WARSAW UNIVERSITY OF TECHNOLOGY

FACULTY OF MECHATRONICS

Ph.D. Thesis

Maciej Przybylski, M.Sc.

Mobile Robot Motion Planning in a Dynamic Environment

Supervisor
Professor Barbara Putz, Ph.D., D.Sc.

WARSAW 2018

I dedicate this work to my family.
I thank my parents for their support,
beloved wife Paloma for understanding
and daughter Zosia for her uplifting cheerfulness.

Acknowledgements

The work described in this thesis was conducted in part within the project 2012/05/B/ST6/03094 financed by the National Science Centre, and within the grants 504M-1141-0103-00, 504M-1141-0115-000, 504/02803, and 504/03270, financed by the Dean of the Faculty of Mechatronics, which are gratefully acknowledged.

Streszczenie

Niniejsza rozprawa poświęcona jest planowaniu ruchu robotów mobilnych w otoczeniu ruchomych przeszkód, co wymaga wprowadzenia zależności od czasu. W związku z tym zaproponowano zdarzeniowy opis przestrzeni poszukiwań, w którym planowanie ruchu można postrzegać jako synchronizację akcji robota ze zdarzeniami generowanymi przez ruchome obiekty—wjazdem lub wyjazdem z danej komórki mapy 2D.

Przewodnim celem pracy było opracowanie algorytmów wyznaczających ścieżkę bliską optymalnej w czasie obliczeń umożliwiającym działanie robota w świecie rzeczywistym. Postawiony cel osiągnięto za pomocą zaproponowanego w pracy hierarchicznego meta-algorytmu Real-time Switchback, który wykorzystuje przełączanie kierunków przeszukiwania grafu w celu obliczenia heurystyki dla niższych poziomów hierarchii. Ponadto opracowano dwa nowe algorytmy heurystycznego przyrostowego przeszukiwania grafu: optymalny D* Extra Lite i sub-optymalny AD*-Cut, działające szybciej niż ich odpowiedniki—wiodące algorytmy D* Lite i AD*. Nowe algorytmy wykorzystują technikę przycinania drzewa przeszukiwań w celu reinicjalizacji gałęzi, których koszty mogły ulec zmianie w wyniku pojawienia się lub zniknięcia przeszkód.

Celem potwierdzenia koncepcji, zaproponowano i zrealizowano system planowania ruchu robota mobilnego w środowisku dynamicznym. Centralnym elementem systemu jest trójpoziomowy algorytm oparty na Real-time Switchback, który wykorzystuje fakt, że wynik planowania ruchu w przestrzeni pomijającej ruchome przeszkody może być wykorzystany jako heurystyka do planowania ruchu zależnego od czasu. Proponowany algorytm łączy na kolejnych poziomach hierarchii regularne przeszukiwanie mapy za pomocą A*, przeszukiwanie przestrzeni konfiguracji za pomocą AD*-Cut oraz działające w czasie rzeczywistym heurystyczne przeszukiwanie sześciowymiarowej przestrzeni stanów z warstwami czasowymi opartymi na zdarzeniach. Cały system został przetestowany w realistycznej symulacji ruchu robota w pomieszczeniach biurowych.

Chociaż pojedyncze elementy proponowanego systemu można odnaleźć w systemach opracowanych przez ostatnie dziesięciolecia, hierarchiczna kompozycja algorytmów, kompaktowy opis przestrzeni poszukiwań oparty na zdarzeniach oraz dwa nowe algorytmy, D* Extra Lite i AD*-Cut, są oryginalnym wkładem autora w dziedzinie sztucznej inteligencji. Co więcej, proponowane metody i algorytmy mają charakter uniwersalny, dzięki czemu odnoszą się do dowolnego problemu, który może być sprowadzony do poszukiwania najkrótszej ścieżki w grafie (np. planowanie ruchu w grach wideo, czy nawigacja samochodów).

Słowa kluczowe: *hierarchiczne planowanie ścieżki, heurystyczne przeszukiwanie grafów.*

Abstract

In this thesis, the problem of mobile robot motion planning in a dynamic environment is addressed. The presence of moving obstacles introduces time dependency, which significantly complicates the problem. Therefore, in this thesis, an event-based state-time space decomposition is proposed, in which events are state-time points describing a moment of entering or leaving a map cell by a moving obstacle. Then, motion planning among moving obstacles can be considered an action-event synchronization, which is investigated for applicability to robot motion planning (including kinematic and dynamic constraints).

The leading objective of the thesis is to develop motion planning algorithms that provide a good trade-off between optimality and computation time. Thus, paths with bounded sub-optimality that are obtained in a time to allow action in the real world are of interest. This has been achieved using the proposed Real-time Switchback algorithm, a hierarchical search algorithm that utilizes alternating search directions to compute abstraction-based heuristics. In addition, the two new incremental heuristic search algorithms, optimal D* Extra Lite and sub-optimal AD*-Cut, are presented. The algorithms utilize a search-tree cutting technique to reinitialize search-tree branches affected by obstacle appearance and disappearance. Both novel algorithms outperform the state of the art D* Lite and AD* algorithms.

Finally, as a proof of concept, the system for a differential-drive robot motion planning in a dynamic environment is proposed. The core of the system is the three-level hierarchical algorithm based on the Real-time Switchback algorithm, which utilizes the fact that a non-temporal search space omitting moving obstacles can be used as an abstraction for time-dependent planning. This hierarchical algorithm combines regular A* searching in a 2D grid, AD*-Cut searching in a non-temporal state lattice, and a real-time A* running in a 6D state lattice with event-based temporal layers. The entire system was tested in a realistic simulation of an indoor environment.

Although single elements of the proposed system can be identified across systems developed over the decades, the hierarchical composition of the algorithms, a compact event-based search-space description, and the two new incremental search algorithms, D* Extra Lite and AD*-Cut are the author's novel contributions to the field of artificial intelligence. Moreover, the algorithms and an event-based description of dynamic environments are general purpose. Thus, all the proposed algorithms apply to any path-searching problem that can be represented as a graph (e.g., motion planning in video games or vehicle routing).

Keywords: *hierarchical path planning, incremental heuristic search.*

Contents

Nomenclature **13**

1 Introduction **15**
1.1 Robot Motion Planning System in a Changeable Environment 16
1.2 Heuristic Search Algorithms 17
1.3 Search-space Representation for Time-dependent Planning 19
1.4 Other Approaches 19

2 Research Scope **21**

3 Motion Planning for a Mobile Robot: A Review **23**
3.1 Space Representation for Motion Planning 23
3.1.1 Collision Detection 25
3.1.2 Configuration-space Sampling 26
3.1.3 Transitions in a Configuration Space and State Space 27
3.2 Motion Planning as a State-space Search 29
3.3 Sampling-based Motion Planning 33
3.3.1 Probabilistic Roadmaps 34
3.3.2 Rapidly Exploring Random Tree 35
3.4 Other Motion Planning Methods 37
3.5 Conclusions 38

4 D* Extra Lite: Incremental Planning **39**
4.1 Incremental planning 39
4.2 Intuition 42
4.3 D* Extra Lite Algorithm 44
4.4 Discussion of the Algorithm 50
4.5 Example 54
4.6 Benchmark results 57

4.6.1 Planning with freespace assumption . . . 59
4.6.2 Planning on maps with shortcuts and barriers . . . 62
4.6.3 Benchmark results summary . . . 64
4.7 Conclusions . . . 65

5 AD*-Cut: Anytime Incremental Planning 67
5.1 Introduction . . . 67
5.1.1 Anytime Planning . . . 67
5.1.2 Anytime Incremental Planning . . . 69
5.1.3 Conclusions . . . 71
5.2 AD*-Cut Algorithm . . . 71
5.3 Benchmark Results . . . 72
5.4 Conclusions . . . 76

6 Planning in a Dynamic Environment 77
6.1 State-time space definition . . . 77
6.2 Related Work . . . 81
6.3 Event-based State-time Space Decomposition . . . 84
6.4 Principles of a Minimum-time Path Search . . . 87
6.5 Minimum-time Path Search in a Safe Interval Graph . . . 90
6.6 Action-event Synchronization for Mobile Robot Motion Planning . . . 92
6.6.1 Simple Action-event Synchronization . . . 93
6.6.2 Simple Action-event Synchronization for Long Actions . . . 94
6.6.3 Action-event Synchronization with Acceleration Limits . . . 95
6.6.4 Action-event Synchronization with Cost-map Constraints . . . 96
6.7 Variants of a Time-dependent Heuristic Search . . . 98
6.7.1 Real-time Planning . . . 98
6.7.2 Anytime Time-dependent Planning . . . 102
6.8 Experimental Results . . . 102
6.9 Conclusions . . . 103

7 Hierarchical Planning in a Dynamic Environment 107
7.1 Hierarchical Planning: Background . . . 107
7.1.1 Abstraction Hierarchies . . . 107
7.1.2 Refinement Planning . . . 109
7.1.3 Planning with Abstraction-based Heuristics . . . 111
7.1.4 Switchback: Optimal Bottom-up Search . . . 113

7.1.5 Optimal Top-down Search . . . 114
7.1.6 Conclusions . . . 116
7.2 Search Space for Hierarchical Planning in a Dynamic Environment . . . 117
7.3 Hierarchical Planning in a Fully-known Dynamic Environment . . . 118
7.4 Real-time Hierarchical Planning in an Unknown Dynamic Environment . . . 121
7.4.1 Real-time Switchback . . . 122
7.4.2 Incremental Real-time Switchback . . . 122
7.4.3 Experimental Results . . . 125
7.5 Conclusions . . . 129

8 Hierarchical Motion Planning in a Dynamic Environment for a Mobile Robot 131
8.1 Differential-drive Mobile Robot Model . . . 131
8.2 System Overview . . . 134
8.2.1 Two-dimensional Global Grid Search . . . 135
8.2.2 Global State-lattice Search . . . 135
8.2.3 Local Real-time State-lattice Search in a Dynamic Environment . . . 136
8.2.4 Partial Motion Planning Using a Safe Interval Graph . . . 138
8.2.5 Obstacle Motion Detection . . . 140
8.2.6 Trajectory Tracking Algorithm . . . 140
8.3 Experiments . . . 141
8.4 Conclusions . . . 145

9 Summary 147
9.1 Main Contributions . . . 147
9.2 Conclusions and Future Work . . . 150

References 151

List of Algorithms 163

Index 164

Nomenclature

$\mathcal{A}$	robot
$\mathcal{A}(q)$	robot at configuration q
$\mathcal{A}(s)$	robot at state s
A	action space
a	action, specifically, a_{s_1,s_2} is an action transiting a robot from state s_1 to state s_2
arc	directed-graph edge
digraph	directed graph
dof	degrees of freedom
search space	space in which a solution of a given problem is sought; depending on a particular problem, this could be either a configuration space, a state space, or a graph
$\mathcal{C}$, C-space	configuration space: a set of configurations
$\mathcal{CT}$	configuration-time space
E	set of graph edges (or arcs for a digraph), specifically $E(G)$ for graph G
e	graph edge (or arc for a digraph)
G	graph $G = (V(G), E(G))$
Γ	collision-free plan consisting of actions
$\gamma(a)$	transition function that returns an action end state
$\mathcal{O}$	obstacle region of a workspace
$\mathcal{O}(t)$	obstacle region of a workspace at time t

$\mathcal{O}_i$	i-th obstacle region
$\mathcal{O}_i(t)$	i-th obstacle region at time t
ω	rotational velocity of a robot
$\phi(\cdot)$	homomorphism $\phi(v)$ for graphs or state abstraction mapping $\phi(s)$ for states
$\phi_E(\cdot)$	edge homomorphism $\phi_E(e)$ for graphs or action-abstraction mapping $\phi_A(a)$ for actions
Π	collision-free path consisting of states or graph nodes
q	configuration
S	state space: a set of states
s	state
S^1	$S^1 = [0, 2\pi)$, where 0 and 2π are glued to represent robot orientation
$\mathcal{ST}$	state-time space: a set of state-time, (s, t), pairs
T	set of time points
t	time stamp
$\tau(a, p)$	motion primitive (a continuous trajectory) represented by an action a an parametrized with p
$\tau(p)$	motion primitive (a continuous trajectory) parametrized with p
V	set of graph vertices (nodes), specifically $V(G)$ for graph G
v	graph vertex (node) or longitudinal velocity of a robot
$\mathcal{W}$	workspace, such that $\mathcal{W} \setminus \mathcal{O}$ is a free space

1. Introduction

Recent progress in robotics allows using robots in unstructured and dynamically changing environments (e.g., unmanned ground vehicles, unmanned aerial vehicles, autonomous cars, personal robots, robotic vacuum cleaners, and collaborative robots). A robot working in such an environment should react quickly to occurring changes. At the same time, it is expected to perform an optimal (or near-optimal) plan, which is a nontrivial problem. Therefore, fast and robust algorithms for motion planning in dynamic environments are highly required.

The work of a robot in a real environment entails such aspects as environmental variability and lack of full knowledge. One of the major problems in robotics is planning in an unknown or a partially-known environment (i.e., a new knowledge is gathered during a plan execution, which induces a need for re-planning). Such problems are addressed by incremental search algorithms. Although incremental search algorithms are often referred to as planning in a dynamic environment, most are designed for planning in a static environment. (Incremental search algorithms typically treat all obstacles in an environment as stationary objects.)

In this thesis, the term *static environment* refers to objects that will not move within a predictable time horizon (i.e., objects that cannot move or can change position sporadically, such as office chairs). The term *dynamic environment* refers to both sporadic changes of static objects and changes induced by moving obstacles (e.g., people, cars, and other robots). Planning among moving obstacles introduces a time dependency; therefore, it will also be referred to as *time-dependent planning*. Finally, wherever a particular type of environment variability is not important, the term *changeable environment* will be used.

In both static and dynamic environments, a robot may possess full, partial, or no knowledge about the environment. In this thesis, the problems of planning in an unknown static environment and planning in an unknown, partially-known, and fully-known dynamic environment are addressed.

1.1 Robot Motion Planning System in a Changeable Environment

A system for a robot working in a changeable environment must perform several activities, such as sensory data acquisition, localization, moving-obstacle tracking, global path planning, and local collision avoidance. All of these activities must be performed under time constraints; the more time is consumed by sensory data processing, the less time is left for planning. Typically, to fulfill time requirements, some activities can be performed in parallel, for example, map updating and path planning or global path planning and local collision avoidance. However, a major efficiency gain can be achieved using efficient path-planning algorithms that are designed to work in a dynamic environment.

Most modern robotic systems have a layered architecture. Typically, layers reflect the temporal decomposition of a robotic system (e.g., path planning, local obstacle avoidance, emergency stopping, and motor speed control), where each layer is running with a different bandwidth (Fig. 1.1) [1, Ch. 6].

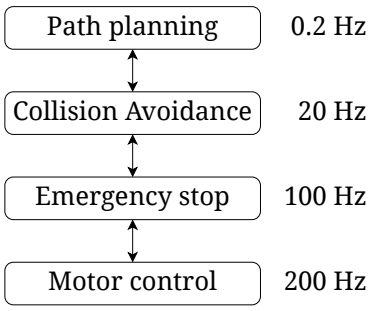


Figure 1.1: Sample temporal decomposition of a robot navigation system. Adapted from [1, Ch. 6].

A more detailed decomposition of system activities for path planning in a changeable environment is shown in Figure 1.2, where activities written in bold are in the scope of the present study (i.e., global path planning, local collision avoidance, and path/trajectory following). These three concerns, with reactive emergency behaviors, are typically considered separately, which simplifies the design and implementation of the robotic system. Furthermore, it is a common approach to utilize a one-direction top-down work-flow, such that higher-level layers provide plans or commands to lower-level layers. However, it may happen that a global planner will generate a plan that is infeasible to execute by a path-following algorithm. Therefore, it is important to design such a planning algorithm or a composition of algorithms that is aware of the robot capabilities and constraints (e.g., kinematic and dynamic constraints or sensory reading inaccuracy).

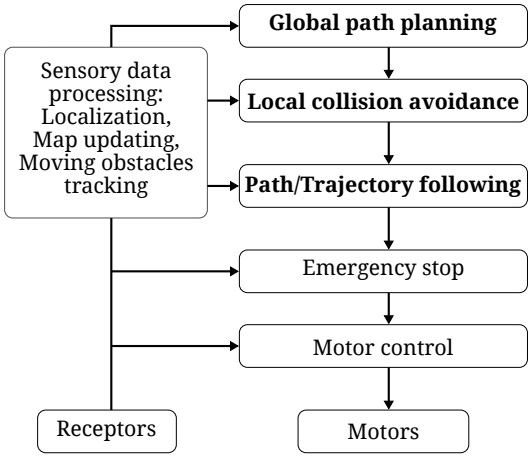


Figure 1.2: Temporal decomposition for path planning in a dynamic environment. The system activities tackled in the present study are written in bold.

Heuristic search, initiated by the seminal work of Hart, Nilsson, and Raphael, who introduced the A* algorithm [2], is one of the basic artificial intelligence (AI) tools to address motion planning problems, including additional concerns, such as limited planning time, plan feasibility, and hierarchical decomposition of planning levels. In the present work, a hierarchical composition of heuristic search algorithms for mobile robot motion planning in a dynamic environment is formally described and thoroughly tested; therefore, heuristic search will be discussed in the following section.

1.2 Heuristic Search Algorithms

In AI, a *robot* is considered an *embodied intelligent agent* [3, Ch. 2], [4], for which interleaving of planning and acting is one of the main problems [5, Ch. 2]. This problem can be solved using a heuristic path search in a graph. Heuristic search algorithms utilize a cost function that guides searching toward a goal, reducing the calculation time.

The optimality of a provided solution is an important property of heuristic search algorithms. It has been shown that A* used with a heuristic cost function (in short, a heuristic) that does not overestimate a true cost (an admissible heuristic) returns an optimal solution (the shortest path). Moreover, sub-optimal or near-optimal heuristic search algorithms based on A* exist. Typically, sub-optimal heuristic search algorithms use an overestimating heuristic that, in some cases, speeds up searching (e.g., anytime search algorithms [6, 7, 8]). The term near-optimal is connected to the limited resolution of a search space (a graph) (i.e., although the path found in a graph is optimal, there can exist a shorter path in a continuous space).

Reactivity and optimality are two desirable properties of robotic systems. However, an optimal solution can be achieved only through global planning, in which computation time depends on the length of a solution. As it is difficult, if even possible, to achieve an optimal solution in real time, it is important to develop techniques that speed up planning.

Heuristic search algorithms can be sped up using several techniques. For example, incremental search algorithms (e.g., Focussed-D* [9], D* Lite [10], IGPPR [11], or MPGAA* [12]) are able to perform quick, optimal re-planning by reusing knowledge from previous searches, which makes these algorithms suitable for planning in unknown or partially-known static environments.

The aforementioned incremental planning algorithms have been successfully used in robotics and video games. The important contribution of this thesis is the D* Extra Lite algorithm [13] that outperforms a state-of-the-art D* Lite in planning in unknown static environments. In the thesis, the AD*-Cut algorithm is also presented, which extends D* Extra Lite to an anytime heuristic search algorithm, similarly as AD* extends D* Lite.

Regardless of the particular domain and problem, it is always possible to reduce the time of planning through a reduction of the search horizon, that is, using a *local search* that can be interrupted before it finds a global solution. Although a local search allows quicker reaction, possibly in real time, in general, it does not give any warranties on solution optimality.

Another serious disadvantage of a local search is that an agent may be trapped in a local minimum. However, this has been overcome in the learning real-time algorithm (LRTA*) [14], which is complete in safely explorable domains (i.e., in domains in which the effects of each action can be reversed). A possible use of real-time search algorithms for planning in a dynamic environment is also investigated in this dissertation. Finally, with the additional assumptions that will be discussed, a real-time search approach has been applied to the task of local collision avoidance.

In complex domains, a simple heuristic cost function may provide too little information to plan quickly. In such cases, it is possible to solve a simplified problem (an abstract problem) and use an abstract plan cost as a heuristic that guides a search in an original search space, which is called *hierarchical planning with abstraction-based heuristics*. The best results are obtained if a search direction is switched between hierarchy levels, which has been utilized by the AltO [15] and Switchback algorithms [16, 17]. This technique is widely used in robot motion planning [11, 18], even in approaches that are not a typical search (e.g., NF1 navigation function in the global dynamic window approach [19]).

However, in the aforementioned algorithms, abstraction-based heuristics are used with, at most, one abstraction level, and the algorithms are not explicitly referred to as a hierarchical search. Another approach to hierarchical planning is the *hierarchical refinement* [20] that solves

a problem in a top-down manner, such that abstract actions are refined at a lower planning level. The main disadvantage of a hierarchical refinement, already mentioned in the context of layered software architectures, is that the refinement of an abstract may not exist; hence, such an algorithm is incomplete.

It should be noted that many robotic systems, by separation of path planning and local collision avoidance, use some form of refinement planning. The final result of this thesis is the system for mobile robot motion planning that utilizes hierarchical planning with an abstraction-based heuristic. The system consists of the following three levels: the forward search in a simplified static environment (A*), the anytime incremental backward search in a static environment including the kinematic constraints of the robot (AD*-Cut) and the real-time, local forward search in a dynamic environment including the kinematic and dynamic constraints of the robot (local A*).

1.3 Search-space Representation for Time-dependent Planning

Planning in an environment with moving obstacles introduces time dependency; thus, a search space needs to be increased by the dimension of time [21]. A generation of a state-time space by regular discretization of time leads to a significant increase in the number of states; therefore, a more compact representation for planning among moving obstacles is desirable (e.g., safe time intervals [22] or obstacle layers presented by the author in an earlier work [23, 24]).

In this thesis, an event-based state-time space decomposition is proposed. In this approach, events that represent obstacles entering and leaving map cells are used to describe obstacle movements. An event-based decomposition generalizes both safe time intervals and obstacle layers methods.

1.4 Other Approaches

Although the approach presented in this dissertation is based on heuristic search algorithms, other efficient motion planning methods, such as sampling-based algorithms, exist. Sampling-based algorithms were proposed as an efficient alternative to exhaustive search algorithms that could struggle in high-dimensional search spaces [25]. To find a collision-free path, algorithms of this kind randomly select configurations from a continuous configuration space. Most sampling-based planning algorithms are non-optimal (i.e., the only objective is to find a collision-free path, regardless of its length). A major disadvantage of these algorithms

is that, for the same problem, they may produce totally different and unpredictable solutions. Moreover, it has been shown that a heuristic search used for dual-arm mobile robot motion planning performs as well as sampling-based methods[26], while providing solutions with a bounded sub-optimality.

Several algorithms related to motion planning in a dynamic environment were developed in the field of autonomous cars, in which a significant boost can be observed in recent years [27, 28]. In this context, a problem of navigation in a changeable environment is considered (local collision avoidance) and has been widely studied. A survey on this topic can be found in [29]. Most local collision avoidance algorithms calculate the best control input (e.g., velocity) that assures collision-free motion for a certain time period (e.g., vector field histogram [30] or dynamic window approach [31]). However, local collision avoidance, if not supported by a global objective function (e.g., a heuristic), may provide sub-optimal solutions or even get stuck in a local minimum [19]. As will be shown in this thesis, a local search using a heuristic provided by a global search can help overcome these problems.

2. Research Scope

The main objective of this thesis is to develop algorithms for mobile robot motion planning in a dynamic environment that provide a good trade-off between optimality and computation time. The planning in a dynamic environment is a complex task that tackles a number of problems. The presented work focuses on two important activities of a robotic system:

- global path planning in a static environment (also considering sporadic changes in static objects positions that occur between planning episodes)
- local path planning in a dynamic environment (with both static and moving obstacles).

It must be noted that each system for motion planning in a dynamic environment requires a robust moving-obstacle detection algorithm; however, this is not discussed here. Other aspects of mobile robotics, such as localization and map building are also out of the scope of the thesis.

Among a number of algorithms for robot motion planning developed over the decades, the following algorithms and methods are especially beneficial regarding the stated objective:

- hierarchical search algorithms that can speed up a search [16],
- incremental search algorithms, able to re-plan quickly by reusing information from previous searches and providing optimal solutions [10],
- anytime search algorithms quickly providing a path with bounded sub-optimality [32],
- local search algorithms, able to provide solutions in a near real time [14],
- compact time representation methods that can be used for planning among moving obstacles [22].

The following theses of this dissertation are proposed.

Proposition 2.1 *A trade-off between optimality and computation time, which allows for mobile robot motion planning in a dynamic environment, can be achieved by a hierarchical composition of heuristic search algorithms of distinct classes (i.e., anytime, incremental, and real-time search algorithms).*

Proposition 2.2 *An incremental and anytime incremental search can be sped up using a search-tree branch cutting technique.*

Proposition 2.3 *An event-based description reduces the size of a search space for minimum-time robot motion planning among moving obstacles.*

The contributions of the present study are five-fold.

- A new incremental search algorithm, D* Extra Lite, has been developed for optimal path planning in a changeable environment (also presented in [13]). Moreover, D* Extra Lite, which utilizes the search-tree branch cutting technique, outperforms the state-of-the-art D* Lite algorithm [10].
- A new anytime incremental search algorithm, AD*-Cut, which extends ideas used by D* Extra Lite, has been developed. In addition, AD*-Cut provides a sub-optimal solution very quickly and constantly improves the solution in the remaining time, outperforming AD* [32], another anytime incremental search algorithm.
- An event-based representation of a search space for time-dependent planning has been proposed, which is an enhancement of obstacle layers [23] that were previously proposed by the author, and is a generalization of other state-of-the-art approaches, such as free intervals [33] and safe intervals [22]. With an event-based description, time-dependent motion planning can be viewed as an action-event synchronization, for which synchronization methods are discussed.
- Based on the author's previous work [24, 34], a Real-time Switchback algorithm was proposed, which is a real-time version of the Switchback algorithm [16], in which higher levels provide a heuristic to lower levels. Switchback and the Real-time Switchback can be considered a general framework for an abstraction-based heuristic search; thus, in the final system for a mobile robot, the three different algorithms, namely, A*, AD*-Cut, and local A*, have been combined at the three consecutive levels of planning.
- The proposed algorithms have been evaluated in the system for mobile robot motion planning among moving obstacles. In addition to the aforementioned algorithms, the system utilizes such approaches as safe, real-time local motion planning [35], and applies an event-based state-space description to dynamically feasible motion planning [18].

This dissertation is organized as follows. In Chapter 3, principles of motion planning for a mobile robot are presented, including state-of-the-art motion planning algorithms. In Chapter 4, a new incremental search algorithm, D* Extra Lite, is presented. Next, AD*-Cut, an algorithm extending D* Extra Lite to an anytime version is introduced (Ch. 5). An event-based representation for motion planning in a dynamic environment is discussed in Chapter 6. Finally, hierarchical methods for motion planning are introduced in Chapter 7 and applied to the task of differential-drive robot motion planning among moving obstacles (Ch. 8). A summary and discussion of future work conclude the dissertation (Ch. 9).

3. Motion Planning for a Mobile Robot: A Review

In this chapter, an introduction to robot motion planning is given, including basic definitions and an algorithm discussion. The chapter begins with a discussion of search-space representations for motion planning, which are a configuration space and a state space (Sec. 3.1). Then, methods for search-space decomposition and sampling are discussed (Sec. 3.1.2). Robot movements in a workspace are reflected by transitions in a configuration space (or a state space), which are described in Section 3.1.3. Transitions in a configuration space can be represented as graph edges, which is a common representation of a search space utilized by motion planning algorithms, among which two major approaches can be distinguished, which are planning as a graph search (Sec. 3.2) and sampling-based planning (Sec. 3.3). Other algorithms related to mobile robot motion planning, such as algorithms for local collision avoidance, are discussed in Section 3.4.

3.1 Space Representation for Motion Planning

In motion planning, a mobile robot $\mathcal{A}$ is considered a rigid body placed in a *workspace* $\mathcal{W} = \mathbb{R}^n$ (e.g., 2D or 3D) [25]. The position of a rigid body in a workspace is the position of all of its points. For a rigid body, as its points cannot change relative positions, it is useful to consider a robot as a shape in 2D space or a volume in 3D space. For example, a wheeled robot can be described as a polygon on a plane ($\mathcal{W}$ is a 2D space) (Fig. 3.1). It is convenient to define the robot local frame of reference as attached to some specific point (e.g., a center of rotation). Thus, a position and orientation of a robot can be described by $q = (x, y, \theta)$ or $q = (x, y, z, \alpha, \beta, \gamma)$ coordinates for 2D and 3D workspaces, respectively, where q is called a *configuration*. In a 2D workspace, a configuration consists of 2D point coordinates $(x, y) \in \mathbb{R}^2$ and a rotation angle θ in $S^1 = [0, 2\pi)$ (Fig. 3.1). A set of all possible configurations is called a *configuration space* or simply a C-space, denoted by $\mathcal{C}$, where $\mathcal{C} = \mathbb{R}^2 \times S^1$.

Typically, a workspace is bounded and may contain an obstacle region $\mathcal{O} \subseteq \mathcal{W}$, such that $\mathcal{W} \setminus \mathcal{O}$ is a free space. If a region occupied by a robot at configuration q intersects with an

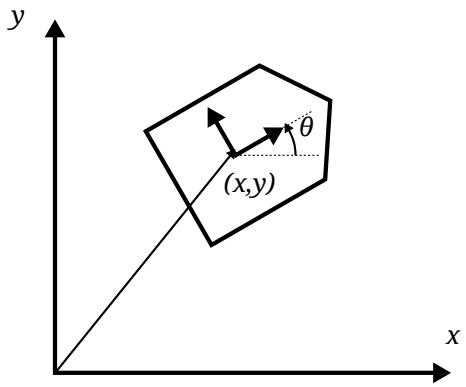
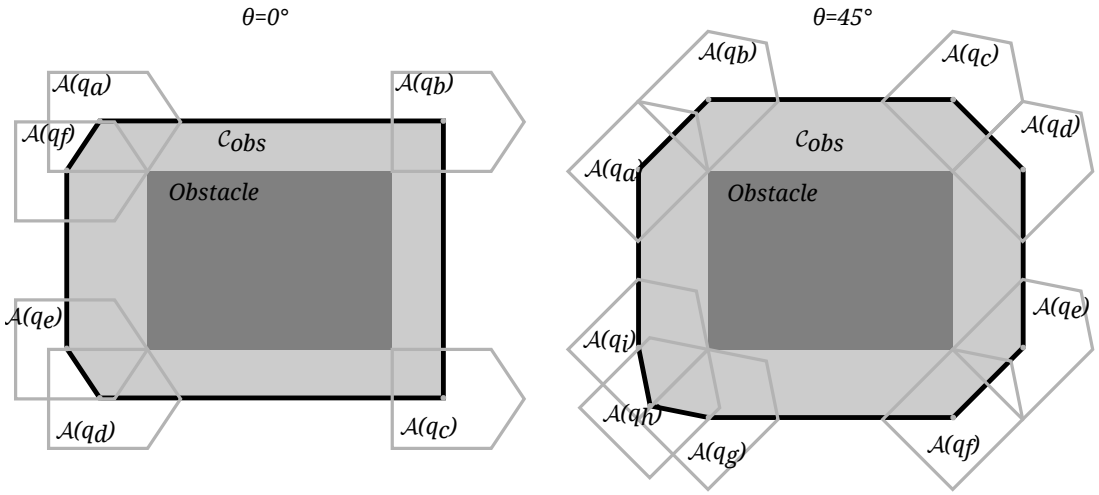


Figure 3.1: Robot in $\mathcal{C} = \mathbb{R}^2 \times S^1$ space.

obstacle region, the robot is in collision. Consequently, a set of configurations exist, $\mathcal{C}_{obs}$, in a configuration space, for which a robot is in collision, namely,

$$\mathcal{C}_{obs} = \{q \in \mathcal{C} | \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}. \tag{3.1}$$

It is important to note that $\mathcal{C}_{obs}$ already considers the shape of the robot $\mathcal{A}(q)$; thus, every configuration q in $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$ must be collision-free. A $\mathcal{C}_{obs}$ region can be constructed as shown in Figure 3.2, that is, by sliding the object contour around the obstacle.



a) b)

Figure 3.2: A C-space construction method for two distinct, fixed robot orientations: a) $\theta = 0°$and b) $\theta = 45°$.

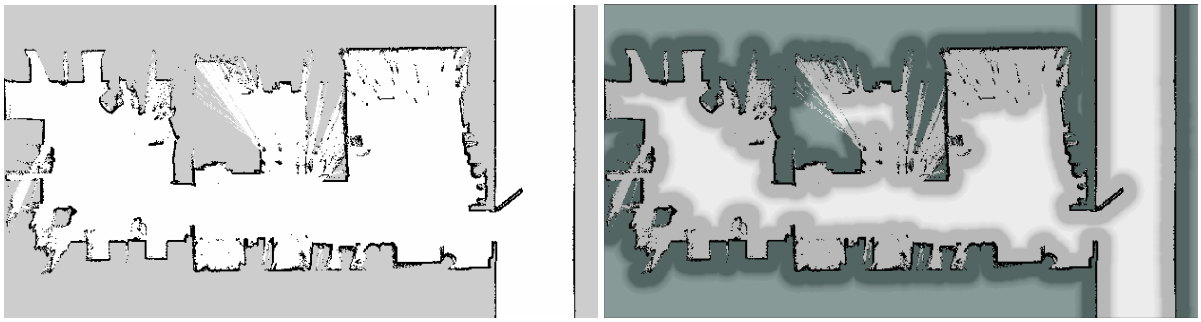
A collision-free path between q_{start} and q_{goal} configurations is a path in a continuous configuration space parametrized with $p \in [0, 1]$, namely, $\tau(p) : [0, 1] \rightarrow \mathcal{C}_{free}$, such that $q_{start} = \tau(0)$ and $q_{goal} = \tau(1)$.

3.1.1 Collision Detection

Although motion planning is held in C-space, usually, there is no need to explicitly construct $\mathcal{C}_{obs}$ or $\mathcal{C}_{free}$. For example, sampling-based planning algorithms perform collision checking only for selected configurations.

Collision checking algorithms for 2D and 3D models have been investigated for decades. A few surveys on this topic can be found in [36, 37, 38]. Such algorithms are usually designed for a particular space, 2D or 3D, and particular object representations (e.g., polygons in 2D space or polyhedra in 3D space). Even though collision checking is a time-consuming operation, some algorithms, such as the Lazy PRM algorithm [39], postpone collision checking until any path is found (if collision along the path is detected the algorithm repeats planning).

A representation of a robot and obstacles by means of polygons or polyhedra allows for an exact collision detection. However, in mobile robotics, an environment map is built upon the range measurements that are represented as point clouds. An object reconstruction from a point cloud is a complex problem. Therefore, a common approach is to store points representing obstacles in occupancy grids, images (2D), voxel-grids (3D), or in tree-like structures with regular grid properties, such as the quad tree (2D) or octree (3D) [40, Ch.14], or in kd-trees that are designed for an efficient nearest-point search [41]. A sample 2D occupancy grid built by a simultaneous localization and mapping (SLAM) algorithm [42] is shown in Figure 3.3a.



a) b)

Figure 3.3: A 2D occupancy grid built with the use of the SLAM algorithm (a) is a typical representation for mobile robot localization and motion planning, where black points denote obstacles, white area denotes free space, and gray area denotes an unknown space). (b) A configuration space represented by a 2D grid with obstacles inflated by the robot circumscribed radius (dimmed regions around black points).

To speed up calculations, collision detection can be performed in two phases: a broad phase used for objects that are far from each other, thus only their bounding regions are checked for collision, and a narrow phase in which exact models are checked [25]. Such an approach is utilized by wheeled robots (like the robot in Figure 3.1), that is, in the broad phase, the

robot contour is approximated with the circumscribed circle. An important advantage of the circumscribed circle used as the bounding region is that $\mathcal{C}_{obs}$ is the same for each orientation θ; therefore, collision checking can be performed in $\mathbb{R}^2$, which is simplified compared to the original, $\mathcal{C} = \mathbb{R}^2 \times S^1$, configuration space (Fig. 3.3b). Another advantage of grid-based configuration spaces is that each grid cell may contain a value representing an occupancy uncertainty or a distance to the nearest obstacle (or a composition), which, in contrast to binary free/occupied information (utilized by most sampling-based planners), eases calculation of a smooth path running at a safe distance from obstacles.

3.1.2 Configuration-space Sampling

As it is impossible to visit all configurations in a $\mathcal{C}_{free}$, a collision-free path search is performed only for selected configurations. Configurations can be picked at random (sampling-based algorithms) (Fig. 3.4a) or can be vertices of a grid (Fig. 3.4b), a special case of a regular lattice. A set of selected configurations with a set of arcs connecting them, which is merely a graph, is called a *roadmap*. Hence, a roadmap constructed from configurations picked at random has been called a probabilistic roadmap, which is also the name of the state-of-the-art algorithm, PRM [43] (discussed in Sec. 3.3.1).

If $\mathcal{C}_{obs}$ can be described with polygons, it is also possible to construct a Voronoi diagram (Fig. 3.4c) or a visibility graph (Fig. 3.4d). While visibility graphs allow the computation of the shortest path that runs through nearby obstacles, planning in Voronoi diagrams results in maximum-clearance paths that run in the middle of a free space.

The relationship between a grid search and probabilistic roadmaps has been investigated in detail by La Valle et al. in [44], who analyzed these two approaches in terms of *dispersion* (i.e., the radius of the largest ball that does not contain any other sample) and concluded that regular grids provide the best possible dispersion. A drawback of regular grids is that the number of samples grows exponentially in dimension number, which makes a grid search intractable for high-dimensional spaces. To overcome the dimensionality issue of a grid search, Lingelbach proposed probabilistic cell decomposition [45], which is a method that utilizes the advantages of the regular neighborhood and probabilistic sampling of the grid. On the other hand, sampling-based methods suffer from other issues. An obstacle region $\mathcal{C}_{obs}$ can take an elaborate shape with narrow passages (Fig. 3.4). Probabilistic methods that focus on filling large uncovered C-space regions may struggle to build a roadmap, passing by such a narrow passage.

Algorithms that explore a configuration space using random sampling (trying to improve sample dispersion) are said to be *probabilistically complete*, that is, if a solution for a given

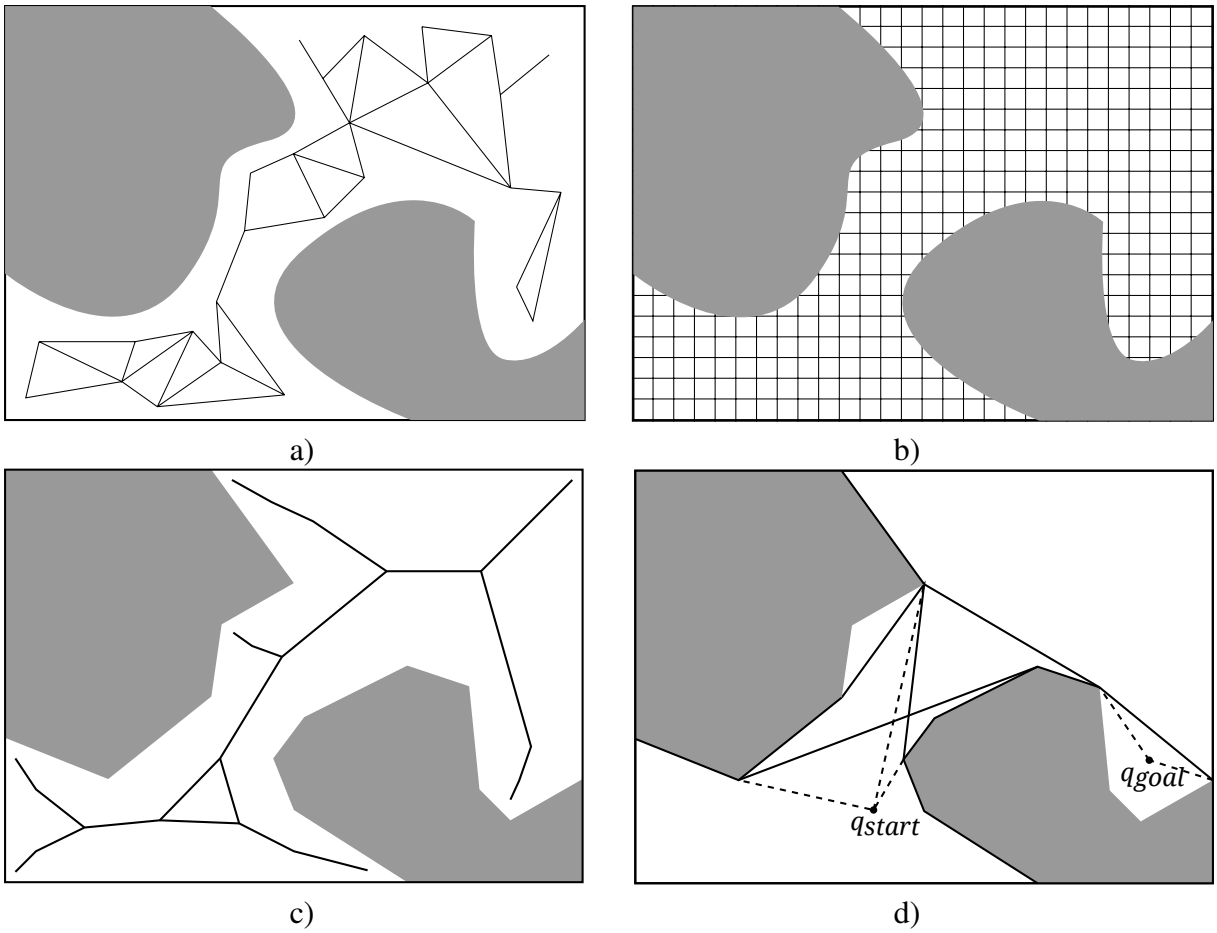


Figure 3.4: Methods of configuration-space sampling: a) a probabilistic roadmap (random sampling), b) a four-connected grid, c) a Voronoi diagram, and d) a visibility graph (the labeled dots are the the start and goal configurations, the solid lines are pre-computed edges, the dashed lines are edges added when the start and goal configurations are known).

problem exists, the algorithm will eventually succeed. On the other hand, if there is no solution, the algorithm will never terminate. For a finite grid-based roadmap, a complete search algorithm can be used (i.e., the algorithm that returns success or failure in a finite time), and if the algorithm is optimal (e.g., A*), it is also optimal up to the grid resolution. However, if the grid-search algorithm subsequently increases resolution after each unsuccessful search, then, if there is no solution, it may run forever, similarly to sampling-based algorithms [46].

3.1.3 Transitions in a Configuration Space and State Space

A single roadmap arc connecting two points in C-space represents transitions in that space. Such a connection can be as simple as a straight line, as shown in Figure 3.4a. However, real robots have limited acceleration, which imposes dynamic and kinematic constraints (*non-holonomic constraints* are imposed on a position and velocity). For example, a car-like robot can move

only along a circle with a limited radius; hence, a path consisting of straight lines may be infeasible. A simple approach to satisfy kinematic constraints is to apply path smoothing in the post-processing phase [25]. Unfortunately, there is no guarantee that a smoothed path will be collision-free and will simultaneously satisfy kinematic or dynamic constraints. A proper solution is to use planning only for feasible transitions between consecutive configurations.

If kinematic and dynamic constraints are considered (which is referred to as kinodynamic planning), the state of a robot is described by a configuration and its first-order time derivative (i.e., $s = (q, \dot{q})$). A set of all possible states is called a *state space* and will be denoted as S. For example, the state of a differential-drive mobile robot is $s = (x,\ y,\ \theta,\ \dot{x},\ \dot{y},\ \dot{\theta})$, though it is more convenient to use $s = (x, y, \theta, v, \omega)$, where v is longitudinal velocity and ω is rotational velocity, defined in the robot local frame of reference. It should be noted that, in the AI community, the term *state* has a more general meaning and refers to any vector (or tuple) consisting of variables that describe a given system [3, 5]; therefore, a simple configuration is also referred to as a state. In this dissertation, the term *state* will be used in that broader meaning.

A transition in a state space is an action $a \in A$, where A is an *action space*. In the context of motion planning, an action represents a motion primitive, that is, a path in a state space, $\tau(a, p) : A \times [0, 1] \rightarrow S$, where p is a parameter with values from $[0, 1]$.

Two main approaches to motion primitive generation can be distinguished as follows. The first approach is used when a motion is allowed only between states that are vertices of a regular lattice (also referred to as a *state lattice*), then a feasible path between state-lattice vertices needs to be computed, which is called local planning. Typically, such local planning requires an inverse kinematic model of a robot or at least a forward kinematic model and a feedback-control algorithm. However, this can be infeasible when a model is complex; in that case, the second approach is preferred. In a given state, it is possible to apply some control input (e.g., velocity or acceleration) for a certain time, resulting in a motion primitive with a final state computed using a forward model. (It should be noted that it is almost impossible to guarantee that the begin and end states of such a motion primitive will belong to lattice vertices.)

While the former approach is utilized by exhaustive search algorithms, the latter approach is common for sampling-based methods (e.g., rapidly exploring random tree (RRT) [47], discussed in Sec. 3.3.2, with an exception to hybrid-state A* [48, 11] that is similar to a state-lattice search), but states expanded during a search do not have to fit the vertices of a lattice. (In hybrid-state A*, a regular grid is used to achieve particular dispersion, that is, only a single state per grid cell can exist.)

Precomputed Motion Primitives

As regular sampling yields a regular neighborhood, it is beneficial for precomputed actions that connect neighboring states. In the most simplified case, motion planning for a mobile robot can be conducted in a 2D state space (only x, y coordinates are represented) in which actions are the arcs of a four- or eight-connected grid (i.e., a robot can move in cardinal or cardinal and diagonal directions to the nearest neighboring state, respectively). The shortest path found by a 2D-grid search will overestimate the true distance (measured along a straight line). To reduce this effect, 16-connected 2D grids can be used. In a 16-connected grid of δ resolution, arc lengths are equal to $1 \cdot \delta$, $\sqrt{2} \cdot \delta$, and $\sqrt{5} \cdot \delta$ for cardinal, diagonal, and remaining arcs, respectively (Fig. 3.5). As shown in [11] the shortest path calculated in a 16-connected 2D grid overestimates the true cost by, at most, 3%.

