to compare the performance of different methods (e.g., DWA versus ND, etc.). Depending on the specific experiment, we use a *typical* environment, or a *critical* environment.

Afterwards, in Section 9.3, we perform some experiments using a real robot, reproducing some of the situations addressed in the simulation experiment, in order to confirm the results of Section 9.2.

Finally, in Section 9.4, we perform a general comparison between a subset of the systems described in this chapter, by using the whole set of MoVeME (simulation) benchmarks. The main motivation of this last set of experiments is to obtain a general evaluation and comparison of the performances of the systems.

9.1 Systems being evaluated

In this section, we describe the different motion systems that will be evaluated in the following of this chapter. Each system is composed by a global representation of the environment, a procedure to compute the local goal and a local algorithm that steers the robot to the global goal. In each scenario we test and/or compare a subset of these motion systems, in order to underline the differences (or the lack of differences) of their performance, either by comparing different methods in the global subsystem or in the local subsystem, or in the systems as a whole.

- **System C-DWA**. The first system exploits a metric map of the environment, composed of a grid of cells, at the global level (see Section 3.2). An A* algorithm is executed on the grid in order to compute the path to the global goal, the A* heuristic used is the classical distance-to-goal. The obstacles on the grid map are expanded in order to obtain the connectivity of the environment (i.e., closing passages that are too narrow for the robot to go through). The Dynamic Window Approach (see Section 2.1.3) is used at the local level, in particular, the random clothoids variant is used in these experiments (see Section 2.1.5). The local target that is given to the DWA is computed along the path in a position that is inside the LSA and at a maximum fixed distance from the robot.

- **System G-DWA**. The environment is globally represented using a geometric model (in particular, a line map, see Section 3.3). The connectivity of the environment is determined using the Voronoi graph computed on this geometric description. The Voronoi graph is a roadmap, furthermore, when a goal is given, both the goal and the current robot pose are connected with the roadmap graph and the path is searched in this graph (e.g., using the Dijkstra algorithm or A*). The local target computation and the local subsystem being used are the same as in System C-DWA.

- **System T-DWA**. A topological description is adopted by this system (see Section 3.4), which does not contain any metrical information. In particular, places are recognizable junctions on a local Voronoi graph and the edges that connect places mean that it is possible to navigate directly from one to the other. Places can be labeled and the goal have to be specified using these labels. The local target is computed using the method explained in Section 3.4, where the topological model gives the direction that has to be followed at each junction. The local subsystem employs the DWA algorithm as in the two previous systems.

- **System T-VFH**. This system is exactly the same of the one above, but it uses a Vector Field Histogram algorithm (see Section 2.1.2) at the local level. The major difference between the two local algorithms is that VFH outputs a *direction*, without considering the robot model constraints, such as non-holonomy and dynamics. This means that the direction must be converted into motion commands, but the final trajectory is not guaranteed to be collision free. In order to avoid collisions, we limit the robot movements to two types of maneuvers: turn on place and go (almost) straight, in this way the direction produced by VFH is reliably followed.

- **System THint-DWA**. At the global level, a topological representation is used; the usual method to compute the target direction along the trajectory is modified by including a *direction* that follows the Voronoi graph. This acts as a "hint" for local algorithms that are able to deal also with the direction of the local target pose. The local algorithm used is the DWA.

- **System THint-DTT**. The topological representation is used by an algorithm to compute the local target pose that includes also a direction, as in the system above. This system uses the Dynamic Trajectory Trees, described in Chapter 5, that is based on the Randomized Kinodynamic Tree algorithm and is thus able to plan a local trajectory consisting of more than one step in the future.

- **System H-DWA** and **System H-DTT**. These are very similar to the previous System T-DWA and System H-DTT, with the only difference that the global subsystem make use of the hybrid topological/geometric representation that we described in Chapter 6, and the local goal is computed using the technique shown in the same chapter, i.e., using the frontiers of the LSA.

- **System H-ND**. This system uses the hybrid global representation of the environment and the local subsystem consists of a slightly modified version of the Nearness Diagram, that has been described in Section 2.1.4. Although the basic ND algorithm produces a direction, as explained above for the VFH, there was no need to limit the robot maneuvers as we do in that case, because, with ND, collisions are unlikely to happen.

- **System H-DBT**. Finally, this system uses the same hybrid representation of the previous systems, but the local subsystem consists in the Dynamic Behavior Tree algorithm, that has been described in Chapter 5: in particular, it presents a set of behaviors that are randomly chosen and evaluated (in our implementation, the local subsystem can choose among the Dynamic Window Approach and the Nearness Diagram), and it has the possibility to increase the look-ahead, depending on the time allowed for each iteration.

The first systems, with the varying global level representation are compared, in particular in Section 9.2.1, to evaluate the effect of the global representation on the overall performances. Systems in which additional information is provided to the local subsystem (e.g., a direction "hint"), are tested to assess the value of this information with respect to the performance. Finally, other experiments compare the performance of different local subsystems (e.g., pure Dynamic Window Approach, pure Vector Field Histogram versus Dynamic Behavior Tree).

9.2 Simulation experiments

In this section, we perform a set of experiments using some of the environments taken from the MoVeME benchmarks set.

9.2.1 A typical environment: the "hospital" map

The first environment is well suited for the evaluation of motion system in a *typical* situation. The "hospital" map (that has been taken from the example maps of the Player/Stage suite and added to the MoVeME benchmarks set) can be seen in Figure 9.2 and represents a large indoor environment (40m wide) with many rooms and corridors. We tested four motion systems in this environment: System C-DWA, System G-DWA, System T-DWA and System T-VFH. In this environment we want to measure how the use of the different environment models can affect the performance of the motion system. For this reason, three of the systems use different environment models at the global level, and the same local algorithm. The fourth system, System T-VFH, making use of a different local method, is added in order to show how the local method affects the performances of the whole system.

The goal in this scenario, as shown by the trails in Figure 9.2, is to go from the pose marked as "start" to the pose marked as "goal". For the systems that use a topological representation, the task is described as going to the "place" of the topological map that represents the room that contains the "goal" pose, and, as soon as this pose becomes visible by the sensors, giving it as local target to the local algorithm. This simulates the fact that the final goal has a position that can be recognized using local information (e.g., landmark-based). In Figure 9.1 we show the performance of the systems with respect to the MoVeME metrics (the values are averaged over 5 experiments for each system).

The most important result is that, despite the use of very different representations, the performance of the four systems do not present significant differences. This is due to the fact that the output of the global algorithm, i.e., the local target, can be computed using a very limited previously acquired knowledge: it can be computed on the fly using only the current local environment spanned by the sensors.

Moreover, additional knowledge can be gathered from these results. Although the commonly used performance metrics, i.e., *Time* and *Len*, show that System C-DWA performance are very similar to those of the other systems, the *Curvature Change* metric is much higher. This can be explained as follows: the A* algorithm, computed on the grid map, outputs a path that is very close to the walls, and this

Metric/System	**System C-DWA**	**System G-DWA**	**System T-DWA**	**System T-VFH**
Accur (m)	0.09 (0.00)	0.10 (0.01)	0.12 (0.01)	0.10 (0.00)
Time (s)	194.27 (2.84)	196.22 (2.75)	196.35 (2.00)	235.77 (2.92)
Len (m)	34.56 (0.18)	34.55 (0.12)	34.55 (0.11)	34.42 (0.13)
Risk (m^{-1})	335.3 (7.99)	334.23 (8.02)	283.47 (0.96)	301.18 (5.29)
CC (rad/m)	1601.7 (210.2)	145.12 (36.12)	168.24 (39.32)	746.78 (108.67)
TS $(N \cdot s)$	0.84 (0.36)	0.35 (0.20)	0.38 (0.01)	0.27 (0.12)
LS $(N \cdot s)$	127.35 (6.06)	130.43 (7.17)	132.98 (2.38)	94.72 (2.14)

Figure 9.1: Performance evaluation in the "hospital" environment of the four systems described in the text, the values are averaged over five experiments in the same conditions, standard deviations are given in parenthesis.

makes its trajectory more sensitive to corners and small obstacles along the walls. The final behavior, as explained by the large *CC* value, is an oscillating trajectory. If the mobile robot were a car-like vehicle, the steering wheel should have been moved much more than in other cases. Indeed, the fact that the robot moves near the corners, hides the local environment to the local algorithm, that often faces unexpected obstacles, requiring sudden changes of direction (i.e., of curvature of the trajectory), and, as explained by the higher *Tangential Stress* (*TS*) value, also to sudden braking. This means also that, even though the global path computed by A* on the global grid map is shorter with respect to the others, this benefit is canceled by the oscillations on the trajectory performed by the local algorithm: their final length (*Len*) is almost the same, as well as the time needed to conclude the task (*Time*).

Other differences among the performance values are more straightforwardly explained. For example, the path followed by the Vector Field Histogram is made up almost of straight segments, that makes it longer than the others, but with a lower *Lateral Stress* (on-place turns does not count for the computation of *LS* metric).

Summarizing, this experiment shows that the use of different environment models leads to similar performance of the motion system. This is important to understand, since some environment models require more resources to be built, maintained and used. Moreover, it shows that the most critical building block of a motion system, with respect to its performance, is the local method.

9.2.2 A critical environment: the "zig-zag" map

The second environment can be seen as a *critical* environment (as we said before, the "hospital" environment is a *typical* benchmark). This specific map is created by introducing a sequence of difficult situations, in order to test the behavior of different motion systems. The environment, called "Zig-zag", is shown in Figure 9.3. In this case, the goal is to move the robot from the pose marked with "start" to the grey region marked with "goal", i.e., the cardinality of $\mathcal{G}$ is greater than 1.

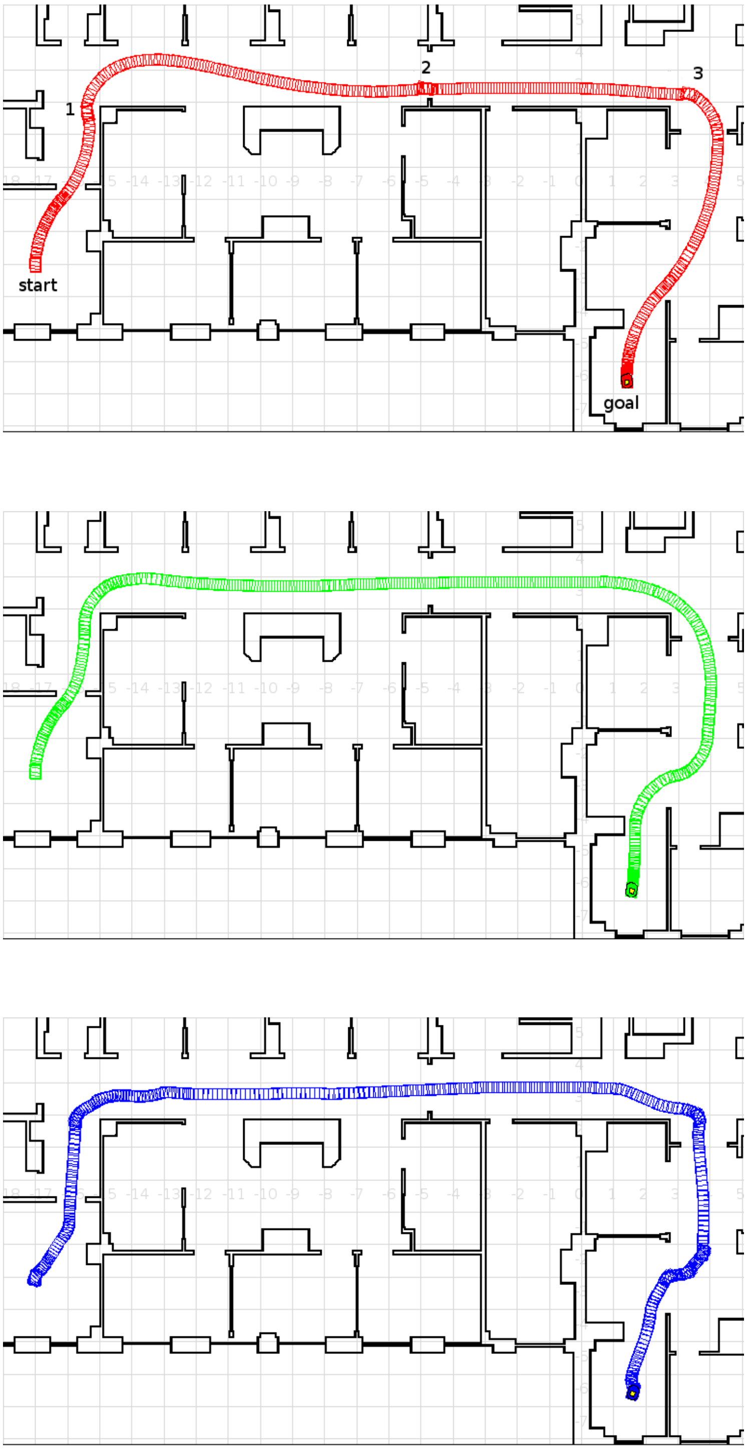


Figure 9.2: The "hospital" environment, with the trails followed by three of the systems described in the text. The first one is System C-DWA: the oscillations described in the text are marked as 1, 2 and 3; the second is System T-DWA, while the last is System T-VFH, in which the trajectory presents more straight lines than the others