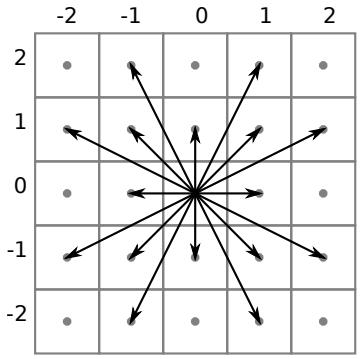


Figure 3.5: Arcs corresponding to node successors in a 16-connected grid.

Although planning in a 2D state space is very quick, paths computed in such a space are not continuous with respect to an orientation; therefore, they are impossible or hard to follow by non-holonomic mobile robots (e.g., differential-drive robots or car-like robots). Planning for such robots is therefore held in a (x, y, θ) state space. Furthermore, with the use of a path-tracking algorithm or an optimization technique [49], it is possible to pre-compute actions satisfying differential constraints that connect neighboring states [50, 18]. Due to a state's regularity, motion primitives generated for some characteristic configurations, as shown in Figure 3.6, can be duplicated for all other states of a state space. This technique was originally proposed in [50], and is referred to as a *state-lattice search*.

3.2 Motion Planning as a State-space Search

A set of states S and a set of applicable actions A can be represented as a directed graph $G = (V, E)$, assuming that a direct state-to-node $s \in S \rightarrow v \in V$ and action-to-edge

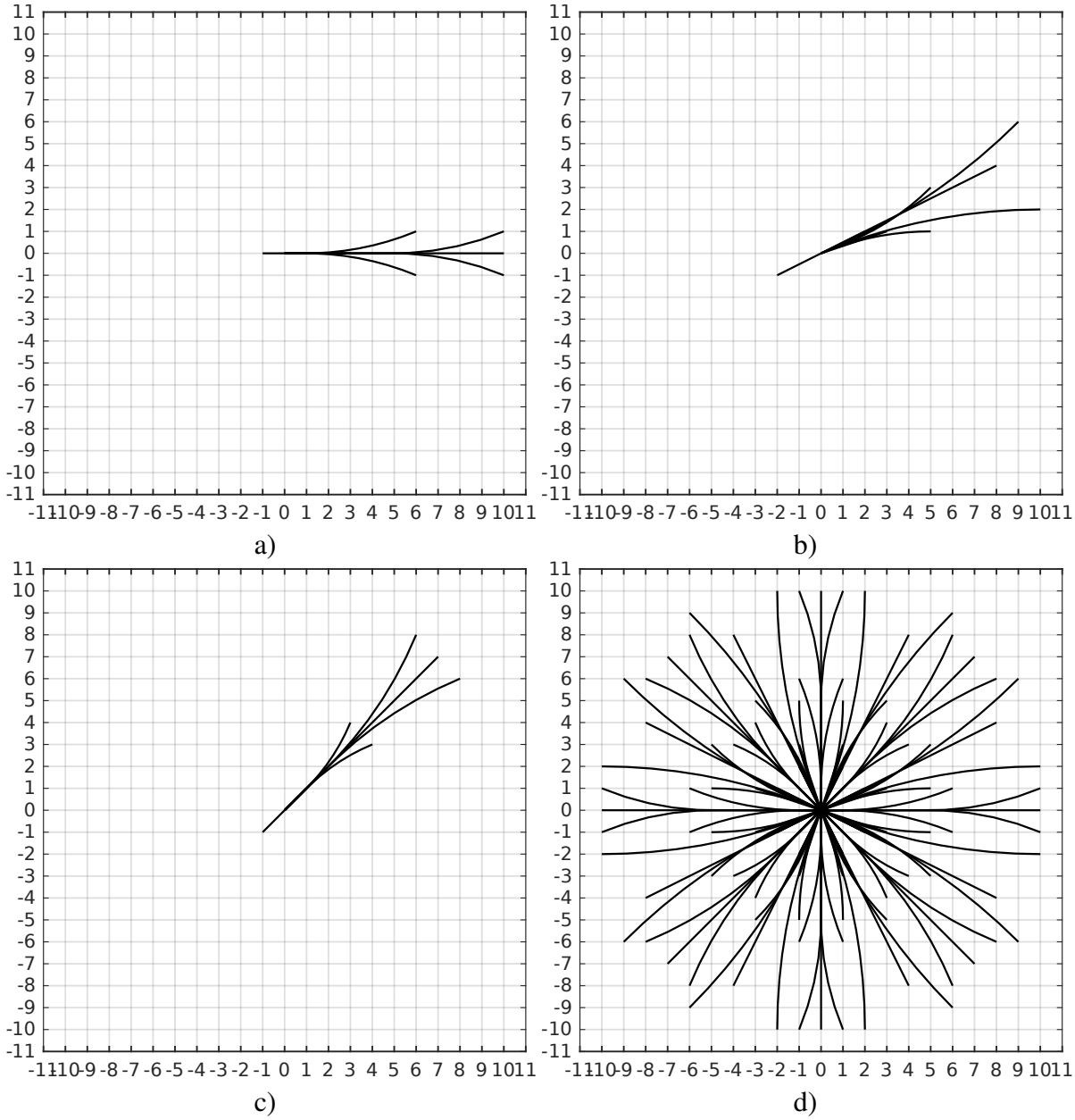


Figure 3.6: Motion primitives generated using the optimization technique described in [49] and implemented in [51] for (a) $\theta = 0°$, (b) $\theta = 22.5°$, (c) $\theta = 45°$, (d) all θ in $\{0°, 22.5°, \ldots, 337.5°\}$. The center of rotation of the robot is situated at $(0, 0)$ point.

$a \in A \rightarrow e \in E$ mapping exist. (Therefore, throughout this thesis, the terms *state* and *node*, and *action* and *edge* are used interchangeably.) Additionally, if a *cost function* is defined for each action, $cost(a_{s,s'}) \equiv cost(s, s') : A \rightarrow \mathbb{R}^+$, that is used as an edge weight, the graph is weighted. A path from s_1 to s_n is a sequence of states, such that an action $a_{s_i,s_{i+1}} \in A$ must exist for each pair of consecutive states $\langle s_i, s_{i+1} \rangle$ $a_{s_i,s_{i+1}} \in A$ must exist.

A path search can be conducted from the starting node (forward search), and from the goal node (backward search), which requires the introduction of the following definitions. A

transition function $\gamma(a_{s,s'}) : S \times A \rightarrow S$ returns the state s' achieved by execution of action $a_{s,s'}$. An *inverse transition function* $\gamma^{-1}(a_{s',s}) : S \times A \rightarrow S$ returns the state s' from which state s can be achieved. With the transition function, we can define a set of *successors* $Succ(s) = \{s' \in S | s' = \gamma(a_{s,s'})\}$ and a set of *predecessors* $Pred(s) = \{s' \in S | s' = \gamma^{-1}(a_{s',s})\}$ (Fig. 3.7).

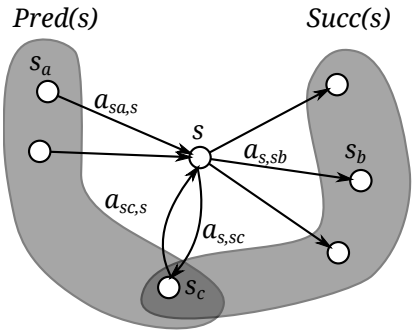


Figure 3.7: State s with successors $Succ(s)$ and predecessors $Pred(s)$ connected by actions, which can be represented as a directed graph.

With the implementation of graph-searching algorithms, nodes hold additional information, such as the following:

- $parent(s)$, a neighboring node that leads to the start state (or goal state, in the case of a backward search),
- $g(s)$, a value that represents the cost from the starting node to s (or the cost from the goal node to s, in the case of a backward search), calculated as follows $g(s) = g(parent(s)) + cost(parent(s), s)$. In the case in which the state s can be omitted, this value is referred to as the g value.

In many path-searching algorithms, common elements can be recognized. In [25] (cf. p. 33) and [52], one can find two generalized (abstract) algorithms depicting common steps of different searching methods. The former algorithm is more suited to typical graph searching, while the latter derives from general action planning problems.

A pseudocode of such a generalized forward search in a state space is shown in Algorithm 3.1. This algorithm has typical elements, such as a list of states to expand ($open\_list$) and a list of states that are already visited (indicated by $visited(s)$). A forward search commences a search from a start state that has the g value initialized with 0, or in general, the lowest cost (lines 5–6 in Alg. 3.1). The search loop is running until a solution is found (line 11 in Alg. 3.1) or there are no more states to expand (i.e., the open list is empty; line 8 in Alg. 3.1). A state expansion (lines 13–18 in Alg. 3.1) involves visiting all successor states. During an expansion, the $g(s')$ value of a successor state is set to $g(s') = cost(s, s') + g(s)$, either when s' is not visited or a new $g(s')$ value is lower than the previous. Whenever a new

$g(s')$ value is set to a state, a $parent(s')$ is also set to s, pointing to the state upon which $g(s')$ was set. It is essential that each state has only one parent state, with the exception of a start state, which is the only state that does not have a parent. As a forward-search algorithm visits new states, it builds a search tree rooted at the start state, in which the parents are the tree branches.

As the FORWARD-SEARCH procedure finishes, a plan can be obtained with the procedure GETPLAN. This procedure retrieves the plan by backtracking from the goal state to the initial state. While backtracking, each preceding action is inserted at the beginning of the plan Π.

With few modifications, the FORWARD-SEARCH (Alg. 3.1) can be easily modified into the BACKWARD-SEARCH algorithm shown in Algorithm 3.2, in which modifications with respect to the FORWARD-SEARCH are marked with (*). The most important change is made in line 13 of the algorithm. In the backward search version, instead of selecting all successor states (i.e., $Succ(s)$), the algorithm must select all states that precede a given state (i.e., $Pred(s)$). A search-direction change imposes modifications in the GETPLAN procedure, where the plan is retrieved starting from the initial state. Thus, each successor is appended to the constructed plan Π.

The discussed algorithms (Alg. 3.1 and Alg. 3.2) expand all successors (predecessors) and add them to $open\_list$ until it finds a goal; thus, both algorithms perform an *exhaustive search* [52]. Moreover, for a finite graph, the algorithms are complete (i.e., they return a solution, if any exists, or return failure). Other properties of the algorithms depend on a particular implementation of an $open\_list$. For example, if $open\_list$ is implemented as a first-in-first-out (FIFO) queue, the presented algorithms work in a breadth-first search (BFS) manner.

Depending on the implementation of the KEY function, if an $open\_list$ is a heap, then two remarkable algorithms can be obtained, namely, the algorithm developed by Dijkstra [53] and A* developed by Hart, Nilsson, and Raphael [2]. With the function $\text{KEY}(s) = g(s)$, the algorithms Alg. 3.1 and Alg. 3.2 are implementations of Dijkstra's algorithm, who showed that, if graph nodes are expanded in ascending order of g-value, then g-values of all expanded nodes will be minimal, and they will be the costs of the shortest paths to these nodes, providing an optimal solution.

An observation made by Hart et al. [2] that a search can be significantly sped-up by the use of the $\text{KEY}(s) = g(s) + h(s, s_{goal})$ (or $\text{KEY}(s) = g(s) + h(s, s_{start})$ in case of a backward search), where $h(s, s_{goal})$ is called a *heuristic function*, was fundamental for a wide range of algorithms based on A* that are the subject of a *heuristic search*, an important branch of AI [54]. Furthermore, Hart et al. showed that, if a heuristic $h(s, s_{goal})$ returns a cost that is not greater than the true cost to the goal, then the A* algorithm provides an optimal solution, and such a heuristic is said to be *admissible*. The all new algorithms proposed in this thesis are based on a heuristic search.

Algorithm 3.1 Forward search.

```
1:  function FORWARD-SEARCH(s_start, s_goal)
2:      for each state s ∈ S do
3:          visited(s) = false
4:          parent(s) = NULL
5:      visited(s_start) = true
6:      g(s_start) = 0
7:      PUSH(open_list, KEY(s_start))
8:      while NOT EMPTY(open_list) do
9:          s = TOP(open_list)
10:         POP(open_list)
11:         if SOLUTIONFOUND(s) then
12:             return success
13:         for all s' ∈ Succ(s) do
14:             if NOT visited(s') OR
15:                     g(s') > cost(s, s') + g(s)   then
16:                 parent(s') = s
17:                 g(s') = cost(s, s') + g(s)
18:                 if NOT visited(s') then
19:                     visited(s') = true
20:                 PUSH(open_list, KEY(s'))
21:     return failure
22: function GETPLAN(s_goal)
23:     Π = ⟨⟩
24:     s = s_goal
25:     while s ≠ NULL do
26:         Π = ⟨s, Π⟩
27:         s = parent(s)
28:     return Π
```

Algorithm 3.2 Backward search.

```
1:  function BACKWARD-SEARCH(s_start, s_goal)
2:      for each state s ∈ S do
3:          visited(s) = false
4:          parent(s) = NULL
5:      visited(s_goal) = true                    ▷ *[a]
6:      g(s_goal) = 0                             ▷ *
7:      PUSH(open_list, KEY(s_goal))              ▷ *
8:      while NOT EMPTY(open_list) do
9:          s = TOP(open_list)
10:         POP(open_list)
11:         if SOLUTIONFOUND(s) then
12:             return success
13:         for all s' ∈ Pred(s) do               ▷ *
14:             if NOT visited(s') OR
15:                     g(s') > cost(s', s) + g(s)   then
16:                 parent(s') = s
17:                 g(s') = cost(s', s) + g(s)
18:                 if NOT visited(s') then
19:                     visited(s') = true
20:                 PUSH(open_list, KEY(s'))
21:     return failure
22: function GETPLAN(s_start)
23:     Π = ⟨⟩
24:     s = s_start
25:     while s ≠ NULL do
26:         Π = ⟨Π, s⟩                            ▷ *
27:         s = parent(s)
28:     return Π
```

[a] * — line modified with respect to a forward search.

3.3 Sampling-based Motion Planning

Sampling-based motion planning refers to methods that explore continuous configuration space by picking configurations at random. To ensure probabilistic completeness, such algorithms try to evenly cover a configuration space with probes. A number of sampling-based algorithms have been developed over the decades (for a survey refer to [25, 55]). Herein, two recognizable algorithms will be discussed, namely, probabilistic roadmaps (PRM) [43] and RRT [47]. The former algorithm is an example of a multiple-query planning algorithm, as a roadmap spanned by the algorithm can be reused for a new query, while the latter is a single-query planning algorithm, as a search tree is rooted in the current state (which is necessary for kinodynamic planning).

3.3.1 Probabilistic Roadmaps

A PRM [43] consists of two phases: a learning phase (a preprocessing phase) that aims to build a roadmap and a query phase that finds a path in the roadmap for given start and goal configurations. As PRM is a multiple-query algorithm, typically, a roadmap is an undirected graph that is easy to use for solving different problems.

The CONSTRUCTROADMAP procedure (Alg. 3.3, adapted from [43]) is the main procedure in the learning phase, where $G = (V, E)$ is an undirected graph, V is a set of graph nodes, and E is a set of graph edges. This procedure incrementally adds random, collision-free configuration q_{rand} to the graph G, until the time reserved for these steps elapses. At this step, it is important to ensure uniform distribution of sample configurations (line 5). For a selected configuration q_{rand}, a set of neighboring nodes N from V is selected (line 6). In the next step, set N is sorted by an increasing distance from q_{rand}, which is not obligatory; however, it is likely that this operation reduces the computation time (in particular the time used by the CONNECT procedure) and helps to construct a compact graph. Each neighbor configuration q_n is then examined for being in the same component as q_{rand}, which prevents cycles in the graph (a roadmap is a forest then). This requirement can also be omitted.

Algorithm 3.3 Roadmap construction.

```
1:  function CONSTRUCTROADMAP
2:      V = ∅
3:      E = ∅
4:      while NOT construction time elapsed do
5:          q_rand = PICKATRANDOM(C_free)
6:          N = GETNEIGHBOURHOOD(q_rand, V)
7:          N = SORTBYINCREASINGDISTANCE(N, q_rand)
8:          V = V ∪ q_rand
9:          for each configuration q_n ∈ N do
10:             if NOT SAMECOMPONENT(q_rand, q_n, G) AND CONNECT(q_rand, q_n) then
11:                 E = E ∪ edge(q_rand, q_n)
```

Finally, if a collision-free path between q_{rand} and q_n exists, which is reported by the CONNECT procedure, a new edge (q_{rand}, q_n) is added to the graph. As the construction of a path between q_{rand} and q_n is a planning problem of itself, the CONNECT procedure is often called a local motion planner; however, it is much simpler than local motion planners used for real-time local collision avoidance. It is desirable to provide a fast and deterministic local planner (i.e., planner that provides the same solution each time). This way it is sufficient to memorize only graph nodes and edges and to calculate configurations that lie along graph edges when necessary. In practice, for holonomic robots (e.g., omnidirectional mobile robots), a local planner can simply provide a line segment in a C-space.

In the original PRM [43], the learning phase splits into the construction step (Alg. 3.3) and the expansion step. The objective of the expansion step is to add more samples in the neighborhood of nodes lying in regions that are difficult (for example, a narrow passage). The "difficulty" of a region is reflected by the weight $w(c)$ (where c is a configuration). In this step, nodes are selected with a probability of $w(c)$. A selected node is expanded by a random-bounce walk in a short range (i.e., for a given configuration a robot follows a direction picked at random and repeats this behavior whenever it hits an obstacle). The end point of such a walk is checked for possible connections with other components of the graph, which is a desirable effect of the expansion step. Small components are removed from the graph, which ends the learning phase.

The learning phase does not consider a start q_{start} and goal q_{goal} configuration. Therefore, at the beginning of the query phase, q_{start} and q_{goal} are examined for connection with graph components. If both configurations can be added to the same component, the shortest-path search can be used to construct a path. Otherwise, a failure is reported.

3.3.2 Rapidly Exploring Random Tree

In contrast to PRM, the RRT algorithm [47] requires start and goal configurations (or states) to be known from the beginning. This is because RRT begins a search-tree construction from the start configuration and gradually adds new configurations that can be picked at random or belong to a deterministic sequence [25, Ch. 5]. A basic idea for a search-tree construction utilized by RRT algorithm is shown in Algorithm 3.4 (adapted from [25, Ch. 5]). The search

Algorithm 3.4 RRT construction.

```
1:  function RRTCONSTRUCT(q_start, K)
2:      V = q_start
3:      E = ∅
4:      for i = 1 to K do
5:          q_rand = PICKATRANDOM(C_free)
6:          q_near = NEAREST(q_rand, T)
7:          q_c = CONNECT(q_near, q_rand)
8:          if q_c ≠ q_near then
9:              V = V ∪ q_c
10:             E = E ∪ edge(q_near, q_c)
```

tree is initialized with a start configuration q_{start} (line 2). For an arbitrarily chosen number of steps K, the following steps are conducted. For a new configuration that is picked at random (line 5), the nearest configuration q_{near} that lies on the search tree is found (line 6). The nearest-configuration search is not restricted to tree nodes V; it may also be any configuration at the edges of the tree, denoted as T. Then, the algorithm examines the connection between q_{near} and q_{rand}. If there is no collision-free connection, the CONNECT procedure returns the last

point that belongs to $\mathcal{C}_{free}$ (line 7). If this point is not already in s, it is added to the tree with (q_{near}, q_c) edge (lines 8–10).

Originally, RRT was designed to perform motion planning for a robot with non-holonomic constraints, including dynamic constraints (i.e., kinodynamic planning) [47, 56]. A full RRT for kinodynamic planning is shown in Algorithm 3.5 (adapted from [25, Ch. 5]). To

Algorithm 3.5 Kinodynamic RRT.

```
 1: function KINODYNAMICRRT(s_start, s_goal, K, Δt)
 2:     V = s_start
 3:     E = ∅
 4:     for i = 1 to K do
 5:         s_rand = PICKATRANDOM(S_free)
 6:         s_near = NEAREST(s_rand, T)
 7:         (s_new, u_new) = SELECTCONTROL(s_near, s_rand, Δt)
 8:         if s_new ∉ S_obs then
 9:             V = V ∪ s_new
10:             E = E ∪ edge(s_near, s_new, u_new)
11:             if ‖s_new − s_goal‖ < ε then
12:                 return success
```

perform kinodynamic planning, the KINODYNAMICRRT has few modifications regarding RRTCONSTRUCT (Alg. 3.4). First, planning is held in a state space S. Second, state s_{new}, which is finally added to the search tree, is obtained by an application of a control input u_{new} (e.g., a left and right wheel speed for a differential-drive robot) for an arbitrarily chosen time period Δt, which ensures that the tree consists of feasible motion segments only. Furthermore, for a robot with non-holonomic constraints, it is difficult to provide such a SELECTCONTROL procedure that would guarantee that s_{rand} will be achieved in Δt; thus, the algorithm reports success when it finds a state near the goal state (line 10). Finally, the edges of the constructed tree, in addition to the end node s_{new}, have to memorize an associated control input u_{new}.

Although, it is almost impossible to achieve $s_{new} = s_{rand}$, it is crucial to pick s_{rand} with a probability of 1 from a random sequence that is dense (i.e., a state space will be uniformly covered by samples), which is necessary to make such an algorithm probabilistically complete. In contrast, a simple approach that selects random control inputs rather than random states will generate many samples near s_{start} and will poorly cover the rest of the state space.

A very important observation on the completeness of kinodynamic RRT planning has been made in [57], which is that kinodynamic RRT-like algorithms with fixed time step and best-input extension are not probabilistically complete.

3.4 Other Motion Planning Methods

To this point, the two most popular motion planning methods, search-based and sampling-based methods, have been discussed, both aiming to perform complete global planning, at least probabilistically. There are also methods that are not complete (i.e., they can get stuck at a local minimum; yet, they are very quick and suitable for local collision avoidance).

One such method is the potential field method that assumes the existence of repulsive and attractive forces that influence robot motion [58]. A sum of these forces ($U(q)$ in Eq. 3.2) results in a vector field with a gradient ($\vec{F}(q)$ in Eq. 3.2) that forces a robot to move in the goal direction (attractive force), while avoiding obstacles (repulsive forces).

$$\begin{aligned} U(q) &= U_{rep}(q) + U_{attr}(q) \\ \vec{F}(q) &= -\vec{\nabla} U(q) \end{aligned} \tag{3.2}$$

The potential field method with an attractive force based on the Euclidean distance to the goal can get stuck in local minima. This issue has been overcome with the use of the NF1 navigation function that is constructed as a backward search from a goal state. In fact, the calculation of NF1 is merely a global grid-based search. Moreover, a potential field generated by repulsive forces can be easily incorporated into a lattice-based search as vertices with inflated action costs, which is a common practice [18, 48]. The inflation of an action cost in the vicinity of obstacles allows achieving maximum-clearance path planning, similarly to the Voronoi diagram; hence, such a potential field is called a Voronoi field [48].

To deal with changes in an environment, global path planning methods are typically supported by local collision avoidance algorithms. Most local collision avoidance algorithms calculate the best control input (e.g., velocity) that assures feasible collision-free motion for a certain time period. For example, a vector-field histogram (VFH) [30] calculates the best motion direction and adjusts the rotational velocity.

The dynamic window approach (DWA) [31], another collision avoidance algorithm, selects the best longitudinal v and rotational ω velocities across velocities that are applicable at a current state (i.e., current velocities and allowed accelerations are considered, which limits the v, ω velocity space to a rectangular window). In this method, input velocities are calculated with a multivariate objective function optimization, in which the following navigation function is maximized [19]:

$$nf(\vec{q}, \vec{v}, \vec{a}) = \alpha \cdot nf1(\vec{q}, \vec{v}) + \beta \cdot vel(\vec{v}) + \gamma \cdot goal(\vec{q}, \vec{v}, \vec{a}) + \delta \cdot \Delta nf1(\vec{q}, \vec{v}, \vec{a}), \tag{3.3}$$

where α, β, γ, and δ are weights, $\vec{q}, \vec{v}$, and $\vec{a}$ are the current position, velocity, and acceleration, respectively, vel is a function rewarding maximal velocity, $goal$ rewards progress toward a local goal, $nf1$ rewards progress toward a global goal, and $\Delta nf1$ rewards dynamics of progress toward a global goal.

The aforementioned collision avoidance method aims to find new control input that is the best with respect to a short time horizon, which makes the sub-optimality more pronounced. This effect can be reduced by a local search in a $(q, \dot{q})$ state space (state consisting of position and velocity), that is limited to the vicinity of a global path. Such an approach is also used in this thesis.

3.5 Conclusions

Basically, motion planning is a problem of finding a collision-free path in a configuration space. If kinematic and dynamic constraints are considered, the problem is solved in a state space which typically consists of configuration and velocity variables. Most motion planning methods are based on either exhaustive lattice search or random sampling. While the advantages of lattice-search methods are completeness and optimality (or bounded sub-optimality), sampling-based methods are better for solving complex problems in high-dimensional spaces. In the context of mobile robots, specifically differential-drive robots, motion planning as a heuristic state-space search on a state lattice is the state of the art. Many real-world applications were developed using this approach (e.g., autonomous car driving systems [18, 48]). Regarding the subject of this thesis, that is, mobile robot motion planning among moving obstacles, state-lattice heuristic search algorithms have been chosen for further analysis and usage.

4. D* Extra Lite: Incremental Planning

This chapter contains a description of the D* Extra Lite algorithm previously published in [13]. The D* Extra Lite algorithm is an optimal incremental search algorithm utilizing a search-tree branch cutting technique, that allows quick reinitialization of parts of a search space that were affected by changes in an environment. Thus, the knowledge from previous searches is reused to speed up re-planning. The discussion of related incremental search algorithms can be found in Section 4.1. A branch cutting technique is presented in Section 4.2. A pseudocode of D* Extra Lite is presented in Section 4.3. A theoretical discussion of the properties of D* Extra Lite is given in Section 4.4. Then, the algorithm is explained on a complex example (Sec. 4.5). Finally, the benchmark results are presented (Sec. 4.6).

4.1 Incremental planning

Goal-directed navigation without accurate knowledge of the environment is a common problem in robotics and video games. As an agent follows a path to a stationary goal, they may discover changes within a certain range of sensors, which will require re-planning. Incremental heuristic search algorithms are beneficial in this context; able to reuse knowledge from previous searches, substantially less computation time is needed for re-planning.

Incremental shortest-path searching algorithms are typically used in a sense-plan-act scheme (Alg. 4.1). During the planning phase a stationary snapshot of the environment is used. Discrepancies between known-map and accurate-map relate to the appearance and disappearance of obstacles. Although in both cases (appearance and disappearance) a previous search can be reused, some techniques may be optimal in some cases and not in others.

The work of Stentz, who developed the D* [59] and Focussed D* [9] algorithms, and that of Koenig and Likhachev, who developed D* Lite [10], are the most recognizable contributions to-date to address the problem of incremental shortest-path planning. All three of the above-mentioned algorithms run backwards, making them especially useful, as only the start-state changes between search episodes, and the goal-state remain unchanged, therefore, a significant part of the explored search-space remain relevant. The general aim behind these algorithms is to repair only the nodes (map cells) in the affected part of the map, that is, unless

Algorithm 4.1 Procedures common for the D* Lite and D* Extra Lite algorithms.

```
 1: function MAIN()
 2:     s_last = s_start
 3:     MAPUPDATE()
 4:     INITIALIZE()
 5:     while s_start ≠ s_goal do
 6:         if NOT SEARCH() then
 7:             return goal is not reachable
 8:         s_start =ACTIONSELECTION(s_start)
 9:         MAPUPDATE()
10:         REINITIALIZE()
11: function SEARCH()
12:     while open-list is not empty do
13:         if SOLUTIONFOUND() then
14:             return true
15:         SEARCHSTEP()
16:     return false
17: function ACTIONSELECTION(s_start)
18:     return argmin_{s'∈Succ(s_start)}(cost(s_start, s') + g(s'))
```

the current state of the agent is achieved by the searching algorithm, in which case, preference is to nodes that will lead to the current state of the agent.

Due to changes in the environment, a part of the search-space may become inconsistent. While one part may include under-consistent nodes with underestimated g values (when new obstacles appear), the other part may feature over-consistent nodes with overestimated g values (when obstacles disappear). In case of underestimated g values, the nodes need to be reinitialized. This allows the algorithm to assign new values, which will most likely be higher. In the D*, Focussed D* and D* Lite, reinitialization occurs during the search. In order to detect an inconsistent node, each node has an additional value, denoted as k value in Focussed D*, and rhs value in D* Lite. Moreover, both algorithms utilize heuristic cost to the agent's current state in order to guide searching. In the searching phase, both algorithms perform the following two operations for each underestimated node. Before the new g value is set, each underestimated node is reinitialized. To assure that reinitialization precedes setting of the g value, the list of open nodes must be sorted using a complex key value (i.e., $min(rhs(s), g(s))$).

A different approach is to reinitialize the affected portion of the map and then to conduct a new search of only that part. As argued by Stentz [9], such an approach is "inefficient when the robot is near the goal and the affected portions of the map have long 'shadows'." This approach, i.e., reinitialization of the entire affected section of the search-space, can be found in the work of Podsedkowski [60, 11], as well as in the Differential A* algorithm proposed by Trovato [61], revisited and extended in [62]. Differential A* may be the most similar algorithm to the D* Extra Lite presented in this paper. Unfortunately, research to-date has neglected to make an experimental comparison of Differential A* with Focussed D* or D* Lite. However, with the

understanding gained from the work on the D* Extra Lite, also stated by Koenig [10], it can be proposed that the reinitialization of the entire open-list before each search episode inhibits efficient functionality of the Differential A* algorithm ($recompute\_OPEN()$ procedure in the pseudocode presented by Trovato [62]).

In order to avoid re-computation of the entire open-list, Focussed D*, D* Lite and D* Extra Lite use a biased key value. A biased key value, in addition to calculated cost g and heuristic cost h, includes k_m value, which grows proportionally to the cost of agent's transition between each search episode. Accordingly, it is ensured that nodes that were pushed to the open-list in the previous and subsequent episodes, will be popped in an order that will satisfy optimality requirements without reordering of the entire open-list.

D*, Focussed D* and D* Lite each run backwards, from the goal to the current state of the agent, which allows for the unaffected search-tree to be easily reused. However, it should be noted that it is also possible for forward-search algorithms to reuse a previously explored search-tree, e.g., LPA* [63] or Fringe-Saving A* [64]. Moreover, adaptive algorithms, which do not reuse search-tree, are continuously running from scratch in a forward direction. These algorithms improve their h values (they learn heuristics from previous searches), which substantially accelerates subsequent search episodes. The basic algorithm for this type is Adaptive A* [65]. Findings strongly suggest that AA* is quicker than repeated A*, and can be faster then D* Lite, however only with the use of buckets in place of heap as a priority queue for open-list managing.

In addition to algorithms using search-tree reuse and adaptive heuristic learning techniques, there are algorithms based on AA* that make use of previously found paths, i.e., Path Adaptive A* [66] and Multi-Path Adaptive A* (MPAA*) [67]. While these algorithms also run forward from the starting state, they can terminate before achieving the goal node. As these algorithms record previously constructed path(s), it is sufficient to construct the path that remains connected with the goal node (it has not been disconnected through changes in the environment). The most complex adaptive algorithm may be Tree Adaptive A* [68], which combines a reusable tree (like D* or LPA* algorithms) with reusable paths, and accordingly, heuristic improving.

All adaptive algorithms mentioned above are limited to freespace assumption. Consequently, although they can manage new obstacles, where there is a shortcut available, their solutions fail to remain optimal. This problem has been solved with Generalized Adaptive A* (GAA*) [69], which in the case of decrease to edge-cost, reestablishes the consistency of h values by performing uninformed backward-search throughout the explored search-space. The recent Multi-Path Generalized Adaptive A* (MPGAA*) [12] algorithm demonstrates the benefits to be gained from the path reuse technique, by combining it with the GAA* algorithm.

It is worth noting that GAA* (and MPGAA*) is similar to the LSS-LRTA* [70]. LSS-LRTA* is a learning real-time algorithm that searches forwards and can be stopped before it finds the global solution. The learning phase is a function equal to the consistency reestablishing performed by GAA*. Without a computation time-limit, LSS-LRTA* undertakes a global search, which makes it comparable to incremental planning algorithms. LSS-LRTA* has been shown to outperform D* Lite in some settings, for a discussion, refer to [70].

4.2 Intuition

In most of incremental heuristic search algorithms, the first search episode is equivalent to the regular A* algorithm, which expands consecutive search-space nodes until it reaches the goal node. This can be in the case of a forward-search (e.g., LPA*, MPGAA*) or a backward-search (e.g., D*, D* Lite). If the algorithm sets parent pointers (for example D* Lite does not), these pointers form a tree with a root in the node from which the search originated; this tree is referred to as the search-tree.

If any change is observed to affect the explored search-space, particularly, an edge-cost $e(s_1, s_2)$ has changed, a part of the visited search-space (a branch of the search-tree) has become inconsistent and must be re-explored. The inconsistent part of a search-tree can be defined as a branch of a search-tree that contains nodes supported by an edge $e(s_1, s_2)$. A node s_2 is supported by an edge $e(s_1, s_2)$ if the node s_1 is a parent of node s_2, furthermore, if a node s_2 is a parent of node s_3 and s_2 is supported by $e(s_1, s_2)$, then s_3 must also be supported by $e(s_1, s_2)$.

The g values of nodes that belong to an inconsistent search-tree branch, are either too high or too low. At that point, all incremental algorithms (such as D*, D* Lite, MPGAA*) distinguish between two situations — when the cost of an edge has increased and when it has decreased.

If the root of the search-tree has not changed, such is the case of incremental search algorithms running backwards (e.g., D* and D* Lite), following observations can be made.

In the situation in which the cost of an edge $e(s_1, s_2)$ decreases, it is sufficient to reopen the s_1 node and continue search. This is owing to the fact that g values in the inconsistent part of the search-tree are higher than should be (nodes are over-consistent). Optimal path-searching algorithms change g value only if $g(s_2) > g(s_1) + cost(s_1, s_2)$, which also functions to prevent the algorithm from re-exploration of consistent nodes. Moreover, in this situation, the affected branch of the search-tree cannot shrink (it can grow or stay unchanged). In Figure 4.1, an example of edge-cost decrease is depicted. As cell $C4$ became free, the cost of corresponding edges $e(s_{C3}, s_{C4})$, $e(s_{B4}, s_{C4})$ and $e(s_{C5}, s_{C4})$ decreased from infinity to one (Fig. 4.1a). As

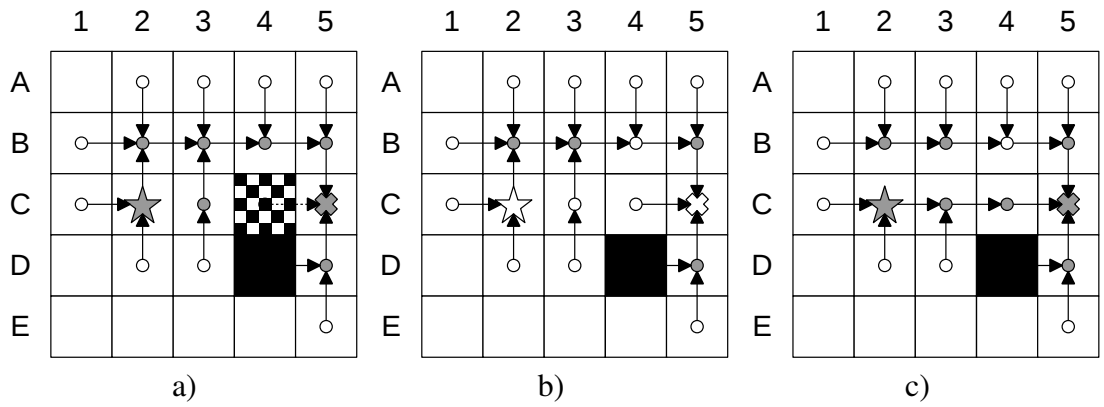


Figure 4.1: The agent (star), following the move from $C1$ to $C2$, observes cell $C4$ to become free (chessboard) (a). Therefore, nodes s_{C3}, s_{B4} and s_{C5} have to be re-opened (b). Figure (c) illustrates search-space following re-planning. White inner shape — open nodes, gray inner shape — closed nodes, arrows — parent node pointers, cross — goal node, black squares — obstacles, dashed line — affected edges.

explained, there is no need to cut any branch, however, nodes s_{C3}, s_{B4} and s_{C5} must be re-opened (Fig. 4.1b)[1].

If the cost of an edge $e(s_1, s_2)$ increases, all nodes in the branch of the search-tree supported by this edge become under-consistent, which means that its g values are lower than they should be. As the condition $g(s_2) > g(s_1) + cost(s_1, s_2)$ is not fulfilled, simple reopening of s_1 will not lead the algorithm to re-establish consistency. Therefore, before the algorithm begins new search, it must make such nodes over-consistent. This is achieved by setting its g values to infinity or by marking them as unvisited. If the cost of the $e(s_1, s_2)$ edge increases, the affected search-tree branch may shrink or even — covered by other unaffected branches — it may disappear. Thus, parent nodes of nodes that belong to the affected area may change radically, as shown in Figure 4.2.

The main issue is to decide which nodes should be made over-consistent and which nodes should be re-explored. The idea behind the D* Lite algorithm is to make over-consistent and to re-explore only those nodes that lead towards the starting-node (i.e., current state of the agent). For both operations (making node over-consistent and node re-exploration) the same open-list is used. This is made possible by introduction of the rhs value for each node, which ensures that nodes will be made over-consistent and re-explored in the correct order. An advantage of this approach is that only necessary nodes are reinitialized and re-explored. A disadvantage of D* Lite is that making some nodes inconsistent and reopening their neighbors is a part of a search-loop that involves operations on the open-list, and this may hinder the efficiency of the algorithm.

[1]In the example, in addition to nodes s_{C3}, s_{B4} and s_{C5}, a start node s_{C2} has been opened to properly handle the termination condition, which is a property of the D* Extra Lite algorithm explained later.

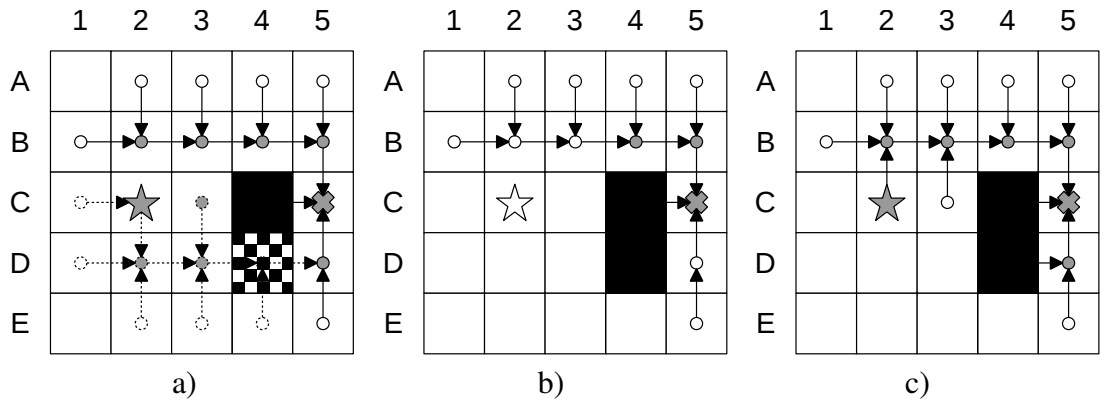


Figure 4.2: The agent (star), following the move from $C1$ to $C2$, observes cell $D4$ to be occupied (chessboard) (a). The entire branch supported by the edge $e(s_{C4}, s_{C5})$, must then be cut (b). As the space left by the cut branch may be filled by another branch, nodes that neighbor the cut branch are re-opened. A new optimal solution emerges that is on a different branch from the initial branch (c).

As already discussed, following an increase in the edge $e(s_1, s_2)$ cost, the entire affected search-tree branch becomes inconsistent. The main idea behind the D* Extra Lite algorithm is to cut the whole branch at once. This is unlike D* Lite, which while searching, reinitializes single nodes. Branch cutting is a simple recursive operation that makes nodes unvisited without employing the open-list. After the branch cut there will be a gap in the frontier fringe to be repaired. Therefore, apart from reopening the s_1 node, all nodes belonging to neighboring branches will also need to be reopened. Only then is the search-space reinitialized and ready for a new search episode.

4.3 D* Extra Lite Algorithm

D* Extra Lite, like other incremental algorithms, operates on procedures that utilize a sense-plan-act scheme. The implementation of D* Extra Lite shares such basic procedures with D* Lite (Alg. 4.1).

The algorithms start with an initial map update and a search-space initialization (lines 3–4 in Alg. 4.1). The main loop (lines 5–10 in Alg. 4.1) iteratively runs searching, action selection and execution, map update and reinitialization. The SEARCH() procedure consists of another loop that repeatedly performs SEARCHSTEP() while a goal condition has not been met, and the open-list is not empty. The ACTIONSELECTION() procedure chooses the action (leading to successive state) that will achieve the goal with the least cost. The REINITIALIZE() procedure will instantly terminate if no change is observed.

Procedures that distinguish D* Lite from D* Extra Lite are shown in listings Alg. 4.2[2] and Alg. 4.3, respectively, but even within these procedures, several elements are similar. In the pseudocode, the following functions are also used:

- TOPOPEN(): returns the node with the lowest key in the open-list,
- POPOPEN(): removes the node with the lowest key in the open-list,
- PUSHOPEN(s, k): if node s is not open, it inserts s to the open-list with key k, if node s is open, it updates the priority (if necessary),
- REMOVEOPEN(s): removes node s from the open-list,
- $open(s)$: indicates if node s is in the open-list.

Algorithm 4.2 D* Lite (optimized version) procedures.

```
1:  function CALCULATEKEY(s)
2:      return [min(g(s), rhs(s)) + h(s_start, s) + k_m; min(g(s), rhs(s))]
3:  function SOLUTIONFOUND()
4:      return key(TOPOPEN()) >= CALCULATEKEY(s_start) AND rhs(s_start) <= g(s_start)
5:  function INITIALIZE()
6:      k_m = 0
7:      for all s ∈ S do
8:          g(s) = rhs(s) = ∞
9:      rhs(s_goal) = 0
10:     PUSHOPEN(s_goal, CALCULATEKEY(s_goal))
11: function SEARCHSTEP()
12:     s =TOPOPEN()
13:     POPOPEN()
14:     k_old = key(s)
15:     k_new = CALCULATEKEY(s)
16:     if k_old < k_new then
17:         PUSHOPEN(s, CALCULATEKEY(s))
18:     else if g(s) > rhs(s) then
19:         g(s) = rhs(s)
20:         REMOVEOPEN(s)
21:         for all s' ∈ Pred(s) do
22:             if s' ≠ s_goal then
23:                 rhs(s') = min(rhs(s'), cost(s', s) + g(s))
24:             UPDATEVERTEX(s')
25:     else
26:         g_old = g(s)
27:         g(s) = ∞
28:         for all s' ∈ Pred(s) ∪ s do
29:             if rhs(s') = cost(s', s) + g_old AND s' ≠ s_goal then
30:                 EVALUATERHS(s')
31:             UPDATEVERTEX(s')
```

Both algorithms must operate while the agent's start state changes between searching episodes. In heuristic search algorithms, the key value to prioritizing an open-list is usually

[2]The pseudocode of D* Lite (optimized version) presented here is not an exact copy of the pseudocode presented in [10], however, it is the same algorithm without any modifications.

```
32: function REINITIALIZE()
33:     if any edge cost changed then
34:         k_m = k_m + h(s_last, s_start)
35:         s_last = s_start
36:         for all directed edges (u, v) with changed cost do
37:             c_old = cost(u, v)
38:             update edge cost cost(u, v)
39:             if c_old > cost(u, v) then
40:                 if u ≠ s_goal then
41:                     rhs(u) = min(rhs(u), cost(u, v) + g(v))
42:             else if rhs(u) = c_old + g(v) then
43:                 if u ≠ s_goal then
44:                     EVALUATERHS(u)
45:             UPDATEVERTEX(u)
46: function UPDATEVERTEX(s)
47:     if g(s) ≠ rhs(s) then
48:         PUSHOPEN(s, CALCULATEKEY(s))
49:     else if g(s) = rhs(s) AND open(s) then
50:         REMOVEOPEN(s)
51: function EVALUATERHS(s)
52:     rhs(s) = ∞
53:     for all s' ∈ Succ(s) do
54:         rhs(s) = min (rhs(s), cost(s, s') + g(s'))
```

calculated as a sum of an heuristic value $h(s_{start}, s)$, which in the case of a backward-search is the cost-to-start, and the $g(s)$ value, is the cost-from-goal. Owing to an agent's transitions toward decreasing g values, h values should be recalculated. Recalculation of the key for each open node, and reordering of an open-list, would hinder the efficiency of the search algorithm. Therefore, another solution is used.

If the agent changes its state from the previous start state s_{t0} to the new start state s_{t1}, than for some nodes previously calculated h values are underestimated, while other are overestimated. If the h value is underestimated, i.e., $h(s_{t0}, s) < h(s_{t1}, s)$, the node will be removed from the top of the open-list too early, thus its key has to be recalculated and the node has to be re-pushed to the open-list (lines 14–17 in Alg. 4.2 and in Alg. 4.3). More serious is when the h value of a node is overestimated, i.e., $h(s_{t0}, s) > h(s_{t1}, s)$. In this case, the node might be removed from the top of the open-list too late and the algorithm will not find the optimal solution. In the worst case, the h value will be overestimated by $h(s_{t0}, s_{t1})$ (Fig. 4.3). To avoid overestimated h values, a bias value k_m can be added to the heuristic calculated for nodes added after agent's transition (line 2 in Alg. 4.2 and in Alg. 4.3). If the k_m value is increased by $h(s_{t0}, s_{t1})$, all nodes pushed to the open list before the agent's transition will have been underestimated (or exactly) h values, i.e., $h(s_{t0}, s) \leq h(s_{t1}, s) + k_m$. The k_m value is increased at each reinitialization step (line 34 in Alg. 4.2, line 30 in Alg. 4.3).

Algorithm 4.3 D* Extra Lite procedures.

```
1:  function CALCULATEKEY(s)
2:      return [g(s) + h(s_start, s) + k_m; g(s)]
3:  function SOLUTIONFOUND()
4:      return TOPOPEN() = s_start OR (visited(s_start) AND NOT open(s_start))
5:  function INITIALIZE()
6:      k_m = 0
7:      visited(s_goal) = true
8:      parent(s_goal) = NULL
9:      g(s_goal) = 0
10:     PUSHOPEN(s_goal, CALCULATEKEY(s_goal))
11: function SEARCHSTEP()
12:     s =TOPOPEN()
13:     POPOPEN()
14:     k_old = key(s)
15:     k_new = CALCULATEKEY(s)
16:     if k_old < k_new then
17:         PUSHOPEN(s, CALCULATEKEY(s))
18:     else
19:         for all s' ∈ Pred(s) do
20:             if NOT visited(s') OR g(s') > cost(s', s) + g(s) then
21:                 parent(s') = s
22:                 g(s') = cost(s', s) + g(s)
23:                 if NOT visited(s') then
24:                     visited(s') = true
25:                 PUSHOPEN(s', CALCULATEKEY(s'))
26: function REINITIALIZE()
27:     if any edge cost changed then
28:         CUTBRANCHES()
29:         if seeds ≠ ∅ then
30:             k_m = k_m + h(s_last, s_start)
31:             s_last = s_start
32:             for all s ∈ seeds do
33:                 if visited(s) AND NOT open(s) then
34:                     PUSHOPEN(s, CALCULATEKEY(s))
35:             seeds = ∅
36: function CUTBRANCHES()
37:     reopen_start = false
38:     for all directed edges (u, v) with changed cost do
39:         if visited(u) AND visited(v) then
40:             c_old = cost(u, v)
41:             update edge cost cost(u, v)
42:             if c_old > cost(u, v) then
43:                 if g(s_start) > g(v) + cost(u, v) + h(s_start, u) then
44:                     reopen_start = true
45:                 seeds = seeds ∪ v
46:             else if c_old < cost(u, v) then
47:                 if parent(u) = v then
48:                     CUTBRANCH(u)
49:     if reopen_start = true AND visited(s_start) then
50:         seeds = seeds ∪ s_start
```

```
51: function CUTBRANCH(s)
52:     visited(s) = false
53:     parent(s) = NULL
54:     REMOVEOPEN(s)
55:     for all s' ∈ Succ(s) do
56:         if visited(s') AND NOT parent(s') = s then
57:             seeds = seeds ∪ s'
58:     for all s' ∈ Pred(s) do
59:         if visited(s') AND parent(s') = s then
60:             CUTBRANCH(s')
```

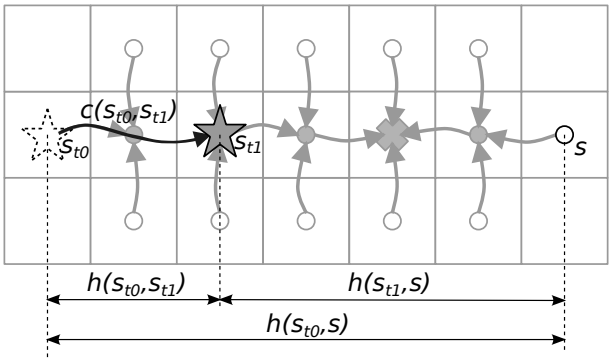


Figure 4.3: If the agent (star) moves from s_{t0} to s_{t1} then h value of node s will change. For the s node, the previous h value was $h(s_{t0}, s)$, however, following the agent's transition, a new h value should be $h(s_{t1}, s)$. Assuming that for the heuristic function, triangle inequality $h(s_{t0}, s_{t1}) + h(s_{t1}, s) \geq h(s_{t0}, s)$ holds, the worst case of overestimation of the h value for node s, i.e., $h(s_{t0}, s) - h(s_{t1}, s)$, will be equal to $h(s_{t0}, s_{t1})$, though, in general, traveled path cost $c(s_{t0}, s_{t1}) \geq h(s_{t0}, s_{t1})$. If, in the next search episode, the old h value is used for open-list prioritizing, s node may be removed from the top of the open-list too late. White inner shape — open nodes, gray inner shape — closed nodes.

Another common element of D* Lite and D* Extra Lite is that a key used for open-list sorting is a pair of values: $key(s) = [key_1(s), key_2(s)]$ (line 2 in Alg. 4.2 and in Alg. 4.3). As both algorithms allow for the reopening of previously closed nodes, in the case of multiple nodes with equal $key_1(s)$ values, the tie-breaking rule should be to favor nodes with lower $key_2(s)$, which is $g(s)$ in D* Extra Lite and $min(rhs(s), g(s))$ in D* Lite. This measure preserves optimality.

Referring to the previous explanations, changes in the environment affect branches of the search-tree. The affected branch must be re-explored with special attention given to the case of edge cost increase. To re-explore such a branch, it must be reinitialized. D* Lite recognizes and reinitializes affected nodes while searching. Such an approach requires the use of the $rhs(s)$ value, which can be understood as a pre-$g(s)$ value (the $g(s)$ takes $rhs(s)$ value when s is over-consistent; line 19 in Alg. 4.2). Comparing $rhs(s)$ and $g(s)$ values is a basic method for recognizing affected nodes (specifically under-consistent).

In contrast to D* Lite, the D* Extra Lite algorithm always reinitializes the entire affected under-consistent branch of a search-tree. As a consequence, at the beginning of searching, the search-space has no under-consistent nodes, but over-consistent, consistent and unvisited nodes only. Therefore, it is possible to keep the SEARCHSTEP() (Alg. 4.3) almost as simple as the A* algorithm.

The REINITIALIZE() procedure (Alg. 4.3) is executed if the cost of any visited edge has changed. Tree-cutting is the first step of reinitialization (CUTBRANCHES() in Alg. 4.3). For each edge with changed cost, the CUTBRANCHES() procedure does one of two possible operations. If the cost of the $e(u, v)$ edge has decreased, the v node is added to the list of seeds to be reopened later (lines 42, 45 in Alg. 4.3). If the cost of the $e(u, v)$ edge has increased and node v is the parent of node u, the branch is cut starting from u (lines 46–48 in Alg. 4.3). The cutting operation is simple, marking nodes unvisited. The CUTBRANCH() procedure is the recursive procedure which traverses throughout the branch, i.e., a next node to cut s' has to be such predecessor of a current node s, so the s is the parent of s' (lines 58–60 in Alg. 4.3).

Each successor node s', such that $s \neq parent(s')$ is placed in the list of seeds (lines 55–57 in Alg. 4.3). Although seeds are simply nodes to reopen, as they might be cut later, they cannot be merely pushed to the open-list. Following the CUTBRANCHES() procedure, the REINITIALIZE() procedure pushes to the open-list only these nodes from the $seeds$ list that remain visited and are not already open (lines 32–34 in Alg. 4.3). This operation repairs the frontier-gap made by branch cutting.

D* Extra Lite succeeds if the start node is on the top of the open-list, such as in the A* algorithm, or, if the start node has been closed in some previous searching episode and has not been cut in the reinitialization.

In the case of edge-cost decrease, there may be a shorter path. Therefore, to preserve optimality, the start node should be reopened. However, in not every case of edge-cost decrease does the start node need to be reopened. Herein, another optimization of the algorithm is possible. Assuming that $h(s_{start}, u)$ is admissible, for decreased $e(u, v)$ edge cost, the start node s_{start} requires reopening only if $g(s_{start}) > g(v) + cost(u, v) + h(s_{start}, u)$. Otherwise, it is impossible for a path containing an $e(u, v)$ edge to be shorter. This condition is checked and applied in lines 42–44 and 49–50 of the Alg. 4.3. Such a situation is depicted in Figure 4.4 in episode five.

4.4 Discussion of the Algorithm

In [71], it was proven that the use of a biased key value does not affect the optimality of the D* Lite algorithm. As D* Extra Lite differs from D* Lite only in the reinitialization, the proof presented by Koenig [71], to some extent, can also be adapted for D* Extra Lite.

Proof 4.1 *For a nonnegative admissible heuristic $h(u, v)$, heuristic search algorithms, such as A*, D* Lite, and D* Extra Lite, provide an optimal solution if and only if they expand nodes in non-decreasing order of the f value, which is calculated as $f(s) = g(s) + h(s_{start}, s)$ in the case of a backward search. Moreover, D* Lite and D* Extra Lite are able to reuse nodes from previous searches. However, due to agent transitions, the start state s_{start} changes; thus, the f value is different at each i-th time point. Hence, $f_i(s) = g(s) + h(s_{start,i}, s)$. To avoid re-computation of the f value and reordering of the open list, D* Lite and D* Extra Lite use a biased key, which is defined as follows:*

$$key_i(s) = [f_i(s) + k_{m,i}; g(s)], \tag{4.1}$$

where $k_{m,i}$ is increased with each i-th search episode (line 30 in Alg. 4.2) in accordance with (4.2).

$$k_{m,i+1} = k_{m,i} + h(s_{start,i}, s_{start,i+1}) \tag{4.2}$$

To prove that the search algorithm using a key defined by 4.1 is optimal, we have to show that, for each pair of nodes s and s' for which $f_i(s) < f_i(s')$ holds, node s' will not be expanded before s for any $key_j(s)$ and $key_k(s')$, where $j \leq i$ and $k \leq i$.

Case 1. *For $j \leq k = i$, we can prove that $key_j(s) < key_k(s') = key_i(s')$. Starting from assumption the $f_i(s) < f_i(s')$, we have the following:*

$$f_i(s) + k_{m,i} < f_i(s') + k_{m,i}.$$

From (4.2), we have the following:

$$\begin{aligned} k_{m,i} &= k_{m,j} + h(s_{start,j}, s_{start,j+1}) + \cdots + h(s_{start,i-1}, s_{start,i}) \\ &= k_{m,j} + \Delta k_{m,j,i}. \end{aligned}$$

Thus,

$$f_i(s) + k_{m,j} + \Delta k_{m,j,i} < f_i(s') + k_{m,i}$$
$$g(s) + h(s, s_{start,i}) + k_{m,j} + \Delta k_{m,j,i} < g(s') + h(s', s_{start,i}) + k_{m,i}.$$

Since

$$h(s, s_{start,j}) \leq h(s, s_{start,i}) + h(s_{start,j}, s_{start,j+1}) + \cdots + h(s_{start,i-1}, s_{start,i})$$
$$h(s, s_{start,j}) \leq h(s, s_{start,i}) + \Delta k_{m,j,i},$$

we have the following:

$$g(s) + h(s, s_{start,j}) + k_{m,j} < g(s) + h(s, s_{start,i}) + k_{m,j} + \Delta k_{m,j,i}.$$

Thus,

$$g(s) + h(s, s_{start,j}) + k_{m,j} < g(s') + h(s', s_{start,i}) + k_{m,i}$$
$$[g(s) + h(s, s_{start,j}) + k_{m,j}; g(s)] < [g(s') + h(s', s_{start,i}) + k_{m,i}; g(s')]$$
$$key_j(s) < key_i(s').$$

Case 2. *For any $k < i$, it may occur that $key_j(s) > key_k(s')$; thus, node s' will pop before node s. However, if $key_{k,old}(s') < key_{i,new}(s')$, node s' is re-pushed to the open list with an updated key without expansion. This is guaranteed by lines 14–17 in Algorithm 4.2. For the updated $key_i(s')$, from Case 1, the relation $key_j(s) < key_i(s')$ holds; thus, node s' will not be expanded before node s.* □

D* Extra Lite is very similar to D* Lite, thus these algorithms have similar both time and space complexity. For example, implementation using a binary heap has $O(n \log n)$ time-complexity, where n is the number of expanded nodes. If there are only over-consistent or uninitialized nodes, both algorithms are almost equivalent. In this case, for D* Lite, only lines 12–24 (Alg. 4.2) of the SEARCHSTEP() function are used, while the SEARCHSTEP() function of the D* Extra Lite algorithm is designed for such a case exclusively. If edge costs decrease, reinitialization is also similar for both algorithms — both reopen node that support a changed edge, (lines 39–41 in Alg. 4.2 and lines 42–45 in Alg. 4.3).

The main difference between the D* Lite and the D* Extra Lite algorithm is in edge-cost increase. D* Lite reinitializes and re-expands only those nodes that lead toward the agent's current state, while D* Extra Lite always reinitializes the entire under-consistent branch of the

search-tree. Therefore, herein, only the complexity of reinitialization is investigated. This is done for the n-th searching episode. Let us consider a sufficiently large graph and an agent with a finite observation range. Since the observation range is finite, the number of changed edges is negligibly small. Therefore, the cost of operations in lines 32–45 (Alg. 4.2) of D* Lite and lines 36–46 (Alg. 4.3) of D* Extra Lite can also be neglected. The crucial operations are in lines 26–31 (Alg. 4.2, which are a part of the SEARCHSTEP() function) and 32–34, 48 and 51–60 (Alg. 4.3, are mainly CUTBRANCH() function).

Now, let us introduce the following numbers relevant to D* Extra Lite:

- n_{to_cut} — number of under-consistent nodes to cut,
- n_{to_open} — number of nodes to open in order to repair the frontier gap,
- $n_{open,DEL}$ — number of nodes in the open-list,
- n_{tree} — number of nodes in the search tree after the previous search episode,

such that $n_{to\_cut} + n_{to\_open} \leq n_{tree}$ and $n_{open,DEL} \leq n_{tree}$. For D* Lite, the following numbers can be defined:

- n_{to_reinit} — number of nodes to be reinitialized for which $key(s) \leq key(s_{start})$,
- $n_{open,DL}$ — number of nodes in the open-list,
- $n_{ever\_visited}$ — number of nodes visited,

such that $n_{to\_reinit} \leq n_{ever\_visited} - 1$ and $n_{open,DL} \leq n_{ever\_visited}$.

In some cases, all visited nodes must be reinitialized. For example, this could happen when, after traversing a long corridor, right before reaching the goal, an agent encounters a dead-end and must take a new path through a corridor that was not initially chosen.

Furthermore, at each n-th step of the agent, the $n_{tree} \leq n_{ever\_visited}$ relation holds. As D* Extra Lite instantly cuts all under-consistent branches, it is likely that $n_{tree} < n_{ever\_visited}$.

The computation time of under-consistent nodes reinitialization for D* Lite and D* Extra Lite is shown in equations (4.3) and (4.4), respectively. As expected, iterations over changed edges have been omitted. Computation times of particular functions are marked as $c_{\text{<function name>;<code lines>}}$, where calculation times of the open-list operations, such as push, pop and delete, may depend on the number of open elements. b is a domain-specific branching factor (number of neighbors).

$$\begin{aligned} T_{reinit,DL} \approx n_{to\_reinit} \cdot (\quad & c_{pop;13}(n_{open,DL}) + c_{push;31,48}(n_{open,DL}) \\ & + c_{h;31,48} + c_{preds;28} + b \cdot c_{cost;29} \quad + \log b \cdot (c_{succs;30,53} + b \cdot c_{cost;30,53}) \\ & + \log b \cdot (c_{push|del;31,48|50}(n_{open,DL}) + c_{h;31,48})) \end{aligned} \tag{4.3}$$

$$\begin{aligned} T_{reinit,DEL} \approx \quad & n_{to\_cut} \cdot (c_{del;54}(n_{open,DEL}) + c_{preds;58} + c_{succs;55}) \\ & + n_{to\_open} \cdot (c_{push;34}(n_{open,DEL}) + c_{h;34}) \end{aligned} \tag{4.4}$$

Now, let us consider the worst-case scenario, in which all nodes that have ever been visited must be reinitialized. In such a case, D* Lite $n_{to\_reinit} = n_{ever\_visited} - 1$, and D* Extra Lite $n_{to\_cut} = n_{ever\_visited} - 1$. Additionally, for D* Extra Lite, $n_{to\_open} = 1$, i.e., only the root (the goal node) will be reopened. The worst-case computation times are represented below, by equations (4.5) and (4.6).

$$\begin{aligned} T_{reinit,DL} \approx n_{ever\_visited} \cdot (\quad & c_{pop;13}(n_{open,DL}) + c_{push;31,48}(n_{open,DL}) \\ & + c_{h;32,48} + c_{preds;28} + b \cdot c_{cost;29} \quad + \log b \cdot (c_{succs;30,53} + b \cdot c_{cost;30,53}) \\ & + \log b \cdot (c_{push|del;31,48|50}(n_{open,DL}) + c_{h;31,48})) \end{aligned} \tag{4.5}$$

$$T_{reinit,DEL} \approx n_{ever\_visited} \cdot (\quad c_{del;54}(n_{open,DEL}) + c_{preds;58} + c_{succs;55}) \tag{4.6}$$

Although there exists no general relationship between $n_{open,DL}$ and $n_{open,DEL}$, for both algorithms, those numbers may be large; for example, while D* Lite removes only consistent nodes, hence it may keep many inconsistent nodes from any of previous searches, D* Extra Lite, due to branch cutting, may maintain a rugged frontier. However, for a sufficiently large graph, it can be assumed that $n_{open,DL} \approx n_{open,DEL} \approx n_{open}$. Furthermore, assuming that an open-list is implemented using a binary heap, the time of pushing, popping and deletion operations is the same, namely $c_{heap}(n_{open}) \approx \log n_{open}$. Equations (4.5) and (4.6) can be transformed as follows.

$$\begin{aligned} T_{reinit,DL} \approx n_{ever\_visited} \cdot ((2 + \log b) \cdot c_{heap}(n_{open}) \\ + (1 + \log b) \cdot c_h + c_{preds} + \log b \cdot c_{succs} + b \cdot (1 + \log b) \cdot c_{cost}) \end{aligned} \tag{4.7}$$

$$T_{reinit,DEL} \approx n_{ever\_visited} \cdot (c_{heap}(n_{open}) + c_{preds} + c_{succs}) \tag{4.8}$$

From (4.7) and (4.8) it is clear that in the worst-case scenario, the following relation holds: $T_{reinit,DEL} < T_{reinit,DL}$. Moreover, as reinitialization time (4.3) depends on branching factor b, D* Lite is more domain sensitive than is D* Extra Lite.

In contrast, there exists a scenario that D* Lite could solve easily while D* Extra Lite would struggle. Let us assume that there is only one under-consistent node for which $key(s) < key(s_{start})$. In such a case $n_{to\_reinit} = 1$. D* Extra Lite would need to cut an entire branch and reopen all nodes that are neighboring to this branch. For a sufficiently large graph, it is likely that $n_{to\_cut} + n_{to\_open} > n_{to\_reinit}$, thus, from equations (4.3) and (4.4), $T_{reinit,DEL} > T_{reinit,DL}$. Indeed, such a scenario is observable on random maps with low fill-ratio. However, these

random maps are artificial, and therefore specific with their salt-and-pepper-like changes. For typical maps from video games as well as better structured maps of rooms, D* Extra Lite remains quicker than both D* Lite and MPGAA*.

Further improvement of D* Extra Lite is possible for undirected graphs. For domains such as presented here path-planning on a grid-map, in which $Succ(s) \equiv Pred(s)$, the CUTBRANCH() function can be simplified. This is demonstrated in Alg. 4.4.

Algorithm 4.4 CUTBRANCH() procedure of the D* Extra Lite algorithm for domains in which $Succ(s) \equiv Pred(s)$.

```
1: function CUTBRANCH(s)
2:     visited(s) = false
3:     parent(s) = NULL
4:     REMOVEOPEN(s)
5:     for all s' ∈ Pred(s) do
6:         if visited(s') AND parent(s') = s then
7:             CUTBRANCH(s')
8:         else
9:             seeds = seeds ∪ s'
```

In the author's opinion, in addition to superior reinitialization-time, D* Extra Lite is easier to implement and more reliable than D* Lite. For example, while to ascertain if a node is a parent of another node, D* Extra Lite relies on topological relations only, (i.e., $parent(s)$; lines 47, 56 and 59 in Alg. 4.3), D* Lite uses $rhs(s) = cost(s, s') + g(s')$ comparison (lines 29, 42 in Alg. 4.2). Comparison in the case of a cost expressed with real numbers, is an error-prone operation for computers. If, due to numerical issues, an admissibility of a heuristic is broken, D* Lite may produce local minima. This would make it impossible to reconstruct path. MPGAA*, another algorithm implemented and used in the benchmark, is also vulnerable to numerical errors; numerical issues may affect output of the GOALCONDITION() function, which would lead the algorithm to run unnecessary search steps.

4.5 Example

In this example, a grid-world domain is used. An agent can move in cardinal directions only. The cost of motion between two unoccupied neighboring cells is one. An occupied cell is treated as regular cell, however, the cost of entering or leaving such a cell is infinite. The heuristic function uses Manhattan distance.

D* Extra Lite in action is presented in Figure 4.4. Each sub-figure depicts the complete state of a search-space. Consecutive episodes are organized in rows. Each episode begins with the initialization/reinitialization of a search-space, after which searching commences. When a solution is found, an agent follows decreasing g values leading towards the goal. After each

step, observation is performed. Any change observed in the explored search-space ends the current episode and starts a new episode from reinitialization.

In Figure 4.4, each visited grid cell (i.e., visited node) has been assigned four values, which are the h value in the bottom-left, the g value in the top-left, the f value in the top-right, and the k_m value in the bottom-right. The $f = h + g + k_m$ value is the first part of a key value calculated in line 2 in Alg. 4.3. Closed nodes have gray-filled inner shape. Nodes with a white-filled inner shape remain on the open-list. An arrow between nodes always points to the parent node.

Episode 1 commences with initialization. In Figure 4.4 the goal cell is marked with a cross sign. Initially, the goal node has $h = 3$, $g = 0$, $f = 3$. The k_m value is set to 0. The current state of an agent is depicted with a star sign. The second sub-figure in the row in Figure 4.4 depicts the state of the search-space after searching. If the start node is visited and it is on the top of the open-list or has already been closed in a previous search episode, the searching algorithm succeeds (line 4 in Alg. 4.3). After searching, the agent follows decreasing g values, until it notices any change in the environment. In Episode 1, after two steps of the agent, the cell $C3$ changed its state to occupied. The affected search-tree branch is indicated with a dashed line (the third sub-figure of Episode 1 in Figure 4.4). Observation of this change ends Episode 1.

Episode 2 commences with reinitialization. Due to change in the $C3$ cell, the costs of edges $e(s_{C3}, s_{B3})$, $e(s_{C2}, s_{C3})$ and $e(s_{D3}, s_{C3})$ have increased. Consequently, nodes supported by these edges became under-consistent (theirs g values are lower than they should be). In this case, the reinitialization procedure will cut the entire affected branch (lines 46–48 in Alg. 4.3). As no particular order is required, cutting may start from any of the $C2$, $D3$ or $C3$ nodes. If the cutting procedure starts from node $C2$, the branches supported by $D3$ or $C3$ will be cut later. During branch cutting, neighboring nodes are pushed to the list of seeds (line 57 in Alg. 4.3, or line 9 in Alg. 4.4 for undirected graph). According to the CUTBRANCHES() procedure, the $seeds$ contains a number of nodes of which only a select few remain visited. Node that are not yet on the open-list are reopened (lines 32–34 in Alg. 4.3). Following the reinitialization that began Episode 2, nodes $B2$ and $B3$ have been reopened. During the reinitialization, a k_m has been increased by $h(s_{E2}, s_{D3}) = 2$ (line 30 in Alg. 4.3). Episode 2 ends after the change observed in cell $D4$.

In the reinitialization at the beginning of Episode 3, the k_m is increased by $h(s_{D3}, s_{C2}) = 2$. The node corresponding to the cell $D5$ is reopened (lines 45, 32–34 in Alg. 4.3). The values of the $D2$ node has been set to $h = 4$, $g = 1$, $k_m = 3$ which results in $f = 8$. As the $g(s_{start}) > g(s_{D5}) + cost(s_{D4}, s_{D5}) + h(s_{start}, s_{D4})$, the start node must be reopened (lines 43–44, 49–50 in Alg. 4.3).

Episode 3 ends with observation of cell $B1$ becoming occupied. At the beginning of Episode 4, the affected branch is cut. However, the node corresponding to the agent's state is closed

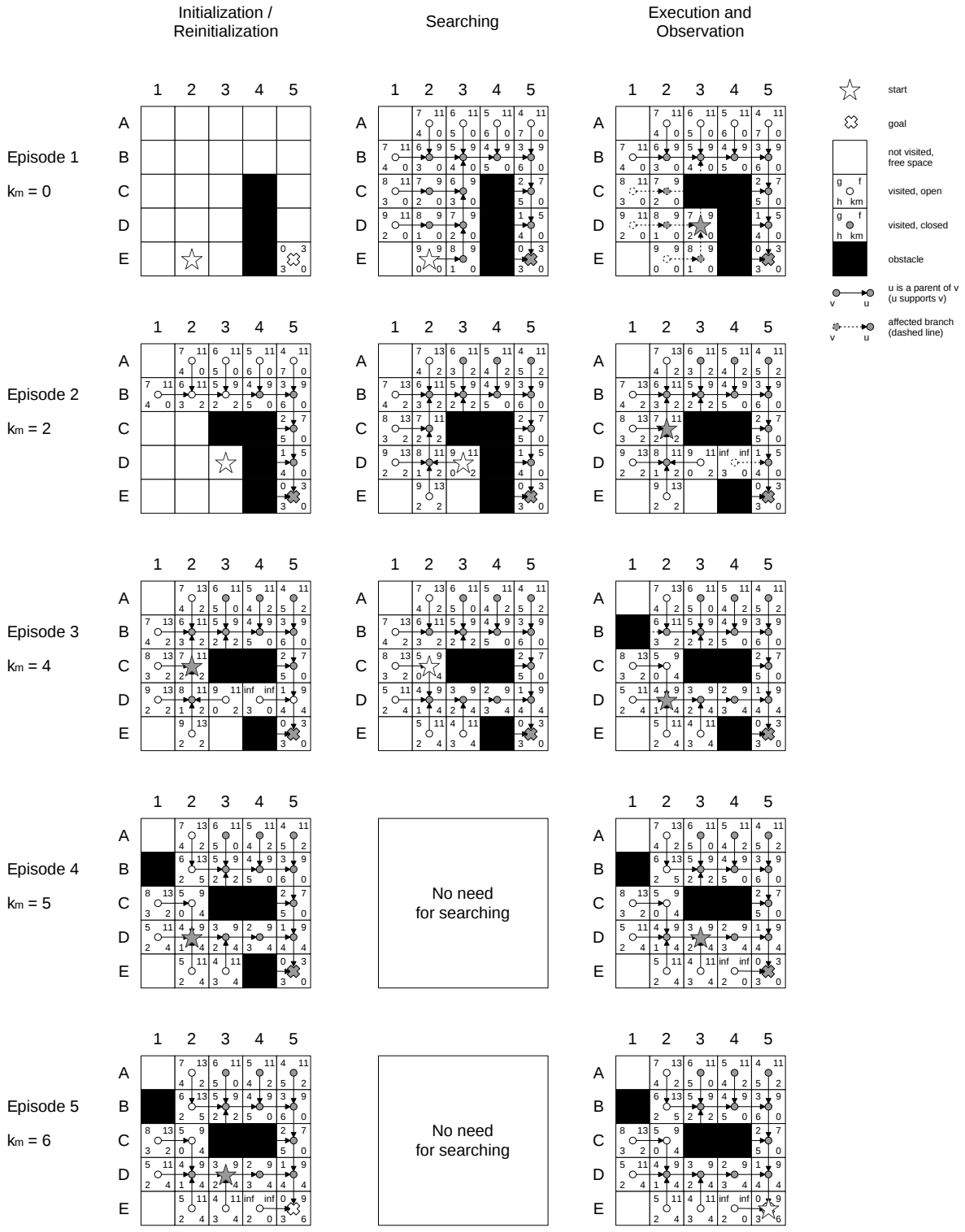


Figure 4.4: D* Extra Lite example in action; white inner shape — open nodes, gray inner shape — closed nodes.

and unaffected, therefore the success condition is realized (line 4 in Alg. 4.3) and no further searching is required.

Following transition from $D2$ to $D3$, the agent observes that the cost of the $e(s_{E4}, s_{E5})$ edge has decreased, which ends the Episode 4. However, there is no need for further searching. This is because $g(s_{start}) = g(s_{E5}) + cost(s_{E4}, s_{E5}) + h(s_{start}, s_{E4})$, and no shorter path can exist (no need to reopen the start node, which is checked in line 43 in Alg. 4.3).

4.6 Benchmark results

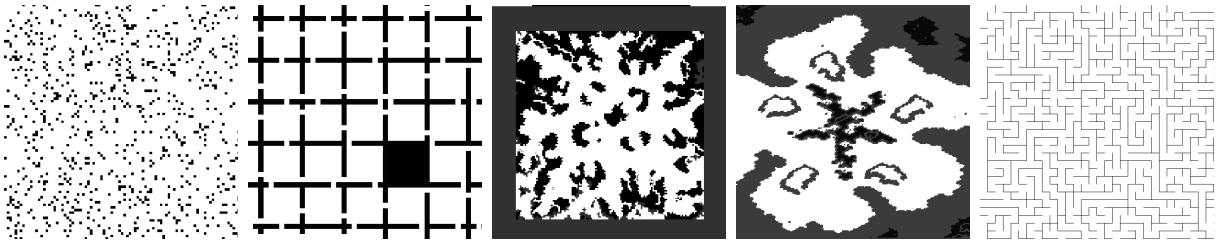
In the experiments the following three algorithms: D* Lite (optimized version) [10], MPGAA* [12] and D* Extra Lite were compared. These experiments were run on an Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz machine, with 8GB of RAM, running 64-Bit Linux.

All three algorithms have been implemented in C++ within the same programming framework[3] and compiled using gcc (4.8.4) compiler with o3 level of optimization.

This framework, in addition to the algorithms, provides a heap implementation. Heap implementation realizes lazy node removal and update, i.e., the REMOVEOPEN() function (line 50 in Alg. 4.2, and line 54 in Alg. 4.3) marks only the node(s) to be removed. Actual node removal takes place when the marked node is on the top of the open list. The following domain-specific functions for 2D grid-based path planning are also provided: benchmark maps and problem-loading, $cost$ and $heuristic$ functions (both use Euclidean distance represented with integer numbers multiplied by factor of 1000), a neighborhood selection function (eight-neighbor grid) and the MAPUPDATE() function, which simulates 360° rangefinder working with a resolution of 1° at a specific observation range. For that reason, simple ray tracing is used. (If a laser beam encounters an obstacle, ray tracing for that beam is stopped.) For each algorithm tested within the framework, the graph representing the entire search-space (equal to the size of the map) is allocated at the beginning.

Every benchmark problem has been solved in accordance with the main function presented in Alg. 4.1, that is, the map is updated after each step of an agent. If any change observed in the map affects the shortest path, reinitialization and a consecutive search are performed, though such necessity is checked by each algorithm in a different way. While the main function is running, a number of parameters are logged, these are: search function running time, reinitialization running time, search steps count, an open-list operations count, predecessor list query count, successor list query count, and traveled path cost. The total running time is

[3] Source code is available at `https://bitbucket.org/maciej_przybylski/heuristic_search`



a) b) c) d) e)

Figure 4.5: Sample maps from the Sturtevant's [72] benchmark: a) a portion of a random map, b) a portion of a rooms map, c) wc3 (World of Warcraft 3), d) sc (Starcraft), e) maze.

simply the sum of the search and reinitialization function running time, thus it does not include map-update time.

These algorithms have been tested within the following two settings: planning with freespace assumption (setting 1), in which obstacles are only added, and planning on maps with shortcuts and barriers (setting 2), in which obstacles may appear or disappear.

In the experiments, maps and problems from the benchmark prepared by Sturtevant [72] have been used. This benchmark provides a number of maps and randomly generated problems for 2D grid-based path planning, of which the following map-sets were used: *random_10*, artificially generated maps with a fill-ratio of 10% (Fig. 4.5a); *rooms*, artificially generated maps consisting of square rooms of different size (8–64 pixels) with narrow — of a single pixel size — passages in walls (Fig. 4.5b); *wc3*, maps from the World of Warcraft 3 video game (Fig. 4.5c); *sc*, maps from the Starcraft video game (Fig. 4.5d); *mazes*, artificially generated maps with passages of varying widths (1–32 pixels) (Fig. 4.5e).

As Sturtevant [72] argues, the *wc3* and *sc* maps are a good approximation of outdoor environments, and while the *rooms* maps simulate indoor environments very well, *random* map problems are typically used for benchmarking of incremental path planning algorithms [9, 10, 12]. Finally, this *mazes* map set benefits from many difficult problems with dead-ends.

Each data-set features distinct characteristics. In the *mazes* map-set, it is possible for even a minor change to cause extensive modification to the shortest path. Additionally, this *random* dataset is characterized by many small changes that do not significantly affect the path. Such a property of maps is related to the dimension parameter described and calculated for each map-set by Sturtevant [72]. (Refer to Table 4.1, for a dimension value for each map-set used in the experiments.) The dimension of a map-set describes the increase in the number of nodes at each depth of searching. As explained by Sturtevant [72], this dimension is an estimation of the branching factor of the search-tree generated while searching (not to be confused with the number of applicable actions, which is domain specific).

In each setting, maps sized 512x512 were used (except for the *sc* map-set in which few maps is larger), with an observation range of 10 map cells. For each original map, a map with modifications was prepared. According to the suggestions of Sturtevant [72], the experimental results were ordered by the problem-length (plots in Figures 4.6 through 4.9). In setting 1, in Figures 4.6 through 4.8, x-coordinate indicates the true shortest path calculated by A* algorithm running on a modified map, which is initially unknown for the agent. In setting 2, in Figure 4.9, x-coordinate indicates the overhead of the initially known shortest path over the true shortest path, that is, $true\_shortest\_path\_cost - initial\_shortest\_path\_cost$.

The plots in Figures 4.6 through 4.9 were created by grouping problems into buckets, this was according to Sturtevant [72]'s proposition, except that bucket size may vary between map-sets. Additionally, in the background of each plot, a histogram illustrates a coverage by problems. Plot values were calculated for buckets that contained a minimum of seven successfully solved problems. For each bucket, a mean was calculated to be presented in the plot. This value excluded the two most extreme values in that bucket.

4.6.1 Planning with freespace assumption

Planning with freespace assumption is a scenario in which the first planning episode is performed on an empty map. Obstacles are added to the map only if they are observed within the observation range, which means that action costs can only increase[4]. As there is no need to reestablish values consistency for h, in this case MPGAA* [12] behaves similar to MPAA* [67]. Although this is a preferable situation for MPGAA*, as it requires reinitialization of underestimated nodes, it is the most demanding scenario for D* Lite and D* Extra Lite.

In Table 4.1, average parameters values logged for planning with freespace assumption are shown. For each map-set, 10,000 randomly selected problems have been solved. In the majority of cases, D* Extra Lite is the quickest (highlighted T_t values in Table 4.1), only for random maps with 10% fill-ratio does D* Lite outperform the other two algorithms. Given that artificial maps have the highest dimension, generally, are less cluttered. As discussed in the complexity analysis, this is the most preferable scenario for D* Lite.

Other parameters listed in Table 4.1 offer further support to results of the complexity analysis. Since D* Lite performs reinitialization while searching, the number of search steps for D* Extra Lite is lower than for D* Lite, as well as the number of operations on the heap. In turn, due to search-tree branch cutting, D* Extra Lite performs many more iterations over the predecessors list. The number of iterations over the successors list for the *mazes* map-set is also

[4] A video demonstrating the three algorithms tested in this study in planning with freespace assumption is available at `https://youtu.be/al2L_TJXnoY`

Table 4.1: The average experimental results for planning with freespace assumption; Dim. — dimension [72], T_r — reinitialization time [ms], T_s — search time [ms], T_t — total time [ms], R_t — total time ratio, #S.Steps — number of search steps, #Heap — number of heap operations, #Preds — number of iterations over predecessors, #Succs — number of iterations over successors, P. Cost — traveled path cost [map cells].

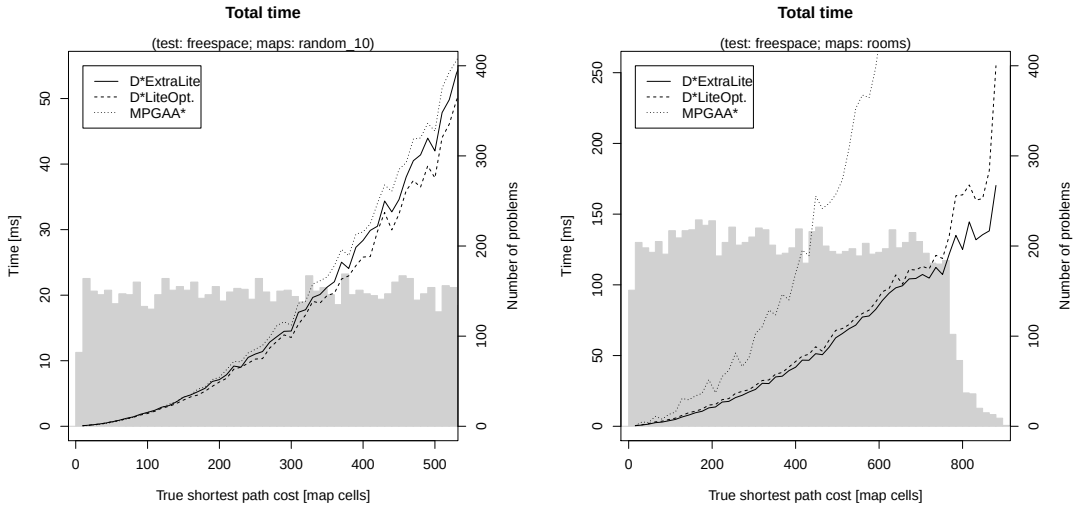
Map set	Dim.	T_r	T_s	T_t	R_t	#S.Steps	#Heap	#Preds	#Succs	P. Cost	Algorithm
random 10%	1.13	3.53	21.51	25.04	1.00	26151	86745	41032	17773	378.844	D* Extra Lite
		0.57	22.55	**23.11**	0.92	26119	90777	23387	972	378.844	**D* Lite Opt.**
		0.13	27.64	27.77	1.11	32621	92960	0	32621	378.031	MPGAA*
rooms	0.88	6.58	42.66	**49.24**	1.00	81348	189873	72708	34510	891.093	**D* Extra Lite**
		1.51	51.75	53.27	1.08	89149	250085	40511	16464	891.093	D* Lite Opt.
		0.30	176.85	177.15	3.60	277328	834970	0	277328	909.645	MPGAA*
wc3	0.75	2.19	19.93	**22.12**	1.00	33384	75607	28994	10502	461.109	**D* Extra Lite**
		0.66	24.28	24.94	1.13	35867	104626	19348	6241	461.109	D* Lite Opt.
		0.13	69.44	69.57	3.15	100582	296381	0	100582	463.155	MPGAA*
sc	0.41	10.90	87.56	**98.46**	1.00	163964	381328	149447	59992	1621.865	**D* Extra Lite**
		2.60	114.67	117.28	1.19	186328	541962	96924	49940	1621.865	D* Lite Opt.
		0.49	586.66	587.16	5.96	947837	2770506	0	947837	1616.717	MPGAA*
random 40%	0.09	22.04	89.26	**111.31**	1.00	225760	564389	232905	126502	6613.036	**D* Extra Lite**
		10.27	153.53	163.80	1.47	332119	898930	129432	164981	6613.036	D* Lite Opt.
		1.75	416.04	417.79	3.75	902292	2692626	0	902292	6531.051	MPGAA*
mazes	0.02	80.85	365.23	**446.08**	1.00	714783	1843893	826248	441713	18254.245	**D* Extra Lite**
		26.75	839.49	866.24	1.94	1093015	3328033	535517	1041180	18254.245	D* Lite Opt.
		4.99	4013.32	4018.31	9.01	6806971	19593377	0	6806971	18396.122	MPGAA*

interesting, since it is higher for D* Lite than for D* Extra Lite. These results are consistent with the worst-case time complexity of D* Lite (Eq. 4.5) and D* Extra Lite (Eq. 4.6), such that D* Lite may perform the iteration over successors up to $\log b$ times more often than D* Extra Lite.

Finally, in Table 4.1, the traveled-path cost is presented. D* Lite and D* Extra Lite are equivalent, however, the traveled path cost varies for MPGAA*. This is due to the fact that, while many paths of the same length exist in the 2D grid domain, different algorithms break ties in different ways. (For example, D* Lite and D* Extra Lite break ties as shown in line 18 of Algorithm 4.1.) Although all three algorithms are optimal, the consequences of the selected next step are unpredictable, and a chosen path could be a dead-end.

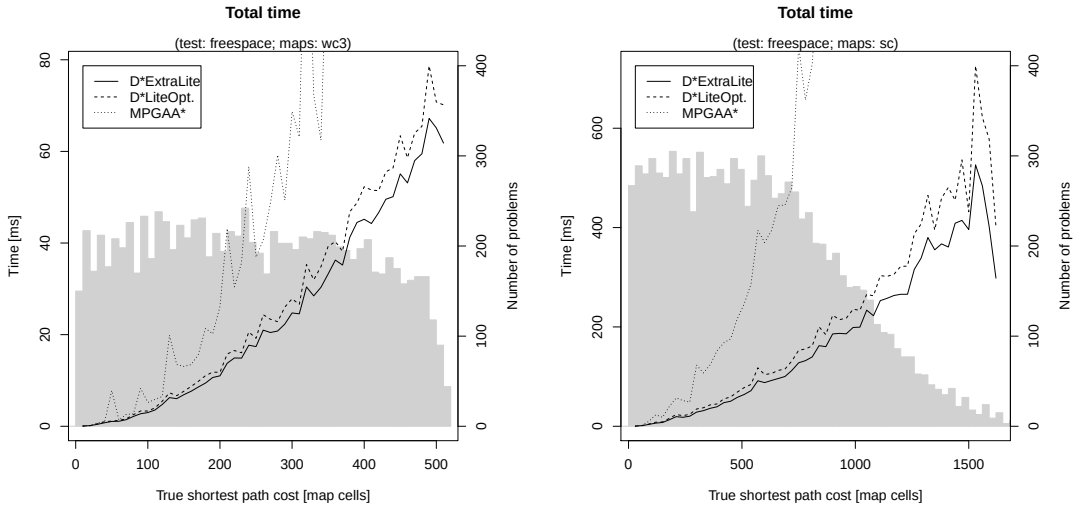
In Figures 4.6 through 4.8 the total time in the function of problem length is shown. For problems of a short length, the algorithms finished missions in a comparable time, although with increasing path length, differences become more pronounced. For the *random_10* maps (Fig. 4.6a), D* Lite is noticeably the quickest. However, with increasing problem complexity, differences between the algorithms increase in favor of D* Extra Lite, including for random maps. In the case of random maps with a fill-ratio of 40% (Fig. 4.8a), which due to obstacle density are more similar to mazes, D* Extra Lite is 1.47 times faster than D* Lite. MPGAA* seems more case-sensitive than D* Lite and D* Extra Lite; total-time plots for MPGAA* in

Figures 4.6b, 4.7a, b, and 4.8a, are more uneven than are corresponding time plots for the other algorithms.



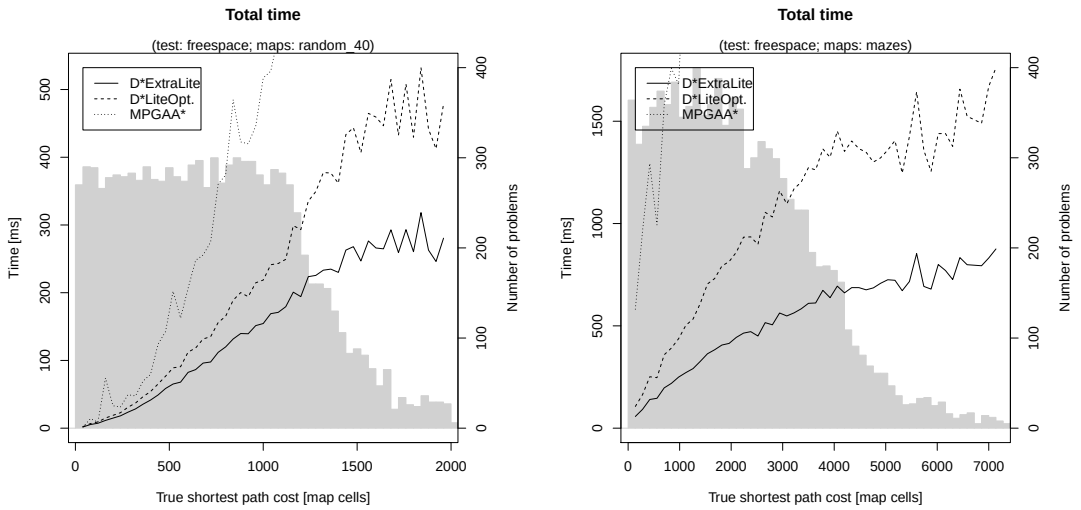
a) b)

Figure 4.6: Total running time for planning with freespace assumption, for *random_10* (a), *rooms* (b) map-sets; in the background the histogram of problems plotted in gray.



a) b)

Figure 4.7: Total running time for planning with freespace assumption, for *wc3* (a), *sc* (b) map-sets; in the background the histogram of problems plotted in gray.



a) b)

Figure 4.8: Total running time for planning with freespace assumption, for *random_40* (a) and *mazes* (b) map-sets; in the background the histogram of problems plotted in gray.

4.6.2 Planning on maps with shortcuts and barriers

A characteristic property of D* Lite, D* Extra Lite and MPGAA* is that they reveal different behaviors when confronted with action-cost increase and decrease, therefore, a new test is proposed. Considering that in the first half of the problems, obstacles were added only (barriers), and in the second half, obstacles were removed only (shortcuts), problems were resolved simply by solving the first half with the freespace assumption, and in the second half, assuming the opposite, such that, although to begin with the agent knows the entire map, as the true map is empty, obstacles can only disappear. Using this approach, for each of three representative map-sets (namely *random_10*, *rooms* and *wc3*), 5,000 problems with shortcuts and 5,000 problems with barriers were solved. In Figure 4.9, results with a negative path cost overhead correspond to problems with shortcuts, while results with a positive path cost overhead correspond to problems with barriers.

In Figure 4.9a, d and g represent the total time for *random_10*, *wc3* and *rooms* map-sets are shown, respectively. D* Extra Lite outperforms D* Lite and MPGAA* algorithms for barriers (positive path cost overheads in Figure 4.9 d and g). Moreover, the superiority of D* Extra Lite increases with the difference between true path-length and initial path-length. Next to the total time charts, the search time (Fig. 4.9b, e, h) and the reinitialization time plots for each map-set are presented (Fig. 4.9c, f, i). Search time is a dominating component for all three algorithms, however in case of MPGAA* and D* Extra Lite, reinitialization time is also meaningful.