The goal of these experiments is to measure the effect of a "hint" of the global planner to the local algorithm. This additional information concerns a part of the environment that is not contained in the local model used by the local algorithm. To this end, the first system is taken from the previous experiments (System T-DWA), while a second one is obtained by adding additional information from the global planner (System THint-DWA). A third system is also considered, which contains the DTT local planner, instead of a pure reactive method (System THint-DTT). The purpose of measuring the performance of this latter system is to have a comparison term with a longer planning horizon and understand if it can get more benefits from the same global "hint".

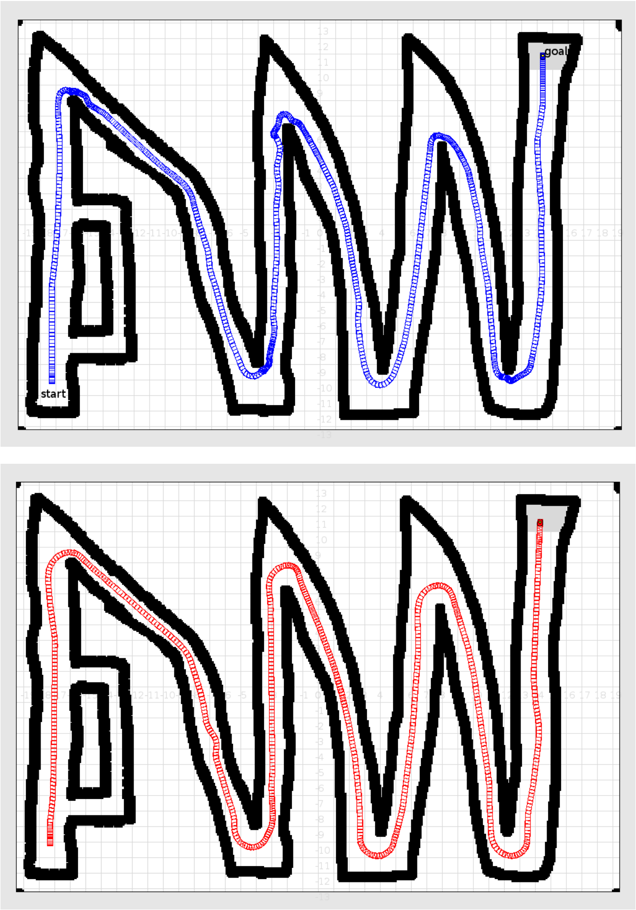


Figure 9.3: The "zig-zag" environment with the trails followed by two of the systems described in the text. The first is System T-DWA, in which the long and narrow corridor after each sharp turning cannot be predicted in time by the algorithm. System THint-DWA, shown below, can benefit from the direction of the local target, that makes it possible for the local algorithm to better adjust the trajectory and enter the narrow corridor with the right orientation, without the need of sharp turns.

Metric/System	**System T-DWA**	**System THint-DWA**	**System THint-DTT**
Time (s)	608.48 (10.40)	533.9 (9.33)	552.19 (10.31)
Len (m)	135.07 (0.17)	142.75 (0.02)	141.22 (0.03)
Risk (m^{-1})	1419 (62.2)	992.69 (44.63)	989.12 (40.32)
CC (rad/s)	916.86 (37.6)	190.15 (14.69)	211.23 (12.18)
TS $(N \cdot s)$	0.52 (0.04)	0.28 (0.01)	0.17 (0.01)
LS $(N \cdot s)$	530.92 (15.24)	493.00 (12.68)	482.18 (10.78)

Figure 9.4: Performance evaluation in the "Zig-zag" environment of the three systems described in the text (*Accur* values are not given, because the goal area is large), the values are averaged over ten experiments in the same conditions (algorithms use probabilistic techniques), standard deviations are reported in parentheses.

The behavior of System T-DWA at the sharp turnings can be seen both in the trails depicted in Figure 9.3 and in the values in Figure 9.4: since there is a limited planning horizon and no heuristics on the right orientation of the corridor is given, the system is forced to change the curvature of the trajectory in a very short time and space. The values of the *Tangential Stress* and the *Curvature Change* both reflect this behavior. Moreover, due to the speed acquired before the turning, the trajectory becomes unstable and oscillates (as can be seen from the *Curvature Change* metric). Finally, also the *Lateral Stress* is higher, meaning that the algorithm is not able to reduce the speed in time before performing the turning.

Fewer differences can be seen between System THint-DWA and System THint-DTT, meaning that the longer planning horizon does not achieve much higher performances, in the presence of the global orientation hint. The larger time of System THint-DTT is basically due to the reduced speed allowed by the planner in order to be able to plan and execute actions without stopping. The low *Curvature Change* value reflects the fact that being able to plan with a longer horizon, the maneuvers at the sharp turn can be more accurate.

9.2.3 A critical task: parallel parking

This experiment regards a parallel-parking maneuver. In order to properly perform the experiment, we restrict the allowed turning radius, in order to simulate a car-like vehicle. Three systems are tested in the experiment: System H-DWA, System H-DTT and System H-DBT. The goal of the experiment is to show how the local subsystem of System H-DBT selects the proper behavior autonomously. In particular, three behaviors are considered in the random choice for the Dynamic Behavior Tree expansion: the Dynamic Window Approach, the Nearness Diagram (that, in our implementation, is forced to perform only forward maneuvers) and generic feedback controlled constant-acceleration curves (clothoids). The orientation of the goal has a high coefficient in the goal fitness evaluation function ϕ: in this way, we force the motion system to prefer maneuvers that move the robot to a position that is parallel to the wall.

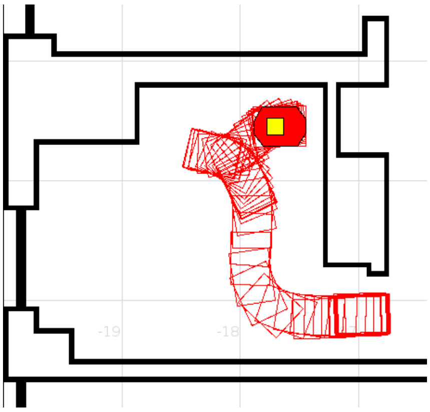


Figure 9.5: A trail of the robot performing a parallel parking maneuver, using System H-DBT.

Metric/System	**System H-DBT**	**System H-DWA**	**System H-DTT**
Success%	72.43	0.0	73.20
Accur%	89.48	-	90.12
Time%	810.98	-	949.18

Figure 9.6: The results of the experiments of parallel parking.

Moreover, in order to increase the success rate, we attached to the global environment representation a hint regarding the maximum speed allowed in the proximity of the parking lot: in particular, maximum speed is reduced when the robot gets closer to the final goal pose, in order to give more time to the planner to produce a proper maneuver.

The experiments are repeated 10 times for each system and their results are shown in Figure 9.6. Moreover, a trail of the robot, using System H-DBT, is shown in Figure 9.5. The results show that the DWA algorithm is not able to perform the appropriate maneuver to move the robot to the desired position (actually, it is not designed for this kind of tasks, at least using its canonical utility function and the additional minimum curvature radius constraint), while both the Dynamic Trajectory Tree and System H-DBT are able to move the robot to the parking lot. The main difference between these last two systems is that the latter can take advantage of faster behaviors (e.g., DWA) while it is approaching to the parking area, thus reducing the overall time to complete the task (i.e., *Time%*).

Summarizing, the use of a system that is able to autonomously select the appropriate behavior (i.e., a fast and short look-ahead pure-reactive method versus a slower long look-ahead local planner) results in an increase of performance with respect to the use of single methods.

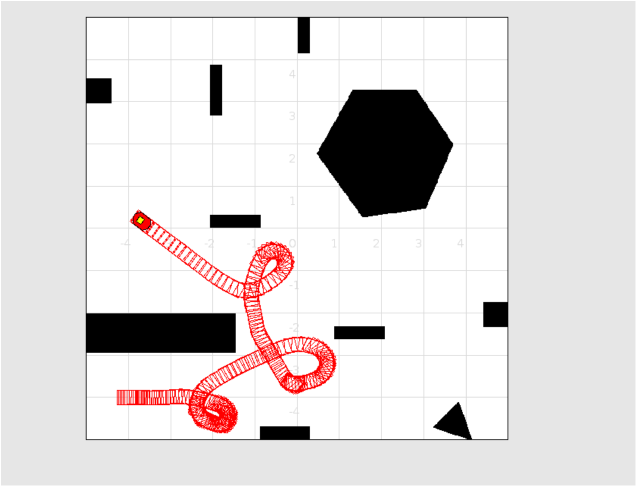


Figure 9.7: A trail of the robot in one of the simulation experiments using System H-DBT, in which the robot is constrained to move only forward and turn left.

Metric/System	System H-DWA	System H-DTT	System H-DBT
Success%	0.00	75.43	76.21
Len%	-	415.43	405.95
Time%	-	814.12	803.23

Figure 9.8: Results of the constrained motion experiments described in Section 9.2.4.

9.2.4 A critical task: constrained motion

The map shown in Figure 9.7 has been taken from Steven LaValle's RRT webpage[1] and added to the MoVeME benchmarks set. The peculiarity of this experiment is that the robot maneuvers are constrained to be forward-only and right-only (the minimum turning radius is also constrained to a constant > 0, in order to simulate a car-like vehicle). In this example, a large look-ahead is needed to properly evaluate the maneuvers, and most of the time the pure-reactive behaviors fail in constructing a good solution. Three systems have been tested in this environment: System H-DWA, System H-DTT and System H-DBT. As in the parallel parking experiments, we can see that the Dynamic Window Approach is not suitable for this kind of setup. The problem is that when the target is on the left of the robot, the right-only constraint filters out all the left turns: furthermore, the best trajectory, chosen through the DWA utility function, is the straight forward motion, that continues as long as the local goal is at a direction $\theta_{goal} \in (0, \pi/2)$. As soon as θ_{goal} reaches $\pi/2$, i.e., it is exactly at the left of the robot, the best behavior computed by the DWA utility function is to stay still and the motion systems stops.