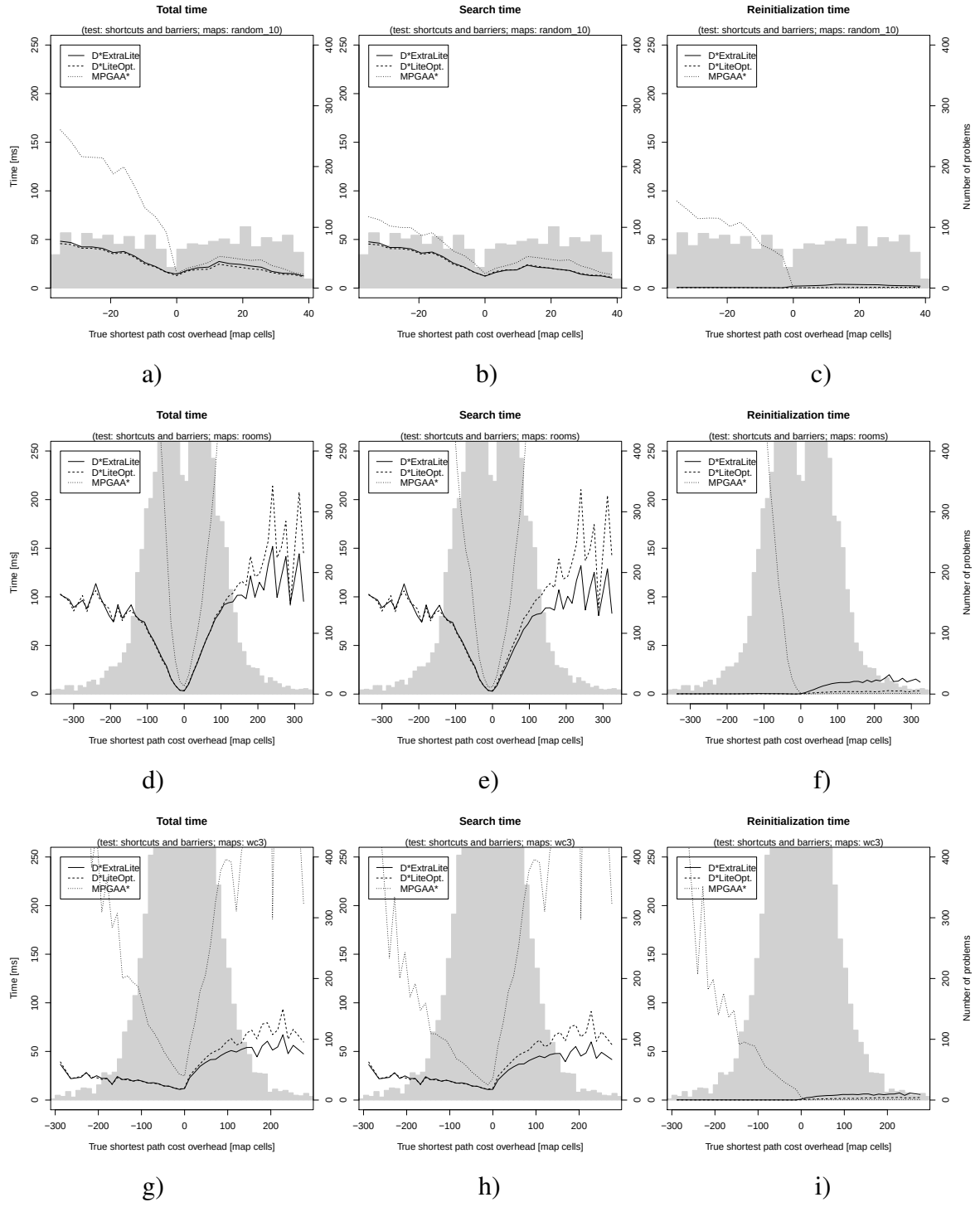


Figure 4.9: Running times for planning with shortcuts and barriers, for *random_10* (a,b,c), *rooms* (d,e,f) and *wc3* (g,h,i) map-sets; in the background the histogram of problems plotted in gray.

Table 4.2: The average experimental results for planning with shortcuts and barriers; T_r — reinitialization time [ms], T_s — search time [ms], T_t — total time [ms], R_t — total time ratio, #S.Steps — number of search steps.

Map set	Shortcuts					Barriers					Algorithm
	T_r	T_s	T_t	R_t	#S.Steps	T_r	T_s	T_t	R_t	#S.Steps	
random 10%	0.64	42.38	43.02	1.00	42337	3.44	20.29	23.73	1.00	23295	D* Extra Lite
	0.46	41.92	**42.38**	0.99	42097	0.76	20.83	**21.59**	0.91	23991	**D* Lite Opt.**
	79.08	63.17	142.25	3.31	219139	0.19	28.68	28.87	1.22	31006	MPGAA*
rooms	0.34	44.28	**44.62**	1.00	46932	7.21	46.70	**53.91**	1.00	79038	**D* Extra Lite**
	0.25	45.23	45.48	1.02	46815	1.63	56.71	58.34	1.08	86192	D* Lite Opt.
	213.40	155.09	368.49	8.26	558437	0.35	164.00	164.35	3.05	250108	MPGAA*
wc3	0.17	22.68	**22.85**	1.00	27166	4.41	36.99	**41.40**	1.00	71476	**D* Extra Lite**
	0.12	23.50	23.62	1.03	27085	1.43	48.71	50.14	1.21	79794	D* Lite Opt.
	70.53	68.81	139.34	6.10	234615	0.30	179.73	180.03	4.35	305136	MPGAA*

A property of the D* Extra Lite algorithm is that reinitialization time is high only in case of added obstacles. This is because under-consistent nodes have to be made over-consistent, which is achieved using the CUTBRANCH() procedure (Alg. 4.3). In contrast to D* Extra Lite, MPGAA*'s reinitialization time is higher when obstacles are removed. This is owing to its reestablishing procedure for h value consistency. In the case of shortcuts, D* Lite and D* Extra Lite only reopen affected nodes, and perform regular searches in the same manner. Therefore, the total time for these two algorithms is almost equal.

In Table 4.2 average parameters for the shortcuts and barriers setting are reported. The average values of particular parameters for barriers are similar to the ones presented in Table 4.1. Where results illustrate the effect of shortcuts, it can be seen, as expected, that the total time, as well as the other parameters, are similar for D* Lite and D* Extra Lite, and noticeably larger for MPGAA*.

4.6.3 Benchmark results summary

The experiments were conducted within two settings, in which maps sized 512x512 from six different map-sets were used. For each map-set 10,000 randomly selected problems were solved. In most experiments, D* Extra Lite performed on average from 1.08 to 1.94 times faster than D* Lite (optimized version), and from 1.11 up to 9.01 times faster than MPGAA*. Only in tests on random maps with a 10% fill-ratio was D* Lite 1.08 times faster than D* Extra Lite.

The weakness of D* Extra Lite is the number of iterations over predecessors and successors, which for all map-sets except the *mazes*, was higher than for the other two algorithms. This property should be taken into consideration when selecting an algorithm for particular domain. In domains with finite search-space, such as 2D video game maps, it is possible to initialize

Table 4.3: Average total time [ms] (T_t) and total-time ratios with regard to D* Extra Lite (R_t) across different observation ranges [map cells]; test conducted with freespace assumption on 100 problems from the *wc3* map-set.

Observation range	D* Extra Lite	D* Lite Opt.		MPGAA*	
	T_t	T_t	R_t	T_t	R_t
10	22.42	24.83	**1.11**	125.41	**5.59**
20	24.82	26.45	**1.07**	129.08	**5.20**
50	23.19	25.48	**1.10**	147.59	**6.36**
100	18.66	20.51	**1.10**	94.34	**5.06**

the entire search-space at the beginning. An iteration over a node's neighbors is then a simple operation with pointers. However, as in domains with a large or infinite search-space nodes are expanded bit-by-bit, domain-dependent implementations of $Pred(s)$ and $Succ(s)$ have to be called instead of operations with pointers, therefore, results may differ. Although it should be noted that the common technique of caching, can amortize running time. Nevertheless, the results of the present study show that in the worst-case scenario D* Extra Lite performs less operations than D* Lite. Therefore, irrespective of implementation, D* Extra Lite is the best choice for difficult, dynamic problems.

The experimental results presented in the paper were gained from tests conducted with an observation range of 10 map cells. In Table 4.3 average total running times are presented for the planning with freespace assumption on *wc3* maps with different observation ranges. Across all three algorithms, running time changed slightly with longer observation ranges. This was the result of more observed changes in the environment, which increased the extent to which the search-tree became inconsistent. However, with a longer observation range, more information was gathered. Nevertheless, the observed relationship between the total running time of each of the analyzed algorithms, was maintained across the different observation ranges.

4.7 Conclusions

In this chapter, several incremental path-planning algorithms, including the recent MPGAA* [12], the popular Focussed D* [9], and the currently state-of-the-art D* Lite [10], were analyzed. In addition, older ideas, such as Differential A* [61], were revisited. In order to provide improved insight into properties of incremental heuristic search algorithms, a new benchmark scenario was proposed. This scenario involved planning for both shortcuts and barriers. Considering the results of the analysis, a novel robust D* Extra Lite algorithm was proposed. In typical two-dimensional navigation problems D* Extra Lite outperforms both D* Lite (optimized version) and MPGAA*. In addition to comprehensive tests, the worst-case

complexity analysis was conducted, which showed that, independently of a particular domain and implementation, D* Extra Lite is faster than D* Lite.

D* Extra Lite is a general purpose, incremental shortest-path algorithm able to work on directed and undirected graphs. It is almost as simple as a regular A* algorithm, only extended with search-tree cutting and frontier-gap repairing. A strong advantage of the D* Extra Lite algorithm to D* Lite, is that it performs branch-cutting as a simple, recursive operation that makes nodes unvisited without the use of complex operations on the open-list.

Additionally, the D* Extra Lite algorithm can be extended easily to an anytime incremental search, as it will be shown in Chapter 5.

5. AD*-Cut: Anytime Incremental Planning

In complex environments, in which a short computation time is more important than the optimality, anytime planning can be used that typically quickly provides a sub-optimal solution to improve the remaining time. A combination of anytime and incremental search algorithms allows for solving complex problems in a changeable environment.

In this chapter, the AD*-Cut algorithm, an anytime version of D* Extra Lite (described in Ch. 4), is proposed. To the author's best knowledge, search-tree branch cutting has not been used for an anytime incremental search yet; hence, AD*-Cut, which is presented in this chapter, is a novel approach. The algorithm is tested on benchmark problems, and it is compared with AD* [32], a state-of-the-art anytime incremental shortest path search algorithm.

5.1 Introduction

5.1.1 Anytime Planning

Anytime planning refers to algorithms that aim to find any sub-optimal solution as quickly as possible and to incrementally improve it in the remaining time. Although anytime algorithms do not provide any warranties on execution time, they are able to perform global planning very quickly, even for complex planning problems, such as 20-degrees-of-freedom robotic arm motion planning [6] or alignment of multiple DNA or protein sequences [7]. This is possibly due to a specific property of a heuristic search. If an overestimating (inadmissible) heuristic cost $f(s) = g(s) + \epsilon \cdot h(s)$ with $\epsilon \geq 1$ is used (weighted A* [6, 7]), then the higher the ϵ, the more the search-tree expansion is focused toward a goal state (it is more greedy). Although the solution is no longer optimal, its sub-optimality is bounded by a factor of ϵ.

A naive implementation of anytime A* may run a new search from scratch, decreasing ϵ each time, until time for planning is over. However, in such an approach, a subsequent search does not reuse information from previous searches; thus, it unnecessarily revisits some nodes. A basic idea to improve the overall performance of an anytime search is to prune states that cannot

belong to the path that is shorter than the path initially found. This is straightforward; if the g_ϵ is the cost of the path initially found with the heuristic inflated by ϵ, then, g^* it must be that $g^* \leq g_\epsilon$ (g_ϵ is the upper bound) for the true least cost g^*; hence, assuming that $h(s)$ is admissible, a state with $f(s) = g(s) + h(s) > g_\epsilon$ cannot belong to the shortest path. This technique has been used in Anytime A* [73] and its improved version Anytime-WA* [7]. An upper bound of the f-value obtained from a previous search is also used for a search-loop termination in Anytime Repairing A* (ARA*) [6], another anytime heuristic search algorithm.

Algorithm 5.1 Anytime repairing A* (ARA*)

```
 1: function F-VALUE(s)
 2:     return g(s) + ε · h(s)
 3: function IMPROVEPATH()
 4:     while F-VALUE(s_goal) > min_{s∈OPEN}(F-VALUE(s)) do
 5:         s =TOPOPEN()
 6:         POPOPEN()
 7:         closed(s) = true
 8:         for all s' ∈ Succ(s) do
 9:             if NOT visited(s') then
10:                 g(s') = ∞
11:             if g(s') > g(s) + cost(s, s') then
12:                 g(s') = g(s) + cost(s, s')
13:                 if NOTclosed(s') then
14:                     PUSHOPEN(s',F-VALUE(s'))
15:                 else
16:                     PUSH(s',INCONS)
17: function MAIN(s_start, s_goal, ε, ε_step)
18:     g(s_goal) = ∞
19:     g(s_start) = 0
20:     PUSHOPEN(s_start,F-VALUE(s_start))
21:     IMPROVEPATH()
22:     ε' = min(ε, g(s_goal)/ min_{s∈OPEN∪INCONS}(g(s) + h(s)))
23:     PUBLISHCURRENTSOLUTION()                    ▷ ε'-suboptimal solution
24:     while ε' > 1 do
25:         ε = ε − ε_step
26:         TO_REOPEN = INCONS ∪ OPEN
27:         CLOSED = OPEN = ∅
28:         for all s ∈ TO_OPEN do
29:             PUSHOPEN(s,F-VALUE(s))
30:         IMPROVEPATH()
31:         ε' = min(ε, g(s_goal)/ min_{s∈OPEN∪INCONS}(g(s) + h(s)))
32:         PUBLISHCURRENTSOLUTION()                ▷ ε'-suboptimal solution
```

The pseudocode of ARA* is shown in Algorithm 5.1. In general, a heuristic search with an inflated heuristic expands new states in an order that does not ensure the g-value will be minimal. Therefore, it is likely that already closed states are reopened due to g-value improvement. In ARA*, such states are placed on the inconsistent list ($INCONS$) and are reopened in a subsequent search with decreased ϵ, which significantly speeds up the algorithm.

The anytime algorithms discussed so far require additional parameters to be set beforehand (e.g., the initial ϵ and decrease step ϵ_{step}). This is not the case of the Anytime Nonparametric A* (ANA*) that outperforms ARA* in most tests [8].

The ANA* also uses an f-value upper bound for state pruning; however, it owes its predominance over ARA* to the novel open-list maintenance technique. In contrast to typical heuristic search algorithms that, in each loop iteration, pop a state with the least f-value, in ANA*, a state with the highest e-value is popped. The e-value is defined by Eq. 5.1, where G is the upper bound of the f-value obtained from the previous search (initially set to a very large number).

$$e(s) = \frac{G - g(s)}{h(s)} \tag{5.1}$$

As $e(s)$ is growing with decreasing $h(s)$ and it is decreasing with growing $g(s)$, ANA* is greedily progressing toward a goal state and simultaneously avoiding search directions in which the both $h(s)$ and $g(s)$ values are growing. This greedy effect gradually fades with decreasing G, which is continuously improved at the end of each search episode.

5.1.2 Anytime Incremental Planning

As already mentioned, incremental search algorithms can be combined with anytime search methods. Anytime D* (AD*) [32], is based on ARA* and D* Lite, is the most recognizable contribution to date to address the problem of anytime incremental search. Other incremental search algorithms that are also based on ARA* are Anytime Truncated D* [74] and the most recent Anytime Tree-Restoring A* [75], which extends Tree-Restoring A*. As presented in this thesis, the AD*-Cut (Sec. 5.2) algorithm is compared to AD*, which will be discussed in detail.

The pseudocode of AD* is shown in Algorithm 5.2. As AD* is based on ARA* and D* Lite, common procedures can be recognized. The method for affected node reinitialization, which relies on checking the rhs-value, is taken from D* Lite; thus, the procedures KEY, INITIALIZE, UPDATESTATE, and SEARCH are similar to those of D* Lite (Alg. 4.2). Similarly to ARA*, AD* avoids reopening nodes by placing them on the $INCONS$ list (line 18 in Alg. 5.2) and performs subsequent searches with a decreased heuristic inflation factor, ϵ, which is preceded by an open-list re-evaluation (lines 30–35 and 41–45 in Alg. 5.2).

In connection with anytime incremental search algorithms, a question arises regarding how to interleave map updates with path improvements. As proposed in AD* [32], there is one main loop that is responsible for map updates, algorithm reinitialization, and navigation toward the goal and one search loop that is responsible for path improvement with the given ϵ. The decision to be made is to decrease ϵ and continue the path improvement or to restart the search algorithm

Algorithm 5.2 Anytime D* (AD*)

```
1: function KEY(s)
2:     if g(s) > rhs(s) then return [g(s) + ε · h(s); rhs(s)]
3:     else return [g(s) + h(s); g(s)]
4: function INITIALIZE()
5:     g(s_start) = rhs(s_start) = ∞
6:     g(s_goal) = ∞
7:     rhs(s_goal) = 0
8:     PUSHOPEN(s_goal,KEY(s_goal))
9: function EVALUATERHS(s)
10:    rhs(s) = ∞
11:    for all s' ∈ Succ(s) do rhs(s) = min (rhs(s), cost(s, s') + g(s'))
12: function UPDATESTATE(s)
13:    if NOT visited(s) then g(s) = ∞
14:    if s ≠ s_goal then EVALUATERHS(s)
15:    if open(s) then REMOVEOPEN(s)
16:    if g(s) ≠ rhs(s) then
17:        if NOT closed(s) then PUSHOPEN(s, KEY(s))
18:        else PUSH(s', INCONS)
19: function SEARCH()
20:    while KEY(s_start) > min_{s∈OPEN}(KEY(s)) OR rhs_start ≠ g(g_start) do
21:        s =TOPOPEN()
22:        POPOPEN()
23:        if g(s) > rhs(s) then
24:            g(s) = rhs(s)
25:            closed(s) = true
26:            for all s' ∈ Pred(s) do UPDATESTATE(s')
27:        else
28:            g(s) = ∞
29:            for all s' ∈ Pred(s) ∪ {s} do UPDATESTATE(s')
30: function REEVALUATEOPEN()
31:    TO_OPEN = INCONS ∪ OPEN
32:    CLOSED = OPEN = ∅
33:    for all s ∈ TO_OPEN do
34:        if visited(s) AND NOT open(s) then
35:            PUSHOPEN(s, CALCULATEKEY(s))
36: function REINITIALIZE()
37:    if any edge cost changed then
38:        for all directed edges (u, v) with changed cost do
39:            update edge cost cost(u, v)
40:            UPDATESTATE(u)
41:    if significant edge-cost changes then
42:        ε = ε_init AND/OR plan from scratch
43:    else if ε > 1 then
44:        ε = ε − ε_step
45:     REEVALUATEOPEN()
46: function MAIN()
47:    (...)                                          ▷ Same as in Alg. 4.1.
```

with increased (possibly initial) ϵ, which may be a better choice due to big changes in the map; AD* makes such a decision after each path improvement.

5.1.3 Conclusions

All algorithms based on a weighted A* owe their efficiency to the greedy nature of searching with an inflated heuristic; however, this is not always true. Wilt and Ruml [76] have shown that a greedy search performs well in problems in which a heuristic cost does not differ significantly from the actual cost; thus, examples can be found for which heuristic inflation slows the search. Nevertheless, for typical robot motion planning applications, such as robotic manipulator motion planning [6, 32] or autonomous car navigation [18], good heuristics can be computed.

5.2 AD*-Cut Algorithm

The AD*-Cut algorithm is designed to perform a time-limited anytime search followed by a map update and affected node reinitialization, in accordance to the MAIN procedure in Algorithm 4.1. The pseudocode of AD*-Cut is shown in Algorithm 5.3. The AD*-Cut algorithm combines the search-tree cutting technique used by D* Extra Lite with anytime repairing used by ARA*. Consequently, the procedures KEY, SOLUTIONFOUND, SEARCHSTEP, SEARCH, and REEVALUATEOPEN (lines 1–47 in Alg. 5.3), to a large extent, correspond to instructions of ARA* (cf. procedures FVALUE, IMPROVEPATH, and MAIN [6]). Furthermore, the procedures REINITIALIZE, CUTBRANCH, and CUTBRANCHES (lines 48–84 in Alg. 5.3) correspond to the procedures of D* Extra Lite (Ch. 4, also in [13]). In the remainder of this section, the overall operation of the algorithm is explained with a discussion of modifications that are specific to AD*-Cut.

An anytime search loop (function SEARCH in Algorithm 5.3) performs multiple searches, starting with $\epsilon = \epsilon_{init}$ and decreasing by ϵ_{step} in subsequent searches, which is merely ARA*. The ϵ value is the factor by which a heuristic is inflated, making the algorithm more greedy and possibly quicker (line 2, Alg. 5.3). Moreover, the algorithm does not allow for reopening states; such states are placed in the list of inconsistent nodes (lines 23–26, Alg. 5.3), which additionally speeds up the algorithm. After a solution with a given ϵ is found, ϵ is decreased (line 36, Alg. 5.3) and all nodes from $OPEN$ and $INCONS$ lists are reopened with new keys (lines 37 and 42–47, Alg. 5.3). The search loop runs until the optimal solution is found (then $\epsilon = 1$) or granted time elapses but not before the first solution is found (line 38, Alg. 5.3). To this point, the only modification with respect to ARA* is that each node holds an additional flag

$visited(s)$ that is maintained to recognize nodes cut in the reinitialization (lines 21–22 and 46, Alg. 5.3).

In the reinitialization, branch cutting is executed if the cost of any visited edge has changed (lines 49–56, Alg. 5.3), otherwise only the open-list re-evaluation is done (line 57, Alg. 5.3). For each edge with a changed cost, the CUTBRANCHES procedure does one of two possible operations.

If the cost of the $e(u, v)$ edge has decreased, the v node is added to the list of seeds to be reopened later (lines 75, 78 in Alg. 5.3). In the case of an edge-cost decrease, there may be a shorter path. Therefore, to preserve optimality, the start node should be reopened. However, not in every case of an edge-cost decrease, the start node need to be reopened. Assuming that $h(s_{start}, u)$ is admissible, for a decreased $e(u, v)$ edge cost, the start node s_{start} requires reopening only if $g(s_{start}) > g(v)+cost(u, v)+\epsilon \cdot h(s_{start}, u)$ (lines 76–77 and 82–83, Alg. 5.3).

If the cost of the $e(u, v)$ edge has increased and node v is the parent of node u, the branch is cut, starting from u (lines 79–81 in Alg. 5.3). The cutting operation marks a node as unvisited, resets its parent, and removes it from the $OPEN$ or $INCONS$, wherever it is placed (lines 59–62, Alg. 5.3). The CUTBRANCH() procedure is the recursive procedure that traverses throughout the branch (i.e., the next node to cut s' must be such a predecessor of a current node s that the s is the parent of s' (lines 66–68 in Alg. 5.3). Each successor node s', such that $s \neq parent(s')$, is placed in the list of seeds (lines 64–66 in Alg. 5.3). Although seeds are simply nodes to reopen, as they might be cut later, they cannot be merely pushed to the open list. Following the CUTBRANCHES() procedure, the REINITIALIZE() procedure pushes to the open list only these nodes from the $seeds$ list that remain visited and are not already open (lines 54–56 in Alg. 5.3). This operation repairs the frontier gap made by branch cutting.

The AD*-Cut algorithm reserves a certain time for as much path improvement as possible, each with decreased ϵ. After a timeout, it returns to the main loop (procedure SEARCH in Alg. 5.3), in which the map update is performed. Although AD*-Cut, as presented in Algorithm 5.3, does not perform ϵ reinitialization, it is possible to apply some reinitialization rules, for example, based on a measure of map changes, like in AD* (lines 41–42 in Alg. 5.2).

5.3 Benchmark Results

In the experiments, AD*-Cut and AD* [32] were compared. Both algorithms were using the same implementation of a heap and domain-specific functions, such as successor and

Algorithm 5.3 AD*-Cut. Required parameters: $\epsilon_{init}, \epsilon_{step}$.

```
 1: function CALCULATEKEY(s)
 2:     return g(s) + ε · h(s_start, s)
 3: function SOLUTIONFOUND()
 4:     k_start = CALCULATEKEY(s_start)
 5:     k_top = CALCULATEKEY(TOPOPEN())
 6:     return visited(s_start) AND k_start ≤ k_top
 7: function INITIALIZE()
 8:     ε = ε_init
 9:     visited(s_goal) = true
10:     parent(s_goal) = NULL
11:     g(s_goal) = 0
12:     PUSHOPEN(s_goal, CALCULATEKEY(s_goal))
13: function SEARCHSTEP()
14:     s =TOPOPEN()
15:     POPOPEN()
16:     closed(s) = true
17:     for all s' ∈ Pred(s) do
18:         if NOT visited(s') OR g(s') > cost(s', s) + g(s) then
19:             parent(s') = s
20:             g(s') = cost(s', s) + g(s)
21:             if NOT visited(s') then
22:                 visited(s') = true
23:             if closed(s') AND ε > 1 then
24:                 if NOT inconsistent(s') then
25:                     inconsistent(s') = true
26:                     PUSH(s', INCONS)
27:             else
28:                 PUSHOPEN(s', CALCULATEKEY(s'))
29: function SEARCH()
30:     found = false
31:     while open-list is not empty do
32:         if SOLUTIONFOUND() then
33:             found = true
34:             if ε = 1 then                          ▷ As the optimal solution is found,
35:                 return found                       ▷ the search loop terminates.
36:             ε = ε − ε_step
37:             REEVALUATEOPEN()
38:         if found AND time elapsed then             ▷ The search loop runs until
39:             return found                           ▷ the first solution or a timeout.
40:         SEARCHSTEP()
41:     return found
```

predecessor generation, action cost, heuristic (the Euclidean distance) and affected state computation.[1]

[1]The implementation of AD* and domain-specific functions were obtained from the SBPL library `http://sbpl.net/`.

```
42: function REEVALUATEOPEN()
43:     TO_OPEN = INCONS ∪ OPEN
44:     CLOSED = OPEN = ∅
45:     for all s ∈ TO_OPEN do
46:         if visited(s) AND NOT open(s) then
47:             PUSHOPEN(s, CALCULATEKEY(s))
48: function REINITIALIZE()
49:     if any edge cost changed then
50:         CUTBRANCHES()
51:         REEVALUATEOPEN()
52:         if SEEDS ≠ ∅ then
53:             for all s ∈ SEEDS do
54:                 if visited(s) AND NOT open(s) then
55:                     PUSHOPEN(s, CALCULATEKEY(s))
56:             SEEDS = ∅
57:     else REEVALUATEOPEN()
58: function CUTBRANCH(s)
59:     visited(s) = false
60:     inconsistent(s) = false
61:     parent(s) = NULL
62:     REMOVEOPEN(s)
63:     for all s' ∈ Succ(s) do
64:         if visited(s') AND NOT parent(s') = s then
65:             SEEDS = SEEDS ∪ s'
66:     for all s' ∈ Pred(s) do
67:         if visited(s') AND parent(s') = s then
68:             CUTBRANCH(s')
69: function CUTBRANCHES()
70:     reopen_start = false
71:     for all directed edges (u, v) with changed cost do
72:         if visited(u) AND visited(v) then
73:             c_old = cost(u, v)
74:             update edge cost cost(u, v)
75:             if c_old > cost(u, v) then
76:                 if g(s_start) > g(v) + cost(u, v) + ε · h(s_start, u) then
77:                     reopen_start = true
78:                 SEEDS = SEEDS ∪ v
79:             else if c_old < cost(u, v) then
80:                 if parent(u) = v then
81:                     CUTBRANCH(u)
82:     if reopen_start = true AND visited(s_start) then
83:         SEEDS = SEEDS ∪ s_start
84: function MAIN()
85:     (...)                                          ▷ Same as in Alg. 4.1.
```

The benchmark problems and map sets *wc3*, *rooms*, and *mazes16* (mazes with the corridor width of 16 map cells) were obtained from the benchmark prepared by [72]. For each map set, 100 problems of similar length (from 400 to 440 map cells) were solved.

The algorithms were tested with the freespace assumption, in which obstacles are only added, which is a harder case for D*-like algorithms than an obstacle disappearance (Sec. 4.4, also in [13]). To ensure comparable conditions for both algorithms, a robot was moving along a precomputed optimal path, which is a common assumption [75]. After each robot step, a map-update function was called. The map-update function simulates a 360° rangefinder working with a resolution of 1° and an observation range of 100 map cells.

The search space was a (x, y, yaw) state lattice (a 2D position with rotation) with a total size of 512x512x16 states (16 possible orientations) and seven applicable actions (motion primitives) per state. The longest motion primitive was eight map cells long. The simulated robot was 10 map cells wide and long, with the exception for tests on the *rooms* map set, in which the robot was the size of a single map cell (due to narrow passages of a single cell width).

In Table 5.1, the results of planning within a 1 s time limit with $\epsilon_{init} = 5$ and $\epsilon_{step} = 0.2$ are presented. The algorithms were allowed to exceed the 1 s time limit when they were searching for the first solution. The following parameters were logged: reinitialization time (excluding time necessary for map-change computation), time until first solution, overall search time, loop time (total time spent in a single iteration of the main loop, including map updates), search step counts, average ϵ, and average path cost. The overall performance ratio is presented at the bottom of Table 5.1.

On average, AD*-Cut provided the first solution 1.47 times faster than AD*. Furthermore, it performed path improvements 4.65 faster and accomplished each main loop iteration 1.41 faster than AD*. The average ϵ value achieved by AD* was 1.35 times higher than that of AD*-Cut. Consequently, the average path cost returned by AD* was 1.05 higher than that of AD*-Cut.

The possible advantage of AD* over AD*-Cut is seen when the maximum times are analyzed. The AD*-Cut algorithm is more sensitive to situations in which a large part of the search tree needs to be cut, but without substantial change in the cost of a re-planned path. However, unlike AD*, AD*-Cut was run without additional improvements, such as planning from scratch in the case of large changes in the map.

Table 5.1: The experimental results for planning with freespace assumption on (x, y, yaw) state lattice. The presented values are calculated as average values per single re-planning episode after solving 100 problems for each map set.

Map Set	Algorithm	Reinit. Time [s]		First Solution Time [s]		Search Time [s]		Loop Time [s]		#Search Steps	Avg. ϵ	Avg. Path Cost
		Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max			Ratio
wc3	AD*-Cut	0.274	4.037	0.115	2.677	0.171	2.819	0.708	5.372	41327	1.14	1.00
	AD*	0.057	0.161	0.152	2.038	0.697	2.239	0.757	2.335	30003	1.58	1.11
rooms	AD*-Cut	0.183	6.258	0.068	2.363	0.099	2.469	0.293	6.618	29292	1.07	1.00
	AD*	0.004	0.031	0.069	0.871	0.646	1.323	0.650	1.332	21453	1.26	1.03
mazes 16	AD*-Cut	0.277	11.951	0.147	6.974	0.176	6.996	0.535	13.059	39664	1.08	1.00
	AD*	0.029	0.145	0.265	6.693	0.729	6.816	0.759	6.903	33996	1.59	1.02
Performance Ratio	AD* / AD*-Cut	0.12	0.02	**1.47**	0.80	**4.65**	0.84	**1.41**	0.42	0.77	**1.35**	**1.05**

5.4 Conclusions

Search-tree branch cutting is a simple reinitialization technique utilized by the D* Extra Lite algorithm (Ch. 4). This technique can be easily used in combination with the ARA* algorithm [6] resulting in AD*-Cut, a new anytime incremental search algorithm. The results of planning on (x, y, yaw) state lattices suggest that AD*-Cut is quicker and achieves shorter paths than AD*, the state-of-the-art anytime incremental search algorithm [32].

In the future, AD*-Cut will be compared with Anytime Truncated D* [74] and Anytime Tree-Restoring A* [75], and the problem of time-consuming reinitialization will be addressed.

6. Planning in a Dynamic Environment

In this chapter, the problem of path planning in the presence of moving obstacles is discussed. As the presence of moving obstacles imposes time as an additional dimension of a state space, this problem will be referred to as time-dependent planning. In particular, the presented discussion focuses on the time-dependent planning with the use of heuristic search algorithms working on grid-based maps and state lattices that are the state-of-the-art search-space representations for mobile robot motion planning.

A basic notation for time-dependent planning is given in Section 6.1. The next section (Sec. 6.2) presents related work. In Section 6.3, an event-based state-time space decomposition is described, which is a generalization of the safe intervals method (SIPP)[22] and obstacle layers method presented by the author in [23] and can also be used in action planning. Using an event-based state-time space decomposition, time-dependent planning is analyzed for its applicability in a forward heuristic search (Sec. 6.4), a learning real-time heuristic search (Sec. 6.7.1), and an anytime heuristic search (Sec. 6.7.2).

6.1 State-time space definition

A basic notation for motion planning used in this chapter is similar to the one used in Chapter 3, which is as follows:

- $\mathcal{W}$: a workspace,
- $\mathcal{A}$: a robot (a robotic car in Fig. 6.1a),
- $\mathcal{O}$: an obstacle region in $\mathcal{W}$,
- $\mathcal{C} = \{q_1, q_2, \ldots\}$: a configuration space,
- $S = \{s_1, s_2, \ldots\}$: a state space, such that $s = (q, \dot{q})$, where $q \in \mathcal{C}$,
- $A = \{a_1, a_2, \ldots\}$: a set of actions applicable in S, such that for each action $a = \langle s_1, s_2 \rangle$, the begin state $s_1 \in S$ and the end state $s_2 \in S$ are defined,
- $\tau(a, p) : A \times [0, 1] \rightarrow S$: a motion primitive (a continuous trajectory in a state space) represented by an action a,
- $\gamma(a = \langle s_1, s_2 \rangle) = s_2 : A \rightarrow S$: a transition function that returns an action end state,

- $\gamma^{-1}(a = \langle s_1, s_2 \rangle) = s_1 : A \to S$: an inverse transition function that returns an action begin state.

A problem of path planning with moving obstacles requires time as an additional dimension of a search space; thus, we have $\mathcal{CT} = \mathcal{C} \times T$ and $\mathcal{ST} = S \times T$ for a configuration-time space [77] and state-time space [21], respectively. (As state-time space is an enhancement of a configuration-time space, further discussion is conducted only for $\mathcal{ST}$.) As obstacles can move, an obstacle region is a function of time $\mathcal{O}(t)$. In particular, the moving obstacle region is an union of regions occupied by many obstacles $\mathcal{O}_{dynamic}(t) = \bigcup_{i=1}^{n} \mathcal{O}_i(t)$, such that $\mathcal{O}_i(t)$ reflects an occupied workspace along i-th obstacle trajectory.

In the dynamic situation shown in Figure 6.2a, that will be used throughout this chapter as a leading example, it is possible to control a robotic car $\mathcal{A}$ that can move only forward and backward along a road, x, which generates a 1D state space. The road is crossing two tracks along which trains, $\mathcal{O}_1$ and $\mathcal{O}_2$, are moving. For such a situation, a 2D state-time space can be constructed (Fig. 6.2b, c). As the car is initially crossing the first track (Fig. 6.2a), depending on train speeds and the dynamic constraints of the car, it can drive back and let both trains cross the road (Fig. 6.2b), or it can accelerate to cross the track before the first train and slow to let the second train cross the road (Fig. 6.2c).

After the introduction, definitions specific to time-dependent planning can be given:

- $T = \{t_1, t_2, \ldots\}$: a set of time points,
- $\mathcal{O}(t) = \bigcup_{i=1}^{n} \mathcal{O}_i(t) \cup \mathcal{O}_{static}$: an obstacle region at a given time point t that is a union of regions occupied by moving obstacles $\mathcal{O}_i(t)$ at t and the static obstacle region $\mathcal{O}_{static}$,
- $\mathcal{ST} = S \times T$: a state-time space, where $\tilde{s} \in \mathcal{ST}$ is a state holding at time t defined as $\tilde{s} = (s, t)$,
- $\mathcal{ST}_{obs} = \{(s, t) \in \mathcal{ST} | \mathcal{A}(s, t) \cap \mathcal{O}(t) \neq \emptyset\}$: a set of collision states that will be considered an open set,
- $\mathcal{ST}_{free} = \mathcal{ST} \setminus \mathcal{ST}_{obs}$: a set of collision-free states that is a complement of $\mathcal{ST}_{obs}$.

It is important to note that $\mathcal{ST}_{free}$ and $\mathcal{ST}_{obs}$ can be viewed as static regions in a state-time space (e.g., Fig. 6.1b,c).

For a state-time space, an action-time space $\mathcal{AT}$ can be defined as $\mathcal{AT} = A \times T$, such that an action $\tilde{a} \in \mathcal{AT}$ is defined as $\tilde{a} = \langle \tilde{s_1}, \tilde{s_2} \rangle$, where $\tilde{s}_1 = (s_1, t_1), \tilde{s}_2 = (s_2, t_2) \in \mathcal{ST}$. In time-dependent planning, it is assumed that, for each action, end time t_2 must be greater than begin time t_1. In other words, actions cannot have zero or negative duration.

For an action $\tilde{a} \in \mathcal{AT}$, the following functions can be defined:

- $begin(\tilde{a}) : \mathcal{AT} \to T$ returns an action begin time t_1,
- $end(\tilde{a}) : \mathcal{AT} \to T$ returns an action end time t_2,
- $duration(\tilde{a}) = end(\tilde{a}) - begin(\tilde{a}) : \mathcal{AT} \to T$ returns an action duration,

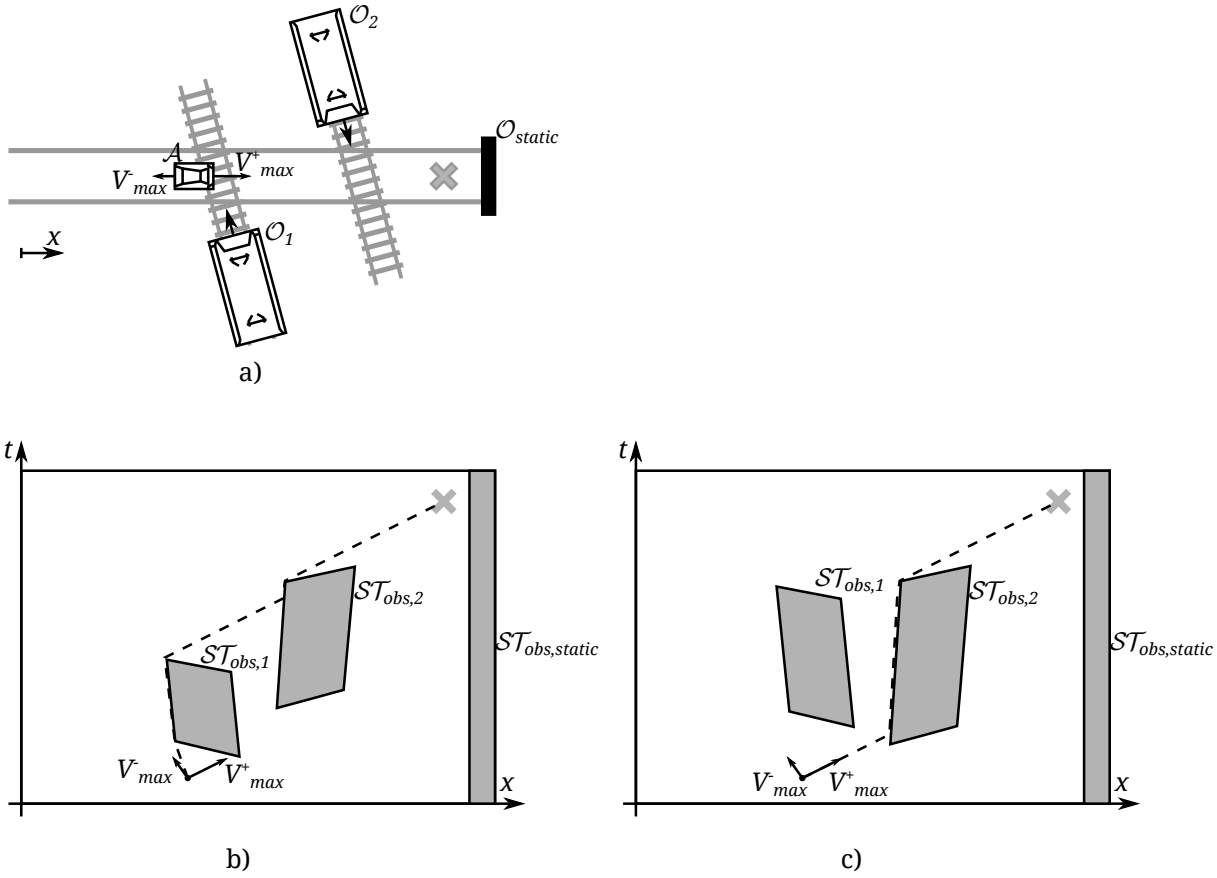


Figure 6.1: Example of (a) time-dependent planning and (b and c) the state-time spaces for the two sample situations. $\mathcal{A}$: a car, $\mathcal{O}_1$,$\mathcal{O}_2$: two trains, $\mathcal{O}_{static}$: a static obstacle, v_{max}: the maximum speed of the car, v_{min}: the minimum speed of the car (reverse speed), dot: a start state, cross: a goal state, dashed-line; the shortest-path in a state-time space.

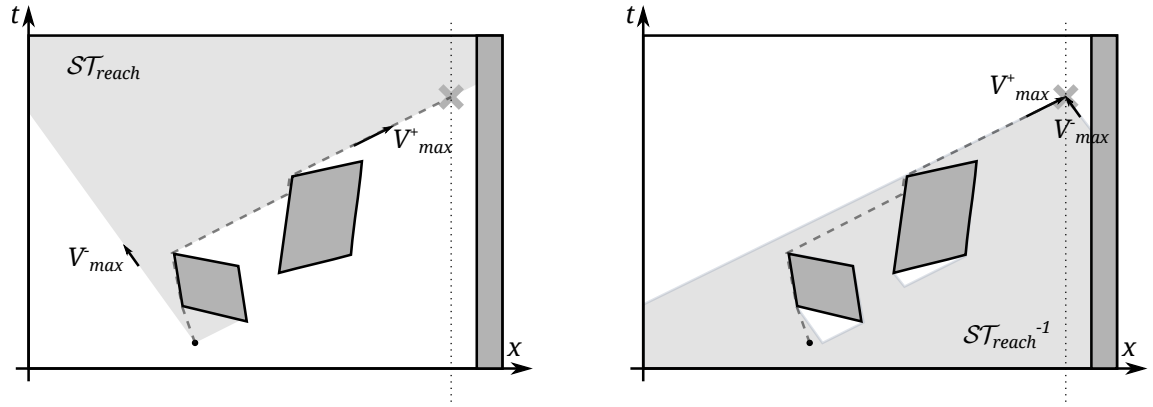
- $\tilde{\tau}(\tilde{a}, p) : \mathcal{AT} \times [0, 1] \rightarrow \mathcal{ST}$ is a motion primitive (continuous trajectory) represented by an action $\tilde{a}$.

As actions have duration, only part of $\mathcal{ST}$ is reachable from a given starting state, namely, a reachable set, $\mathcal{ST}_{reach}$ (Fig. 6.2a). Consequently, a given state is reachable only from some part of $\mathcal{ST}$, namely, a backward-reachable set, $\mathcal{ST}^{-1}_{reach}$ (Fig. 6.2b).

Reachable sets can be obtained with the following recursive functions:

- $reach(\tilde{s}) = \{\tilde{s}\} \cup \bigcup_{\tilde{a}=\langle \tilde{s}, \tilde{s}' \rangle \in \mathcal{AT} | \tilde{s}' \in \mathcal{ST}_{free}} reach(\tilde{s}') : \mathcal{ST}_{free} \rightarrow 2^{\mathcal{ST}_{free}}$: a function that returns a set of all future collision-free states reachable starting from a given state,
- $reach^{-1}(\tilde{s}) = \{\tilde{s}\} \cup \bigcup_{\tilde{a}=\langle \tilde{s}', \tilde{s} \rangle \in \mathcal{AT} | \tilde{s}' \in \mathcal{ST}_{free}} reach^{-1}(\tilde{s}') : \mathcal{ST}_{free} \rightarrow 2^{\mathcal{ST}_{free}}$: a function that returns a set of all past collision-free states from which a given state is reachable.

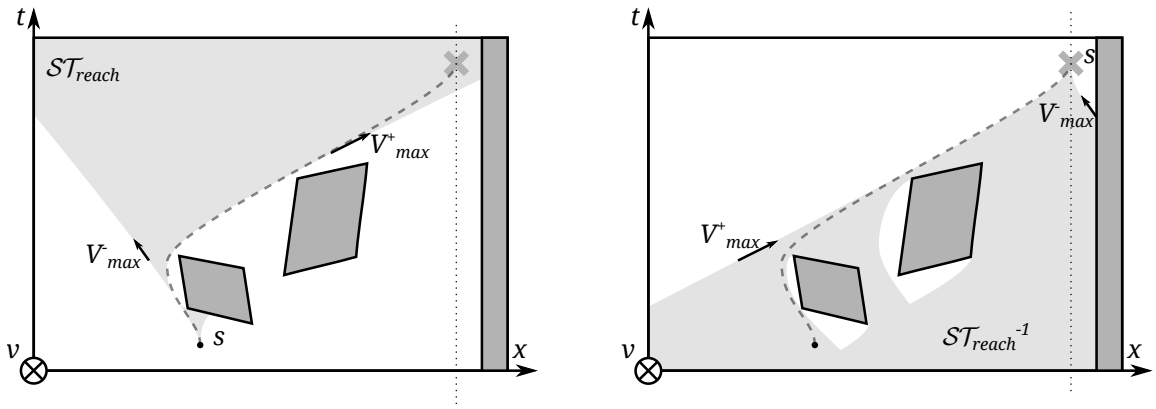
By definition, reachable sets are subsets of $\mathcal{ST}_{free}$; hence, $reach(\tilde{s}) \in 2^{\mathcal{ST}_{free}}$ and $reach^{-1}(\tilde{s}) \in 2^{\mathcal{ST}_{free}}$, where $2^{\mathcal{ST}_{free}}$ is a power set of $\mathcal{ST}_{free}$. Reachable sets are constructed recursively (i.e., starting from a given state $\tilde{s}$, $\tilde{s}$ is added to a reachability set, then the same



a) b)

Figure 6.2: Sample reachable sets in a state-time space: a) $\mathcal{ST}_{reach} = reach(\tilde{s})$ is a reachable set starting from state $\tilde{s}$, b) $\mathcal{ST}^{-1}_{reach} = reach^{-1}(\tilde{s})$ is backward reachable (i.e., is a set from which state $\tilde{s}$ is reachable).

procedure is repeated for each successor state $\tilde{s}'$ of $\tilde{s}$ (or predecessor state $\tilde{s}'$, in case of $reach^{-1}(\tilde{s})$)). In addition to that, the shape of a reachable set depends on constraints on actions, which can be of different types (e.g., acceleration limits or domain-specific constraints, such as one-way roads). In Figure 6.2, the reachable sets are constrained only by the maximum forward and reverse speeds (i.e., there are no limits on a speed change). If acceleration limits are considered, a reachable set can have a more elaborate shape (Fig. 6.3). In the presence of dynamic constraints, for the example from Figure 6.1a, a state-time space is 3D, such that $\tilde{s} = (q, \dot{q}, t)$; therefore, Figure 6.3 depicts a projection on the q, t plane.



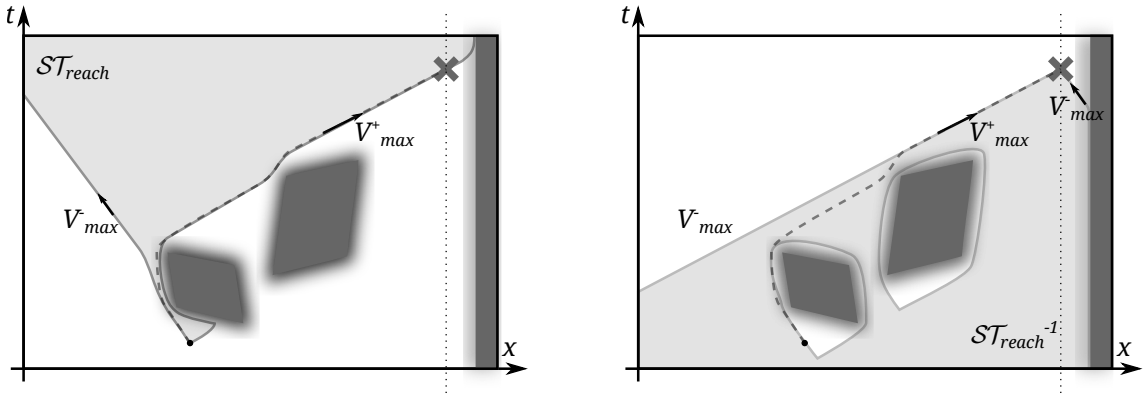
a) b)

Figure 6.3: Sample reachable sets in a 3D state-time space (i.e., $(q, \dot{q}, t)$-space) under velocity and acceleration limits (i.e., $\dot{q} \in < v_{min}, v_{max} >$ and $\ddot{q} \in < a_{min}, a_{max} >$), where (a) $\mathcal{ST}_{reach} = reach(\tilde{s})$ is a reachable set starting from a state $\tilde{s}$ (with a speed equal to 0) and (b) $\mathcal{ST}^{-1}_{reach} = reach^{-1}(\tilde{s})$ is a set from which a state $\tilde{s}$ (with 0 velocity) is reachable.

In mobile robot motion planning, it is a common approach to use a cost map that provides costs of the robot presence at a given point in a workspace (consequently, at a state in a state space). A cost may reflect the collision probability or distance from an obstacle. Cost-map values, provided by function $cm(\tilde{s}) : \mathcal{ST} \rightarrow [0, 1]$, can be incorporated into time-dependent planning as a constraint on the maximum speed at a given state $\tilde{s}$ in a state space, such that, for each action $\tilde{a} = \langle \tilde{s_1}, \tilde{s_2} \rangle$, where $\tilde{s_1} = (s_1, t_1)$ and $\tilde{s_2} = (s_2, t_2)$, the following holds:

$$\frac{\|s_2 - s_1\|}{t_2 - t_1} \leq cm(\tilde{s_1}) \cdot v_{max}. \tag{6.1}$$

The sample reachable sets under cost-map constraints are shown in Figure 6.4, in which one can note the smoothed borders of these sets.



a) b)

Figure 6.4: Sample reachable sets under cost-map constraints: a) $\mathcal{ST}_{reach} = reach(\tilde{s})$ is a reachable set starting from a state $\tilde{s}$, b) $\mathcal{ST}^{-1}_{reach} = reach^{-1}(\tilde{s})$ is a set from which a state $\tilde{s}$ is reachable. Blurred regions depict cost-map values ranging from 0 to 1, depending on the distance from $\mathcal{ST}_{obs}$.

In Figures 6.2b, 6.3b, and 6.4b, which represent backward-reachable sets, unreachable regions around obstacles can be observed. In fact, these are regions of inevitable collision $\mathcal{ST}_{ric}$, as proposed in [78]. It can be proposed that for each state $\tilde{s}$ in a region of inevitable collision $\mathcal{ST}_{ric}$ a reachable set obtained with $reach(\tilde{s})$ is bounded (in space by the robot velocity limits and in time by $\mathcal{ST}_{obs}$).

6.2 Related Work

State-space sampling or state-space decomposition methods, discussed in Section 3.1.2, which are cell decomposition, Voronoi diagrams, visibility diagrams, random sampling, and discretization (regular grids or lattices), can also be applied to a state-time space. Particular

algorithms for motion planning amid moving obstacles can combine different sampling or decomposition methods for a space and for time dimensions. For example, while the algorithms described in [33, 79] combine space random sampling with evenly discretized time, the algorithms described in [22, 80, 23] use regular grids for a space representation and combinatorial cell decomposition for the time dimension.

Combinatorial Search Methods

If obstacles and a robot are polygons (or polyhedra), and obstacle movements are piecewise linear, then $\mathcal{ST}_{obs}$ is also a polygon (or a polyhedron). In such a case, exact methods, such as a cylindrical decomposition [81] or visibility graph [77], can be used to construct a search graph. In both methods, time is sliced at points of the obstacle velocity change (corners of $\mathcal{ST}_{obs}$). A similar approach has been presented in [82], in which a modified visibility graph method (modified to obey the speed limit) is spanned over a configuration-time space. The graph is used to calculate multiple time intervals in which a goal configuration is reachable, such that the solution is valid for multiple start intervals. A common drawback of the aforementioned methods is that the complexity of cell decomposition or visibility graph construction makes these algorithms impractical, even for 2D motion planning.

Sampling-based Methods

As sampling-based methods are typically used for solving hard and high-dimensional motion planning problems, they were also used for planning in dynamic environments (e.g., approaches based on PRM [83, 84, 85] and approaches based on RRT [35, 86]). A method proposed in [83] is presented as PRM-based (i.e., it picks at random samples with some probability distribution). However, it constructs a tree by sampling a time milestone for which a random control is applied, which makes it similar to RRT. In [84], this approach has been extended to solve the boundary value problem by construction of a secondary tree, rooted at the goal state, which is expanded backward in time.

An approach that combines PRM and RRT algorithms was proposed in [85], in which, a PRM with lazy collision checking is initially constructed for a static environment. In the query phase, a static path is checked for collision with moving obstacles, and the roadmap is locally repaired with RRT if necessary. If a local repair fails, the roadmap construction algorithm continues. A similar method that uses a roadmap constructed for a static environment was proposed in [33, 87, 79]; however, time is discretized, such that the state-time search-graph can be viewed as a set of regular grids spanned along roadmap edges.

In [88], a space-time exploration guided heuristic search (STEHS) for autonomous cars has been proposed. The STEHS utilizes a two-stage hierarchical heuristic search. The first stage is a heuristically guided exploration that is responsible for construction of a collision-free corridor consisting of overlapping cylinders in a state-time space. The second stage is an RRT-like kinodynamic motion planning algorithm that constructs a collision-free trajectory consisting of dynamically feasible motion primitives within the corridor expanded at the first stage.

Local Planning

The solutions that purely rely on RRT are described in two papers [35, 86]. What is important is that, in both cases, kinodynamic planning is performed locally. A safe motion planning in a dynamic environment was thoroughly investigated in [35], with the conclusion that planning needs to be performed in real time. As no global planning methods can ensure real-time performance, it is necessary to provide such a local planner that returns plans that do not lead to a collision in a certain time horizon. A path provided by a local planner cannot contain inevitable-collision states [35].

In [86], the problem of inevitable collision states has been resolved using the modified RRT algorithm that uses a potential field computed around the trajectory of each moving object. When the time for RRT is over, a local path is reconstructed starting from the node with the least cost.

Within a finite time horizon it is also possible to apply a numerical optimization method that provides smooth trajectory with a minimized collision cost [89]. This method works in a continuous state space, but with discretized time.

In addition to real-time requirements, another argument for local planning in a dynamic environment is that, in most situations, it is impossible to accurately predict trajectories of observed obstacles to longer time horizons.

Grid-based Search and State-lattice Search

Time-dependent planning is also possible with regular grids [21, 90] and state lattices [91, 92, 93]. The natural idea is to use both space and time discretization. If a robot has limited speed (and acceleration), a space resolution depends on these limits and a given time step [21]. Due to the large size of discretized state-time space, methods of this kind are typically used for a local search [21, 90] or as part of a global search graph that performs time-dependent planning only within a fixed time horizon around a start state [91, 92].

Global time-dependent motion planning on a regular grid has been achieved using the safe interval path planning (SIPP) method [22, 80]. In SIPP, to avoid a state-time space explosion,

a time is decomposed using a combinatorial decomposition (i.e., each grid cell is multiplied by the number of disjoint time intervals in which this cell is free). A similar method was proposed by the author in [23], in which obstacle layers were introduced, such that each i-th obstacle layer represents the time at which the i-th obstacle leaves a grid cell. In this thesis, safe intervals and obstacle layers are generalized using an event-based state-time space decomposition and are applied to a state-lattice search.

Approximate Methods

Finally, methods that solve simplified or modified problems of motion planning in the presence of moving obstacles can be found. For example, in [94], a hierarchical approach was proposed that divides the path-planning problem into path planning in a static environment and velocity planning, which is applied along a static path. This approach falls into a group of methods referred to as velocity-tuning methods [25, Ch. 7.1].

Other methods avoid motion planning in a dynamic environment by reducing the problem to reactive collision avoidance. Such an approach has been proposed in [95], where the velocity obstacle (VO) concept has been introduced. In this concept, assuming that the obstacle motion model is known, a region of possible collision is represented in the velocity space of a robot, such that the only decision to be made is to select the velocity that avoids such a region. In [96] the concept of reciprocal VO has been proposed, which, in addition to VO, assumes that moving objects are agents utilizing a similar collision-avoidance policy as the robot.

6.3 Event-based State-time Space Decomposition

Dynamic changes in a state space can be modeled by events from the set $\mathcal{E} = \mathcal{E}_{obs} \cup \mathcal{E}_{free}$ (Fig. 6.5), where:

- $\mathcal{E}_{obs} = \{e = (s,t) | (s,t) \in \mathcal{ST}_{free} \wedge (s,t+1) \in \mathcal{ST}_{obs}\}$: a set of events denoting that a state s became blocked by an obstacle starting from $t+1$ upwards,
- $\mathcal{E}_{free} = \{e = (s,t) | (s,t-1) \in \mathcal{ST}_{obs} \wedge (s,t) \in \mathcal{ST}_{free}\}$: a set of events denoting that a state s became free, starting from t upwards.

For an event, function $at(e) : \mathcal{E} \rightarrow T$ can be defined that returns a time at which the event occurs. However, it will be convenient to omit at when events are used with relation and interval operators (e.g., $e_1 < e_2$ instead of $at(e_1) < at(e_2)$ and $[e_1, e_2]$ instead of $[at(e_1), at(e_2)]$).

As it moves, a convex obstacle leaves a trace in the workspace $\mathcal{W}$. For each point p on the trace, two events can be observed: first, when the obstacle enters point p and, second, when the obstacle leaves point p. The obstacle trace in $\mathcal{W}$ corresponds to a trace in $\mathcal{ST}_{obs}$, such that the

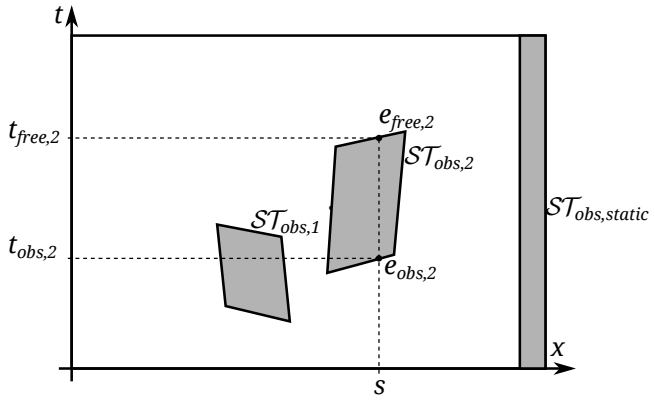


Figure 6.5: Sample Events in a State-time Space.

trace in $\mathcal{ST}_{obs}$ is bounded along the time dimension by the two aforementioned events (e.g., $(e_{obs,2}, e_{free,2})$ in Fig. 6.5); hence, the following natural observations can be made (Lemma 6.1 and Corollary 6.2).

Lemma 6.1 *For each state $s \in S$, the motion of a moving obstacle induces s in S, at least one pair of events $e_1 \in \mathcal{E}_{obs}$ and $e_2 \in \mathcal{E}_{free}$, such that $e_2 \geq e_1$; hence, (e_1, e_2) is a time interval in which a state s is occupied by the moving obstacle.*

Corollary 6.2 *For a state $s \in S$, there is an event $e_1 \in \mathcal{E}_{obs}$, if and only if it is followed by an event $e_2 \in \mathcal{E}_{free}$.*

With the event-based notation provided and Corollary 6.2, a safe interval proposed in [22] can be defined (Definition 6.1).

Definition 6.1 *Let s be a state in S. A safe interval of s is an interval $[t_1, t_2]$, such that $t_1 \in \{0\} \cup \{t|(s,t) \in \mathcal{E}_{free}\}$ and $t_2 \in \{t|(s,t) \in \mathcal{E}_{obs}\} \cup \{+\infty\}$, for which $\nexists e = (s,t) \in \mathcal{E}, e_1 < e < e_2$.*

The sample safe intervals bounded by events are shown in Figure 6.6a. With respect to this definition, safe intervals correspond to disjoint subsets of $\mathcal{ST}_{free}$; hence, they split $\mathcal{ST}_{free}$ into partitions. Consequently, such a partitioning can be used as a mapping to an abstract graph. This approach is similar to a cylindrical decomposition (Fig. 6.6b) described in [25, Ch. 7.1]; however, herein cylinders are formed over states, not time.

With the definition of a safe interval, Proposition 6.3 can be given (the proposition was originally proposed in [22] in a slightly different form and was discussed in [23], yet not formalized):

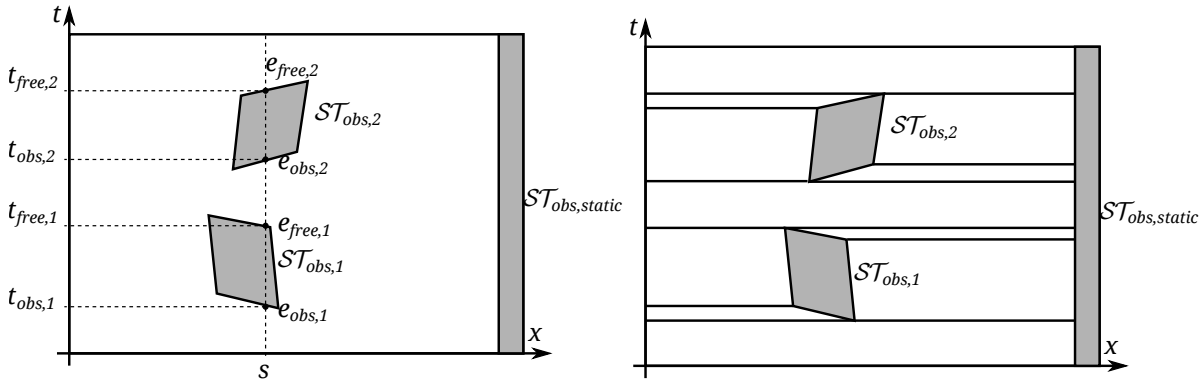


Figure 6.6: State-time space decomposition methods; sample events and time intervals, $[0, t_{obs,1}], [t_{free,1}, t_{obs,2}], [t_{free,2}, +\infty]$, (a) in a state-time space and (b) cylindrical decomposition [25, Ch. 7.1].

Proposition 6.3 *For n convex moving obstacles such that each is moving along P_k path in a workspace $\mathcal{W}$, such that $\forall i \neq j, P_k[i] \neq P_k[j]$ (the moving obstacle visits each point along the path at most once), at least one and at most $n+1$ safe intervals for each state $s \in S$ exist.*

A sketch of the proof of Proposition 6.3 is given in [22]. The example supporting this proposition is shown in Figure 6.6a.

From Proposition 6.3, it stands that, for n convex moving obstacles that do not visit any point in a workspace more than once, a state-time space can be decomposed into $n+1$ qualitative layers that include all safe intervals in $\mathcal{ST}$. This observation has been used in the author's previous work [23], in which obstacle layers have been proposed. In this approach, path planning amid moving obstacles is held in a $\mathcal{SI} = S \times \mathbb{Z}$ space (that is similar to a mode-space concept [25, Ch. 7.3]) for which $\phi_{\mathcal{SI}}(\tilde{s}) : \mathcal{ST} \rightarrow \mathcal{SI}$ mapping is defined as follows:

$$\phi_{\mathcal{SI}}(\tilde{s} = (s,t)) = \begin{cases} (s, \operatorname{argmin}_i(t - t_i)) & \text{if } \exists e = (s, t_i) \in \mathcal{E}_{free}, t_i \leq t, \\ (s, 0) & \text{otherwise,} \end{cases} \tag{6.2}$$

where $i \in \{1, 2, \ldots\}$ is a moving obstacle number. Obstacle layers, as defined by Eq. 6.2, are qualitative, such that the 0 layer that begins at time 0 represents all situations before the presence of any obstacle at any place, and the i-th layer that begins at time t_free, i, represents all situations immediately after leaving some place by an i-th obstacle.

Both approaches, safe intervals [22] and obstacle layers [23], abstract an original state-time space to a *safe interval graph* G, such that nodes of G correspond to safe intervals in $\mathcal{ST}$. Therefore, each node $v \in V(G)$ has labels, $interval^-(v)$ and $interval^+(v)$, that are the begin and end times of the safe interval, respectively. The number of nodes in G is $|G| = |S| \cdot (n+1)$,

where n is the number of convex moving obstacles, and S is a discretized state space. Two nodes in G are adjacent only if their safe intervals overlap.

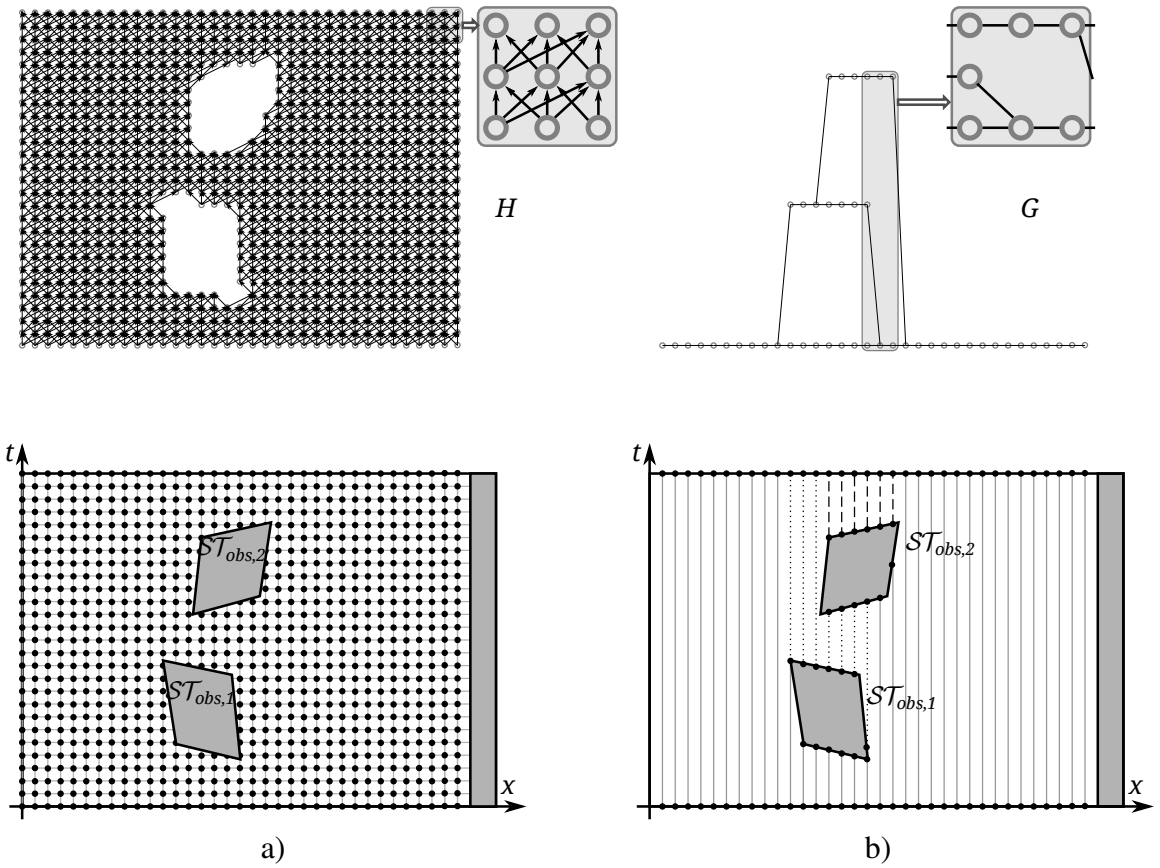


Figure 6.7: A state-time space (bottom) abstracted into (top): a) an acyclic directed graph H representing points picked from a regular grid embedded in $\mathcal{ST}$, b) an undirected graph G representing safe intervals; solid, dotted, and dashed lines depict safe intervals of 0th, 1st, and 2nd obstacle layers, respectively.

A sample safe interval graph G is shown in Figure 6.7b, next to an alternative approach in which an abstract graph H consists of points picked from a regular grid embedded in $\mathcal{ST}$ (Fig. 6.7a), such that the number of nodes in H is $|H| = |S| \cdot |T|$. As H is an acyclic directed graph, a linear-time shortest-path search algorithm exists. However, the complexity of such an approach is $O(|S| \cdot |T|)$, which usually will be higher than $O(|G| \cdot log|G|) \approx O(|S| \cdot log|S|)$, which is necessary for the shortest-path search in a safe interval graph G.

6.4 Principles of a Minimum-time Path Search

A motion planning problem in a state-time space is a pair $P = (\tilde{s}_{start}, s_{goal})$ such that:

- $\tilde{s}_{start} \in \mathcal{ST}$: a start state in a state-time space,

- $s_{goal} \in S$: a goal state is defined in a state space, not in a state-time space, as the arrival time is unknown until planning is finished.

A solution for a problem $P = (\tilde{s}_{start}, s_{goal})$ is as follows:

- $\tilde{\Gamma} = \langle \tilde{a}_1, \ldots, \tilde{a}_n \rangle$ is a collision-free plan such that:
  - $\gamma^{-1}(\tilde{a}_1) = \tilde{s}_{start}$,
  - $\forall i < n, \gamma(\tilde{a}_i) = \gamma^{-1}(\tilde{a}_{i+1})$,
  - $\gamma(\tilde{a}_n) = (s_{goal}, t_{goal})$,
- $\tilde{\Pi} = \langle \tilde{s}_0, \ldots, \tilde{s}_n \rangle$ is a collision-free discrete trajectory such that $\forall 1 \leq i \leq n, \langle \tilde{s}_{i-1}, \tilde{s}_i \rangle = a_i \in \tilde{\Gamma}$.

As this thesis is focused on graph-based motion planning methods, a collision-free discrete trajectory $\tilde{\Pi}$ is desired. In graph theory, structures with time-dependent edge costs are referred to as *time-dependent networks*. Dean [97] and Foschini et al. [98] provided the complexity analysis for the shortest-path planning in time-dependent networks. In both cases, FIFO networks are considered, in which waiting an arbitrarily long time is allowed at any node. In the case of motion planning in the presence of moving obstacles, waiting at a point along the path of a moving object will lead to a collision; therefore, such an assumption cannot be made.

Although FIFO networks do not apply to time-dependent planning as defined in this chapter, graph representations can still be used. For example, the acyclic directed graph, as shown in Figure 6.7a, is known as a time-expanded network [97] or time-layered graph [98]. In addition, a safe interval graph (Fig. 6.7b) is a form of time-expanded graph, though it is a qualitative expansion.

Optimal planning for a search is possible if Bellman's *principle of optimality* is fulfilled, that is, a solution plan is optimal if any partial solution is optimal. After Edelkamp and Shrödl [54, Ch. 17.2], this property holds for time-dependent planning if a graph is time consistent.

Definition 6.2 *The graph is time consistent if, for all time points $t_1 \leq t_2$ and every pair of actions $\tilde{a}_1 = \langle (s_1, t_1), (s_2, t_1') \rangle$ and $\tilde{a}_2 = \langle (s_1, t_2), (s_2, t_2') \rangle$, the following relation holds:*

$$t_1 + duration(\tilde{a}_1) + t^*(\tilde{s}_1') \leq t_2 + duration(\tilde{a}_2) + t^*(\tilde{s}_2'), \tag{6.3}$$

where $t^*(s) : \mathcal{ST} \rightarrow T$ returns the shortest time to goal.

In short, it must hold that:

$$t_1 + duration(\tilde{a}_1) \leq t_2 + duration(\tilde{a}_2). \tag{6.4}$$

In other words, a time-consistency property says that it is impossible to achieve the goal earlier by starting later. The example of broken time consistency is shown in Figure 6.8.

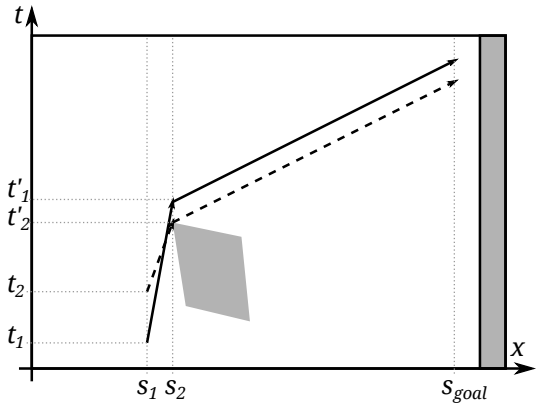


Figure 6.8: Example of broken time consistency for actions $\tilde{a}_1 = \langle (s_1, t_1), (s_2, t'_1) \rangle$ and $\tilde{a}_2 = \langle (s_1, t_2), (s_2, t'_2) \rangle$, $t_1 < t_2$, but $t'_1 > t'_2$.

Figures 6.9a and 6.9b depict states corresponding to nodes visited by the minimum-time trajectory search in a state-time graph (obtained from a regular grid; Fig. 6.7a) and a safe interval graph, respectively.

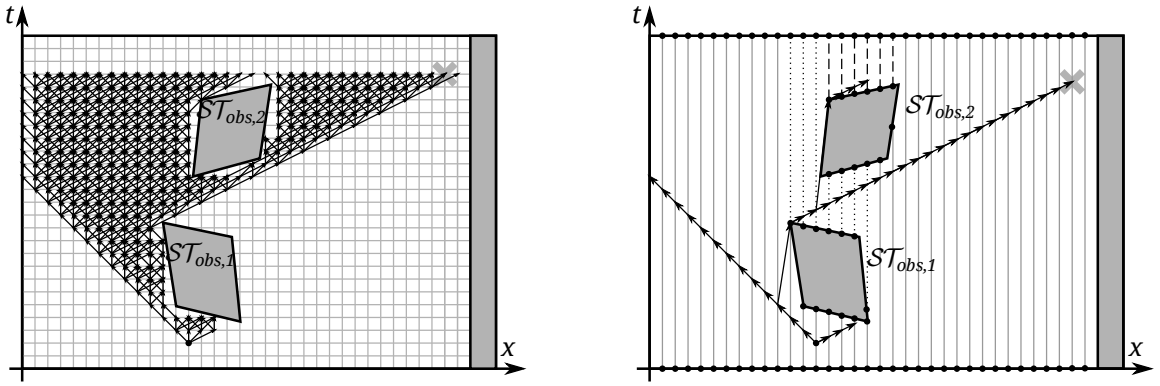


Figure 6.9: The forward breadth-first search: a) in a state-time graph, b) in a safe interval graph; solid, dotted, and dashed lines depict safe intervals of 0th, 1st, and 2nd obstacle layers, respectively.

An approach, as proposed in [21], that uses a state-time graph is straightforward. The only difficulty is to choose a state-space resolution, such that, for a given time step and the maximum acceleration, each action $\tilde{a} \in \mathcal{AT}$ leads to a state that has a corresponding node in a state-time graph H (i.e., $\gamma(\tilde{a}) \in V(H)$). Although in Figure 6.9a the outcome of a BFS is shown, in [21], A* has been used to reduce the number of visited nodes.

6.5 Minimum-time Path Search in a Safe Interval Graph

In this section, the minimum-time path search in a safe interval graph [22, 23, 24] is discussed from the new perspective of an event-based state-time space decomposition.

A safe interval graph search can be performed using A* [22, 24], any other forward-search algorithm described in Section 3.2, or dynamic programming described in the algorithm presented by the author in [23]. However, as a safe interval corresponds to a set of states, the problem of state-to-node and action-to-edge mapping arises. To deal with this issue, each safe interval graph node $v \in V(G)$ needs to correspond to a single state $\tilde{s} = (s, t) \in \mathcal{ST}$ (it is labeled with a state $\tilde{s}$), such that t is the earliest possible time at which a safe interval can be reached. Graph-node labeling with states is valid only for a particular problem $P = (\tilde{s}_{start}, s_{goal})$ and can be obtained by searching that is conducted from a start node labeled with a start state $\tilde{s}_{start}$. Initially, safe interval graph nodes can be labeled only with safe interval bounds $[t_1, t_2]$. Finally, the following notation will be used to describe a safe interval graph search:

- $\phi_G(\tilde{s}) : \mathcal{ST}_{free} \rightarrow V(G)$ is a state-to-node mapping, such that $v = \phi_G(\tilde{s})$ is a graph node; however, in most cases, it will be omitted,
- $g(\phi_G(\tilde{s}))$, $h(\phi_G(\tilde{s}))$, and $f(\phi_G(\tilde{s}))$ (in short, $g(\tilde{s})$, $h(\tilde{s})$, and $f(\tilde{s})$,) are typical labels used by A* denoting the cost from the start, the estimated cost to the goal, and their sum, respectively,
- $interval^-(\phi_G(\tilde{s})) : V(G) \rightarrow T$ (in short, $interval^-(\tilde{s})$) returns a safe interval lower bound,
- $interval^+(\phi_G(\tilde{s})) : V(G) \rightarrow T$ (in short, $interval^+(\tilde{s})$) returns a safe interval upper bound,
- $\tilde{s}_{min}(\phi_G(\tilde{s})) : V(G) \rightarrow \mathcal{ST}_{free}$ (in short, $\tilde{s}_{min}(\tilde{s})$), the earliest reachable state within $[interval^-(\tilde{s}), interval^+(\tilde{s})]$ time interval.

A forward-search algorithm (Alg. 3.1), presented in Section 3.2 that can be used for a safe interval graph search, requires minor modification in a successor expansion loop (i.e., instruction $\tilde{s}_{min}(\tilde{s}') = \tilde{s}'$ needs to be added between lines 17 and 18 of the algorithm (Alg. 3.1)).

To label a graph node with the earliest reachable state, only the quickest possible actions need to be used. Let us define $\tilde{a}^*(\tilde{s}_1, s_2) : \mathcal{ST} \times S \rightarrow \mathcal{AT}$, a function that returns the minimum-time action that leads from $\tilde{s_1}$ to $(s_2, t_2) \in \mathcal{ST}$ with the minimum time t_2 as:

$$\tilde{a}^*(\tilde{s}_1, s_2) = \underset{\tilde{a}=\langle \tilde{s}_1, (s_2, t_2) \rangle \in \mathcal{AT}}{\operatorname{argmin}} (t_2). \tag{6.5}$$

Clearly, the minimum-time action must obey the velocity limits and acceleration limits (if such are considered) at any time of its duration.

Referring to the car example, only motions at maximum speeds, v_{min} and v_{max}, are of interest; however, these are not always applicable due to the presence of moving obstacles. In such a case, the only possible solution is to delay a transition to a neighboring state (a graph node) until it is free. This can be viewed as a robot motion synchronization with a moving obstacle, in particular, with an event $e \in \mathcal{E}_{free}$. At this point, it will be useful to define an action-event synchronization function.

Definition 6.3 *Event-action synchronization function,* $sync(\tilde{s}_1, e) : \mathcal{ST} \times \mathcal{E}_{free} \rightarrow \mathcal{AT}$*:*

$$sync(\tilde{s}_1, e) = \underset{\tilde{a}=\langle \tilde{s}_1, \tilde{s}_2 \rangle \in \mathcal{AT} \wedge t_2 \geq t_e}{\operatorname{argmin}} (t_2), \tag{6.6}$$

where $\tilde{s}_1 = (s_1, t_1), \tilde{s}_2 = (s_2, t_2) \in \mathcal{ST}$*,* $e = (s_2, t_e) \in \mathcal{E}_{free}$*, such that:*

- $t_1 < t_e \leq t_2$*,*
- $t_1, t_2 \in [interval^-(\tilde{s}_1), interval^+(\tilde{s}_1)]$*,*
- $t_2 \in [interval^-(\tilde{s}_2), interval^+(\tilde{s}_2)]$ *(i.e., that safe intervals of two adjacent nodes must overlap),*
- $duration(sync(\tilde{s}_1, e)) \geq duration(\tilde{a}^*(\tilde{s}_1, s_2))$ *(i.e., an event-synchronized action cannot be faster than the fastest action applicable at a state* $\tilde{s}_1$*),*
- $\forall_{t \in [t_1, t_2]} \dot{\tilde{\tau}}(sync(\tilde{s}_1, e), t) \in [v_{min}, v_{max}]$ *(i.e., an event-synchronized action must obey velocity limits),*
- $\forall_{t \in [t_1, t_2]} \ddot{\tilde{\tau}}(sync(\tilde{s}_1, e), t) \in [a_{min}, a_{max}]$ *(i.e., an event-synchronized action must obey acceleration limits, if such are considered).*

With the definitions given above, an action cost (Eq. 6.7) and a successors set (Eq. 6.8) that are needed for the shortest-path search can be defined as follows:

$$cost(\tilde{a}) = duration(\tilde{a}), \tag{6.7}$$

$$\begin{aligned} Succ(\tilde{s}) = \{\tilde{s}' = (s', t') \in \mathcal{ST}_{free} | \langle \tilde{s}, \tilde{s}' \rangle = \tilde{a}^*(\tilde{s}, s') \vee \langle \tilde{s}, \tilde{s}' \rangle = sync(\tilde{s}, e), \\ e = (s', t_e) \in \mathcal{E}_{free} \wedge t_e \geq end(\tilde{a}^*(\tilde{s}, s'))\}. \end{aligned} \tag{6.8}$$

From Eq. 6.8, we show that successors of a given state are states reached by minimum-time actions or actions obtained by synchronization with future events.

Now, it must be determined whether actions that are used to generate a successor set (Eq. 6.8) create a time-consistent safe interval graph. With respect to Eq. 6.4, it must be shown that there is no action that begins later but ends earlier than any other action.

Proposition 6.4 *Let G be a directed graph consisting of nodes corresponding to safe intervals. If the arcs of G are chosen among the minimum-time or event-synchronized actions, the G is time consistent.*

Proof 6.1 *Let us consider two actions $\tilde{a}_1 = \langle (s_1, t_1), (s_2, t'_1) \rangle$ and $\tilde{a}_2 = \langle (s_1, t_2), (s_2, t'_2) \rangle$, such that both begin at the same state s_1 at two different time points, $t_1 < t_2$, and both actions lead to the same state s_2 at time points t'_1 and t'_2.*

Case 1. *If both actions are the minimum-time actions, then their durations are equal, $duration(\tilde{a}_1) = duration(\tilde{a}_2)$; hence, as $t_1 < t_2$, it must be that $t'_1 \leq t'_2$.*

Case 2. *If both actions, $\tilde{a}_1 = sync((s_1, t_1), e)$ and $\tilde{a}_2 = sync((s_1, t_2), e)$, are synchronized with a future event e, then, by definition (Eq. 6.6), $t_1 = t_2 = \text{argmin}_{\tilde{a}=\langle (s_1,\cdot),(s_2,t') \rangle \in \mathcal{AT} \wedge t' \geq t_e}(t')$.*

Case 3. *Let $\tilde{a}_1 = sync((s_1, t_1), e)$ be the action synchronized with a future event e and $\tilde{a}_2 = \tilde{a}^*((s_1, t_2), s_2)$ be the minimum-time action. By definition (Eq. 6.6), $e \in \mathcal{E}_{free}$, and thus from Corollary 6.2, there must be an event $e' = (s_2, t'_e) \in \mathcal{E}_{obs}$ such that $\{(s_2, t') | t' \in (t'_e, t_e)\} \subseteq \mathcal{ST}_{obs}$. Hence:*

3.1. if $t'_2 \geq t'_1$, the time consistency is preserved,

3.2. if $t'_e < t'_2 < t_e$, it means that action $\tilde{s}_2$ leads to a collision,

3.3. if $t'_2 \leq t'_e$, then action $\tilde{s}_2$ leads to a different safe interval than action $\tilde{a}_1$ (i.e., $\phi_G(\gamma(\tilde{a}_1)) \neq \phi_G(\gamma(\tilde{a}_2))$). If so, there must be another action $\tilde{a}_3 = \langle (s_1, t_1), (s_2, t_3) \rangle$, such that $\phi_G(\gamma(\tilde{a}_2)) = \phi_G(\gamma(\tilde{a}_3))$, for which Case 1 or Case 3.1 applies.

□

6.6 Action-event Synchronization for Mobile Robot Motion Planning

A calculation of an event-synchronized action (Eq. 6.6) is a local planning problem of itself. However, with additional assumptions, it can be simplified, making global safe interval planning tractable. It is a common technique to pre-compute motion primitives (e.g., arcs of a state lattice [50, 18]) considering kinematic constraints. Thus, in this dissertation, it is proposed to reuse such motion primitives and synchronize them with events. Throughout this section, two types of actions (motion primitives) will be distinguished:

- non-temporal action (non-temporal motion primitive), a, that is, a continuous path in the configuration space represented by $\tau(a, p)$, with a distance function returning its length,

namely:

$$distance(a) = \int_0^1 \tau(a,p)\,\mathrm{d}p \tag{6.9}$$

- temporal action (temporal motion primitive), $\tilde{a}$, that is, a continuous trajectory in the state-time space represented by $\tilde{\tau}(\tilde{a},p)$ being an outcome of a non-temporal action synchronization.

In the remainder of this section, a few forms of an action-event synchronization that are useful for mobile robot motion planning are discussed.

6.6.1 Simple Action-event Synchronization

We assume that a function $a^*(s_1,s_2) : S \times S \to A$ returns a minimum-time action in a static environment, which is defined as follows:

$$a^*(s_1,s_2) = \underset{a=\langle s_1,s_2\rangle \in A}{\operatorname{argmin}} (duration(a)). \tag{6.10}$$

An event-synchronized action can be viewed as a protraction of the minimum-time action, such that the event-synchronized action lasts longer than the minimum-time action, $duration(sync((s_1,t_1),(s_2,t_e))) \geq duration(a^*(s_1,s_2))$, without changing a motion primitive (i.e., the path in S is unchanged).

In the simplest case, the event-synchronization function can be defined as:

$$\begin{aligned} sync(\tilde{s}_1,e) = \langle \tilde{s}_1,e\rangle \quad & |t_e \geq t_1 + duration(a^*(s_1,s_2)) \wedge \\ & distance(sync(\tilde{s}_1,e)) = distance(a^*(s_1,s_2)), \end{aligned} \tag{6.11}$$

where $\tilde{s}_1 = (s_1,t_1) \in \mathcal{ST}$ and $e = (s_2,t_e) \in \mathcal{E}_{free}$ (Fig. 6.10). From Eq. 6.11 it follows that:

$$\begin{aligned} t_e - t_1 &\geq duration(a^*(s_1,s_2)), \text{ hence} \\ \frac{distance(sync(\tilde{s}_1,e))}{t_e - t_1} &\leq \frac{distance(a^*(s_1,s_2))}{duration(a^*(s_1,s_2))} \\ \frac{distance(sync(\tilde{s}_1,e))}{t_e - t_1} &\leq v_{max}. \end{aligned} \tag{6.12}$$

Therefore, an action returned by Eq. 6.11 preserves velocity limits.

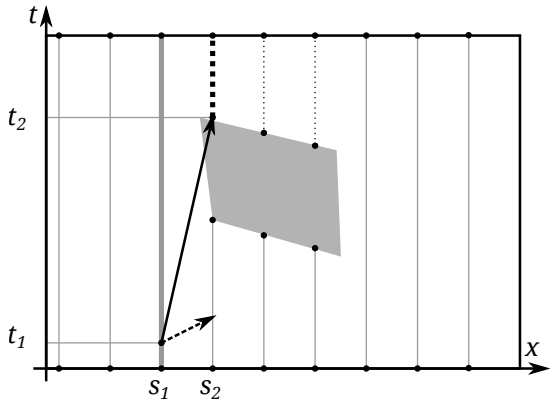


Figure 6.10: Example of an action-event synchronization, where a dashed arc denotes minimum-time action, a solid arc denotes synchronized action, and thick vertical lines (both solid and dotted) denote safe intervals considered in the synchronization.

6.6.2 Simple Action-event Synchronization for Long Actions

Typically, a state lattice includes motion primitives that are longer than a state-lattice resolution. A simple synchronization applied to a long motion primitive may produce an action, $\tilde{a} = \langle \tilde{s}_1, \tilde{s}_2 \rangle$, that leads to a collision, as shown in Figure 6.11a, even though, from Definition 6.3, the following holds: $t_1, t_2 \in [interval^-(\tilde{s}_1), interval^+(\tilde{s}_1)] \wedge t_2 \in [interval^-(\tilde{s}_4), interval^+(\tilde{s}_4)]$. Furthermore, in some cases (for example, Fig. 6.11b), the overlapping condition of the safe intervals is too restrictive; therefore, it needs to be generalized.

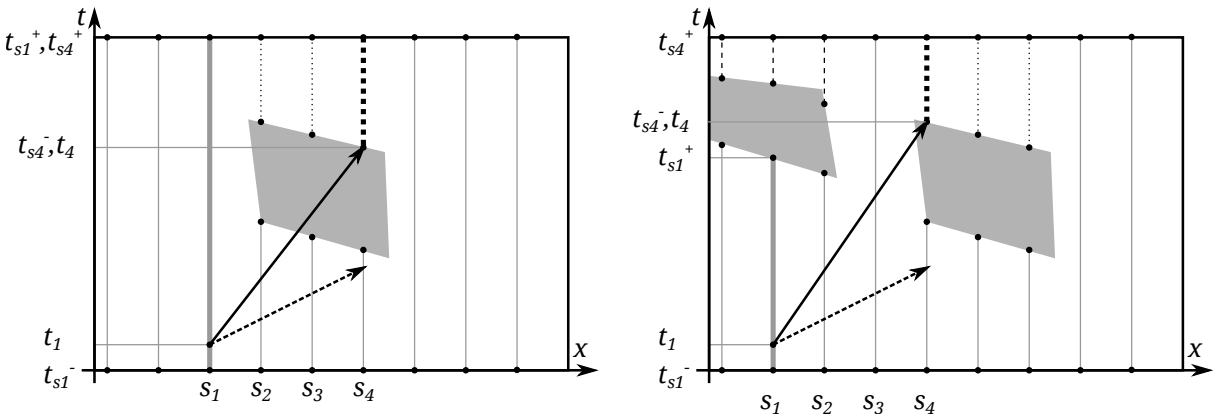


Figure 6.11: Example of a failed action-event synchronization: a) although safe intervals of $\tilde{s}_1$ and $\tilde{s}_4$ states overlap, the synchronization leads to the collision in the mid-states; b) although safe intervals of $\tilde{s}_1$ and $\tilde{s}_4$ states do not overlap, the synchronization is collision-free. Dashed arcs denote minimum-time actions, solid arcs denote synchronized actions, and thick vertical lines denote safe intervals considered in the synchronization.

As a motion primitive runs nearby state-lattice nodes (states), it can be represented as a sequence of states $P_{\tilde{a}} = \langle \tilde{s}_1, \ldots, \tilde{s}_k \rangle$, which next can be mapped to a path in a safe interval

graph. It can be proposed that a long motion primitive, $\tilde{a}$, is a collision-free path in a safe interval graph if the following holds:

$$\begin{aligned} \forall_{\tilde{s}_i, \tilde{s}_{i+1} \in P_{\tilde{a}}} \quad & t_i, t_{i+1} \in [interval^-(\tilde{s}_i), interval^+(\tilde{s}_i)] \\ \wedge \quad & t_{i+1} \in [interval^-(\tilde{s}_{i+1}), interval^+(\tilde{s}_{i+1})]. \end{aligned} \tag{6.13}$$

Although with the condition in Eq. 6.13 the synchronization examples shown in Figures 6.11a and 6.11b are resolved properly, the condition is still restrictive. For example, in the situation shown in Figure 6.12, it will not allow for synchronization between states $\tilde{s}_1$ and $\tilde{s}_3$. However, the search for a solution of more elaborate problems can be part of a global search. The exemplified problem is possible to solve, if a state lattice includes motion primitives of a lattice resolution length.