---

[1] `http://msl.cs.uiuc.edu/rrt/`

The results given in Figure 9.8 shows very high values for *Time%* and *Len%*: this is due to the fact that the minimum path length, used to compute the percentages, is calculated geometrically and do not consider the right-only maneuvers. Additionally, in this case, the difference between the DTT-based system and the DBT-system are not so relevant as they have been measured in Section 9.2.3. The reason is that this scenario (specifically, the constraints on the allowed motion) is particularly hard and require a large look-ahead to be solved. The two algorithms, DTT and DBT, becomes thus very similar, because the benefits of using reactive behaviors become negligible (actually, behavior-based trajectory arcs are seldom selected for tree extension).

9.3 Real robot experiments

In this section we present a set of experiments using real robots. As noticed in Chapter 8, the results of simulation benchmarks have to be confirmed by real-world examples. Indeed, it is likely that some real-world issues, that are not suitably simulated, yield to unexpected differences.

We use a Videre Design's[2] Erratic mobile platform, equipped with a laser range finder. In particular, two different devices are used: a SICK's[3] LMS laser range finder, that can measure distances up to 80 meters, and a Hokuyo's[4] URG 04-LX, that can measure up to 4 meters. Systems using the former (SICK, 80 meters) are named with a -long suffix, otherwise, the range finder used is the latter (Hokuyo, 4 meters).

The environment model used in this experiment has been acquired by the robot before the actual experiment.

9.3.1 A typical environment: corridor experiment

In this experiment we would like to confirm the results obtained through simulation experiments reported in Section 9.2.1 and Section 9.2.2. Specifically, the results should confirm that the global representation has a negligible effect on the overall performance (provided that it is able to determine an appropriate local goal), and that, on the contrary, if important hints are attached to the local goal, the performance increases.

Four motion systems have been evaluated: System C-DWA, System H-DWA, System HHint-DWA and System HHint-DWA-long (the difference between the last two systems is only in the sensors being used).

The experiments in the real environment shown in Figure 9.9 yield the results given in Figure 9.10, that comply with the findings of the experiments in simulation environments. System C-DWA and System H-DWA have a very similar per-

---

[2] `http://www.videredesign.com`

[3] `http://www.sick.de`

[4] `http://www.hokuyo-aut.jp`

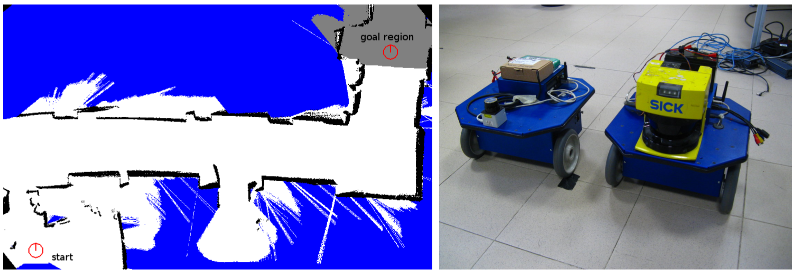


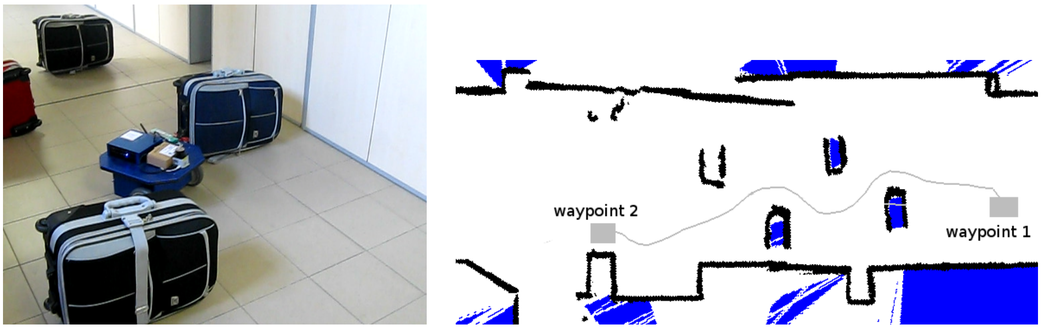
Figure 9.9: The map of the real environment used for the experiments (on the left) and the robots used (on the right). The robot starts the task from the position marked with "start" and the goal is to reach the grey area, detected using local landmarks.

Metric/System	**Sys. C-DWA**	**Sys. H-DWA**	**Sys. HHint-DWA**	**Sys. HHint-DWA-long**
Time (s)	94.512 (2.45)	95.395 (3.01)	91.77 (2.32)	84.21 (2.03)
Len (m)	21.68 (0.27)	21.82 (0.17)	22.62 (0.22)	23.25 (0.03)
Risk (m^{-1})	182.77 (6.13)	181.12 (5.18)	132.32 (4.12)	120.66 (4.19)
CC (rad/m)	1916.79 (42.1)	1842.86 (43.2)	108.52 (12.18)	67.64 (5.13)
TS $(N \cdot s)$	0.31 (0.02)	0.39 (0.03)	0.18 (0.01)	0.06 (0.00)
LS $(N \cdot s)$	62.43 (2.54)	62.60 (2.52)	69.02 (2.37)	79.20 (2.28)

Figure 9.10: Performance evaluation in the real experiment of the four systems described in the text (*Accur* values are not given, because the goal area is large), the values are averaged over ten experiments in the same conditions, standard deviations are reported in parentheses.

formance, despite the use of different environment models, as we found in Section 9.2.1. System C-DWA and System H-DWA present a high value of *CC*, that is mainly due to the fact that the trajectory goes too close to the obstacles. In particular, this is especially apparent when tackling the sharp turns, where, since the map is built during the motion execution, new obstacles appear and require the robot to correct the control commands, leading to oscillations. This results also in a longer time to complete the task.

Since System HHint-DWA and System HHint-DWA-long have higher performances with respect to the other two systems, we can confirm that the direction hint, provided by the global planner, is an important piece of information for increasing the performance of the motion system. However, due to the limited range of the sensor (and, consequently, of the area in which the local target pose can be given), System HHint-DWA-long outperforms System HHint-DWA.



(a) The real robot performing the second phase of the experiment (b) The trail of the robot course among the obstacles

Figure 9.11: A robot trail of the execution of System H-DBT, during the "slalom" phase of the experiment described in Section 9.3.2.

Metric/System	**System H-ND**	**System H-DTT**	**System H-DBT**
Time (s)	32.43	89.57	45.18
Len (m)	3.48	3.03	3.66
AvgRisk (m^{-1})	2.45	3.48	3.03
AvgCC $((rad/m)/s)$	578.40	84.23	619.89
AvgTS (N)	0.01	0.65	0.02
AvgLS (N)	0.89	0.12	0.85

Figure 9.12: The results of the "moving obstacles" phase of the experiment described in Section 9.3.2.

9.3.2 A critical environment: moving obstacles, slalom and parallel parking

We decomposed the experiment into three phases, using the two way-points. The central phase of the experiment, in which the robot is required to navigate between the first way-point to the second way-point, is shown in Figure 9.11: way-points are denoted with gray squares. The goal of the first two phases is to reach the related way-point, i.e., the gray square, with any orientation. As soon as the robot reaches the way-point, the next goal is provided. Three motion systems have been tested in this environment: System H-ND, System H-DTT and System H-DBT.

Each of the three phases has its own peculiarities: the first phase, that we call "moving obstacles", requires the robot to go through a portion of the corridor where moving obstacles (people walking) are present. The contextual knowledge about the presence of moving obstacles is attached to the map, and a suitable set of contextual adaptations of the systems are performed. The presence of moving obstacles could have been also detected on-line, but, in order to simplify the experiment, we preferred to consider this information as *a priori* knowledge about the environment. The results of this first phase are reported in Figure 9.12. The high

Metric/System	**System H-ND**	**System H-DTT**	**System H-DBT**
Time (s)	65.3	75.30	68.23
Len (m)	12.90	12.33	12.87
AvgRisk (m^{-1})	2.62	3.21	3.18
AvgCC $((rad/m)/s)$	662.73	642.89	660.12
AvgTS (N)	0.01	0.01	0.01
AvgLS (N)	0.93	0.98	0.95

Figure 9.13: The results of the "slalom" phase of the experiment described in Section 9.3.2.

Metric/System	**System H-ND**	**System H-DTT**	**System H-DBT**
Accuracy%	43.18	83.29	82.18
Time (s)	59.91	36.28	38.27
Len (m)	3.29	1.50	1.56
AvgRisk (m^{-1})	1.55	4.15	5.18
MaxRisk (m^{-1})	10.50	11.05	10.49
AvgCC $((rad/m)/s)$	7.20	19.68	18.58
AvgTS (N)	0.003	0.01	0.01
AvgLS (N)	0.32	0.14	0.15

Figure 9.14: The results of the "parallel parking" phase of the experiment described in Section 9.3.2.

Metric/System	**System H-ND**	**System H-DTT**	**System H-DBT**
Accuracy%	43.18	83.29	82.18
Time (s)	157.64	201.15	151.68
Len (m)	19.67	16.86	18.09
AvgRisk (m^{-1})	2.18	3.5	3.64
AvgCC $((rad/m)/s)$	396.25	281.72	486.27
AvgTS (N)	0.01	0.29	0.01
AvgLS (N)	0.69	0.45	0.72

Figure 9.15: The overall results of the experiment described in Section 9.3.2.

value of *AvgTS*, presented by System H-DTT reflects the fact that the system needs frequent sudden stops in order to avoid to collide with the moving obstacles. This is mainly due to the low reactivity of the method: the plans are often invalidated by the obstacles crossing the planned trajectories. Moreover, its low values of *AvgCC* and *AvgLS* mean that the overall behavior of the system is to go roughly straight and stop suddenly, if an obstacle crosses its way. System T-ND, on the contrary, being based on a pure-reactive behavior, presents higher performances with respect to the other two systems.

In the second phase, that we call "slalom", the robot moves through a set of obstacles that form very narrow passages between each other. No contextual knowledge is used in this area, and the motion systems are required to adapt to the situation (e.g., by reducing the speed) automatically. The results of this second phase are reported in Figure 9.13. All systems are able to accomplish the task and very little differences can be seen from the metrics. Actually, the behavior of the three systems among the obstacles is almost the same. Little differences can be found in how the Nearness Diagram keeps the robot away from the obstacles, with respect to the DTT. A straightforward effect of this fact is that the trajectory followed by the ND is longer. However, it is executed in less time, i.e., at a higher speed: this is probably due to the randomized component in the other two algorithms, that results in a sub-optimal trajectory.

The third phase requires the robot to accomplish a parallel parking maneuver, i.e., this experiment is the real-robot counterpart of the one described in Section 9.2.3. The task is the same: the robot is required to park parallel to the wall and its motion is constrained by a minimum turning radius (in this case, this is implemented as a contextual rule, attached to this area). The detail of the trajectory followed by System H-DBT during one of the experiments is shown in Figure 9.16. The results of the last phase of the experiments are shown in Figure 9.14: the most apparent difference between System H-ND and the other two is that, since the former is not suitable for this kind of maneuvers, performed a very long circular trajectory in order to reach the final goal. The effects of this behavior is reflected in many metrics. The most apparent ones are the length of the trajectory (*Len*) and the related time to reach the goal (*Time*): the long maneuver results obviously also in a much longer time to reach the goal. Since the maneuver moves the robot to the center of the corridor, before going back to the parking lot, this long trajectory also reduces the average risk (*AvgRisk*); however, the maximum risk (*MaxRisk*) is similar to the other two systems, and is reached in the goal pose for all the systems. Finally, the maneuver of System H-DTT and System H-DBT requires also a larger change of curvature (if the robot were a car, the driving wheel should be turned a lot at the cusp of the trajectory, as it is shown in Figure 9.16), with respect to the long and time consuming trajectory of System H-ND (that basically performs a long circular trajectory, i.e., the curvature is not changed at all). Finally, the main problem with the ND trajectory is that it causes the robot to reach the goal with the wrong orientation, and that the system is not able to correct it: this is apparent to the low *Accuracy* value.