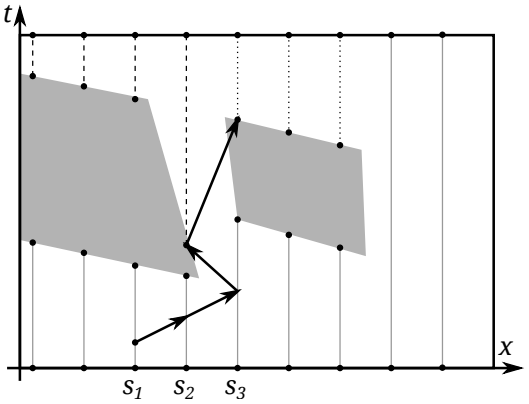


Figure 6.12: Sample situation in which a simple action-event synchronization cannot be applied. Synchronization between $\tilde{s}_1$ and $\tilde{s}_3$ is impossible with respect to Eq. 6.13; however, a solution can be found if a state lattice includes shorter motion primitives.

6.6.3 Action-event Synchronization with Acceleration Limits

To check whether an action returned by Eq. 6.11 preserves acceleration limits, we consider the worst case in which a robot moving at the maximum speed must stop to avoid a collision with a moving obstacle. If this is part of the minimum-time path, once the collision is avoided, the robot should achieve the maximum speed as soon as possible, that is, at time t_e. If a control input has a bang-coast-bang profile (i.e., it is a sequence of input controls $\langle a_{min}, 0, a_{max} \rangle$)[21], the duration of this action can be calculated as follows:

$$duration(sync(\tilde{s}_1, e)) = \frac{v_{max}}{a_{min}} + \frac{v_{max}}{a_{max}} + t_{wait}, \tag{6.14}$$

where t_{wait} is a time of in-place waiting, which is necessary to synchronize with t_e. Then, the traveled distance is calculated as follows:

$$distance(sync(\tilde{s}_1, e)) = \frac{v_{max}^2}{2a_{min}} + \frac{v_{max}^2}{2a_{max}}. \tag{6.15}$$

If $distance(sync(\tilde{s}_1, e)) < \frac{v_{max}^2}{2a_{min}}$, the robot will not have enough distance to stop; hence, $\tilde{s}_1$ is in the inevitable collision region. If $\frac{v_{max}^2}{2a_{min}} \leq distance(sync(\tilde{s}_1, e)) < \frac{v_{max}^2}{2a_{min}} + \frac{v_{max}^2}{2a_{max}}$, the robot will not achieve the maximum speed at time t_e; hence, such an action cannot be part of the minimum-time path.

In the general case, it is possible to preserve acceleration limits along an event-synchronized action if it is constructed from a motion primitive, $a^*(s_1, s_2)$, of a length greater than the distance required for stopping from and accelerating to the maximum speed, that is:

$$distance(a^*(s_1, s_2)) \geq \frac{V_{max}^2}{2dec_{max}} + \frac{V_{max}^2}{2acc_{max}}. \tag{6.16}$$

If a state lattice includes such motion primitives, it can be used for minimum-time motion planning in a safe interval graph.

The discussion on acceleration limits given in this section is under the assumption that safe interval graph nodes are decomposed only with respect to space and time intervals (i.e., it is assumed that at each node a robot has the maximum velocity). However, it is possible to construct a safe interval graph that is decomposed with respect to the space, time intervals, and velocity. Then, acceleration limits can be considered in a global search by avoiding connections between nodes that would require infeasible accelerations.

6.6.4 Action-event Synchronization with Cost-map Constraints

An action-event synchronization becomes complicated if speed constraints imposed by a cost map are considered. As proposed in Section 6.1, a cost map provides a function $cm(\tilde{s} = (s, t))$ that returns a value from 0 to 1. This value is used to reduce the maximum speed that is applicable at a given state, at a given time point (Eq. 6.1). If a cost map is defined for a static map, for a basic motion primitive $a^*(s_1, s_2)$ that is used for an event-synchronized action generation, the following holds:

$$\frac{distance(a^*(s_1, s_2))}{duration(a^*(s_1, s_2))} = cm(s_1) \cdot V_{max}, \tag{6.17}$$

hence (as in Eq. 6.11)

$$\begin{aligned} \frac{distance(sync(\tilde{s}_1, e))}{t_e - t_1} &\leq \frac{distance(a^*(s_1, s_2))}{duration(a^*(s_1, s_2))} \\ \frac{distance(sync(\tilde{s}_1, e))}{t_e - t_1} &\leq cm(s_1) \cdot V_{max}. \end{aligned} \tag{6.18}$$

Thus, an event-synchronized action preserves velocity constraints provided by a static cost map.

Difficulties arise when a cost map is defined in the neighborhood of a moving obstacle. The cm function is supposed to return a 0 value at the boundary of a moving obstacle region and to return values increasing proportionally to the distance from a moving obstacle at a given time point (i.e., $cm(\tilde{s} = (s,t)) \propto distance(s, \mathcal{ST}_{obs}(t))$). Then, for a point in $\mathcal{ST}$ that is at the boundary of an obstacle region, the maximum allowed speed is equal to zero. If such a point is an event with which the action ought to be synchronized, a simple synchronization function (Eq. 6.11) cannot be used, as the end time of such an action would be obviously greater than the event time, $t_2 = \infty > t_e$. Therefore, for the minimum-time planning with cost-map constraints, another synchronization function must be defined, one that returns an action with an end time t_2 that is greater than t_e but is also the earliest achievable. This can be obtained by a local search in a regular grid (with fixed time and space steps) spanned along a motion primitive that is used for synchronization (Fig. 6.13).

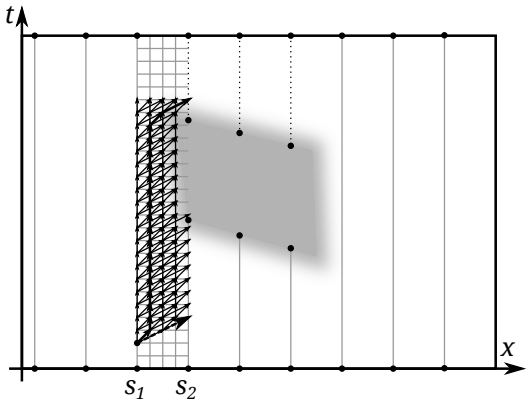


Figure 6.13: Example of an action-event synchronization as a local search, where a dashed arc denotes minimum-time action (to be synchronized), solid arcs denote local actions, and thick arcs denote the trajectory of a synchronized action.

6.7 Variants of a Time-dependent Heuristic Search

6.7.1 Real-time Planning

Real-time planning refers to problems in which planning is interleaved with acting and the time allocated for planning is limited. The problem of real-time searching and acting was originally formulated by Korf [14], who proposed the learning real-time A* (LRTA*). For a survey on real-time planning, also called the agent-centered search, refer to [99] and [54, Ch. 11].

In a real-time search, if the time provided for planning is over, which usually happens before a global solution is found, a search algorithm is stopped, and the best action based on the search tree explored to that point is performed by an agent. Typically, the best action is the one that minimizes the cost to the goal (Eq. 6.19).

$$\underset{a \in A | \gamma(a) \in Succ(s_{current})}{\operatorname{argmin}} \left(cost(a) + h(\gamma(a)) \right) \tag{6.19}$$

The main scheme for a real-time search is shown in Algorithm 6.1. It consists of four steps: 1) local search-space generation (performed as forward BFS, Dijkstra, or A*; e.g., Alg. 3.1), 2) heuristic values update (the learning step), 3) action selection (Eq. 6.19) and 4) action execution. Since the real-time search performs only a local search, there is a risk of the agent being trapped

Algorithm 6.1 Main procedure common for real-time algorithms.

```
1: function REAL-TIMESEARCH(s_start, s_goal, lookahead)
2:     while s_start ≠ s_goal do
3:         LSS ←SEARCH(s_start, s_goal, lookahead)        ▷ LSS: local search space
4:         H-VALUEUPDATE(LSS)
5:         a =ACTIONSELECTION(s_start)
6:         s_start = γ(a)
```

in a local minimum. In general, the learning step prevents that if and only if there are no states with an infinite cost to the goal. Roughly speaking, there are no traps. Such a search space is called 'safely explorable', for which the real-time algorithm is guaranteed to find the solution [54, Ch. 11].

Except for the third step, all steps may be time-consuming. A performance improvement can be achieved by providing a better heuristic (improving Step 1) or by speeding up the learning step (Step 2). In the original version of LRTA* [14], a heuristic value update is performed only for the current state of an agent, such that a new heuristic is calculated as follows (Eq. 6.20):

$$h(s) = \max \Big(h(s), \min_{a \in A | \gamma(a) \in Succ(s)} \big(cost(a) + h(\gamma(a)) \big) \Big). \tag{6.20}$$

In the LRTA* version presented in [54, Ch. 11], the heuristic value update (Alg. 6.2) is implemented as dynamic programming, where Eq. 6.20 is Bellman's equation. With such an implementation, if the value update step is conducted until convergence, it has $O(n^2)$ complexity. Otherwise, the calculated heuristic values remain admissible but converge more slowly. Regardless of the method, heuristics learned by LRTA* eventually converge to accurate values.

Algorithm 6.2 LRTA* heuristic value update.

1: **function** LRTA*-H-VALUEUPDATE(LSS)
2: **for** each $s \in LSS$ **do**
3: $temp(s) = h(s)$
4: $h(s) = \infty$
5: **while** $\{s \in LSS | h(s) = \infty\} \neq \emptyset$ **do**
6: $s_n = \text{argmin}_{s \in LSS} \max\left(temp(s), \min_{a \in A | \gamma(a) \in Succ(s)} \left(cost(a) + h(\gamma(a))\right)\right)$
7: $h(s_n) = \max\left(temp(s_n), \min_{a \in A | \gamma(a) \in Succ(s_n)} \left(cost(a) + h(\gamma(a))\right)\right)$

In [100], Koenig proposed to improve the h value update step using Dijkstra's algorithm initialized with nodes left in the open list at the end of the local search. This algorithm has been formalized as LSS-LRTA* in [70].

Another extreme simplification of the value-update step is used in the real-time adaptive A* (RTAA*) algorithm [101], in which a new heuristic evaluation is described by Eq. 6.21, where state s_{min} has the lowest f value among the states from an open list at the end of a local search.

$$h(s) = f(s_{min}) - g(s) \tag{6.21}$$

The RTAA* heuristic is well informed (near ideal) for all states lying along the shortest path from the current state to s_{min}. For all other states, the heuristic is still admissible, but is less informative than a heuristic calculated by LRTA*. Moreover, it usually underestimates the cost to the goal.

Although a learning real-time search seems to be ideal for use in robotics and video games, LRTA* and its variants have a significant drawback; an agent tends to revisit the same states of the search space multiple times, which is necessary to escape local h value depression. This effect, which is also called a "scrubbing behavior," has already been reported by Korf [14]. The recent findings strongly suggest that scrubbing is unavoidable [102]. However, it is possible to reduce this effect using better basic heuristics.

Real-time Planning in a Safe Interval Graph

Real-time search algorithms are designed to perform planning in a limited time; hence, they are well suited to mobile robot motion planning in a dynamic environment. In this section, how the algorithms of this class apply to time-dependent planning will be investigated, specifically, to a safe interval graph search.

There are two aspects of concern. First, a search space for real-time planning must not include states with an infinite cost (it must not include traps). In the case of time-dependent planning in the presence of moving obstacles, such states are well recognized. They form inevitable collision regions in a state-time space, which has been thoroughly discussed by [35] and explained in Section 6.1. Second, as real-time algorithms conduct only a local search, to assure completeness, they utilize heuristic learning. However, it is not clear whether such heuristics are admissible when used for time-dependent planning.

Typical heuristic learning rules (e.g., LRTA* or RTAA*) assume that the h value can only grow (Eq. 6.20). In the dynamic environment case, it means that once the h value is increased due to an event occurrence, it will not change even after the event has elapsed. For example, if the robot encounters a dynamic obstacle at some space (a graph node), it will remember increased heuristics permanently; thus, it will visit all nodes with lower h values, even if the actual heuristic cost to the goal is decreasing with time. Such a situation is shown in Figure 6.14. After an initial local search (Fig. 6.14a), that discovers an obstacle region, $\mathcal{ST}_{obs}$, h values are updated in accordance with Eq. 6.20 (Fig. 6.14b). If these h values are applied to states explored during a new search (after transition from s_1 to s_2), the f values will overestimate the total cost (Fig. 6.14c). Therefore, for time-dependent planning, a proper learning rule was proposed by the author in [34] (i.e., instead of h value updates, the lower bound on the f value, $f_{min}(s)$, should be updated):

$$f_{min}(s) = \max\Big(f_{min}(s), \min_{a \in A | \gamma(a) \in Succ(s)} \big(g(s) + cost(a) + f_{min}(\gamma(a)) - g(\gamma(a))\big)\Big). \quad (6.22)$$

With the learning rule defined by Eq. 6.22, f value calculation used in a local search is expressed as follows:

$$f(s) = \max\big(g(s) + h_{static}(s), f_{min}(s)\big), \quad (6.23)$$

where $h_{static}(s)$ is an ideal heuristic calculated by a backward search in a static environment. The example of the f value calculated using Eq. 6.23 is shown in Figure 6.14d.

Transforming Eq. 6.23 into form of Eq. 6.24, the heuristic is not constant and depends on $g(s)$, which indirectly entails time dependency. Such a relationship is characteristic for planning

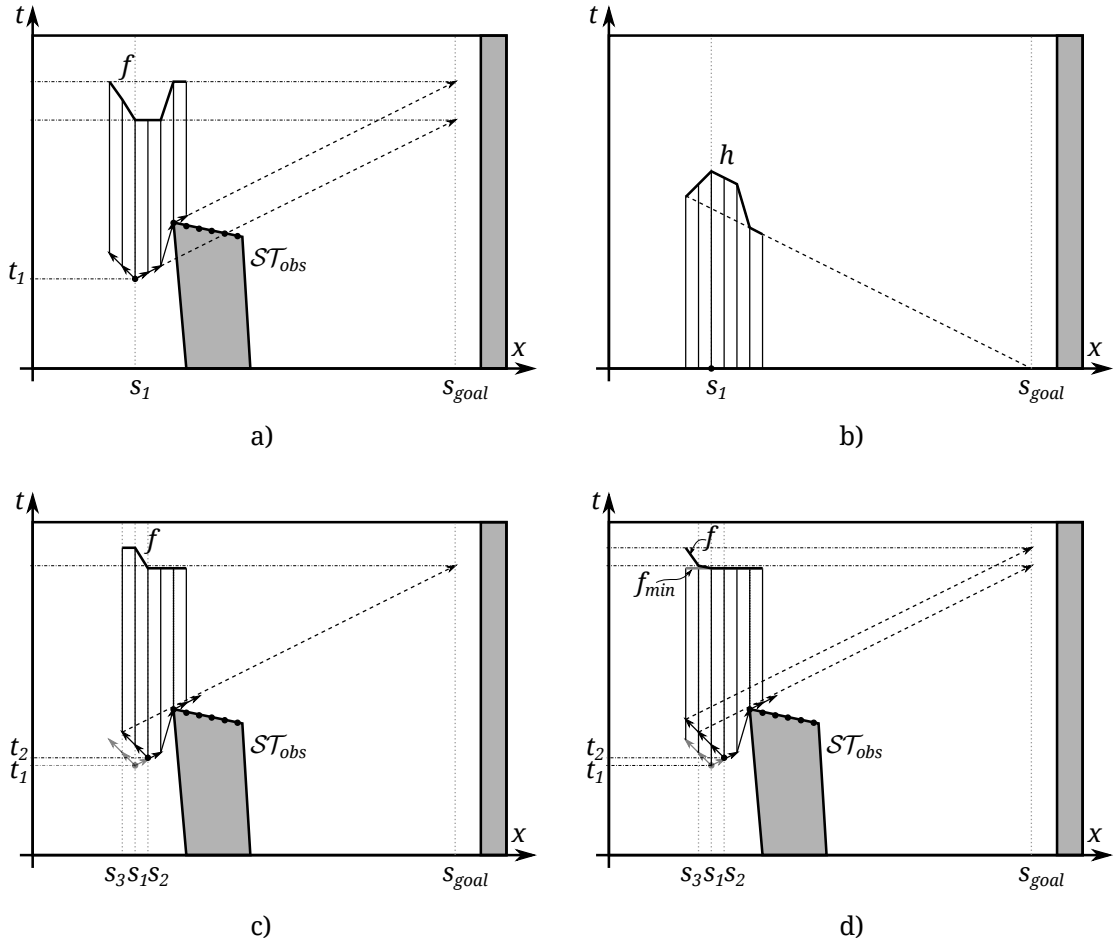


Figure 6.14: Heuristic learning in real-time planning in a safe interval graph: a) an initial search starting from (s_1, t_1) ($\mathcal{ST}_{obs}$ is unknown), b) a heuristic learned in accordance with Eq. 6.20 (dashed line denotes an initial heuristic), c) a consecutive search (after the transition from s_1 to s_2) with $f(s) = g(s) + h(s)$, ($f(s_1)$ and $f(s_3)$ are overestimated), d) a consecutive search (after the transition from s_1 to s_2) with $f(s) = \max\big(g(s) + h(s), f_{min}(s)\big)$. Thick solid arcs denote actions, thick solid lines denote f or h values, and dashed arcs denote the estimated shortest paths to the goal.

in a dynamic environment.

$$f(s) = g(s) + \max\big(h_{static}(s), f_{min}(s) - g(s)\big) \tag{6.24}$$

The difference between the f_{min}-value update and the h-value update is noticeable if the initial g value grows with subsequent planning iterations. Otherwise, if the g value is always initialized with the same value (in the example 0), then the two update rules are equivalent.

The proposed learning and heuristic calculation rules (Eq. 6.22 and Eq. 6.23) are correct under the strong assumption that the ideal static heuristic, $h_{static}(s)$, is provided. It is also possible to combine both learning rules (Eq. 6.22 and Eq. 6.20), as long as it is possible to

distinguish static and moving obstacles. Then, the f value should be calculated as follows:

$$f(s) = g(s) + \max\left(h_{basic}(s), h_{static}(s), f_{min}(s) - g(s)\right), \tag{6.25}$$

where h_{static} is learned with Eq. 6.20, f_{min} is learned with Eq. 6.22, and h_{basic} is a basic heuristic used for nodes that has not been explored before (for example, it can be a time to goal of movement along a straight line at the maximum speed).

6.7.2 Anytime Time-dependent Planning

The anytime version of planning in a safe interval graph has been discussed in [103]. The authors pointed out that using ARA* (Alg. 5.1) for a safe interval graph search is incomplete. This is shown in Figure 6.15. As ARA* does not reopen closed nodes, it may fail to find a

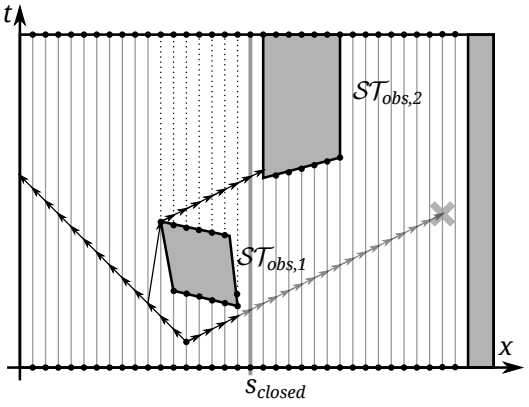


Figure 6.15: A sample problem in which ARA* may fail to find a solution. Due to the use of an inflated heuristic, a state s_{closed} has been expanded and closed with a high g value. Unfortunately, there is no further motion from that state. Light gray arcs depict a proper solution that cannot be found, as a state s_{closed} cannot be reopened.

solution in problems in which actions are feasible only if executed at an early stage (from a state with the lowest possible cost), for example, when batteries are still loaded enough or there is enough time to perform an action. However, such a discussion is limited to ARA*. Any other algorithm that allows for state reopening (e.g., ANA* or A* with an overestimated heuristic) is complete when used with a safe interval graph.

6.8 Experimental Results

In this section, the results of a minimum-time path search in a safe interval graph with a simple action-event synchronization are presented. Path planning was conducted using the A*

algorithm running forward in a fully-known environment (i.e., static obstacle positions) and moving obstacles trajectories were known.

Each single test scenario begins with planning on a map without moving obstacles, which will be called the 0-th run. The next path-search run for another robot is performed on the same map; however, the next robot must avoid collision with a robot moving along a trajectory planned in the preceding run. The same scheme was continued up to 50 runs, such that, in each i-th run, i-th robot avoids collision with all robots with trajectories planned in all preceding runs.

Planning in such a scenario was performed on 512x512 sized maps from *wc3*, *rooms*, and *mazes_16* map sets [72] (Fig. 4.5), such that at least 10 different maps from each map set were used. The robots had a radius of five map cells, which gives traces in a configuration space that are 10 map-cells wide. Each robot had to travel a path that was at least 400 map-cells long, starting at time 0. A search space was a 16-connected 2D grid with inflated travel costs in the vicinity of static obstacles. The dynamic obstacle regions in the C-space were not inflated; hence, the borders of such regions were sharp. A sample result of motion planning for 51 robots on a *battleground* map from the *wc3* map set is shown in Figure 6.16.

The average values of parameters logged during all scenario runs are presented in Table 6.1. The average planning time in a function of the number of moving obstacles is shown in Figure 6.17. The main observation that can be made in this particular benchmark is that planning time with the presence of one moving obstacle is about two times longer than the planning time without moving obstacles, despite the similar average number of search steps. The planning time is clearly doubled due to an additional layer of a search space. The plots presented in Figure 6.17 suggest that planning time is growing linearly with the number of moving obstacles in the environment.

6.9 Conclusions

In this chapter, an event-based state-time space decomposition for motion planning among moving obstacles has been proposed. With two types of events, when a moving obstacle enters and when it leaves a point of a robot workspace, it is possible to obtain other search-space representations, such as safe intervals proposed in [22] or obstacle layers proposed in the author's previous work [23]. Then, motion planning among moving obstacles can be considered an action-event synchronization problem. An action-event synchronization (Sec. 6.6) that assumes arbitrarily long action duration protraction has been investigated against time consistency and feasibility for robot motion planning.

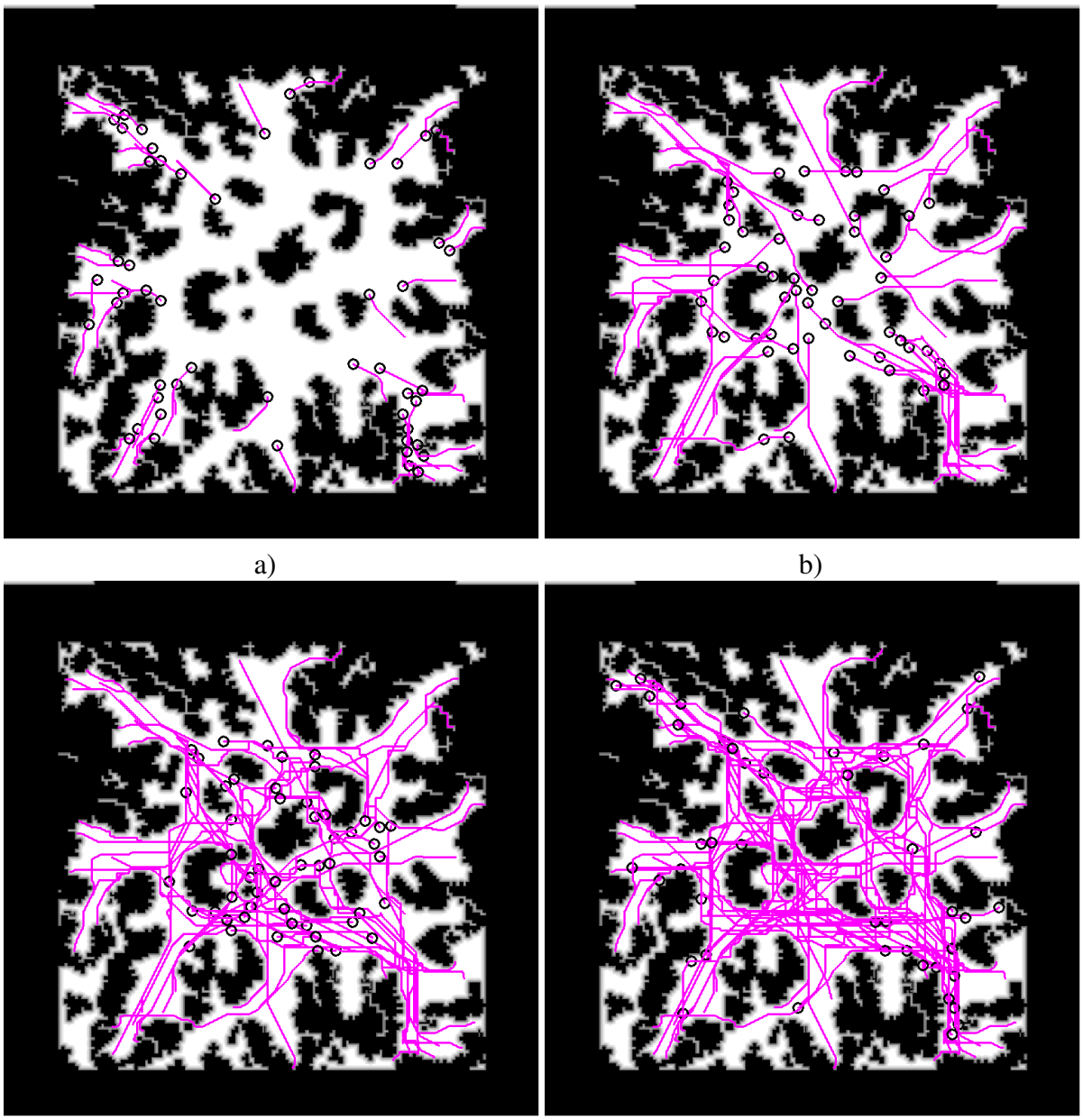


Figure 6.16: A collision-free motion of 51 robots on a *battleground* map from the *wc3* map set, at four consecutive stages (a, b, c, and d), where white fields are a free space, circles are robots, and lines are traveled paths.

In general, heuristic search methods can be used for a minimum-time path search in safe interval graphs; however, the following conditions must be fulfilled. A search algorithm needs to memorize the full state of a robot (i.e., state and time point) alongside of each safe-interval graph node. If a heuristic search is used, it is important to pop nodes from an open list in a lexicographical f, g ascending order. Finally, a search algorithm needs to allow for reopening nodes; therefore, algorithms based on ARA* cannot be directly used for planning in a safe

Table 6.1: The average experimental results for a minimum-time path search in a safe interval graph using A*; #Mov.Obst.: number of moving obstacles, T_p: planning time [ms], #S.Steps: number of search steps, #Heap: number of heap operations, #Succs: number of iterations over successors, P. Cost: path cost [map cells].

Map set	#Maps used	#Mov.Obst.	T_p	#S.Steps	#Heap	#Succs	P. Cost
rooms	30	0	222.87	78520	269444	78520	617.319
	30	1	382.67	85574	288480	85574	621.539
	30	2	386.77	80745	268553	80745	629.534
	30	5	474.83	81804	266366	81804	628.044
	30	10	637.84	81897	256587	81897	627.590
	30	20	1151.16	108650	314137	108650	673.179
	30	50	2748.65	147382	382062	147382	767.307
wc3	36	0	82.04	28488	99537	28488	463.262
	36	1	163.94	34716	118284	34716	465.254
	36	2	174.11	33803	113654	33803	465.843
	36	5	246.32	39740	125999	39740	482.181
	36	10	435.35	53779	161056	53779	498.705
	36	20	722.71	62262	171799	62262	518.894
	34	50	1831.44	90208	221978	90208	587.775
mazes_16	10	0	75.22	26694	89857	26694	484.329
	10	1	170.70	36876	120942	36876	542.923
	10	2	159.17	31651	103481	31651	539.778
	10	5	185.73	33788	107583	33788	553.290
	10	10	254.32	38006	114149	38006	591.769
	10	20	465.07	48359	131945	48359	699.647
	10	50	1179.81	66472	167795	66472	927.526

interval graph. Heuristic improving algorithms, such as LRTA* or RRTA*, can be used for planning in a safe interval graph; however, the heuristic learning needs to be modified to preserve optimality.

An event-based approach with a simple action-event synchronization was implemented and tested. The tests covered the use of A* for robot motion planning among moving obstacles (up to 50) on a variety 2D maps. The benchmark results suggest that with such state-space decomposition planning, time grows linearly with the number of moving obstacles.

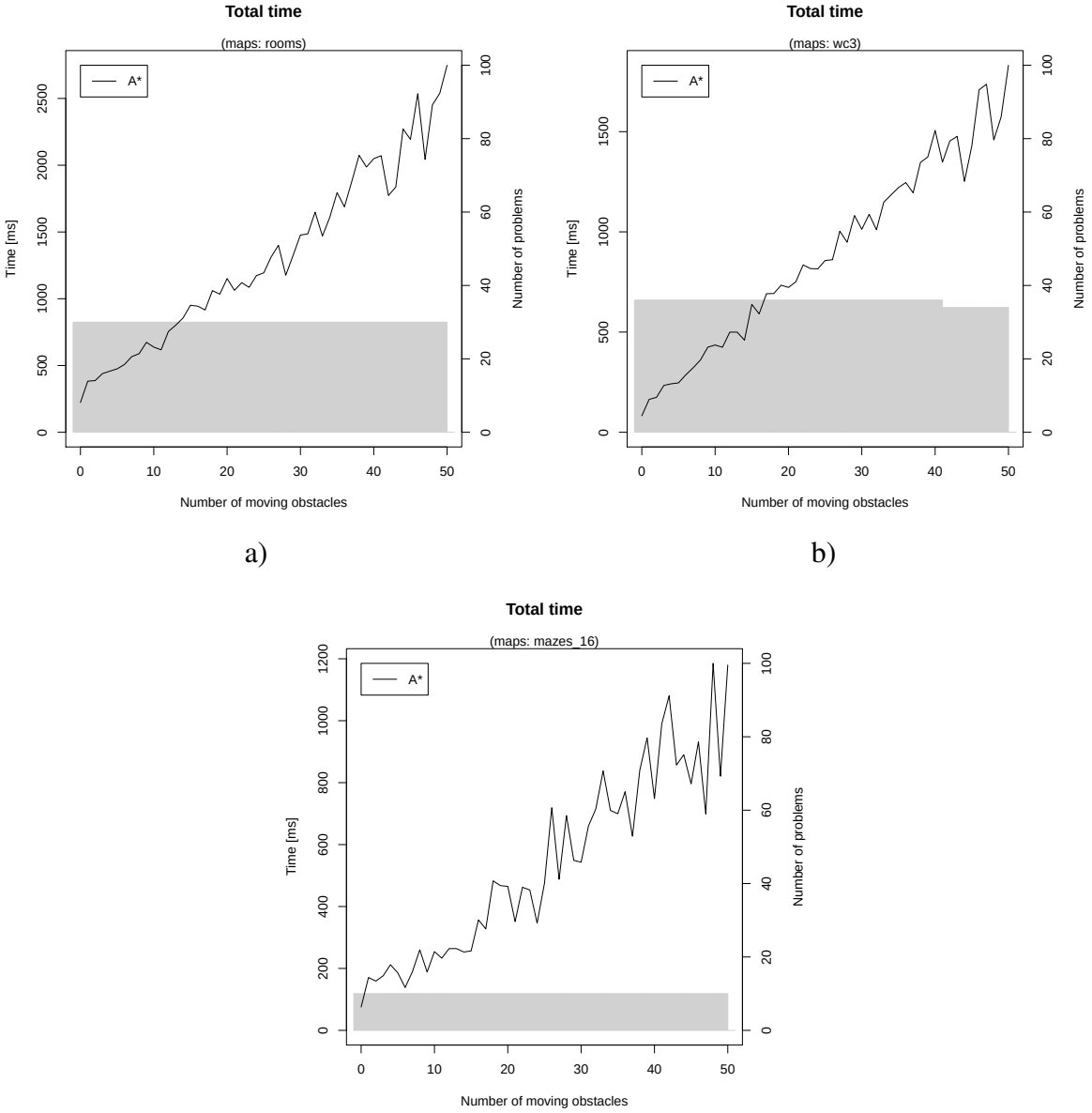


Figure 6.17: Total time for planning amid moving obstacles, for (a) *rooms*, (b) *wc3*, and (c) *mazes_16* map sets; in the background, the histogram of problems is plotted in gray.

7. Hierarchical Planning in a Dynamic Environment

In this chapter, hierarchical planning is used for planning in a safe interval graph. Hierarchical planning aims to speed up searching by solving a simplified problem and using gathered information to solve the original problem. A background for hierarchical planning is provided in Section 7.1, in which abstraction hierarchies are explained (Sec. 7.1.1) and algorithms for hierarchical planning are discussed (Secs. 7.1.2 through 7.1.5), with special attention given to the Switchback algorithm (Sec. 7.1.4). In Section 7.3, the Switchback algorithm is used for motion planning in a fully-known dynamic environment. Next, a new Real-time Switchback algorithm is proposed (Sec. 7.4) that is used for real-time motion planning in an unknown dynamic environment.

7.1 Hierarchical Planning: Background

Hierarchical planning can be achieved with various techniques, for example, the refinement of a simplified plan [15, 104, 105, 106] or the use of knowledge from an explored part of a simplified search space as a heuristic for a search in an original search space [107, 108, 16, 17].

As a search-space simplification is connected with abstraction transformation, in this section, an abstraction transformation and types of abstraction are discussed first (Sec. 7.1.1). Next, the algorithms for hierarchical planning are presented (Secs. 7.1.2 through 7.1.5).

7.1.1 Abstraction Hierarchies

In [54, Ch. 4.1], an *abstraction transformation* is defined as the mapping $\phi : S \rightarrow S'$ that maps each state $s \in S$ to an abstract state $\phi(s) \in S'$ and each action $a \in A$ to an abstract action $\phi(a) \in A'$. In [15], Holte proposed analyzing existing abstraction transformations using a graph-oriented perspective. The definition of the abstraction transformation given above is coherent with the definition of a *homomorphism* used in graph theory [109]:

Definition 7.1 *Let G and H be graphs. A function $\phi : V(G) \to V(H)$ is a homomorphism from G to H if it preserves edges, that is, if for any edge $[u, v]$ of G, $[\phi(u), \phi(v)]$ is an edge of H.*

It should be emphasized that, with the homomorphism definition, it is possible to have $\phi(u) = \phi(v)$, which induces loops in H (but, typically these are ignored while planning). In particular, graph H can be a *quotient* of graph G (Fig. 7.1), which is defined as follows [109]:

Definition 7.2 *Let G be a graph and let $\mathcal{P} = \{V_1, \ldots, V_2\}$ be a partition of the vertex set of G into non-empty classes. The* quotient *$G/\mathcal{P}$ of G by $\mathcal{P}$ is the graph whose vertices are the sets $V_1, \ldots, V_k$ and whose edges are the pairs $[V_i, V_j]$, $i \neq j$, such that there are $u_i \in V_i$, $u_j \in V_j$ with $[u_i, u_j] \in E(G)$. The mapping $\pi_{\mathcal{P}} : V(G) \to V(G/\mathcal{P})$ defined by $\pi_{\mathcal{P}}(u) = V_i$ such that $u \in V_i$, is the* natural *map for $\mathcal{P}$. (...) $\pi_{\mathcal{P}}$ is a homomorphism, if and only if V_i is an independent set for each i.*

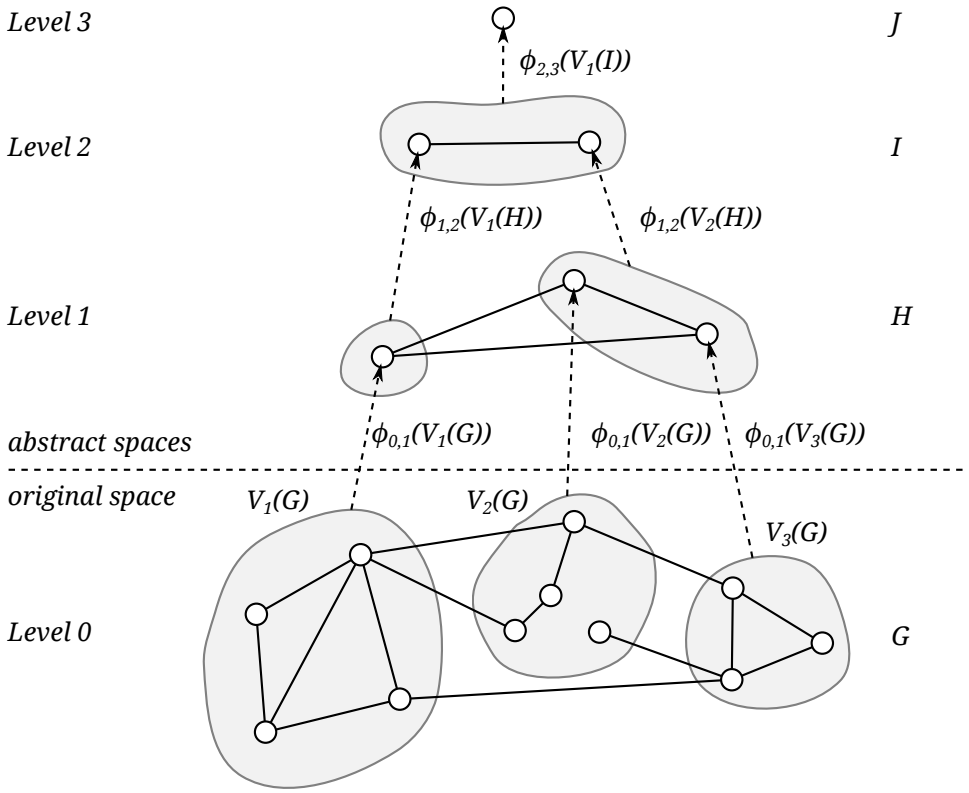


Figure 7.1: Example of multi-level hierarchy of quotient graphs.

Typically, it is desirable to create an abstraction hierarchy that reduces the search-space size (the number of nodes and edges); therefore, quotient graphs are commonly used in many hierarchical planning applications [15, 107, 105, 106, 110].

In action planning and motion planning, graph nodes hold some domain-specific *labels* (e.g., in action planning it could be a set of predicates, while in 2D motion planning it could be a robot position in a plane (x, y)). As a quotient graph is constructed using surjective mapping, the question arises 'how does this mapping apply to labels?' Answering this question is not trivial, and preferable methods differ between domains.

For some domains, it is possible to automatically compute a single abstract label for all nodes that are mapped to the same abstract node, for example, by calculation of a center of mass of points in n-dimensional space. Another simple approach is to provide such mapping $\phi : V(G) \to V(H)$ for which $V(H) \subseteq V(G)$ (i.e., all nodes in a partition are represented by one selected node in this partition preserving its labels, for example, all robot positions in a partition are mapped to the one selected). In addition, for domains in which attributes are n-tuples representing points in an n-dimensional metric space, it is possible to construct mapping that drops one or a few attributes, resulting in points in k-dimensional space, where $k < n$, which is merely partitioning. For example, a three-dimensional motion planning problem, in which $q = (x, y, z)$, can be simplified to a two-dimensional motion planning by dropping the z coordinate. This is a common method used for kinodynamic motion planning [60, 11, 18], in which a heuristic search held in a state space, in which $s = (x, y, \dot{x}, \dot{y})$ is sped up using a heuristic computed by a search in a 2D grid in which $q = (x, y)$. As shown in Section 7.2, this method can also speed up time-dependent planning, in which a static environment (without moving obstacles) can be considered an abstraction of a dynamic environment.

7.1.2 Refinement Planning

It is a natural approach to start planning in an abstract graph. Next, an abstract path needs to be *refined* at each lower abstraction level, down to an original space (Fig. 7.2b). An abstract path can be represented as a sequence of edges (edge path) or as a sequence of nodes (node path). In [15], it has been shown that the node-path refinement is a better technique since abstract nodes can be connected by more than one edge.

Although refinement planning is performed in a top-down manner, a path-planning problem is typically defined in an original search space. Therefore, unless there is a straightforward mapping to the topmost abstraction level, a refinement algorithm commences with an ascending phase that tries to reach the topmost level, or the level with the first common ancestor (Fig. 7.2a).

A particular partition of an original graph G has a crucial effect on the efficiency of refinement planning. In the worst case, it may happen that an abstract path cannot be refined in an original graph. Such a situation is shown in Figure 7.3, in which the problem is to find a

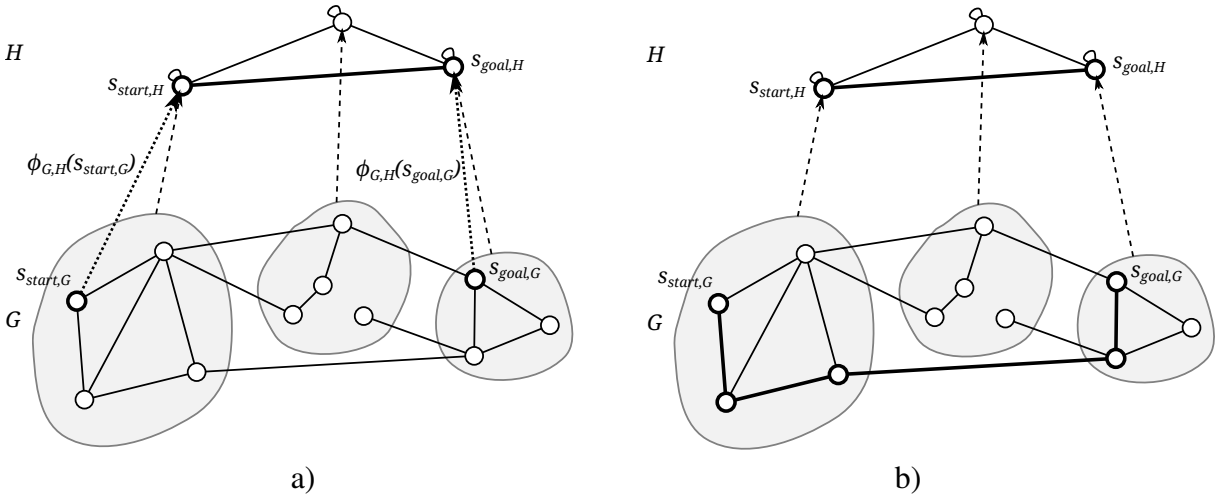


Figure 7.2: Example of refinement planning: a) ascending to find the topmost abstraction level and abstract planning, b) abstract path refinement at the original level.

path from a_{start} to c_{goal}. In H, path $P_H =< v_1, v_3 >$ is found, which needs further refinement. Moreover, P_H corresponds to $P_G =< d, e >$ in G. As set V_3 contains nodes c_{goal} and e that are not adjacent, there is no path to c_{goal} that could contain e. Therefore, the algorithm needs to step back in the hierarchy to find another abstract path without a $\{v_1, v_3\}$ edge. In contrast to this example, a refinement is called a *monotonic refinement* if it does not require stepping back in the hierarchy. In other words, if an abstract path exists, then a path in an original search space also exists.

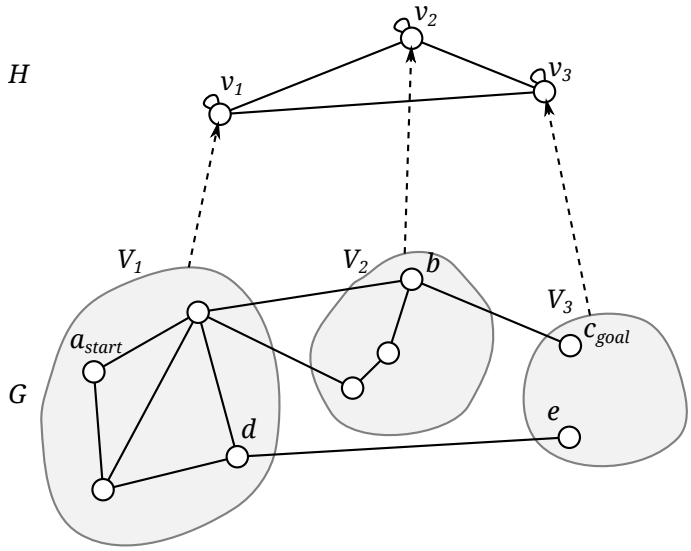


Figure 7.3: Example of refinement planning, which is non-monotonic due to the chosen partitioning.

The *classical refinement* (monotonic node-path refinement) does not require stepping back in the hierarchy, but it may provide highly sub-optimal paths [15]. In hierarchical path-finding

A* (HPA*)[104], a path sub-optimality is reduced in post processing by path smoothing. Although this method applies well to grid-based path-finding, it is not general.

Sub-optimality can be significantly reduced with a technique called *path marking* that uses an abstract path to reduce the search space at the lower abstraction level (e.g., partial-refinement A*, PRA*[105, 106]). At a lower level, a search is performed from scratch; however, it is limited to nodes that belong to abstract nodes along an abstract path.

Alternating search directions, used by the AltO algorithm [20], is another refinement technique that provides shorter paths than a classical refinement. During a search, the cost from a start node to each expanded node is calculated; thus, if the search at the lower abstraction level is performed in an opposite direction, these costs can be used as heuristics. The direction of a search is altered again, while stepping downwards further. This technique outperformed classical and path-marking refinement methods in most tests [20]. In fact, alternating search direction has more in common with planning using abstraction-based heuristics (Fig. 7.4). It can be viewed as a refinement in the sense that a lower-level search is limited to a search space explored by the higher level.

7.1.3 Planning with Abstraction-based Heuristics

As already mentioned in the previous section, the solution found in an abstract search space can be used as a heuristic for the original problem solving. This can be simply implemented as an A* algorithm that is using a heuristic calculated by an abstract search (e.g., BFS). However, such an implementation may not provide the expected efficiency gain, which is a subject of Valtorta's theorem [54, Ch. 4.2]:

> Let u be any node that is necessarily expanded when the problem (s, t) is solved in S with the BFS. Let $\phi : S \to S'$ be any abstraction mapping, and the heuristic estimate $h(u)$ be computed by BFS from $\phi(u)$ to $\phi(t)$. If the problem is solved by the A* algorithm using h, then either u itself will be expanded or $\phi(u)$ will be expanded.

From the theorem, it stands that the evaluation of a heuristic using a blind search in an abstract search space may expand as many nodes as the blind search in an original problem search space. Valtorta's theorem applies to all aforementioned abstract mapping methods.

The approach described by Valtorta's theorem has been called a naive hierarchical A* search [107]. However, it holds only for those hierarchical heuristic search algorithms that (original and all abstract levels) perform searching in the same direction (forward or backward) at all levels.

Heuristic Caching

A lot of effort has been made to break the so-called Valtorta's barrier [15, 107, 20], which means to expand fewer nodes than are expanded by a blind search in the original search space. This has been achieved with two heuristic caching techniques. As described in [107], during the forward search from s_1 to s_g, apart from the cost from s_1 to s_g, costs from s_1 to all other expanded nodes are also calculated. Moreover, as the path cost h^* found in the abstract search is known, it can be used for the heuristic calculation, such that $h(s) = h^* - g(s)$, which has been referred to as h^* caching [107]. Nevertheless, these techniques still require an abstract path for each expanded node to be found.

To decrease the number of abstract searches, *P-g* caching has been proposed [107]. With this technique, an abstract path is calculated only once, providing the abstract path cost P. In the original space, the heuristic for node s is calculated as $h'(s) = P - g(s)$. In [107], it has been proven that such a heuristic is admissible. Moreover, for nodes along the path, it is an ideal heuristic. However, for nodes off the optimal path, *P-g* caching may produce a heuristic value lower than the minimal possible value in the given domain $d(s)$ (e.g., the Euclidean distance for path searching in Euclidean space). Thus, the proper heuristic is calculated as the $max(h'(s), d(s))$. In fact, the adaptive A* (AA*) [65] and the real-time adaptive A* (RTAA*) [101] use *P-g* caching.

Alternating Search Direction

It has been shown that an alternating search direction is an efficient technique that breaks Valtorta's barrier [15, 20]. As already discussed, alternating search direction has been originally proposed as a top-down refinement method (AltO); however, it can also be used in a bottom-up manner, where the higher-level abstract planning is performed (and resumed if necessary) whenever a heuristic at the lower level is required. If a ground-level search performs a complete search and if a chosen abstraction method ensures admissible heuristics, then such a bottom-up algorithm is optimal. This approach has been described in [16, 17] as the Switchback algorithm, which is used in this thesis and will be discussed in Section 7.1.4.

If an abstract search is aimed to provide a heuristic for an original search, the admissibility of such a heuristic is an important issue to consider. A quotient graph and restriction may produce inadmissible heuristics, which may lead to sub-optimal solutions, or admissible but less informed heuristics. Therefore, it is crucial to properly compute abstract edge costs [107, 111]. For example, if abstract nodes represent regions (open subsets of a metric space), an abstract edge cost can be calculated as $cost(V_1, V_2) = \inf_{v_1 \in V_1} \inf_{v_2 \in V_2} d(v_1, v_2)$. Although a heuristic computed using a graph with such edge costs will never overestimate the true shortest-path

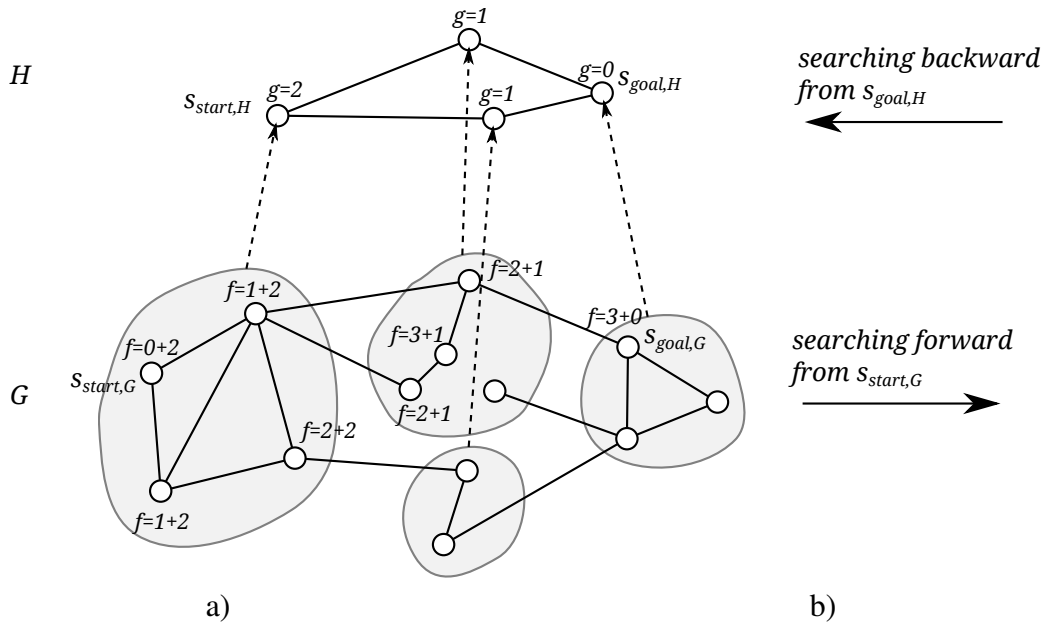


Figure 7.4: Example of alternating search directions in which lower levels use heuristics based on costs computed by a higher level search conducted in opposite direction.

cost, abstract cost computed using this method may significantly underestimate it, which is undesirable. (The less informed heuristic that is used, the bigger the number of nodes expanded by the A* algorithm [3, Ch. 3.6.1].)

Pattern Databases

The heuristics discussed so far are useful for a single-query search. If a search space is initially known and is of a tractable size, it is possible to pre-calculate the shortest paths between all pairs of nodes, which can be used as heuristics for a multiple-query search. This approach is known as a pattern database [112]. Of course, it may be time- and space-consuming; therefore, abstractions can be used to speed up pattern-database calculation [113, 114]. A True distance heuristics (TDHs) [108] are similar to pattern databases; however, they are designed to be used with map-based search spaces.

7.1.4 Switchback: Optimal Bottom-up Search

Switchback [16] is a hierarchical heuristic search algorithm utilizing alternating search directions. The algorithm allows for searching with multiple levels of search-space abstractions, such that the search is conducted in a bottom-up manner, where an original search space is the base level (level 0).

A major advantage of Switchback is its simplicity, as it is built upon the A* algorithm. The main modification in relation to A* is that, in Switchback, a heuristic calculation involves searching at a higher abstraction level. No less important is that Switchback is optimal if a path cost at each abstraction level does not overestimate the true path cost [16].

The pseudocode of the Switchback is shown in Algorithm 7.1. The number of abstraction levels represented by i_{top} is a matter of choice and depends on a particular domain. In addition, the direction of a search at the base level $direction(0)$ can be freely chosen. It is important that the search direction is switched between hierarchy levels (Fig. 7.4).

This allows for the use of $i + 1$-level g values as heuristics in an i-level search (line 11 in Alg. 7.1). Each search level maintains its own open list. As a search direction is changed between abstraction levels, a neighbor set consists of successors or predecessors for a forward search and backward search, respectively (lines 31–32 in Alg. 7.1).

If the SEARCH function is called for the first time, for a given level i, then this level needs to be initialized (line 40 in Alg. 7.1). It is noteworthy that, at the initialization stage, a heuristic for a start state, $s_{start,i}$, is calculated before the state is pushed to an open list (line 26 in Alg. 7.1). Thus, at the initialization of the base level, the higher-level searches need to be accomplished (line 10 in Alg. 7.1). From this point on, if a g value of an abstract node that has not been expanded yet is requested, the higher levels are resumed (lines 10–11 in Alg. 7.1) (i.e., they continue searching until the requested node is popped from an open list (lines 42–43 and 6–7 in Alg. 7.1)). In this thesis, Switchback has been chosen as the basic hierarchical planning algorithm.

7.1.5 Optimal Top-down Search

In Switchback, search levels are tightly coupled, that is, lower levels resume higher-level searches; hence, all search data, such as open lists or closed lists, need to be maintained until the base level finishes. In contrast to Switchback, AltO, which conducts a search in a top-down manner, limits the search node expansion to only nodes that have expanded counterparts at higher abstraction levels [20]. Thus, in AltO, resources required for a higher-level search can be freed; only g values need to be memorized.

To achieve an optimal top-down search, two main approaches are possible. The first approach is to continue the search at a higher-level, even though the solution is found, until the f value at the top of the open list exceeds some multiplicity (overhead) of the optimal solution length. The second approach is similar to *P-g* caching, that is, for nodes in which higher-level nodes were not expanded, it is known that the f value cannot exceed the f value at the top of the open list calculated at the higher-level, $f(\phi(s)) \geq f_{top}$; hence, $g(\phi(s)) \geq f_{top} - h(\phi(s))$.

Algorithm 7.1 Switchback algorithm, where i denotes the abstraction level and $i = 0$ is the base level.

```
1:  function SWITCHBACK(s_start, s_goal)
2:      if SEARCH(s_start, s_goal, 0) = success then
3:          return GETPLAN(s_goal)
4:  function KEY(s, i)
5:      return g(s)+ HEURISTIC(s, i)
6:  function SOLUTIONFOUND(s_query,i)
7:      return TOPOPEN() = s_query,i OR (visited(s_query,i) AND NOT open(s_query,i))
8:  function HEURISTIC(s, i)
9:      if i = i_top then return h(s, s_goal,i)
10:     if SEARCH(φ_{i+1}(s_goal,i), φ_{i+1}(s), i + 1) = success then
11:         return g(φ_{i+1}(s))
12:     else
13:         return ∞
14: function INITIALIZE(s_start,i, s_query,i, i)
15:     for each state s ∈ S_i do
16:         visited(s) = false
17:         parent(s) = NULL
18:     if direction(i) = forward then
19:         s_start,i = φ_i(s_start)
20:         s_goal,i = φ_i(s_query)
21:     if direction(i) = backward then
22:         s_start,i = φ_i(s_query)
23:         s_goal,i = φ_i(s_start)
24:     visited(s_start,i) = true
25:     g(s_start,i) = 0
26:     PUSH(open_list_i, KEY(s_start,i))
27:     initialized(i) = true
28: function SEARCHSTEP()
29:     s = TOP(open_list_i)
30:     POP(open_list_i)
31:     if direction(i) = forward then children = Succ_i(s)
32:     if direction(i) = backward then children = Pred_i(s)
33:     for all s' ∈ children do
34:         if NOT visited(s') OR g(s') > cost(s, s') + g(s) then
35:             parent(s') = s
36:             g(s') = cost(s, s') + g(s)
37:             if NOT visited(s') then visited(s') = true
38:             PUSH(open_list_i, s')
39: function SEARCH(s_start,i, s_query,i, i)
40:     if NOT initialized(i) then INITIALIZE(s_start,i, s_query,i, i)
41:     while NOT EMPTY(open_list_i) AND TOPKEY(open_list_i) ≠ ∞ do
42:         if SOLUTIONFOUND(s_query,i) then
43:             return success
44:         SEARCHSTEP()
45:     return failure
```

These two methods can be easily combined, such that a heuristic calculated at the k-th level is

calculated as follows:

$$h_k(s) = \begin{cases} g_{k+1}(\phi(s)) & \text{if } \phi(s) \text{ has been expanded} \\ \max(f_{top,k+1} - h(\phi(s)), h_{basic}(s)) & \text{otherwise.} \end{cases} \tag{7.1}$$

A heuristic calculated in accordance with Eq. 7.1 is admissible, thus the algorithm allows for an optimal search. However, such a heuristic function, in most cases, has local minima, which, in the context of real-time-search algorithms, are called depression regions. Existence of such regions in a real-time search makes the robot visit the same places multiple times (scrubbing behavior, see Sec. 6.7.1). An approach discussed in this section can be found in the source code of the SBPL library[51], in which it aims to speed up a state-lattice search by providing a heuristic computed as a search in a 2D grid. This approach has been used in [18]; however, to the author's best knowledge, it has not been described in detail.

7.1.6 Conclusions

A hierarchical search is a natural method of solving complex planning problems. The methods discussed in this chapter fall into two categories: refinement planning and planning with heuristic-based abstractions. Regarding the subject of this thesis, planning with heuristic-based abstractions, in particular, alternating the search-direction method utilized by the Switchback algorithm [16] is of interest.

Most hierarchical search methods are used for classical action planning (e.g., STRIPS-like planning) [20, 115, 116, 16, 17], and path-finding for video games [107, 104, 105, 106]. In the context of robot motion planning, hierarchical planning is also used, although it is not directly referred to as such. For example, many mobile robot systems consist of global path planning and a local search for collision avoidance, which are composed and run in a top-down manner. In addition, the alternating search-direction method is a common approach in robot motion planning, that is, a 2D (x, y) grid-search provides a heuristic to speed up the search in a 3D (x, y, θ) or 4D state lattice [60, 11, 48, 18]. In this chapter, the alternating search-direction method is used for global optimal and real-time motion planning among moving obstacles. The planning algorithm consisting of three levels designed for a differential-drive robot will be presented in chapter 8.

7.2 Search Space for Hierarchical Planning in a Dynamic Environment

A static state space (without moving obstacles) can be considered a homomorphic abstraction (a quotient graph) of a state-time space, in which moving obstacle regions of $\mathcal{ST}$ are omitted; thus, $\phi(\tilde{s}) : \mathcal{ST} \rightarrow S$ is a surjective mapping defined as follows:

$$\phi((s,t)) = s, \tag{7.2}$$

such that $(s,t) \in \mathcal{ST}_{free} \cup \mathcal{ST}_{obs} \setminus \mathcal{ST}_{obs,static}$, are mapped to $s \in S_{free}$, where $\mathcal{ST}_{obs,static}$ is the region of collision with static obstacles. Hence, the following is proposed.

Proposition 7.1 *Let $\mathcal{ST}$ be a state-time space that includes static and moving obstacles, and S be a state space space representing only static obstacles that is obtained using Eq. 7.2. Then, costs of the shortest paths computed by searching in S can be used as admissible heuristics for a heuristic shortest-path search in $\mathcal{ST}$.*

Proof 7.1 *Let $\mathcal{ST}' = S \times T$ be a state-time space without moving obstacles constructed upon a non-temporal state space S, and $\mathcal{ST}_{free} = \mathcal{ST} \setminus \mathcal{ST}_{obs}$ be a collision-free region of a state-time space, such that, if there are no moving obstacles in a workspace, then $\mathcal{ST}_{free} = \mathcal{ST}'_{free}$, where $\mathcal{ST}'_{free} \subseteq \mathcal{ST}'$. As the presence of moving obstacles can only enlarge $\mathcal{ST}_{obs}$ thus shrink $\mathcal{ST}_{free}$, it must be that $\mathcal{ST}_{free} \subseteq \mathcal{ST}'_{free}$. The same stands for the number of applicable actions in $\mathcal{ST}$, that is, $\mathcal{AT} \subseteq \mathcal{AT}'$.*

Now, let Π' be the shortest collision-free path in $\mathcal{ST}'_{free}$ connecting some two states. If Π' is not the collision-free path in $\mathcal{ST}_{free}$, that is, $\Pi' \cap \mathcal{ST}_{obs} \neq \emptyset$, then the shortest collision-free path Π in $\mathcal{ST}_{free}$ (if such a path exists) cannot be shorter than Π', as $\mathcal{ST}_{free} \subseteq \mathcal{ST}'_{free}$ and no new shorter actions were introduced to $\mathcal{AT}$. Hence, a cost of the shortest path in $\mathcal{ST}'$ cannot be greater than the cost of the shortest path in $\mathcal{ST}$. Therefore, costs of the shortest paths in $\mathcal{ST}'$ can be used as admissible heuristics for planning in $\mathcal{ST}$. □

In addition to abstractions obtained by omitting moving obstacles, a static state space can have abstraction levels of itself, for example, an (x, y) grid can be used as an abstraction of an (x, y, θ) state lattice; hence, multiple abstraction levels are possible.

Among the hierarchical planning algorithms discussed in Section 7.1, the alternating search-direction algorithms running both top-down [15, 20](Secs. 7.1.2 and 7.1.5) and bottom-up, such as Switchback [16] (Sec. 7.1.4), can be easily applied to planning within a hierarchy consisting of static and dynamic search spaces.

In this thesis, it is proposed to use an abstraction-based heuristic search to compose hierarchical planning algorithms consisting of the following algorithms: A*, local A*, D* Extra Lite, and AD*-Cut (local A* works like LRTA*, but without learning). As robot motion planning in a dynamic environment requires fast algorithms, in Sec. 7.4 a real-time version of the Switchback algorithm is proposed. The algorithms are tested within two scenarios: first, when full trajectories of moving obstacles are known (Sec. 7.3) and, second, when moving and static obstacles are detected in a certain range around a robot in an incremental search (Sec. 7.4).

7.3 Hierarchical Planning in a Fully-known Dynamic Environment

In this section, the algorithms using an abstraction-based heuristic are described and evaluated for motion planning among moving obstacles on a 2D map. The two basic approaches are compared: a top-down optimal search (as discussed in Sec. 7.1.5) and the Switchback algorithm (discussed in Sec. 7.1.4), both using the alternating search-direction technique.

A hierarchical search space used in tests consists of a safe interval graph (Sec. 6.5), which is the base level (level 0), and one abstract level (level 1) representing a 2D map with static obstacles. As action-event synchronization for planning in a safe interval graph proposed in Section 6.6 is designed to be used in a forward search, planning at the base level is performed in that direction, which imposes a backward search at level 1.

The algorithms were tested within the same scenario as described in Section 6.8, that is, motion planning on 2D artificial maps among moving obstacles, from 0 to 50. The experimental results are presented in Table 7.1.

In all tests, both Switchback and top-down algorithms were quicker than A*, regardless of the number of moving obstacles. In total, Switchback was 1.67 times quicker than A*, and top-down was 1.53 times quicker than A*. In general, the total running times that are shown in Figure 7.5 suggest that Switchback is the quickest, regardless of a particular map set and the number of moving obstacles.

In Table 7.1, the number of search steps for hierarchical algorithms is simply the sum of search steps at both hierarchy levels. Although A* does fewer search steps in total, all searching is held in a safe interval graph; hence, each successor generation requires an action-event synchronization with additional computations. The two hierarchical algorithms do most of the work by a backward search in an abstract space, which is reflected in the number of predecessor-generation calls (column #Pred in Tab. 7.1). As supposed, with an abstraction-based heuristic, hierarchical algorithms do significantly fewer steps in an original

Table 7.1: Average experimental results for a minimum-time path search in a safe interval graph; #Mov.Obst.: number of moving obstacles, T_p: planning time [ms], R_p: planning time performance over A*, #S.Steps: number of search steps, #Heap: number of heap operations, #Pred: number of iterations over predecessors, #Succs: number of iterations over successors, P. Cost: path cost [map cells].

#Mov. Obst.	Algorithm	T_p	R_p	#S.Steps	#Heap	#Pred	#Succs	P. Cost
rooms								
1	A*	382.67	1.00	85574	288480	0	85574	621.539
	Switchback	202.90	**1.89**	89824	321310	86960	2864	621.539
	Top-down	214.52	1.78	88945	318413	84383	4562	621.539
10	A*	637.84	1.00	81897	256587	0	81897	627.590
	Switchback	303.31	**2.10**	99865	350136	87481	12384	627.590
	Top-down	306.53	2.08	91155	321721	72411	18744	627.590
50	A*	2748.65	1.00	147382	382062	0	147382	767.307
	Switchback	1383.47	**1.99**	191703	617377	139503	52200	767.307
	Top-down	1636.74	1.68	150781	476787	72880	77901	767.307
wc3								
1	A*	163.94	1.00	34716	118284	0	34716	465.254
	Switchback	99.57	**1.65**	39883	141298	34815	5068	465.254
	Top-down	113.48	1.44	40865	143006	31743	9122	465.254
10	A*	435.35	1.00	53779	161056	0	53779	498.705
	Switchback	365.07	**1.19**	82294	274396	56056	26237	498.705
	Top-down	399.54	1.09	71878	234042	31814	40063	498.705
50	A*	1831.44	1.00	90208	221978	0	90208	587.775
	Switchback	1550.00	**1.18**	148829	445090	85280	63549	587.775
	Top-down	1703.68	1.07	110009	303925	28713	81296	587.775
mazes_16								
1	A*	170.70	1.00	36876	120942	0	36876	542.923
	Switchback	85.15	**2.00**	33985	118585	29924	4061	542.923
	Top-down	91.38	1.87	34261	119784	28391	5870	542.923
10	A*	254.32	1.00	38006	114149	0	38006	591.769
	Switchback	152.79	**1.66**	46833	157270	36612	10221	591.769
	Top-down	180.99	1.41	45054	150812	28647	16407	591.769
50	A*	1179.81	1.00	66472	167795	0	66472	927.526
	Switchback	850.13	**1.39**	100156	307076	65199	34957	927.526
	Top-down	890.83	1.32	77647	228388	31167	46480	927.526
Total Performance Ratio								
A*/Switchback		1.67		0.76	0.67	0.00	3.00	1.00
A*/Top-down		1.53		0.89	0.80	0.00	2.11	1.00

space than the A* algorithm. This is depicted in Figure 7.6, in which nodes expanded at the base level are illustrated.

As a final remark, the superiority of an abstraction-based heuristic search may be even more pronounced when bucket-like structures are used for open-list maintenance, which is the case of the SBPL library [51, 18], or when the novel L* algorithm is used [117]. This is likely, as the number of open-list operations done by hierarchical search algorithms is higher than in

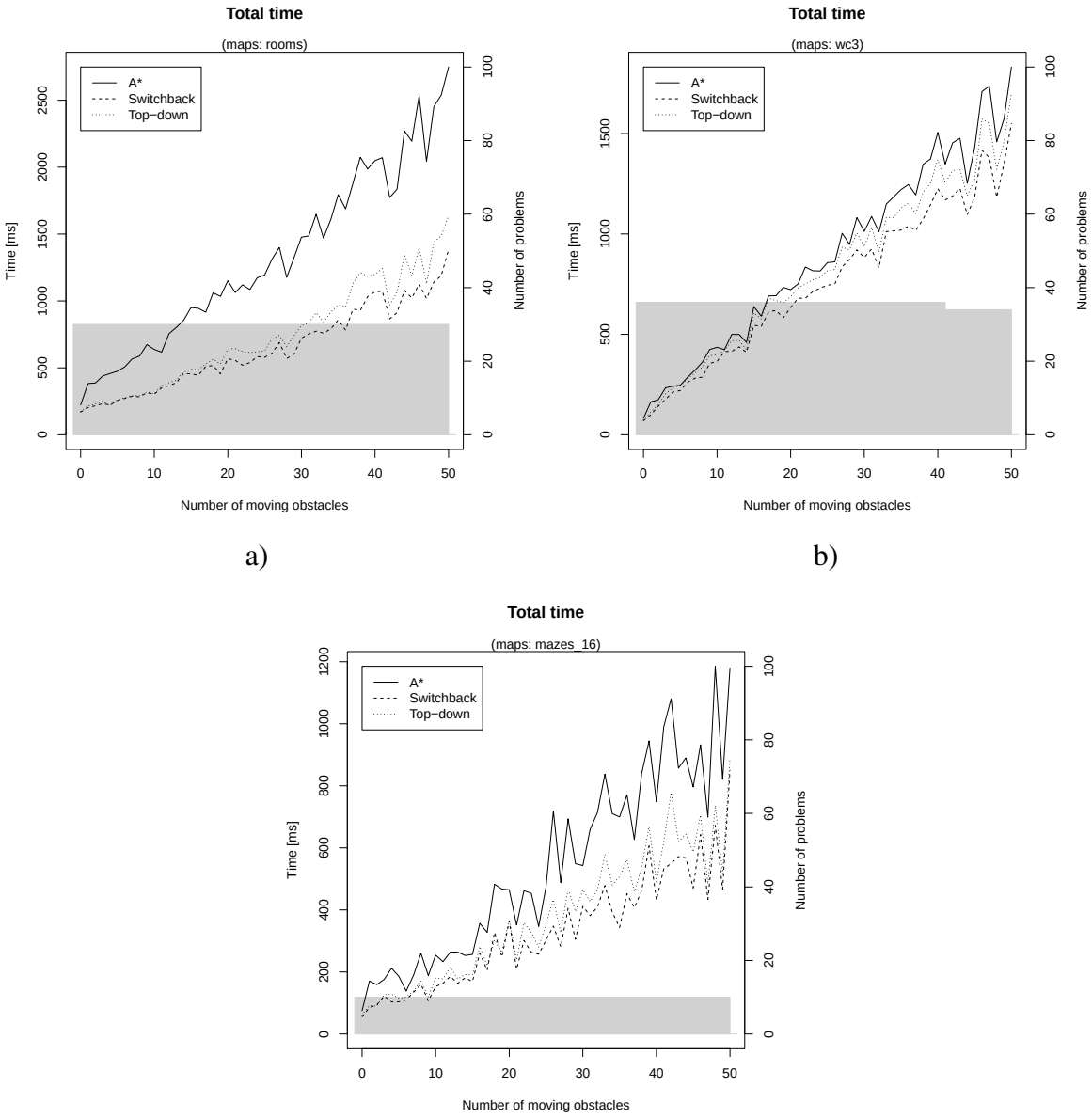
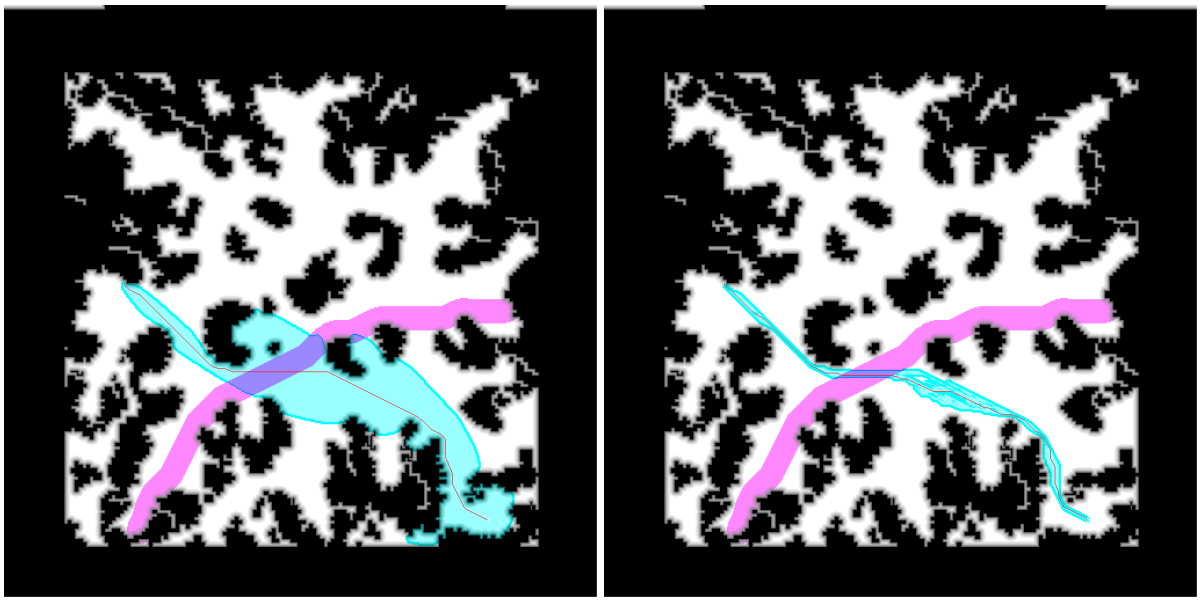


Figure 7.5: Total times for planning amid moving obstacles, for (a) *rooms*, (b) *wc3*, (c) and *mazes_16* map sets; in the background, the histogram of problems is plotted in gray.

the case of A*; hence, the possible increase in speed will be more pronounced for hierarchical algorithms.



(a) *(b)*

Figure 7.6: Nodes expanded by A* (a) and Switchback (b) in an original search space (i.e., a safe interval graph) with one moving obstacle; obstacle layers are projected onto a 2D grid. Blue depicts expanded nodes, red depicts a moving obstacle region ($\mathcal{ST}_{obs}$), a thin line denotes the shortest path, and black denotes static obstacles.

7.4 Real-time Hierarchical Planning in an Unknown Dynamic Environment

The hierarchical search algorithms discussed in Section 7.1 perform a complete global search. However, in some cases, such as in video games or robot navigation, it is more important to act quickly, possibly in real time, than to act optimally. The problem of a real-time hierarchical search has been addressed by Bulitko [110], in which the path-refinement learning real-time search algorithm (PR LRTS) has been proposed. The PR LRTS combines path-marking refinement with an LRTA*, such that the search space of the real-time algorithm is restricted to a corridor along an abstract path. Such a combination of a global A* with a local LRTS helps to reduce the scrubbing behavior that is unavoidable in a local search (Sec. 6.7.1).

The PR LRTS, as it relies on path refinement, does not guarantee that a collision-free path exists in a safe interval graph limited to nodes expanded in a state space representing a static environment. Therefore, in the following section, a new Real-time Switchback algorithm is proposed.

7.4.1 Real-time Switchback

Switchback resumes searching at a higher level whenever a heuristic is required at the lower level; therefore, search levels are tightly coupled. Consequently, a lower-level search cannot be finished until an abstract search is finished.

As a higher-level search may require considerable time to expand a node for which a heuristic is requested, Switchback is not suitable for motion planning in a dynamic environment in which decisions need to be made quickly. In this thesis, a new Real-time Switchback algorithm is proposed, as shown in Algorithm 7.2.

The main modification with respect to the Switchback algorithm is the time limit introduced to a search loop. A timeout at any level of searching results in termination with failure, following the current search step (lines 36–37 in Alg. 7.2). In the case of a higher-level timeout, the only consequence is that infinity is returned instead of the cost to the goal (lines 10–11 in Alg. 7.2). Regardless of the returned value, lower levels will also terminate immediately, as all levels need to meet the same time limit.

If a search is terminated before a global solution at the base level is found, it is still possible to return a partial solution. In fact, this is the usual behavior of real-time algorithms discussed in Section 6.7.1. Typically, real-time algorithms are designed to select one immediate action, the action that minimizes the cost to the goal (Eq. 6.19). Herein, it is proposed that the algorithm returns a local path ending in a state that does not have to be the immediate successor of a current state but can be selected among all states expanded by the local search (LSS: local search space):

$$s_{local\_goal} = \underset{s \in LSS}{\operatorname{argmin}} \; \underset{s' \in Succ(s)}{\operatorname{argmin}} \; (cost(s, s') + h(s')) \,. \tag{7.3}$$

A local-goal selection is conducted only at the base level. It is convenient to select the best local goal, s_{local_goal}, during a node expansion following a successor that is pushed to an open list (lines 23–26 in Alg. 7.2).

A state to be selected for a local goal needs to be safe (line 23 in Alg. 7.2). The notion of a safe state is domain and problem specific. Herein, in a safe-interval graph search, a *safe state* is defined as a state for which no collision with a moving obstacle is possible in the predictable future. As a local goal is selected, a local path can be reconstructed with the function GETPLAN from Algorithm 3.1, which is called from lines 8–9 in Algorithm 7.2.

7.4.2 Incremental Real-time Switchback

Real-time algorithms were originally intended to reduce the total mission time that includes both planning and acting times. In practice, this is hard to achieve. As discussed in Section 6.7.1,

Algorithm 7.2 Real-time Switchback algorithm; functions modified with respect to Algorithm 7.1, where i denotes the abstraction level and $i = 0$ is the base level.

1: **function** KEY(s, i) ... ▷ Functions same as in Alg. 7.1
2: **function** SOLUTIONFOUND($s_{query,i}$) ...
3: **function** HEURISTIC(s, i) ...
4: **function** INITIALIZE($s_{start,i}, s_{query,i}, i$) ...
5: **function** REAL-TIMESWITCHBACK(s_{start}, s_{goal})
6: **if** SEARCH($s_{start}, s_{goal}, 0$) $= success$ **then**
7: **return** GETPLAN(s_{goal})
8: **else if** $s_{local\_goal} \neq NULL$ **then**
9: **return** GETPLAN(s_{local_goal})
10: **else**
11: **return** failure
12: **function** SEARCHSTEP()
13: $s =$ TOP($open\_list_i$)
14: POP($open\_list_i$)
15: **if** $direction(i) = forward$ **then** $children = Succ_i(s)$
16: **if** $direction(i) = backward$ **then** $children = Pred_i(s)$
17: **for** all $s' \in children$ **do**
18: **if** NOT $visited(s')$ OR $g(s') > cost(s, s') + g(s)$ **then**
19: $parent(s') = s$
20: $g(s') = cost(s, s') + g(s)$
21: **if** NOT $visited(s')$ **then** $visited(s') = true$
22: PUSH($open\_list_i, s'$)
23: **if** $i = 0$ AND ISSAFE(s) AND $h(s') + cost(s, s') < \infty$ AND
24: $h(s') + cost(s, s') < h^+_{local\_goal}$ **then** ▷ A safe local goal
25: $s_{local\_goal} = parent(s')$ ▷ with lowest h-value is searched for.
26: $h^+_{local\_goal} = h(s') + cost(s, s')$
27: **function** SEARCH($s_{start,i}, s_{query,i}, i$)
28: **if** NOT $initialized(i)$ **then** INITIALIZE($s_{start,i}, s_{query,i}, i$)
29: **if** $i = 0$ **then** ▷ Initialization of a local goal;
30: $s_{local\_goal} = NULL$ ▷ only at the base level.
31: $h^+_{local\_goal} = \infty$
32: **while** NOT EMPTY($open\_list_i$) AND TOPKEY($open\_list_i$) $\neq \infty$ **do**
33: **if** SOLUTIONFOUND($s_{query,i}$) **then**
34: **return** $success$
35: SEARCHSTEP()
36: **if** timeout **then**
37: **return** $failure$ ▷ A time limit is checked at each search level.
38: **return** $failure$

real-time algorithms suffer from scrubbing behavior [102]; thus, they do not provide the expected increase in speed if an environment is fully known. However, in unknown or partially known environments, real-time algorithms can be used in the same manner as incremental search algorithms [70].

The function MAIN presented in Algorithm 7.3 allows the use of Real-time Switchback for incremental planning. This function is very similar to that of the LRTA* (Alg. 6.1), but with the functions MAPUPDATE and REINITIALIZE added. In the incremental version of the

Real-time Switchback, changes in the environment detected by MAPUPDATE are used in the reinitialization of algorithms at each level (function REINITIALIZE in Algorithm 7.3).

Algorithm 7.3 Real-time Switchback main function.

```
 1: function MAIN()
 2:     MAPUPDATE()
 3:     while s_start ≠ s_goal do
 4:         Π = REAL-TIMESWITCHBACK(s_start, s_goal)        ▷ Real-time Switchback returns local path.
 5:         if Π = ∅ then
 6:             return failure
 7:         s_start =ACTIONSELECTION(s_start, Π)
 8:         MAPUPDATE()
 9:         H-VALUEUPDATE()                                 ▷ Not necessary in time-dependent planning.
10:         REINITIALIZE(s_start, s_goal)
11: function REINITIALIZE(s_start, s_goal)
12:     for each level i starting from top do
13:         if i-level algorithm is incremental then        ▷ E.g., D* Extra Lite, or AD*-Cut
14:             REINITIALIZE(φ_i(s_start), φ_i(s_goal), i)
15:         else
16:             RESET(i)                                    ▷ Resetting algorithm to run new search from scratch.
```

At this point, it is important to note that Switchback can be viewed as a meta-algorithm or framework that is able to compose algorithms of distinct classes. In this chapter, the following algorithm compositions are tested: local A* (level 0) with D* Extra Lite (level 1), and local A* (level 0) with AD*-Cut (level 1). In the final system for mobile robot motion planning in a dynamic environment (Chapter 8), a three-level hierarchical algorithm consisting of local A* (level 0) with AD*-Cut (level 1) and A* (level 2) algorithms is used.

H-value Update for Hierarchical Planning in a Dynamic Environment

The improvement of the h value (i.e., function H-VALUEUPDATE in line 4 in Alg. 6.1 and line 9 in Alg. 7.3) is necessary to guarantee the completeness of real-time algorithms. As real-time algorithms compute a local solution, a robot may get stuck in a local minimum of a heuristic function. The continual improvement of h values allows an escape from a minimum. However, if a heuristic function does not include the local minima, a problem of real-time planning trivializes to the problem of following the h-value gradient descent; then, improvement of the h value is not necessary. As the improvement of the h value requires additional computation, such a situation is desirable. It can be shown that, with additional assumptions, Real-time Switchback used for time-dependent planning does not require improvement of the h value. Hence, the following is proposed.

Proposition 7.2 *Let the state-time space $\mathcal{ST}$ be an original (base) search space, and S, representing a static environment without moving obstacles, such that $\forall_{(s,t)\in\mathcal{ST}} s \in S$ is an*

abstract search space. If Real-time Switchback is given enough time to find a safe local goal and all moving obstacles eventually disappear, then the Real-time Switchback without the H-VALUEUPDATE *function is complete; that is, a robot will eventually achieve the goal state.*

The proof for Proposition 7.2 is based on a simple observation that the most reliable strategy for navigation among moving obstacles is to find a safe place to hide and wait until all moving obstacles are gone, and then to follow a path to the goal.

Proof 7.2 *If a time reserved for a single run of the Real-time Switchback (line 4 in Alg. 7.3) is long enough to find a safe local goal (line 23 in Alg. 7.2), then the algorithm will provide a collision-free local path. From this, it also stands that a robot at each iteration of the main loop (function* MAIN *in Alg. 7.3) will avoid collision with moving obstacles. Hence, the search space is safely explorable (i.e., it does not include terminal states or those with no escape), which is a basic requirement for the use of real-time algorithms.*

Now, it must be shown that a heuristic function provided to the base level (level 0) does not have permanent local minima. In a Real-time Switchback, a heuristic calculated at a higher level (level 1) is an outcome of a global search starting from the goal state. With respect to Proposition 7.2, an abstract search space represents the environment, omitting moving obstacles. Therefore, if all moving obstacles are gone, such a heuristic is ideal (i.e., at each state, it provides the true cost to the goal); hence, it has only one minimum at the goal state. □

Although the requirements stated in Proposition 7.2 seem to be strong, in practice, they are possible to fulfill. For example, the parameters of local-search time limit, search-space resolution, maximal obstacle velocity, and computational power of an onboard computer can all be empirically adjusted to follow requirements with a reasonable probability of success. Otherwise, full knowledge of future events and global planning are required, which is a non-realistic assumption.

7.4.3 Experimental Results

In this section, experimental results of motion planning among moving obstacles using Real-time Switchback are presented. The experiments were conducted within the same settings as described in Sections 6.8 and 7.3, for two versions of a Real-time Switchback, namely, local A* with D* Extra Lite (RT-LA*D*EL, in short) and local A* with AD*-Cut (RT-LA*AD*-Cut, for short), compared to regular Switchback consisting of A* with D* Extra Lite (A*D*EL, in short). As RT-LA*AD*-Cut is an anytime algorithm, it requires the initial ϵ_{init} and ϵ_{step}, which were set to 2 and 0.1, respectively.

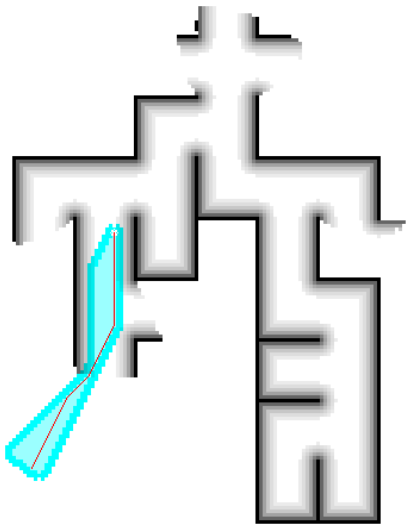


Figure 7.7: Local search space expanded by level 0 of Real-time Switchback (the blue region with dimmed border) and the local path (thin line).

In the presented tests, the allocated time for real-time search was set to 0.01 s, in which it was not always possible to find a safe local goal. If such a situation took place, the current problem was interrupted with failure; therefore, in addition to typical search parameters, the percent of successfully solved problems was recorded.

In Table 7.2, the results for the two tested algorithms are presented. Additionally, Table 7.2 includes the results of an optimal global search with A* for reference (i.e., the environment was fully-known). In the context of a real-time incremental search, the ratio of successful runs, search time per re-planning episode (measured at line 4 in Alg. 7.3), and traveled path cost are the most interesting factors. In addition, the average number of search steps per re-planning episode is presented.

Both tested algorithms were able to solve over 70% of problems with 10 (and less) moving obstacles within the limit of 0.01 s. For higher numbers of moving obstacles, the given time limit is clearly to small, which can be observed in Figures 7.8a, 7.9a, and 7.10a, in which the success rate is presented. It is noteworthy that even LA*D*EL, which always performs global (complete) planning, was not able to solve all problems, which is characteristic for planning without full knowledge; a robot may encounter an unknown dead-end in which it is impossible to avoid collision with a moving obstacle.

As RT-LA*AD*-Cut provides an inadmissible heuristic function, in most tests, a robot controlled by RT-LA*AD*-Cut traveled longer paths. It is interesting that in some cases (e.g., planning with one moving obstacle in the *rooms* map set in Tab. 7.2) a robot

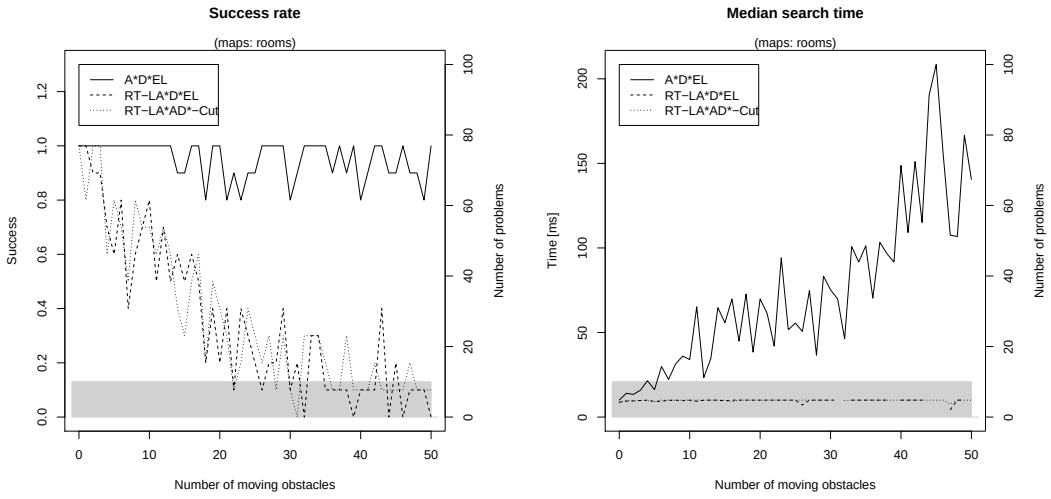
Table 7.2: Experimental results for the search in a safe interval graph with Real-time Switchback in an allocated search time of 0.01 s. The presented values are calculated after solving 10 problems for each number of moving obstacles and each map set.

#Mov. Obst.	Algorithm	Success rate %	Search Time per Re-plan. [ms]		#Search Steps per Re-plan.	Total Time [s]	Path Cost
			Median	Max.			
rooms							
1	A*	100	761.24	761.24	98425	0.76	715.963
	A*D*EL	100	14.05	78.35	2576	11.19	893.401
	RT-LA*D*EL	100	9.36	95.61	944	5.01	883.895
	RT-LA*AD*-Cut	80	10.00	32.34	754	4.60	887.567
10	A*	100	1182.32	1182.32	87090	1.18	682.900
	A*D*EL	100	34.02	249.58	3991	28.73	912.086
	RT-LA*D*EL	80	10.00	106.46	628	6.64	994.764
	RT-LA*AD*-Cut	70	10.00	28.90	582	6.70	1006.529
50	A*	100	4351.72	4351.72	150548	4.35	795.086
	A*D*EL	100	140.50	1045.95	7444	157.65	1045.531
	RT-LA*D*EL	0	NA	NA	NA	NA	NA
	RT-LA*AD*-Cut	10	10.01	20.47	240	6.84	861.136
wc3							
1	A*	100	271.90	0.43	36565	0.27	471.248
	A*D*EL	100	10.12	79.52	2596	9.85	702.330
	RT-LA*D*EL	90	7.60	78.66	851	4.57	689.384
	RT-LA*AD*-Cut	90	8.45	36.23	753	4.23	709.206
10	A*	100	805.92	0.46	66778	0.81	534.240
	A*D*EL	100	34.73	351.14	5506	42.82	825.297
	RT-LA*D*EL	70	8.72	80.59	632	7.33	774.569
	RT-LA*AD*-Cut	70	8.75	33.04	599	8.27	965.616
50	A*	100	2671.00	0.45	83707	2.67	590.485
	A*D*EL	100	181.12	1584.75	10569	267.53	985.984
	RT-LA*D*EL	22	10.02	74.68	277	15.29	1177.815
	RT-LA*AD*-Cut	11	10.01	29.29	339	14.68	1137.998
mazes_16							
1	A*	100	224.88	0.39	36876	0.23	542.923
	A*D*EL	80	7.71	36.88	1395	12.15	989.813
	RT-LA*D*EL	80	6.37	90.42	785	7.87	992.371
	RT-LA*AD*-Cut	90	6.65	58.75	753	6.85	990.196
10	A*	100	430.42	0.49	38006	0.43	591.769
	A*D*EL	90	19.30	116.32	2997	54.63	1228.366
	RT-LA*D*EL	80	8.26	51.04	771	10.81	1193.244
	RT-LA*AD*-Cut	90	9.27	35.60	755	11.96	1350.433
50	A*	100	1360.54	0.40	66472	1.36	927.526
	A*D*EL	100	143.90	711.21	6873	363.51	1827.447
	RT-LA*D*EL	0	NA	NA	NA	NA	NA
	RT-LA*AD*-Cut	0	NA	NA	NA	NA	NA
Total Performance Ratio							
A*D*EL / RT-LA*D*EL		2.81	8.18	7.12	8.67	12.40	1.07
A*D*EL / RT-LA*AD*-Cut		2.92	8.08	16.55	9.36	12.29	1.00

controlled by LA*D*EL traveled longer paths than the robots controlled by RT-LA*D*EL and RT-LA*AD*-Cut. This may suggest that global planning in a dynamic environment without full knowledge may suffer a behavior similar to the "scrubbing" discussed in Section 6.7.1, that is, a robot explores all areas that may contain the shortest path. In contrast, in Real-time Switchback a local path is greedily selected upon a heuristic, which seems to be a better strategy in environments in which multiple paths of similar cost exist.