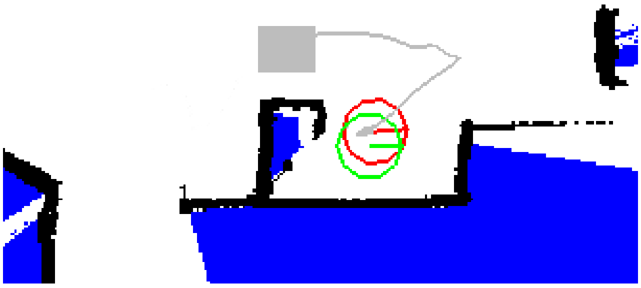


Figure 9.16: The parallel parking with the real robot and the System H-DBT. The trail followed by the robot during the experiment, that ends in the red pose, is shown in gray. The goal pose is shown in green.

Finally, the overall results of the whole experiment are shown in Figure 9.15. These final results show that, even if the other systems perform better in some specific case (i.e., in one of the phases of the experiment), the adaptive System H-DBT has better overall performances. In particular, it is able to accurately position the robot at the final parallel parking pose, with a *Time* and *Len* that are comparable with the best performance achieved in this experiment.

9.4 Experiments with the whole set of MoVeME benchmarks

In Section 9.2, we presented specific tests to show some peculiarities of a selection of motion systems. We used some specialized environments, taken from the MoVeME set of benchmarks, to test individual features, often in the execution of critical tasks. The results of these experiments have been subsequently confirmed using real environments, in Section 9.3.

In this section, we make use of the whole set of benchmark problems that is provided by the MoVeME framework, in order to obtain more general assessments about some motion systems. Indeed, since the MoVeME set of benchmarks is composed by critical tasks (such as those used in Section 9.2), as well as typical situations, testing a motion system on the whole set of benchmarks allows to obtain a *general evaluation* of the specific motion system. The set of MoVeME benchmarks is currently composed by 17 environments and 24 different experiments (i.e., initial pose and goal description). For each setup, 10 repetitions have been performed: in Figure 9.17 we show the values of the MoVeME performance metrics, averaged over the whole set of benchmarks. Obviously this large set of experiments has been performed in simulation, under the assumption, verified in Section 9.3, that the results are accurate with respect to the real case.

All systems tested in this section use the hybrid topological/geometric repre-

Metric/System	**System H-DWA**	**System H-ND**	**System H-RRT**	**System H-DBT**
Success%	84.21	90.90	98.32	98.25
Time%	326.52 (124.65)	295.46 (22.34)	556.39 (233.59)	305.23 (45.34)
Len%	103.25 (5.40)	112.43 (4.10)	120.67 (17.65)	118.42 (3.59)
AvgRisk (m^{-1}))	73.27 (25.12)	24.23 (4.70)	45.09 (4.62)	31.20 (13.49)
AvgLS (N)	3.14 (1.37)	2.15 (1.44)	3.10 (2.39)	2.40 (1.62)
AvgTS (N)	0.03 (0.01)	0.02 (0.00)	0.03 (0.01)	0.02 (0.01)
AvgCC ($rad/m/s$)	162.42 (63.97)	120.30 (43.96)	169.03 (58.99)	124.18 (50.31)

Figure 9.17: Results of the experiments with the whole set of the MoVeME benchmarks.

sentation in the global subsystem. Additional information in this representation are used only in the parallel parking experiment, where the advised maximum speed is reduced in proximity of the parking lot (this also gives more time to the planner to compute an effective plan to the goal pose). Specifically, the following systems are tested: System H-DWA, System H-ND, System H-DTT, System H-DBT.

System H-DWA is the less reliable, with respect to the success rate: this is mainly due to the fact that it uses a randomized approach. System H-ND succeeds in all the scenarios it is designed for, i.e., excluding the parallel parking maneuver and the constrained forward-only left-only scenario. System H-DTT succeeds in almost all scenarios, but the main drawback is the long time needed for planning (the maximum speed has been reduced, in order to decrease the *TS* metric). Finally, System H-DBT can adapt to the situation and obtain the high performance of the fast methods (i.e., H-DWA and H-ND), while increasing the success ratio (i.e., in particular in the two critical scenarios mentioned above): most notably, while running through wide straight corridors, System H-DBT navigates using trajectory arcs that are either DWA behaviors or ND behaviors, without a precise preference (actually, in this situation they behave similarly); however, in narrow spaces, the ND is chosen more frequently.

Beside the differences described above, the four methods are well-performing and other differences are not relevant in the evaluation. The only additional consideration can be done regarding the lower *Len%* and the higher *AvgRisk* presented by the System H-DWA: the limited look-ahead and the utility function do not perform well in keeping a desired lateral distance from the obstacles, instead, they tend to shorten the followed path (at the cost of correcting maneuvers, due to obstacles behind the corners). As a final note, the values given in Figure 9.17 are averaged only over the successful experiments.

Summarizing, the results show that the versatile System H-DBT presents the highest success ratio, while keeping its performance close to the best.

9.5 Summary

In this chapter, we used the MoVeME evaluation framework described in Chapter 8, in order to compare different motion systems in different situations and while achieving different kind of goals. The definition of objective and quantitative performance metrics allows to obtain assessment and analyses that are very difficult or not possible by a simple qualitative inspection of the robot behavior. Moreover, the definition of a set of benchmarks, i.e., well-defined environments and specified tasks, that encompass both common situations and critical tasks, allows to compare motion systems from a general point of view and allows to analyze the behavior of different systems in some critical situations.

All systems are specific instances of the general System H-DBT, that includes the local subsystem described in Chapter 5 and the global subsystem described in Chapter 6. One of the benefits of the use of such a generic system is that it is possible to derive many single methods by appropriately setting some internal parameters of the system. Moreover, System H-DBT allows to analyze the behavior of *single components* of well-known algorithms in the literature, without the need to test them as a whole.

Besides the particular aspects studied in the different experiments, the overall result shown in this chapter is that a flexible system, that can autonomously choose among different algorithms and adapt to the specific situation (context) in which it operates, has the highest average performances. The proposed evaluation framework and the general motion system provide, in our view, a significant contribution in the definition of a system that can adapt to specific situations, by automatically tune its internal parameters and cope with the environment challenges.

Conclusions and future work

Conclusions

In this thesis, we address the robot motion problem, both from a global perspective, by investigating world representations, path-planning and contextual reasoning, and from a local perspective, where trajectory planning and adaptation, as well as behavioral approaches, such as obstacle avoidance techniques, are exploited.

The robot motion problem has been studied from the very beginning of mobile robotics, but is still an open research area: for this reason, there has been a large proliferation of different approaches, focused on particular instances of the problem. The main goal of this thesis was to find a common ground in such a way that all algorithms and methods, regarding mobile robots and vehicles motion problems, could be embedded into a methodical framework. The greatest challenge was to find a common point of view that allows to merge (local) planning techniques with obstacle avoidance techniques or other kind of reactive behaviors.

Therefore, one of the main contribution of this thesis is the formalization of a novel algorithm, the Dynamic Behavior Tree, that allows to include both directly specified maneuvers and reactive behaviors. In this algorithm, different behaviors are automatically selected and mixed as the tree grows. In addition, the DBT comprises many of the RRT variations that can be used to increase the system performances. The benefit of such a general approach is twofold: on the one hand, it allows to test and experiment the single characteristics of different methods, in such a way that it is possible to understand their effects and peculiarities as separate components; on the other hand, this can be considered as the basis for a versatile system that is able to autonomously adapt its internals with respect to the situation and the requested task.

Moreover, the need of an objective evaluation of the motion systems for robots and autonomous vehicles, only recently gained significant attention by the research community. Without widely accepted performance metrics, it becomes difficult to choose the “right” motion system with respect to an application, or even to compare different solutions. For this reason, the second important contribution in this thesis is the definition of an evaluation framework that features both a set of performance metrics, that address the most common issues that should be faced by motion systems, as well as a set of benchmarks that should be used as test-beds, in order to analyze the motion systems both in typical situations and while performing difficult tasks. In this way, one can obtain a general assessment about benefits and drawbacks of different approaches, and understand their behavior in specific situations.

Finally, along the way, as intermediate steps to the thesis goals, other contributions have been presented, as follows:

- A local method cannot work appropriately in large and complex environments, without the presence of a global planner that is able to guide the local methods using global knowledge about the environment. Regarding this topic, we address the problems that can arise from this decomposition in global/local subsystems, and present a solution that makes use of "hints" attached to the global representation of the environments. Moreover, we present a method to compute a hybrid topological/geometric representation, based on a virtual frontier-based exploration, that is also easily extensible to 3D.

- An extended goal definition is also given, that allows for the specification of fuzzy, symbolic and roughly defined global goals. Moreover, when used for defining the local goal, it gives a larger freedom to the local subsystem, that can optimize the local trajectory without being constrained to unnecessarily fixed way-points.

Future work

The aim of this thesis is to make a step forward in the problem of robot motion. However, such a problem, although it has been addressed for years, is still far from being solved. The definition of the framework for robot motion systems presented in this thesis shows that there are still many open points in this topic.

Concerning the global level, learning techniques can be exploited for automatic context discovery, hints for the local algorithms and possibly also semantic labels that should simplify the interaction with human beings (see, e.g., Rottmann et al, 2005; Dornhege and Kleiner, 2007). For example, the local goal hints can be discovered by using procedures similar to the value iteration that are commonly used by the Reinforcement Learning community. Moreover, the representation should allow for 3D modeling, in order to generalize the application of the framework to a larger set of problems (e.g., to flying robots).

The local motion subsystem still needs the use of heuristics in order to speed up the computation, and become suitable for on-line use. Filtering techniques, based on heuristic argumentation, have been presented, that are able to reduce the number of generated trajectories. However, these are often ad-hoc techniques, mainly applied to precise trajectories and are difficult to generalize to reactive behaviors. Context detection and adaptation can be exploited in conjunction with learning techniques: for example, some past work try to learn the most efficient navigation policies together with context classification (Coelho Jr et al, 1998). Moreover, the use of the layered learning approach presented in this thesis, to tune the large amount of parameters of the DBT algorithm can lead to sub-optimal solutions. Other smarter techniques should be tested (e.g., Cherubini et al, 2009).

Finally, uncertainty in execution of a motion (e.g., a reactive behavior) should be used explicitly to model the possible set of trajectories that can be followed by the system. Moreover, the uncertainty management could be used to reduce computation, allowing the framework to use only a rough estimate of the results of a reactive behavior execution, rather than forcing the computation of its exact evolution in time. Very little work has been done in this direction.

Part IV

Appendices

A

The OpenRDK framework

In this Appendix, we describe the OpenRDK software framework for robotic applications, that has been developed primarily by the author of this thesis and is currently actively used in the RoCoCo Laboratory of "Sapienza" University of Rome and in the Intelligent Control Group of University of Madrid. The OpenRDK has been used to implement, develop and test all software modules that are described in this thesis. For details on the OpenRDK framework, refer to Calisi et al (2008b), Calisi et al (2008a) or the OpenRDK website[1].

A.1 The OpenRDK architecture

OpenRDK is written in C++ and it runs on Unix-like operating systems (Linux, OS X). The main entity is a software process called **agent**. Deployment usually consists of multiple agents across different machines. A **module** is a thread inside the agent process; modules are instantiated dynamically at run-time. An agent **configuration** is the list of the modules to instantiate, together with the value of their parameters and their interconnection layout. The agent configuration initially specified in a configuration file. Modules communicate using a blackboard model: each agent has a **repository**, where modules publish some of their internal variables (parameters, inputs and outputs), called **properties** (see Figure A.1). A module defines its properties during initialization and it can access every other modules' properties, in the same process or in remote hosts, through a global URL-like addressing scheme. Access to remote properties is transparent from the module perspective

---

[1] `http://openrdk.sf.net`

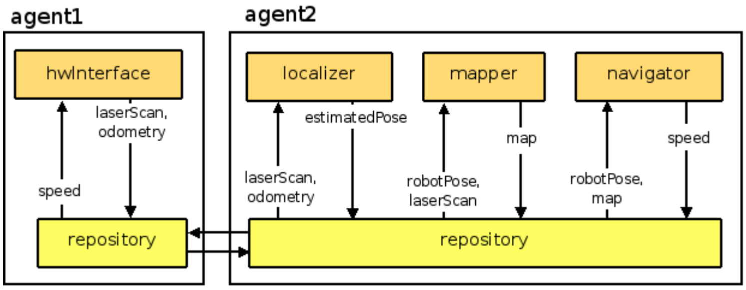


Figure A.1: An example of four modules in the OpenRDK framework. The modules communicate using a centralized structure called repository. The property sharing mechanism allows to access to remote properties in a transparent way.

and it reduces to (regulated) shared memory in the case of local properties. Special queue properties exist to provide producer/consumer behavior (local and remote).

Concurrency model OpenRDK uses a multi-process multi-threaded model. In fact, using callback functions requires a discipline of programming that OpenRDK typical users does not have. The multi-threading solution is a good compromise between efficiency and simplicity of use, although it requires an infrastructure for concurrent data access and synchronization. In the proposed concurrency approach, it is necessary to consider two aspects: real-time requirements and deadlocks with memory locking mechanisms. For the first issue, currently OpenRDK uses POSIX Threads; therefore, hard real-time behavior is not supported. Indeed, the framework is designed for high level functionalities; when real-time is a requirement, such as for fast feedback controllers, the typical approach supported by OpenRDK is to develop the real-time component as a separate entity and make OpenRDK communicate higher level information to it. Accessing shared memory with locking mechanisms can introduce undesired deadlocks. OpenRDK provides for a global locking mechanism that can be used to avoid deadlocks when accessing simple data types. Indeed, in this case the single module is not responsible for locking/unlocking and deadlocks are prevented. However, when data to be exchanged are large (e.g., maps or images) module locking may be necessary and in these cases the implemented solution does not implement a verification mechanism that guarantees deadlock-free behaviors. Nonetheless, these situations are typically limited in a robotic application and deadlock on complex data sharing can be manually verified. At run-time, each module (thread) is typically waiting for some event(s) to happen: new data to consume in a queue, the change of a property value, a timeout, etc.

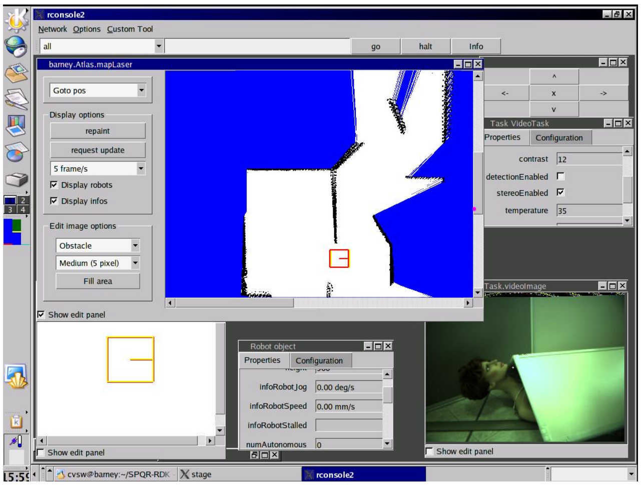
Repository, properties and URLs The repository is the place where modules publish the data they want to share with others. Properties can be inputs, outputs, state, or parameters for the module (there is no hard distinction enforced). Each property has a URL of the type `rdk://agent/module/property`; the agent name is different from the host name as there can be more agents on the same host. By referring to such an URL, modules can transparently access data on any other agent; the repository will take care of establishing connections and negotiating with the remote host.

Property links The mechanism of "property links" is analogous to Unix symbolic links and introduces a level of indirection that allows the modules to be as decoupled as possible. Without links, two modules need to agree on some well-known location (URL) to access the information to be shared, and this creates an unnecessary coupling between them. With links, a module's input property is linked to another module's output property, so that the two modules do not need to be aware of each other. Links are specified in a configuration file; since the data flow is not hard-coded, modules can be easily re-used for different applications. Links can point to remote properties as well, and this allows to distribute the computation in a way which is completely transparent to the module developer.

Configuration and object persistence An agent configuration is an XML file containing the list of modules to be loaded, a description of their interconnections (property links) and the values of their properties, serialized as explained in the previous section. Two kinds of properties are flagged by the developer as to be saved in the configuration file: those representing the proper parameters of the module, and those representing the state. Therefore, this mechanism has the double function of configuring the modules and offering a way to freeze and restart later the execution of the system.

Queues as object dispatchers OpenRDK implements two models for sharing data between modules: publisher/reader and producer/consumer. Regular properties realize the former, and special "queue" properties implement the latter. Queues are very smart FIFO containers: they support multiple readers; thread-safeness is ensured without object duplication; they own the objects that are pushed into them and take care of garbage collection, by destroying the objects when no reader is interested in them anymore; they allow subscribing modules to listen to particular objects entering in the queue, and to be awoken on that event; they are "passive" objects: no additional thread is required to handle them.

Inter-agent information sharing Accessing remote properties is transparent from the module's point of view: typically, modules read/write their own properties, which are then linked to a remote URL via the configuration file. There are some parameters for tuning the transmission behavior: The subscriber can request a property update every time it changes on the remote repository (ON_CHANGE) and optionally set a minimum interval between two subsequent updates. As an alternative, it may request the update to be sent at fixed intervals (PERIODIC in


rconsole2
Network
Options
Custom Tool
all
go
halt
Info
barney.Atlas.mapLaser
Goto pos
Display options
repaint
request update
5 frame/s
Edit image options
Obstacle
Medium (5 pixel)
Fill area
^
<-
x
->
v
Task VideoTask
Properties
Configuration
Task.videoImage
Robot object
Properties
Configuration
cvsw@barney:~/SPQR-RDK
stage
rconsole2

☑ Display robots
☑ Display infos
☑ Show edit panel
☐ Show edit panel
☐ Show edit panel

contrast 12
detectionEnabled ☐
stereoEnabled ☑
temperature 35

infoRobotJog 0.00 deg/s
infoRobotSpeed 0.00 mm/s
infoRobotStalled
numAutonomous 0

Figure A.2: The RConsole GUI, the OpenRDK tool for remote inspection.

OpenRDK terms). The subscriber can request to use one of two transport protocols: UDP or TCP. OpenRDK also partially implements a data reconstruction layer: some objects can be optionally split in multiple packets and reconstructed in the destination repository. Some objects can be transmitted using LOSSLESS (default) or LOSSY compression. For example, if an image has to be sent to an image processing module, a lossless format has to be chosen. On the other hand, if the property is requested just for visualization purposes, a lossy compression is more suited.

Remote inspection RConsole is a graphical tool for remote inspection and management of modules. We use it both for the main control interface of the robot and for debugging while we develop the software. RConsole was very easy to implement thanks to the property sharing mechanism: it is just an agent that happens to have some module that displays a GUI. Through the reflection used in the repository, graphical widgets visualize the internal module state and allow the user to change their parameters while running. Advanced viewers allow to interact with images and maps, moving robot poses, seeing visual debugging information provided by modules, etc. (see Figure A.2).

Integration with simulators OpenRDK provides modules that allow to connect to both USARSim and, through Player, to Stage and Gazebo. The modules expose the same interface of the real ones, thus resulting in a transparent behavior for the modules that connect to them.

Modules for logging and replaying OpenRDK provides a configurable module that, reading from a sensor queue, is able to write a log file containing the sensor data. This file can be processed off-line using third-party tools or used in conjunction with another module that provides the "playback" feature.

A.2 OpenRDK applications

The OpenRDK framework has been successfully used in a wide range of robotic applications. Our group has a long record in RoboCup competitions. OpenRDK has been extensively used in all competitions in which we have been involved: RoboCupRescue Real Robots, RoboCupRescue Virtual Robots, RoboCup@Home, and also RoboCup soccer Standard Platform League (with two-legged humanoid robots). In particular, the latter league makes use of Aldebaran's Nao humanoid robots. OpenRDK currently runs on the Nao's internal computation unit and our team has developed modules for a humanoid robotic soccer application.

A wheeled robot for exploration and search task Our group is involved in rescue robotics, whose goal is to develop robots to assist human rescuers during emergency operations. The main capabilities needed by such a robot are: to build a map on an unknown environment, to move autonomously in a cluttered scenario, and to report to the human rescuers the interesting features found during the exploration (for example, possible human victims, that are entombed or trapped, or possible treats).

The system has been developed as an OpenRDK agent. The real robot was equipped with two personal computers and two agents run on each of them: in this way, we were able to split the computation on two machines. In particular, the first is responsible for the robot mapping and navigation subsystems, as well as the mission manager module; the second machine contains the modules for vision processing. In this application, one example of property sharing is that the vision module published a queue of "possible human sightings" that was read remotely by the mission manager module on the other PC. By simply substituting the real sensor and robot modules by modules that connected to a simulator, we have been able to test exactly the same software system in both real and simulated scenarios. The simulated rescue scenario allowed us to conduct experiments with a large number of robots in large environments.

Assistive robots The RoboCare Project[2] aims at building a system for assistance of the elderly and the impaired person. Such non-invasive technology should be easily integrated in the environment, be able to interact with the person and to monitor his behavior, and act as a distributed and heterogeneous system. For example, some of the main components are a multi-camera system that can follow the human in the environment and track his position, a wheeled robot that can move in the environment and interact with the human through a human-robot interface, and a PDA that the assisted person can use to interact with the system components. In this project, two OpenRDK agents are involved and interconnected to a pre-existent system. One of them is responsible of managing the mobile robot. It includes modules for localization in a known environment as well as path-planning and dynamic obstacle avoidance. Another OpenRDK agent is running connected with the camera tracking system and is responsible for sending the image data to the PDA and to send the tracked human position to the robot agent.

---

[2] `http://robocare.istc.cnr.it`

RoboCup standard (humanoid) league OpenRDK is not designed for a particular kind of robot (e.g., for a wheeled robot, such as the one used in the above example), because it provides only the infrastructure for module concurrency management, information sharing and generic tools. This leads to a great versatility, that has been demonstrated by using it on a humanoid robot, namely for the RoboCup[3] Humanoid Standard league. Although no module could be re-used from the "exploration and search" or the "assistive robots" application, the OpenRDK keeps its features also in this new environment.

---

[3] `http://www.robocup.org`

B

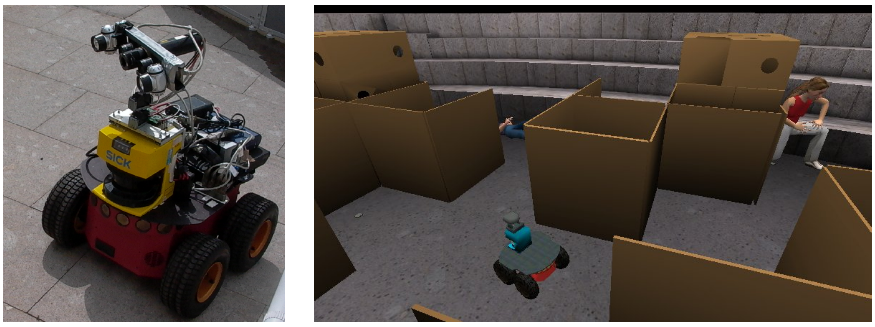
An application: exploration and search missions

This appendix shows an example application for robot motion systems: the exploration and search task. In particular, we adopt a frontier based approach for navigation goal detection and a high-level graph-based language to formalize the high-level exploration plan. The complete system has been presented in Calisi et al (2007c) and in Calisi et al (2007b).

B.1 Introduction

In recent years increasing attention has been devoted to rescue robotics, both from the research community and from rescue operators. Robots can consistently help human operators in dangerous tasks during rescue operations in several ways. Indeed, one of the main services that mobile robots can provide to rescue operators is to work as remote sensing devices reporting information from dangerous places that human operators cannot easily and/or safely reach.

A consistent part of rescue robotic research is focused on providing robots with high mobility capabilities and complex sensing devices. Such kind of robots are usually designed to be tele-operated during the rescue mission and the knowledge about the mission scenario is gathered by the human operator through the displayed output of the sensors (e.g., camera images). Performance evaluation for these kinds of rescue robots is consequently focused on the ability to drive the robot through terrains of measurable complexity (e.g., random step-fields (Jacoff and Messina, 2006)), while taking into account operation constrains, such as lack



(a) the real robot (b) an example of a simulated environment for exploration and search missions

Figure B.1: The real robot that we use for exploration and search missions and an example of a simulated environment.

of visibility of the robot in action, or control by single operator. Communication in this case must be guaranteed at all times, otherwise the robots become out of control.

Another branch of rescue robotics focuses instead on providing mobile bases with a certain degree of autonomy. Autonomous and semi-autonomous robots can process acquired data and build a high-level representation of the surrounding environment. In this mode, robots can act in the environment (e.g., navigate) through a limited interaction with the human operator. This also allow a human operator to easily control multiple robots by providing high level commands (e.g., "explore this area", "reach this point", etc.). Moreover, in case of temporary network breakdown, the mobile robotic platforms can continue the execution of the ongoing task and return to a predefined base position.

In this section, we focus in exploration and search missions, in which a robot is required to explore an unknown environment (in our case, we consider flat environments, due to the mechanical limitations of our mobile base) and search for interesting features during this exploration. In particular, interesting features can be of the class of possible human signs (victims) or other threats (fire, sources of contamination, etc.). The real mobile base that we use for such missions is shown in Figure B.1(a), and an instance of a simulated environment is shown in Figure B.1(b). A detailed description of our exploration and search system can be found in Calisi et al (2007b), and in Calisi et al (2007c).

B.2 Outline of the method

The exploration strategy needs to be very flexible, since the nature of rescue missions can be very different depending on the scenario at hand. Our solution is a hierarchical structure. At a higher level, a complex plan is used to determine the

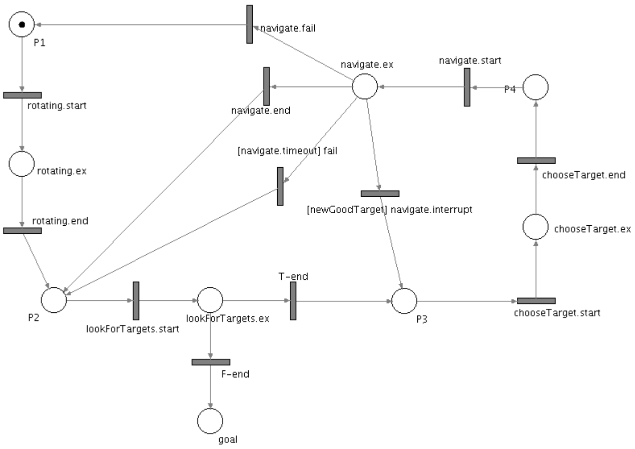

P1
navigate.fail
navigate.ex
navigate.start
P4
rotating.start
navigate.end
[navigate.timeout] fail
rotating.ex
chooseTarget.end
[newGoodTarget] navigate.interrupt
chooseTarget.ex
rotating.end
T-end
P2
lookForTargets.ex
P3
chooseTarget.start
lookForTargets.start
F-end
goal

Figure B.2: An instance of an exploration plan described using the PNP formalism.

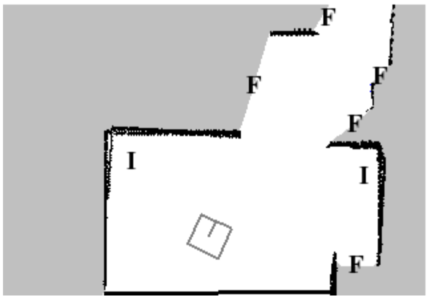


Figure B.3: A possible situation during the exploration mission, in which the system must choose among a set of frontiers (denoted with the letter "F") and a set of interesting places (denoted with the letter "I").

main behavior of the robot. This plan is built using the Petri Net formalism (PNP, see Ziparo and Iocchi (2006) for details), that makes it possible to define qualitative strategic rules to be applied in the mission: for example, if during the navigation the robot detects a new victim, it stops the exploration and goes towards the victim to determine his/her status. Figure B.2 shows an instance of this kind of plans.

The exploration strategy can be divided into two main parts (as in the typical "next best view" algorithm (Pito, 1996; González-Baños and Latombe, 2002)): *i*) decide where to go next, considering that the environment has to be explored as fast as possible and that there are places in which there are more chances to find interesting features; *ii*) move the robot to the target position, that requires the ability to deal with cluttered and rough terrains.

The first decision involves various issues, depending on the kind of environment to be explored. In a rescue mission, a robot has indeed different concurrent goals. In particular, one goal is to explore and build a consistent map of the envi-

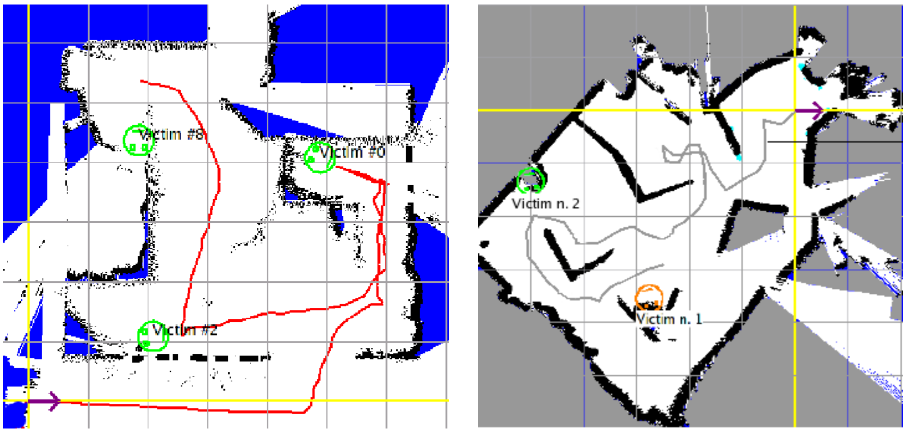


Figure B.4: Two maps that have been built by our autonomous exploration and search system. The rounded smiling faces denote the places where human signs have been found. The path followed by the robot during the mission is also visible.

ronment, another one is to investigate further those areas, already been mapped, where there is the possibility to find victims. In our system, for what concerns the first goal, the choice of which position to explore is based on unexplored frontiers Yamauchi (1997). This method focuses on unexplored areas of the map by taking into account the unexplored frontiers (the bounds between unknown and free space), i.e. the positions that can be seen as "gates" to those areas. Moreover, in order to compute unexplored frontiers, we need only the current map. Another input for the choice of the next target to explore are the "interesting" areas, where it is possible to find victims, i.e., where it is needed a further investigation (e.g., the use of slower algorithms to process data). Figure B.3 shows a situation in which the union of a set of frontiers (denoted with the letter "F") and a set of interesting places (denoted with the letter "I") are the input of the decision process. The problem can be seen as a *multi-objective search problem*, as typical of many robotic tasks IROS06 (2006). Thus, we can use standard techniques based on information gain and make them easy to configure in order to take into account different importance weights for the different features to be measured in the environment.

B.3 Exploiting the motion system in exploration tasks

The motion system is responsible to move the robot to the desired place (e.g., the feature of the environment to be analyzed or a frontier to expand the known map). It is considered as a black box by the exploration strategy, that interact with the motion system only by specifying new places to be reached and retrieving as feedback possible failures (due to unexpected situations or obstacle that are invisible to the sensors mounted on the robot). Finally, the exploration plan can interrupt the motion system in order to change the global goal or simply allow the vision-based feature detection components of the system to work on still images.

Two examples of the final map, that can be built using the whole search and exploration system described in this Appendix, are given in Figure B.4.

Bibliography

Amigoni F, Gasparini S (2008) Analysis of methods for reducing line segments in maps: Towards a general approach. In: Proc. of IEEE/RSJ Int. Conf. on Robots and Intelligent Systems (IROS), pp 2896–2901 33

Arya S, Mount DM, Netanyahu NS, Silverman R, Wu AY (1998) An optimal algorithm for approximate nearest neighbor searching fixed dimensions. Journal of the ACM (JACM) 45(6):891–923 23

Beeson P, Jong NK, Kuipers B (2005) Towards autonomous topological place detection using the extended Voronoi graph. In: Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA), Barcelona, Spain, pp 4373–4379 34

Bhattacharya P, Gavrilova M (2008) Roadmap-based path planning. IEEE Robotics and Automation Magazine pp 58–66 33

Borenstein J (1991) The vector field histogram - fast obstacle avoidance for mobile robots. Robotics and Automation, IEEE Transactions on Volume: 7 Issue: 3:278 –288 14, 95

Borenstein J, Koren Y (1989) Real-time obstacle avoidance for fast mobile robots. IEEE Transactions on Systems, Man, and Cybernetics 19(5):1179–1187 14

Bosse M, Newman P, Leonard J, Soika M, Feiten W, Teller S (2003) An atlas framework for scalable mapping. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) 33, 35

Branicky M, Knepper R, Kuffner J (2008) Path and trajectory diversity: Theory and algorithms. In: IEEE International Conference on Robotics and Automation (ICRA) 25

Brock O, Khatib O (1999) High-speed navigation using the global dynamic window approach. In: IEEE Int. Conf. on Robotics and Automation (ICRA), pp 341–346 31

Brock O, Khatib O (2002) Elastic strips: A framework for motion generation in human environments. The International Journal of Robotics Research 21(12):1031–1052 28

Bruce J, Veloso M (2002) Real-time randomized path planning for robot navigation. In: Proceedings of IROS-2002, Switzerland, October 2002 10, 23

Burgard W, Stachniss C, Grisetti G, Steder B, Kuemmerle R, Dornhege C, Ruhnke M, Kleiner A, Tardos JD (2009) A comparison of slam algorithms based on a graph of relations. In: Proceedings of IEEE/RSJ Conference on Robots and Systems (IROS) 72

Calisi D, Nardi D (2009) Performance evaluation of pure-motion tasks for mobile robots with respect to world models. Autonomous Robots xviii, 87, 88, 97

Calisi D, Farinelli A, Iocchi L, Nardi D (2005a) Autonomous navigation and exploration in a rescue environment. In: Proceedings of IEEE International Workshop on Safety, Security and Rescue Robotics (SSRR), Kobe, Japan, pp 54–59 xvii, 10, 24, 49

Calisi D, Farinelli A, Iocchi L, Nardi D (2005b) Autonomous navigation and exploration in a rescue environment. In: Proceedings of the 2nd European Conference on Mobile Robotics (ECMR), Edizioni Simple s.r.l., Macerata, Italy, pp 110–115 xvii

Calisi D, Farinelli A, Grisetti G, Iocchi L, Nardi D, Pellegrini S, Tipaldi D, Ziparo VA (2007a) Uses of contextual knowledge in mobile robots. In: AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, Springer-Verlag, Lecture Notes in Computer Science, vol 4733, pp 543–554 xviii, 75, 77

Calisi D, Farinelli A, Iocchi L, Nardi D (2007b) Multi-objective exploration and search for autonomous rescue robots. Journal of Field Robotics, Special Issue on Quantitative Performance Evaluation of Robotic and Intelligent Systems 24:763–777 xviii, 127, 128

Calisi D, Farinelli A, Iocchi L, Nardi D (2007c) Multi-objective robotic search and exploration in rescue missions. In: Proc. of Fourth Intl. Workshop on Synthetic Simulation and Robotics to Mitigate Earthquake Disasters (SRMED), pp 27–32, electronic proceedings only xviii, 127, 128

Calisi D, Iocchi L, Leone GR (2007d) Person following through appearance models and stereo vision using a mobile robot. In: Proc. of International Workshop on Robot Vision, pp 46–56 xvii

Calisi D, Censi A, Iocchi L, Nardi D (2008a) OpenRDK: a framework for rapid and concurrent software prototyping. In: Proc. of Int. Workshop on System and Concurrent Engineering for Space Applications (SECESA) xviii, 121

Calisi D, Censi A, Iocchi L, Nardi D (2008b) OpenRDK: a modular framework for robotic software development. In: Proc. of Int. Conf. on Intelligent Robots and Systems (IROS), pp 1872–1877 xviii, 121

Calisi D, Iocchi L, Nardi D (2008c) A unified benchmark framework for autonomous Mobile robots and Vehicles Motion Algorithms (MoVeMA benchmarks). RSS Workshop on Experimental Methodology and Benchmarking, Zurich, Switzerland xviii, 87, 88, 95

Calisi D, Iocchi L, Nardi D, Scalzo CM, Ziparo VA (2008d) Context-based design of robotic systems. Robotics and Autonomous Systems (RAS) - Special Issue on Semantic Knowledge in Robotics 56(11):992–1003 xviii, 77

Calisi D, Iocchi L, Nardi D, Scalzo CM, Ziparo VA (2008e) Contextual navigation and mapping for rescue robots. In: Proc. of IEEE Int. Workshop on Safety, Security & Rescue Robotics (SSRR), Sendai, Japan, pp 19–24 xviii, 77

Calisi D, Iocchi L, Nardi D, Randelli G, Ziparo V (2009) Improving search and rescue using contextual information. Advanced Robotics 23(9):1199–1216 82

Censi A, Calisi D, De Luca A, Oriolo G (2008) A Bayesian framework for optimal motion planning with uncertainty. In: Proc. of the IEEE Int. Conference on Robotics and Automation (ICRA), pp 1798–1805 xvii, 10

Cheng P, Frazzoli E, LaValle S (2003) Exploiting group symmetries to improve precision in kinodynamic and nonholonomic. IEEE/RSJ Int Conf on Intelligent Robots & Systems 26, 53, 55

Cherubini A, Giannone F, Iocchi L, Lombardo M, Oriolo G (2009) Policy gradient learning for a humanoid soccer robot. Robotics and Autonomous System - Special Issue on “Humanoid Soccer Robots” 57(8):808–818 117

Choset H, Nagatani K (2001) Topological simultaneous localization and mapping (SLAM): Toward exact localization without explicit localization. IEEE Transactions on Robotics and Automation 17:125–137 33

Choset H, Lynch KM, Hutchinson S, Kantor G, Burgard W, Kavraki LE, Thrun S (2005) Principles of Robot Motion. The MIT Press 3, 11, 12

Coelho Jr JA, Araujo E, Huber M, Grupen R (1998) Contextual control policy selection. In CONALD’98 - Workshop on Robot Exploration and Learning, Pittsburgh, PA, June 1998. 116

Collet T, MacDonald B, Gerkey B (2005) Player 2.0: Toward a practical robot programming framework. In: Proc. of the Australasian Conf. on Robotics and Automation (ACRA 2005) 95

Coolidge J (1952) The unsatisfactory story of curvature. The American Mathematical Monthly 59(6):375–379 91

Coombs D, Murphy K, Lacaze A, Legowik S (2000) Driving autonomously offroad up to 35 km/h. In: Proc. of Intelligent Vehicles Conference 16

Dornhege C, Kleiner A (2007) Behavior maps for online planning of obstacle negotiation and climbing on rough terrain. Tech. Rep. 233, University of Freiburg 83, 116

Dubins L (1957) 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 9

Durham JW, Bullo F (2008) Smooth nearness-diagram navigation. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) 15

F Lamiraux DB, Lefebvre O (2004) Reactive path deformation for nonholonomic mobile robots. IEEE Transactions on Robotics 20(6):967–977 28, 53, 55

Fabrizi E, Saffiotti A (2002) Augmenting topology-based maps with geometric information. Robotics and Autonomous Systems 40(2):91–97, online at http://www.aass.oru.se/˜asaffio/ 72

Feiten W, Bauer R, Lawitzky G (1994) Robust obstacle avoidance in unknown and cramped environments. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), pp 2412–2417 14

Ferguson D, Kalra N, Stentz A (2006) Replanning with RRTs. In: Proceedings of IEEE International Conference on Robotics and Automation (ICRA), Ieee, pp 1243–1248 24

Ferguson D, Howard T, Lykhachev M (2008) Motion planning in urban environments: Part I. In: Proc. of Int. Conf. on Intelligent Robots and Systems (IROS), pp 1063–1069 17

Fernández J, Sanz R, Benayas J, Diéguez A (2004) Improving collision avoidance for mobile robots in partially known environments: the beam curvature method. Robotics and Autonomous Systems 46(4):205–219 95

Filliat D (2008) Interactive learning of visual topological navigation. In: IEEE/RSJ International Conference con Intelligent Robots and Systems (IROS), Nice, France, pp 248–254 34

Fox D, Burgard W, Thrun S (1997) The dynamic window approach to collision avoidance. IEEE Robotics & Automation Magazine 4(1):23–33 14, 95

Fraichard T, Delsart V (2008) Navigating dynamic environments with trajectory deformation. Journal of Computing and Information Technology pp 1–11 28

Frazzoli E, Dahleh MA, Feron E (2002) Real-time motion planning for agile autonomous vehicles. AIAA Journal of Guidance, Control, and Dynamics 25:116–29 22, 59

Freda L, Oriolo G (2005) Frontier-based probabilistic strategies for sensor-based exploration. In: Proceedings of IEEE International Conference on Robotics and Automation (ICRA) 68

Gayle R, Klingler KR, Xavier PG (2007) Lazy reconfiguration forest (LRF) - an approach for motion planning with multiple tasks in dynamic environments. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), IEEE, pp 1316–1323 24

Giralt G, Sobek R, Chatila R (1979) A multi-level planning and navigation system for a mobile robot: a first approach to hilare. In: Proc. of the Int. Joint Conf. on Artificial Intelligence (IJCAI), pp 335–337 33

González-Baños HH, Latombe JC (2002) Navigation strategies for exploring indoor environments. I J Robotic Res 21(10-11):829–848 129

Grisetti G, Tipaldi GD, Stachniss C, Burgard W, Nardi D (2006) Speeding up rao blackwellized slam. Orlando, FL, USA, pp 442–447 72

Grisetti G, Tipaldi G, Stachniss C, Burgard W, Nardi D (2007) Fast and accurate SLAM with Rao-Blackwellized particle filters. Robotics and Autonomous Systems, Special issue on Simultaneous Localization and Map Building 55(1):30–38 33, 35

Grisetti G, Rizzini DL, Stachniss C, Olson E, Burgard W (2008) Online constraint network optimization for efficient maximum likelihood mapping. In: Proc. of Int. Conference on Robotics and Automation (ICRA), pp 1880–1885 33, 35

Guibas LJ, Hsu1 D, Zhang L (2000) A hierarchical method for real-time distance computation among moving convex bodies. Computational Geometry 15(1-3):51–68 23

Guo Y, Qu Z, Wang J (2003) A new performance-based motion planner for nonholonomic mobile robots. In: Messina E, Meystel A (eds) Proc. of Int. Workshop on Performance Metrics for Intelligent Systems Workshop (PerMIS) 93, 94

Heinrich-Meisner V, Lauer M, Igel C, Riedmiller M (2007) Reinforcement learning in a nutshell. In: European Symposium on Artificial Neural Networks (ESANN), pp 277–288 60

Howard TM, Kelly A (2007) Optimal rough terrain trajectory generation for wheeled mobile robots. The International Journal of Robotics Research 26(2):141–166 17

Hsu D (2000) Randomized single-query motion planning in expansive spaces. PhD thesis, Stanford University 18

Hsu D, Latombe JC, Motwani R (1999) Path planning in expansive configuration spaces. International Journal of Computational Geometry & Applications 9(4-5):495–512 18

Hsu D, Kindel R, Latombe JC, Rock S (2002) Randomized kinodynamic motion planning with moving obstacles. The International Journal of Robotics Research (IJRR) 21(3):233–255 18

ICRA07 (2007) ICRA 2007 Workshop on Semantic Information in Robotics (ICRA-SIR 2007), Rome, Italy 76

Indyk P, Motwani R (1998) Approximate nearest neighbors: Towards removing the curse of dimensionality. In: Proceedings of the Annual ACM Symposium on Theory of Computing, pp 604–613 23

IROS06 (2006) Proceedings of IEEE/RSJ IROS 2006 Workshop on Multi-objective Robotics (IROS-MOR 2006), Beijing, China 130

J P Laumond SS, Lamiraux F (1998) Guidelines in nonholonomic motion planning for mobile robots. In: Robot Motion Planning and Control, Springer Berlin / Heidelberg, pp 1–53 8

Jacoff A, Messina E (2006) DHS/NIST response robot evaluation exercises. In: IEEE International Workshop on Safety Security and Rescue Robots, Gaithersburg, MD, USA 127

Kalisiak M, van de Panne M (2006) Rrt-blossom: Rrt with local flood-fill behavior. In: Proceedings of IEEE International Conference on Robotics and Automation (ICRA) 51

Kamon I, Rivlin E (1997) Sensory-based motion planning with global proofs. IEEE Transactions on Robotics and Automation 13(6):814–822 35, 36

Kant K, Zucker S (1986) Toward efficient trajectory planning: the path-velocity decomposition. International Journal of Robotics Research 5(3):72–89 12

Khatib O (1991) Real-time obstacle avoidance for manipulators and mobile robots. In: Iyengar SS, Elfes A (eds) Autonomous Mobile Robots: Perception, Mapping, and Navigation (Vol. 1), IEEE Computer Society Press, Los Alamitos, CA, pp 428–436 14

Khatib O, Brock O (1999) Elastic strips: A framework for integrated planning and execution. In: Proceedings of the International Symposium on Experimental Robotics, pp 245–254 10, 27

Khatib O, Quinlan S (1993) Elastic bands: Connecting path planning and robot control. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), vol 2, pp 802–807 27

Kohl N, Stone P (2004) Policy gradient reinforcement learning for fast quadrupedal locomotion. In: Proceedings of International Conference on Robotics and Automation (ICRA) 61

Koren Y, Borenstein J (1991) Potential field methods and their inherent limitations for mobile robot navigation. In: Proceedings of the IEEE Conference on Robotics and Automation (ICRA), pp 1398–1404 14, 95

Kostov V, Degtiariova-Kostova E (1995) The planar motion with bounded derivative of the curvature and its suboptimal paths. Acta Mathematica Universitatis Comeianae 64:185–226 91

Kuffner J, LaValle S (2000) RRT-Connect: An efficient approach to single-query path planning. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) 19, 21

Kuipers B (1985) The map-learning critter. Tech. rep., University of Texas at Austin 34

Kuipers B, Byun YT (1991) A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations. Journal of Robotics and Autonomous Systems 8:47–63 34

Lamiraux F, Bonnafous D (2002) Reactive trajectory deformation for nonholonomic systems: Application to mobile robots. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), pp 8–13 28

Lamiraux F, Ferre E, Vallée E (2004) Kinodynamic motion planning: Connecting exploration trees using trajectory optimization methods. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), pp 3987–3992 28

Latombe JC (1991) Robot Motion Planning. Kluwer Academic Publishers xv, 3, 10, 33, 65

LaValle SM (1998) Rapidly-exploring random trees: A new tool for path planning. Tech. rep., Computer Science Dept., Iowa State University 19

LaValle SM (2006) Planning Algorithms. Cambridge University Press, also available at http://msl.cs.uiuc.edu/planning/ 3, 12

LaValle SM, Kuffner JJ (2000) Rapidly-exploring random trees: Progress and prospects. In: Algorithmic and computational robotics: new directions, A K Peters, pp 293–208 17, 21

LaValle SM, Kuffner JJ (2001) Randomized kinodynamic planning. The International Journal of Robotics Research 20:378–400 25

Lazanas A, Latombe JC (1995) Motion planning with uncertainty: a landmark approach. Artificial Intelligence 76(1-2):287–317, planning and Scheduling 41

Li TY, Shie YC (2002) An incremental learning approach to motion planning with roadmap management. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), IEEE, pp 3411–3416 24, 25

Lin MC, Canny JF (1991) A fast algorithm for incremental distance calculation. In: IEEE International Conference on Robotics and Automation (ICRA), pp 1008–1014 23

Luca AD, Oriolo G, Vendittelli M (2001) Wmr control via dynamic feedback linearization. IEEE Transactions on Control Systems Technology vol. 10(no. 6):pp. 835–852 52

Lumelsky V, Stepanov A (1987) Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. Algorithmica 2:403–430 35, 36

Lynch K (1960) The Image of the City. MIT Press, Cambridge, MA, USA 34

McCarthy J, Buvač (1997) Formalizing context (expanded notes). In: Buvač S, Iwańska Ł (eds) Working Papers of the AAAI Fall Symposium on Context in Knowledge Representation and Natural Language, American Association for Artificial Intelligence, Menlo Park, California, pp 99–135 75

Meek D (2004) An arc spline approximation to a clothoid. Journal of Computational and Applied Mathematics 170(1):59–77 16

Minguez J (2005) Integration of planning and reactive obstacle avoidance in autonomous sensor-based navigation. In: Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) 31

Minguez J, Montano L (2004) Nearness diagram (ND) navigation: Collision avoidance in troublesome scenarios. IEEE Transactions on Robotics and Automations 20(1):45–59 15, 70, 95

Minguez J, Montano L, Simeon T, Alami R (2001) Global nearness diagram (GND) navigation. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), vol 1, pp 33–39 31, 32

Minguez J, Osuna J, Montano L (2004) A “divide and conquer” strategy based on situations to achieve reactive collision avoidance in troublesome scenarios. In: Proceedings of IEEE International Conference on Robotics and Automation (ICRA), vol 4, pp 3855–3862 15

Minguez J, Lamiraux F, Laumond JP (2008) Motion planning and obstacle avoidance. In: Siciliano B, Khatib O (eds) Springer Handbook of Robotics, Springer Berlin Heidelberg 9

Mirtich B (1998) V-Clip: fast and robust polyhedral collision detection. ACM Trans Graph 17(3):177–208 23

Mjolsness RC, Swartz B (1987) Some plane curvature approximations. Mathematics of Computation 49(179):215–230 91

Montemerlo M, Thrun S, Koller D, Wegbreit B (2002) FastSLAM: A factored solution to the simultaneous localization and mapping problem. In: Proc. of the Conf. American Association for Artificial Intelligence (AAAI), Edmonton, Canada 41

Montesano L, Minguez J, Montano L (2006) Lessons learned in integration for sensor-based robot navigation systems. International Journal of Advanced Robotic Systems 3(1):85–91 31

Muñoz N, Valencia J, Londoño N (2007) Evaluation of navigation of an autonomous mobile robot. In: Proc. of Int. Workshop on Performance Metrics for Intelligent Systems Workshop (PerMIS), pp 15–21 88, 90, 91, 93, 94

Nilsson NJ (1969) A mobile automaton: an application of artificial intelligence techniques. In: Proc. of Int. Joint Conf. on Artificial Intelligence (IJCAI), pp 509–520 32

Ozguner U, Stiller C, Redmill K (2007) Systems for safety and autonomous behavior in cars: The DARPA Grand Challenge experience. Proceedings of the IEEE 95(2):397–412 92

Petti S, Fraichard T (2005) Safe navigation of a car-like robot within a dynamic environment. In: European Conference on Mobile Robots (ECMR) 22

Pito R (1996) A sensor based solution to the next best view problem. In: Proceedings of International Conference on Pattern Recognition, (ICPR), pp 941–945 129

Quinlan S (1994) Efficient distance computation between non-convex objects. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), vol 4, pp 3324–3329 23

Raño I, Minguez J (2006) Steps towards the automatic evaluation of robot obstacle avoidance algorithms. In: Proc. of Workshop of Benchmarking in Robotics, in the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS) 88, 90, 91, 93, 94

Rawlinson D, Jarvis R (2008) Topologically-directed navigation. Robotica 26(02):189–203 35

Reeds J, Shepp R (1990) Optimal paths for a car that goes both forward backward. Pacific Journal of Mathematics 8

Reif JH (1979) Complexity of the mover's problem and generalization. In: Proceedings of the 20th IEEE Symposium on Foundations of Computer Sciences (FOCS), pp 421–427 4

Riedmiller M (1994) Rprop - description and implementation details. Tech. rep., University of Karlsruhe 55

Riedmiller M, Braun H (1993) A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In: Proceedings of the IEEE International Conference on Neural Networks, pp 586–591 55, 61

Rimon E, Koditschek DE (1992) Exact robot navigation using artificial potential functions. IEEE Transactions on Robotics and Automation 8(5):501–518 65

Rosenblatt J (1997) DAMN: A distributed architecture for mobile navigation. Journal of Experimental and Theoretical Artificial Intelligence 9(1):339–360 92

Rottmann A, Mozos OM, Stachniss C, Burgard W (2005) Semantic place classification of indoor environments with mobile robots using boosting. In: Proc. of the National Conference on Artificial Intelligence (AAAI) 116

Sack D, Burgard W (2004) A comparison of methods for line extraction from range data. In: Proc. of the 5th IFAC Symposium on Intelligent Autonomous Vehicles (IAV) 33

Scheuer A, Fraichard T (1996) Planning continuous-curvature paths for car-like robots. In: Proceedings of the International Conference on Robots and Systems (IROS), vol 3, pp 1304–1311 22

Schröeter C, Höchemer M, Gross HM (2007) A particle filter for the dynamic window approach to mobile robot control. In: Proc. of the 52nd Int. Scientific Colloquium (IWK), vol 1, pp 425–430 16, 51

Shmaglit A, Rinat K, Brand Z, Fischler A, Velger M (2006) Autonomous vehicle control and obstacle avoidance concepts oriented to meet the challenging requirements of realistic missions. In: International Conference on Control, Automation, Robotics and Vision (ICARCV), pp 1–6 92

Simmons R (1996) The curvature-velocity method for local obstacle avoidance. In: Proceedings of IEEE International Conference on Robots and Automation, vol 4, pp 3375–3382 11, 14, 95

Spenko M, Iagnemma K, Dubowsky S (2004) High speed hazard avoidance for mobile robots in rough terrain. In: Proceedings of SPIE Conference of Unmanned Ground Vehicles Technology, pp 439–450 15

Stachniss C, Burgard W (2002) An integrated approach to goal-directed obstacle avoidance under dynamic constraints for dynamic environments. In: Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS) 32, 95

Stone P, Veloso M (2000) Layered learning. In: Mántaras RLD, Plaza E (eds) Machine Learning: ECML 2000 (Proceedings of the Eleventh European Conference on Machine Learning, Springer Verlag, Barcelona, Spain, pp 369–381 61

Taïx M, Malti AC, Lamiraux F (2008) Planning robust landmarks for sensor based motion. In: Bruyninckx H, Preucil L, Kulich M (eds) European Robotics Symposium (EUROS), Springer Berlin / Heidelberg, Springer Tracts in Advanced Robotics, vol 44, pp 195–204 41, 72

Takeuchi E, Calisi D, Ohno K, Tadokoro S, Igarashi H, Kinjo T, Takamori T, Matsuno F (2008) Development of RTCs for mobile robots with autonomy and operability - report 2: Obstacle detection modules. In: Proc. of 26th Annual Conf. of the Robotics Society of Japan (RSJ), pp 1F3–08, in Japanese xvii

Thompson A (1977) The navigation system of the JPL robot. In: Proc. of the Int. Joint Conf. on Artificial Intelligence (IJCAI), pp 749–757 33

Thrun S (1996) An approach to learning mobile robot navigation. Robotics and Autonomous Systems 15:301–319 60

Tomomi K, Jun O, Rie K, Takahisa M, Tamio A, Tsuyoshi U, Tsuyoshi N (2003) Path planning for a mobile robot considering maximum curvature, maximum curvature derivative, and curvature continuity. Transactions of the Japan Society of Mechanical Engineers 69(688):3269–3276 91

Tovar B, Guilamo L, LaValle SM (2004) Gap navigation trees: Minimal representation for visibility-based tasks. In: Workshop on the Algorithmic Foundations of Robotics, pp 11–26 35, 36

Tovar B, Murrieta-cid R, Lavalle SM (2007) Distance-optimal navigation in an unknown environment without sensing distances. IEEE Transactions on Robotics 23(3):506–518 35, 36

Turner RM (1998) Context-mediated behavior for intelligent agents. International Journal of Human-Computer Studies 48(3):307–330 76

Ulrich I, Borenstein J (1998) VFH+: Reliable obstacle avoidance for fast mobile robots. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), pp 1572–1577 14, 95

Ulrich I, Borenstein J (2000) VFH*: Local obstacle avoidance with look-ahead verification. In: Proceedings of the 2000 IEEE International Conference on Robotics and Automation (ICRA), pp 2505–2511 58

Urmson C (2002) Locally randomized kino-dynamic motion planning for robots in extreme terrains. PhD Thesis Proposal, The Robotics Institute, Carnegie Mellon University 22, 59, 64

Urmson C, Simmons R (2003) Approaches for heuristically biasing rrt growth. In: Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp 1178–1183 23

Walton D, Meek D (2005) A controlled clothoid spline. Computers and Graphics 29:353–363 16

Werner F, Gretton C, Maire F, Sitte J (2008) Induction of topological environment maps from sequences of visited places. In: IEEE/RSJ International Conference con Intelligent Robots and Systems (IROS), pp 2890–2895 34

Yamauchi B (1997) A frontier based approach for autonomous exploration. In: IEEE International Symposium on Computational Intelligence in Robotics and Automation 68, 130

Zhang Z (1994) Iterative point matching for registration of free-form curves and surfaces. International Journal of Computer Vision 13(2):119–152 72

Ziparo VA, Iocchi L (2006) Petri net plans. In: Proceedings of Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA), Turku, Finland, pp 267–290, bericht 272, FBI-HH-B-272/06 129

Zucker M, Kuffner J, Branicky M (2007) Multipartite RRTs for rapid replanning in dynamic environments. In: Proceedings of IEEE International Conference on Robotics and Automation (ICRA), IEEE, pp 1603–1609 24, 26

Zwynsvorde DV, Siméon T, Alami R (2000) Incremental topological modeling using local voronoi-like graphs. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) 63

Zwynsvorde DV, Siméon T, Alami R (2001) Building topological models for navigation in large scale environments. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) 63