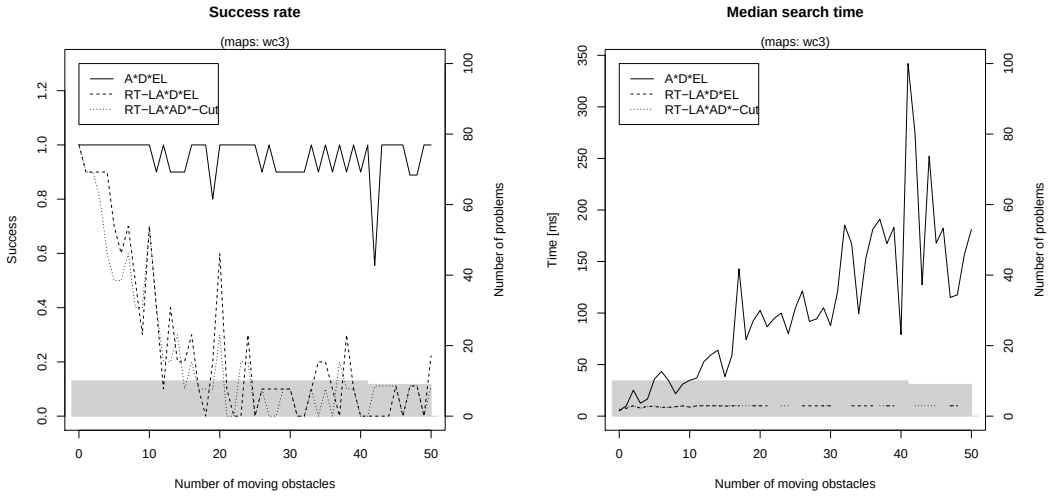
The median of the search time per single re-planning episode for RT-LA*D*EL and RT-LA*AD*-Cut is close to the granted time limit. At this point, it should be noted that the implementations of the tested algorithms are not true real-time. This is reflected by the maximum search time per single re-planning episode included in Table 7.2. (To obtained a true real-time, a number of implementation details need to be considered, such as a choice of a data structure for an open-list maintenance, a memory allocation scheme, and a modification of a reinitialization in incremental algorithms.) However, in practice, the simple implementation of Real-time Switchback is sufficient to provide near real-time performance, in contrast to A*D*EL, for which a search time grows linearly with the number of moving obstacles (Figs. 7.8b, 7.9b, 7.10b).

To conclude, the results of this benchmark support the claim that Real-time Switchback provides a good trade-off between optimality (reflected by the traveled path cost) and computation time (reflected by the search time per single re-planning episode) when used for planning in a dynamic environment.



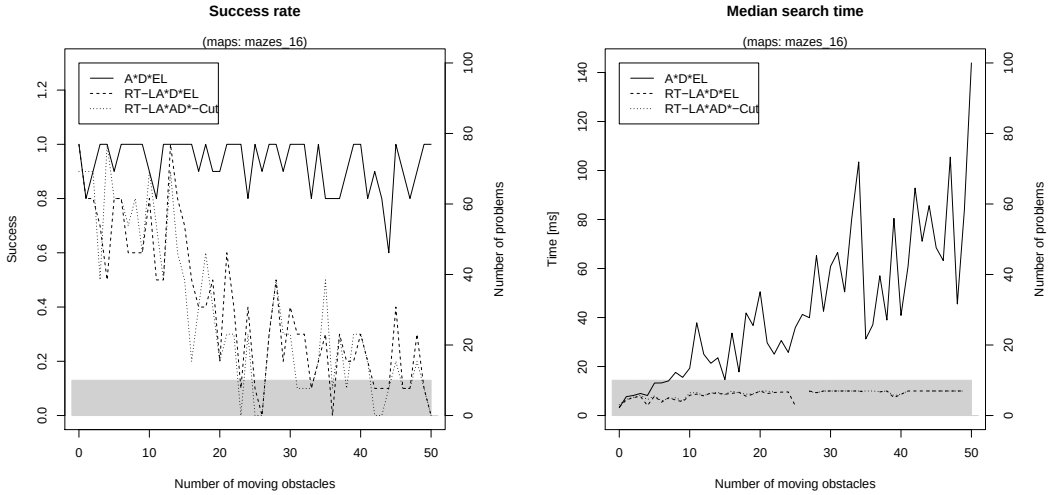
a) b)

Figure 7.8: Results of hierarchical real-time search amid moving obstacles, for the *rooms* map set; in the background, the histogram of problems is plotted in gray.



a) b)

Figure 7.9: Results of hierarchical real-time search amid moving obstacles, for the *wc3* map set; in the background, the histogram of problems is plotted in gray.



a) b)

Figure 7.10: Results of hierarchical real-time search amid moving obstacles, for the *mazes_16* map set; in the background, the histogram of problems is plotted in gray.

7.5 Conclusions

In this chapter, hierarchical search algorithms were investigated against usability for motion planning among moving obstacles. Specifically, the alternating search-direction technique utilized by the Switchback algorithm [16] was implemented and tested. In Section 7.2, an abstraction hierarchy for an optimal hierarchical search in a state-time space was proposed. The experimental results of planning with the proposed abstraction hierarchy supports a proposition

that a search in a search space without moving obstacles provides an admissible heuristic for a search in a state-time space. The proposed approach to hierarchical planning in a dynamic environment is then extended to a real-time incremental search, for which a new Real-time Switchback algorithm is proposed. As Switchback and Real-time Switchback can be viewed as a general framework for an abstraction-based hierarchical search, it was possible to obtain two variants of the Real-time Switchback for a real-time incremental search, namely, the combination of local A* with D* Extra Lite and a combination of local A* with AD*-Cut. Both algorithms were tested, reaching the conclusion that both can be used for real-time motion planning among moving obstacles in an unknown or partially known environment.

8. Hierarchical Motion Planning in a Dynamic Environment for a Mobile Robot

In this chapter, the algorithms and methods presented in Chapters 4 through 7 are applied to a practical problem of mobile robot motion planning in a dynamic environment. Specifically, motion planning for differential-drive robots (which have a simple mechanical construction) working in dynamic indoor environments is considered. The testing setup is developed in the Robot Operating System (ROS) [118], the software framework that provides the inter-process communication mechanisms, robotic simulators (Stage and Gazebo), drivers for many devices used in robotics, and the typical algorithms necessary for robot navigation. The tests were conducted in the dynamic scenario developed in the Stage simulator.

In Section 8.2, the developed system overview is presented. The test scenario and experiments is described in Section 8.3.

8.1 Differential-drive Mobile Robot Model

Typically, a differential-drive robot has two coaxial drive wheels and a single castor wheel (Fig. 8.1). The difference between the left wheel velocity, v_l, and the right wheel velocity, v_r, results in the motion of the robot about an instantaneous center of curvature point, ICC, with an angular velocity, ω. In addition, ICC is always on the common drive-wheel rotation axis. Another characteristic point is P, a point that is also on the common drive-wheel rotation axis in the middle between the drive wheels. The distance between P and ICC is denoted by R. If the wheels rotate with the same rotational velocity, but in the opposite directions ($v_l = -v_r$), then ICC is equal to P, $R = 0$, and the robot is turning in place. If $v_l = v_r$, the robot is moving along a straight line; hence, R is infinite [119].

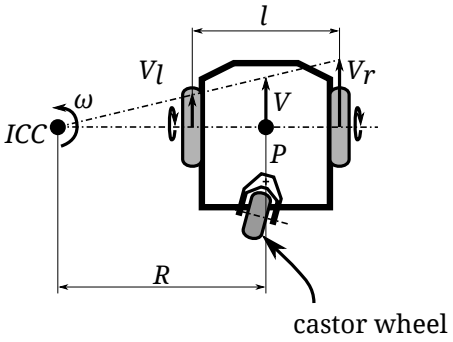


Figure 8.1: Differential-drive robot with two drive wheels and a single castor wheel, where v_l, v_r are left and right wheel velocities, respectively, l denotes the wheel base, R denotes the rotation radius, ICC, is the instantaneous center of curvature, P denotes the point in the middle of the common drive-wheel axis, v denotes the velocity at P, and ω denotes the angular velocity about ICC.

As the wheel velocities, v_l, v_r, and the wheel base, l, are known, R, ω, and v can be calculated as follows:

$$R = \frac{l}{2}\frac{v_l + v_r}{v_r - v_l}, \tag{8.1}$$

$$\omega = \frac{v_r - v_l}{l}, \tag{8.2}$$

$$v = \frac{v_l + v_r}{2}, \tag{8.3}$$

where, naturally:

$$v = \omega R. \tag{8.4}$$

A longitudinal velocity, v, is always perpendicular to the drive-wheel axle. Therefore, it is useful to attach the robot local coordination system, $\{X_R, Y_R\}$, at P with the y-axis pointing to the left wheel (Fig. 8.2). As the robot cannot move in the Y_R-axis direction, its motion in a local frame can represented as a vector $[v \quad 0 \quad \omega]^T$.

The motion of a differential-drive robot in a world frame (Fig. 8.2), $\{X_W, Y_W\}$, is described by:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}_W = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ 0 \\ \omega \end{bmatrix}_R = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ \omega \end{bmatrix}. \tag{8.5}$$

From Eq. 8.5, we have that $\dot{x} = v\cos\theta$ and $\dot{y} = v\sin\theta$; hence:

$$-\dot{x}\sin\theta + \dot{y}\cos\theta = 0, \tag{8.6}$$

which describes nonholonomic constraints.

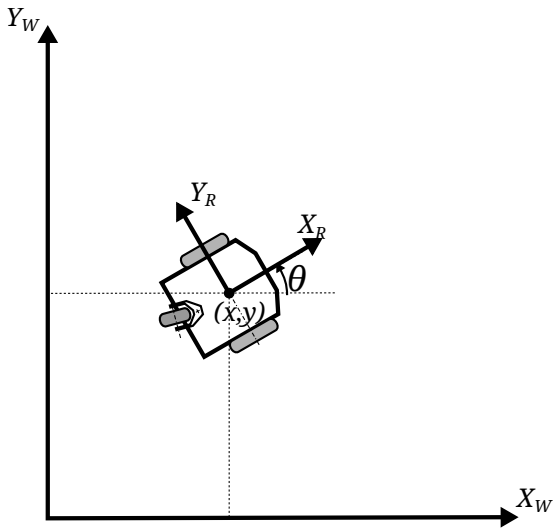


Figure 8.2: Differential-drive robot in a global coordination frame.

The forward kinematics of a differential-drive robot is described by the following equation: [119]:

$$\begin{aligned} x(t+\Delta t) =& x(t) + R(\sin(\theta + \omega\Delta t) - \sin(\theta)) \\ y(t+\Delta t) =& x(t) - R(\cos(\theta + \omega\Delta t) - \cos(\theta)) \\ \theta(t+\Delta t) =& \theta + \omega\Delta t, \end{aligned} \tag{8.7}$$

if $\omega \neq 0$ ($|R| < \infty$), and:

$$\begin{aligned} x(t+\Delta t) =& x(t) + v\Delta t \sin(\theta) \\ y(t+\Delta t) =& x(t) + v\Delta t \cos(\theta) \\ \theta(t+\Delta t) =& \theta, \end{aligned} \tag{8.8}$$

otherwise.

In this chapter, the longitudinal velocity, v, and rotational velocity, ω, will be used, rather than the left and right wheel velocities. If state lattices are used for motion planning, R is imposed by a local curvature of a motion primitive (an arc in a state lattice). Hence, rotational velocity can be calculated from Eq. 8.4, $\omega = \frac{v}{R}$. Therefore, action-event synchronization, applied to avoid a collision with a moving obstacle, merely focuses on adjusting the longitudinal

velocity. Then, Eq. 8.7 can take the form of the following:

$$\begin{aligned} x(t+\Delta t) =& x(t) + R(\sin(\theta + \frac{v}{R}\Delta t) - \sin(\theta)) \\ y(t+\Delta t) =& x(t) - R(\cos(\theta + \frac{v}{R}\Delta t) - \cos(\theta)) \\ \theta(t+\Delta t) =& \theta + \frac{v}{R}\Delta t, \end{aligned} \tag{8.9}$$

if $0 < |R| < \infty$. (If a state lattice includes arcs that represent turning in place, the angular velocity needs to be adjusted.)

8.2 System Overview

The overview diagram presenting the tested robot motion planning system is shown in Figure 8.3. The proposed system consists of the three planning algorithms that form the hierarchical planning algorithm based on the Real-time Switchback algorithm (Sec. 7.4). The two topmost levels (i.e., the 2D global grid search (GGS) and the global anytime incremental state-lattice search (GSLS)), provide heuristics to the consecutive lower levels. Only the lowest level, the local real-time state-lattice search (RTSLS), calculates a trajectory that is sent to the trajectory-following algorithm.

As the system is based on the Real-time Switchback that uses the alternating search-direction technique (Sec. 7.1), search directions are switched at the consecutive search hierarchy levels. These directions are imposed by the lowest level; as RTSLS is a local search in a safe interval graph, it must be conducted forward.

All three levels require a problem definition (i.e., the robot's current state, a goal position, and a global cost map representing the static obstacles). As RTSLS performs planning among moving obstacles, it requires additional information about trajectories of detected moving obstacles that are further transformed into safe intervals collected in a 2D look-up table. The global cost map is obtained from a map previously built by the SLAM algorithm. This cost map contains only static obstacles and is updated as new range data are collected. It is important to pass through to the global cost-map updating process only those distance measurements that represent static obstacles. Otherwise, moving obstacles would be remembered as static ones.

In the entire system, it is assumed that all calculations are done in the global map frame. This is provided by the localization process that, upon actual rangefinder and encoder readings, can localize a robot in the global map.

In the remainder of this section, the three planning levels are discussed in detail.

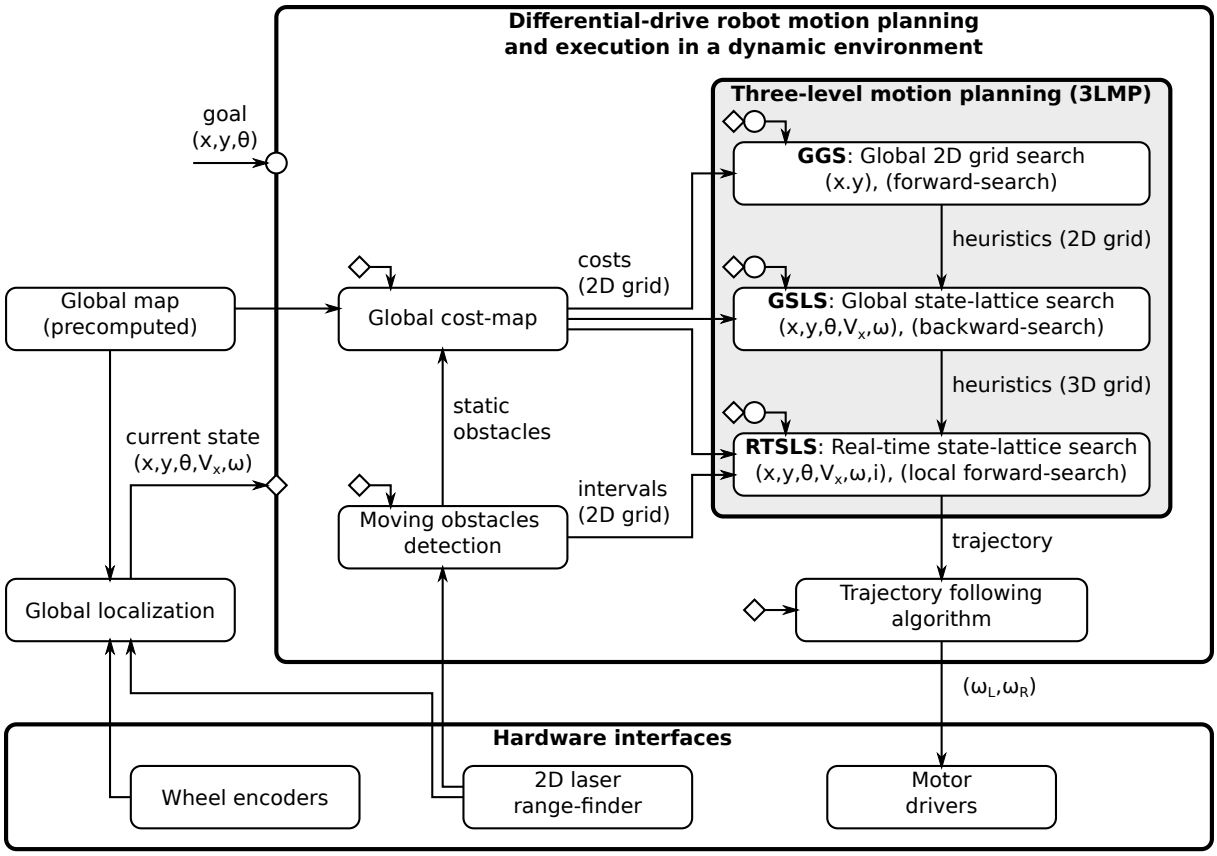


Figure 8.3: Overview of the tested robot motion planning system.

8.2.1 Two-dimensional Global Grid Search

The GGS is performed as a forward-A* search on a 16-connected grid (Fig. 3.5, Sec. 3.1.3) as proposed in [11] and later used in [18]. To use distances calculated by a 2D grid search as heuristics for minimum-time planning, they must correspond to the travel time; therefore, the action cost in the 2D grid search is calculated as follows:

$$cost(s_1, s_2) = \frac{\|s_2 - s_1\| \cdot \delta}{cm(s_1) \cdot v_{max}}. \tag{8.10}$$

A cost calculated with respect to Eq. 8.10 clearly underestimates the true cost; hence, such a search provides admissible heuristics for a state-lattice search.

8.2.2 Global State-lattice Search

The quality of a heuristic cost estimation has a great effect on the size of an explored search space. Therefore, the GSLS aims to provide such a heuristic to the real-time local search that reflects the kinematic constraints of the robot. In GSLS, a backward search in a 3D

state lattice is performed, $s = (x, y, \theta)$, where (x, y, θ) is the robot pose on a plane. As the search algorithm uses a graph, mapping from a state lattice to a graph must be defined, that is, $\phi_{GSLS}(s) = (x, y, \theta)$. The (x, y) values are simply chosen among the vertices of a regular grid that has a δ resolution, and θ takes the multiplicity of $2\pi/16$; hence, mapping these values to G is straightforward. Motion primitives were generated using a script provided with SBPL [51] that implements the optimization method described in [49].

Due to the dimensionality of the used state lattice and its high number of states, in the GSLS, AD*-Cut algorithm, an anytime version of D* Extra Lite, is used. In the experiments, the time reserved for GSLS was set to 2 seconds.

8.2.3 Local Real-time State-lattice Search in a Dynamic Environment

The RTSLS is realized as a local forward-A* in a 6D state-time space, $s = (x, y, \theta, v, \omega, t)$, where (x, y, θ) is the robot pose on a plane, v is the longitudinal velocity, ω is the angular velocity, and t is the time. A 6D state is mapped to a safe interval graph, G, with the mapping function, $\phi_{RTSLS}(\tilde{s}) = (x, y, \theta, \{-1, 0, 1\}, \{-1, 0, 1\}, id)$, where $id \in \mathbb{Z}$ (integer number) identifies a safe interval.

Velocities v and ω are mapped to the values from the set $\{-1, 0, 1\}$ that correspond to negative, zero, and positive values of a given velocity variable. In other words, in the search graph, only nine general situations (going forward, going backward, left turn in place, right turn in place, etc.) are recognized as separated nodes.

In addition to the aforementioned graph node labels, an actual state (with real values) is memorized as a $\tilde{s}_{min}$ label for each state mapped to a graph node (as in Sec. 6.5). Therefore, two states that represent a robot at the same position with different velocities but the same sign (positive, negative, or zero) will be mapped to the same search graph node. As the minimum-time path is sought, only a state that minimizes node cost needs to be memorized.

As proposed in Section 6.6, motion primitives used in RTSLS are generated in two steps, that is, non-temporal motion primitive (actions) generation is followed by an action-event synchronization generating temporal motion primitives.

Non-temporal Motion Primitive Generation

The RTSLS uses the same basic motion primitives as GSLS; however, from one basic motion primitive, a few motion primitives are generated by the application of the maximal velocities. Additionally, states at which the robot is stopped are distinguished as the special states at which it is possible to switch between left turn in place, right turn in place, going forward, and going backward. (A motion type cannot change within a single motion primitive.) Thus, from the

basic action $a = \langle (x_1, y_1, \theta_1), (x_2, y_2, \theta_2) \rangle$, the following actions are obtained, (as in the case of forward (backward) movements, the angular velocity depends on the curvature of the motion primitive, it is omitted in the state description and is replaced with $\cdot$):

- if a is a forward movement:

$$\begin{aligned} &\langle (x_1, y_1, \theta_1, 0, 0), (x_2, y_2, \theta_2, \alpha_2 v^+_{max}, \cdot) \rangle, \\ &\langle (x_1, y_1, \theta_1, \alpha_1 v^+_{max}, \cdot), (x_2, y_2, \theta_2, \alpha_2 v^+_{max}, \cdot) \rangle, \\ &\langle (x_1, y_1, \theta_1, \alpha_1 v^+_{max}, \cdot), (x_2, y_2, \theta_2, 0, 0) \rangle, \end{aligned} \tag{8.11}$$

- if a is a backward movement:

$$\begin{aligned} &\langle (x_1, y_1, \theta_1, 0, 0), (x_2, y_2, \theta_2, \alpha_2 v^-_{max}, \cdot) \rangle, \\ &\langle (x_1, y_1, \theta_1, \alpha_1 v^-_{max}, \cdot), (x_2, y_2, \theta_2, \alpha_2 v^-_{max}, \cdot) \rangle, \\ &\langle (x_1, y_1, \theta_1, \alpha_1 v^-_{max}, \cdot), (x_2, y_2, \theta_2, 0, 0) \rangle, \end{aligned} \tag{8.12}$$

- if a is a left turn in place:

$$\begin{aligned} &\langle (x_1, y_1, \theta_1, 0, 0), (x_2, y_2, \theta_2, 0, \alpha_2 \omega^+_{max}) \rangle, \\ &\langle (x_1, y_1, \theta_1, 0, \alpha_1 \omega^+_{max}), (x_2, y_2, \theta_2, 0, \alpha_2 \omega^+_{max}) \rangle, \\ &\langle (x_1, y_1, \theta_1, 0, \alpha_1 \omega^+_{max}), (x_2, y_2, \theta_2, 0, 0) \rangle, \end{aligned} \tag{8.13}$$

- if a is a right turn in place:

$$\begin{aligned} &\langle (x_1, y_1, \theta_1, 0, 0), (x_2, y_2, \theta_2, 0, \alpha_2 \omega^-_{max}) \rangle, \\ &\langle (x_1, y_1, \theta_1, 0, \alpha_1 \omega^-_{max}), (x_2, y_2, \theta_2, 0, \alpha_2 \omega^-_{max}) \rangle, \\ &\langle (x_1, y_1, \theta_1, 0, \alpha_1 \omega^-_{max}), (x_2, y_2, \theta_2, 0, 0) \rangle, \end{aligned} \tag{8.14}$$

- for all types of movements (a single step):

$$\langle (x_1, y_1, \theta_1, 0, 0), (x_2, y_2, \theta_2, 0, 0) \rangle, \tag{8.15}$$

where $\alpha_1 = cm(x_1, y_1)$, $\alpha_2 = cm(x_2, y_2)$ are cost-map factors.

An action cost is defined as a time of travel along a motion primitive. The motion primitives generated by SBPL [51] are represented as point sets, $\{p_1, \ldots, p_n\} = \{(x_1, y_1, \theta_1), \ldots, (x_n, y_n, \theta_n)\}$, with a fixed number of intermediate poses, n. Therefore, the travel time at the maximal possible velocity, $v_{p,max} \in \{v^+_{max}, v^-_{max}\}$ or $v_{p,max} \in \{\omega^+_{max}, \omega^-_{max}\}$

(depending on the movement type), is calculated as follows:

$$cost_{v_{p,max}}(a) = \sum_{k=1}^{n-1} \frac{2\|p_{k+1} - p_k\|}{cm(p_k)v_{p,max} + cm(p_{k+1})v_{p,max}}, \tag{8.16}$$

which is the sum of travel instances between consecutive points, considering the velocity limits at these points. For the actions of $\langle (p_1, 0), (p_2, cm(p_2)v_{p,max}) \rangle$ and $\langle (p_1, cm(p_1)v_{p,max}), (p_2, 0) \rangle$ types, which are actions representing accelerating from zero velocity and stopping from the maximal velocity, the cost is calculated as follows:

$$cost(a) = \max \left(cost_{v_{p,max}}(a), \sqrt{\frac{2distance(a)}{a_{p,max}}} \right). \tag{8.17}$$

Finally, for step-like actions (of $\langle (p_1, 0), (p_2, 0) \rangle$ type), the cost is calculated as follows:

$$cost(a) = \max \left(cost_{v_{p,max}}(a), \sqrt{\frac{2\frac{distance(a)}{2}}{a_{p,max}}} + \sqrt{\frac{2\frac{distance(a)}{2}}{a_{p,max}}} \right). \tag{8.18}$$

In Eqs. 8.17 and 8.18, $a_{p,max}$ denotes the maximal acceleration (deceleration), such that, to some extent, acceleration (deceleration) limits are considered.

Temporal Safe Interval Motion Primitive Generation

Temporal motion primitives are generated by a simple action-event synchronization (Sec. 6.6.1) based on non-temporal primitives. Furthermore, it is assumed that a robot and the moving obstacles are disk shaped, such that a disk circumscribes an object. Therefore, action-event synchronization is not required for turn-in-place movements.

8.2.4 Partial Motion Planning Using a Safe Interval Graph

In [35], Petti and Fraichard proposed partial motion planning (PMP), a scheme for safe motion planning in dynamic environments. In this scheme, local motion planning is performed in cycles of a fixed duration, t_c (Fig. 8.4). At the beginning of an i-th cycle, t_i, new observations are acquired, and a prediction of moving-obstacle motion is made. Then, a local search is performed until $t_{i+1} = t_i + t_c$. The local search that starts at t_i is rooted at $\tilde{s}_{i+1} = (s_{i+1}, t_{i+1})$, where $\tilde{s}_{i+1}$ can be predicted using the plan from the previous search. A plan returned by the local search is valid for a time interval, $[t_{i+1}, t_{i+1} + t_{v_i}]$, where the validity time, t_{v_i}, should be greater than the cycle duration, t_c.

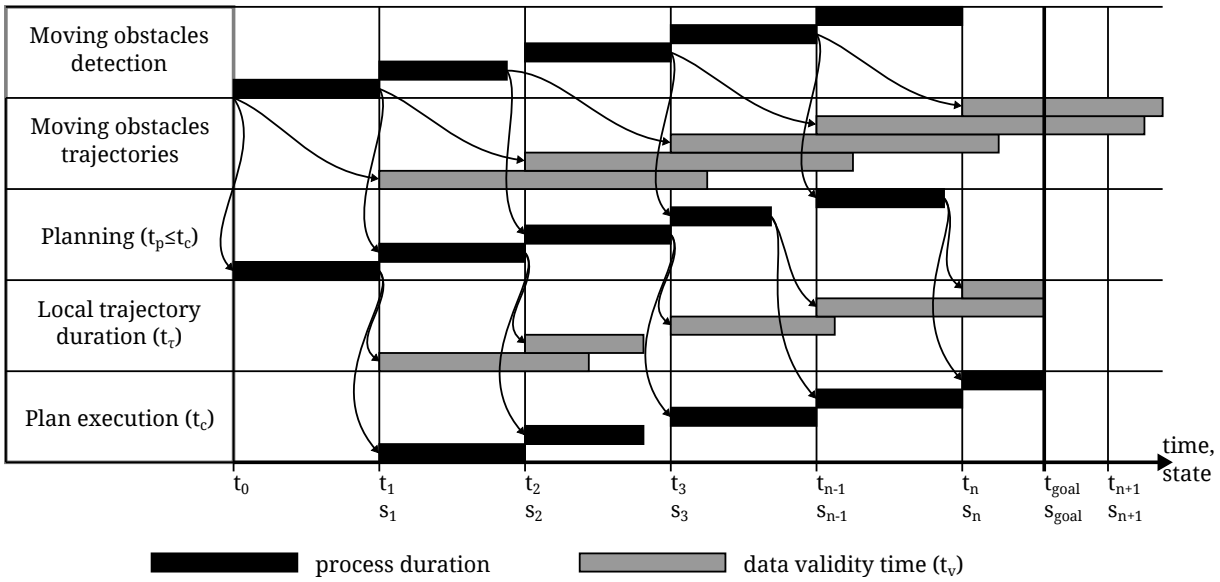


Figure 8.4: Partial motion planning scheme (adapted from [35]).

As stated in [35], a collision-free local trajectory cannot include inevitable collision states (ICS) (i.e., states from which every future trajectory leads to a collision (Sec. 6.1)). The computation of all future trajectories to determine whether a state is an ICS is intractable. Therefore, in [35], it is proposed to determine whether a collision-free braking trajectory for each state along a local plan (generated by an application of the maximum deceleration) exists that does not end in an ICS. Although such an approach provides maximal safety, it is restrictive. Let us consider the example from Figure 8.5. If $\mathcal{ST}_{obs}$ is very wide (i.e., $\mathcal{ST}_{obs}$ projected on the s axis has d length) such that d is greater than the distance necessary for the robot to stop, $d > \frac{v_{max}^2}{2 \cdot a_{min}}$, there is no non-ICS trajectory that leads from state s_1 to s_2.

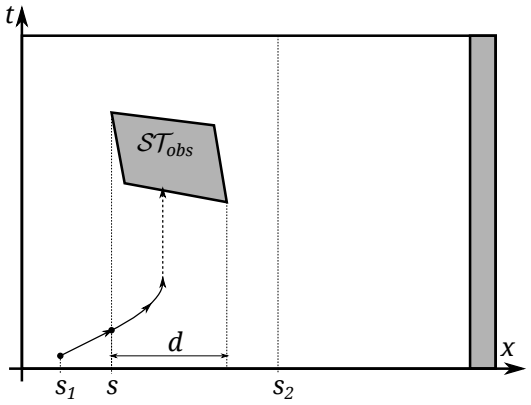


Figure 8.5: Example of an instance in which checking the trajectory of inevitable collision states, proposed in [35], is too restrictive.

Therefore, herein, it is proposed that, if a state at the end of the local trajectory is not an ICS, then the trajectory is collision-free. Furthermore, in RTSLS, states with 0 velocity are represented in a safe interval graph; thus, searching for a collision-free braking trajectory does not have to be performed for each state in an RTSLS search tree. It is sufficient to select (during a local search) a safe state s, as proposed in a Real-time Switchback (Sec. 7.4), that is, a state at which the robot is stopped and there is no future event, $e = (s, t_e)$, such that $e \in \mathcal{E}_{obs}$ (the state is not on a predicted moving obstacle path). This requirement can also be loosened such that a safe state is a state for which there is no collision in a given time horizon.

As a final remark, the duration of the local trajectory, t_{τ_i}, is not the same as the validity time, t_{v_i}, and does not need to be greater than the cycle duration, t_c. If a trajectory duration without a time necessary for stopping is greater than the cycle time (i.e., $t_{\tau_i} - \frac{v_{max}}{a_{min}} \geq t_c$), the overall motion will be smoother; otherwise, a robot will be stopping and waiting for a new plan. A local trajectory validity time, t_{v_i}, reflects an obstacle motion prediction horizon. Optionally, t_{v_i} can be calculated as $t_{v_i} = t_e - t_{i+1}$, where t_e is the earliest collision time (with a moving obstacle) at the end of the local trajectory.

8.2.5 Obstacle Motion Detection

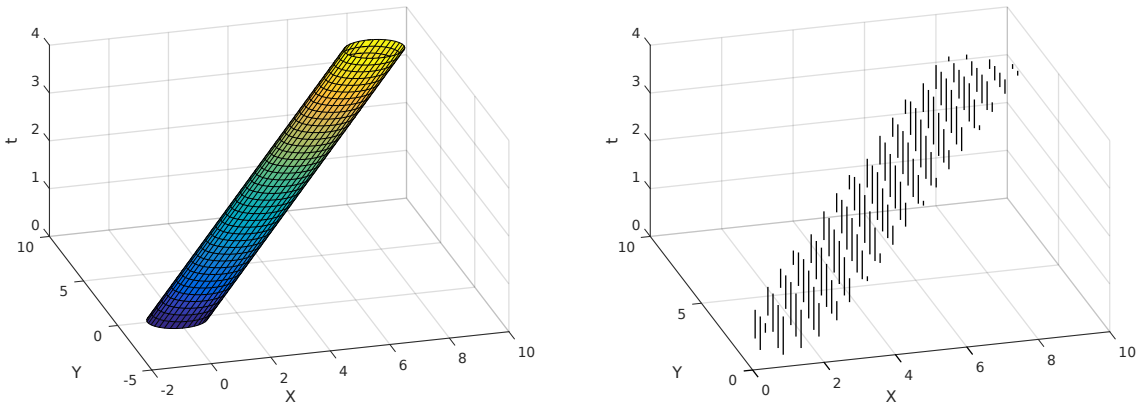
In the presented system, it is assumed that moving obstacles are constantly detected and their future trajectories and velocities can be predicted for a finite time horizon, namely, the data validity time, t_v (Fig. 8.4).

In all tests presented in this thesis, obstacle radii were constant. Then, in the 3D (x, y, t) space, an obstacle leaves a cylindrical trace (Fig. 8.6a). Herein, planning is performed in a state lattice, therefore, moving obstacles are observed only at state lattice vertices, which, in Figure 8.6b, is represented by collision intervals. As safe intervals complement collision intervals, the calculation of safe intervals is straightforward.

8.2.6 Trajectory Tracking Algorithm

A path provided by a state-lattice motion planning algorithm may contain discontinuities at state-lattice vertices [18], especially in angular velocity. Furthermore, real robotic systems suffer from inaccuracies. Therefore, velocities calculated from a trajectory should not be used directly as control inputs. In the system described in this chapter, control inputs (v, ω) are computed by the trajectory tracking algorithm of the following form [120]:

$$\begin{aligned} v_{new} =& v_t \cos(\theta_t - \theta) + k_1\big((x_t - x)\cos(\theta) + (y_t - y)\sin(\theta)\big) \\ \omega_{new} =& \omega_t + k_2 \mathrm{sgn}(v_t)\big(-(x_t - x)\sin(\theta) + (y_t - y)\cos(\theta)\big) + k_3(\theta_t - \theta), \end{aligned} \tag{8.19}$$



a) b)

Figure 8.6: Trace in a state-time space left by a disc-shaped obstacle moving along a straight line: a) exact trace, b) collision intervals.

where (x_t, y_t, θ_t) is the point on the trajectory to follow, v_t and ω_t are robot velocities at the trajectory point, (x, y, θ) is the current robot position (Fig. 8.7), and $k_1 = 0.5$, $k_2 = 1$, and $k_3 = 0.5$ are the empirically chosen controller parameters.

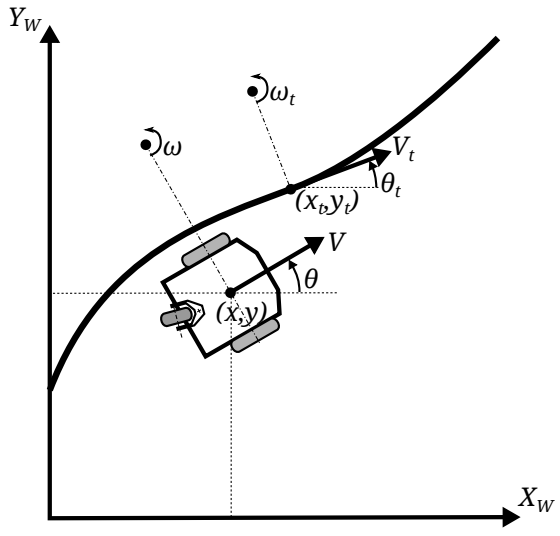


Figure 8.7: Differential-drive robot trajectory tracking.

8.3 Experiments

A simulation in a Stage simulator was prepared as the final evaluation of the three-level planning algorithm (3LMP)[1]. As 3LMP consists of the algorithms of different classes (i.e., real-time

[1] A video demonstrating a sample test run is available at `https://youtu.be/buVdbx046QU`

and anytime incremental) a number of parameters need to be defined. These are gathered in Table 8.1. The most interesting parameters are: maximum allocated time $0.2\,\mathrm{s}$ ($5.0\,\mathrm{s}$, when run for the first time) and AD*-Cut specific parameters, $\epsilon_{init} = 3.0$ and $\epsilon_{step} = 0.2$.

The map used for tests represents a real environment, part of the Faculty of Mechatronics building (Fig. 8.8). In the prepared scenario, a robot needs to navigate from a room to a hall. The path runs through a hallway along which four objects are moving (while two objects move from the left to the right, the other two move the opposite).

The motion of the objects was planned off-line using the approach described in Section 6.8. The moving objects were simulated in Stage as holonomic robots (Fig. 8.9), which do not need to change orientation, thus, a simple control algorithm can be used.

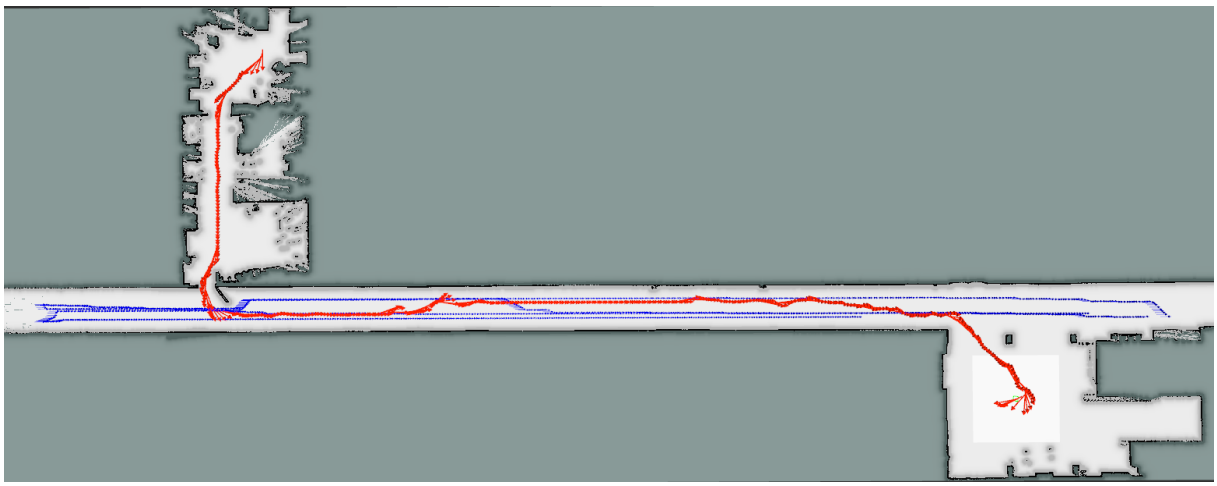


Figure 8.8: View obtained from RViz (ROS visualization) during the test. The robot navigates from a room (top-left) to a hall (bottom-right); the arrows depict poses of the robot along the traveled path. The dashed lines in the hallway represent the trajectories of the four moving obstacles.

The robot is equipped with a laser scanner that works in a range of $10\,\mathrm{m}$; hence an observation range was considerably long, which was 400 map cells. In the prepared simulation, a laser scanner could register only static obstacles; moving obstacles were detected by a mock motion detector, such that the robot could know trajectories of the moving obstacles for 20 seconds into the future.

The same scenario (i.e., navigation from a room to a hall) was run ten times. The average results of these runs are presented in Table 8.2. In all runs the robot was able to accomplish a mission and the median and maximum search times per re-planning episode were close to the granted times. Furthermore, in these runs, the robot traveled similar distance, about $53\,\mathrm{m}$, in the similar time, $190\,\mathrm{s}$.

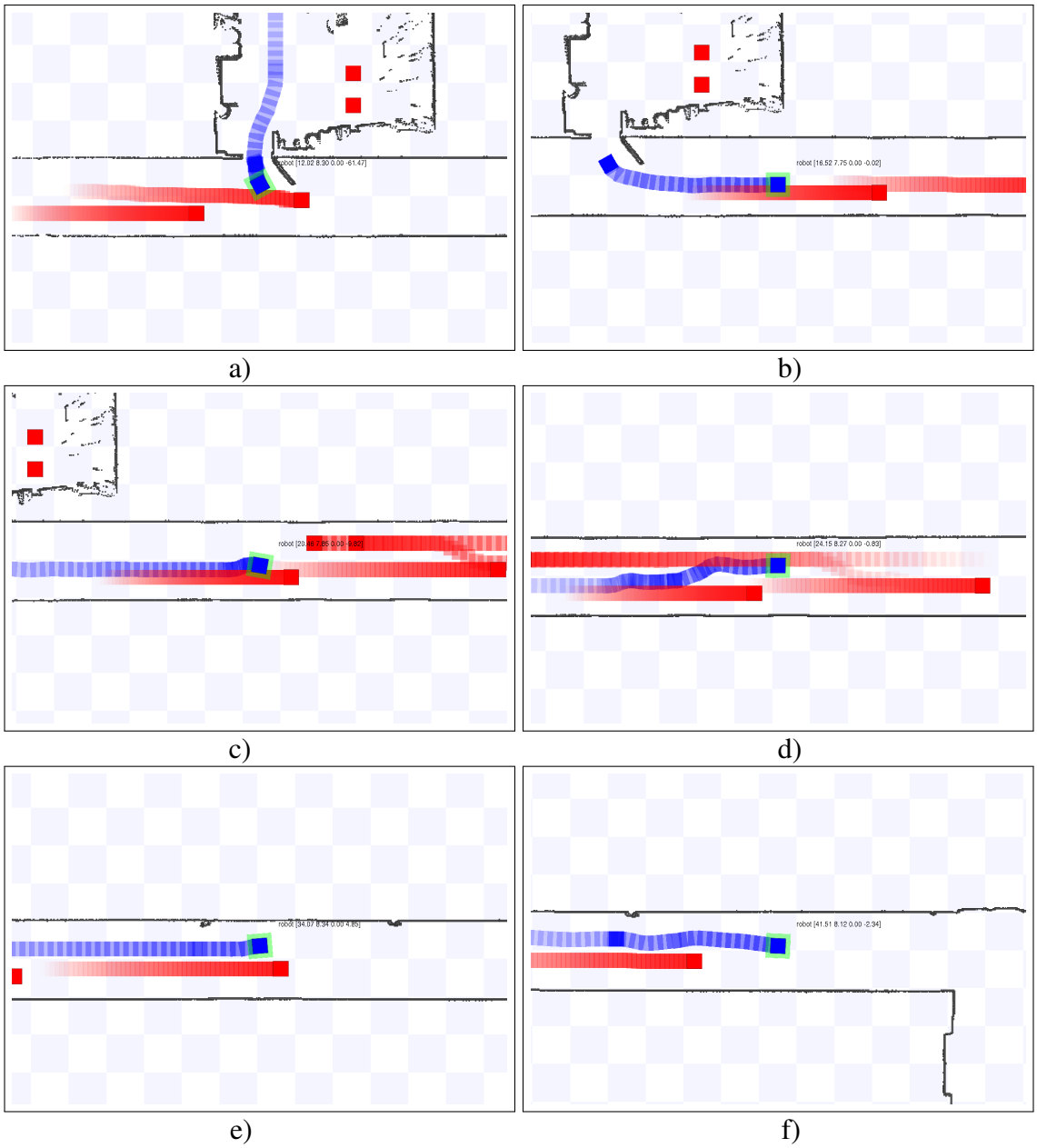


Figure 8.9: Subsequent views obtained from simulator Stage during the test. The square with a label is the controlled robot; the other shapes are the moving obstacles.

The search time per re-planning, traveled distance, longitudinal and angular velocities, for three sample runs, are presented in Figure 8.10. In the traveled distance plot, it can be observed that in all runs the beginning trajectory is almost the same. The differences between runs appear about the 70th second. While, in run 1, the robot was waiting considerably long in the doorway of the room, in run 10, the robot entered the hallway earlier and was moving right after a moving obstacle with the reduced speed.

Table 8.1: Parameters of the three-level motion planning algorithm benchmark.

Stage Setting	
Simulation frequency	$10\,\text{Hz}$
Environment Settings	
Map cell resolution δ	$0.025\,\text{m}$
Map size	$79.175\,\text{m} \times 21.95\,\text{m}$
Map size (cells)	3167×878 cells
Laser scanner range	$10\,\text{m}$
Laser scanner range (cells)	400 cells
Moving obstacles detection horizon	$20\,\text{s}$
Robot Parameters	
Robot size (width $\times$ length)	$0.4\,\text{m} \times 0.4\,\text{m}$
Minimum longitudinal velocity, v_{min}	$-0.1\,\frac{\text{m}}{\text{s}}$
Maximum longitudinal velocity, v_{max}	$0.5\,\frac{\text{m}}{\text{s}}$
Minimum longitudinal acceleration (deceleration), a_{min}	$-1\,\frac{\text{m}}{\text{s}^2}$
Maximum longitudinal acceleration, a_{max}	$1\,\frac{\text{m}}{\text{s}^2}$
Minimum angular velocity, ω_{min}	$-0.78\,\frac{\text{rad}}{\text{s}}$
Maximum angular velocity, ω_{max}	$0.78\,\frac{\text{rad}}{\text{s}}$
Local A* Algorithm (RTSLS: level 0)	
Search direction	forward
Search space	6D: safe interval graph on top of a state lattice with velocity sign (Sec. 8.2.3)
Number of possible robot orientations (angles)	16
Angular resolution of robot orientation	$22.5°$
Number of basic motion primitives per direction	7
Maximum search time	$0.2\,\text{s}$
Maximum search time (initial planning)	$5.0\,\text{s}$
AD*-Cut Algorithm (GSLS: level 1)	
Search direction	backward
Search space	3D state lattice
Number of possible robot orientations (angles)	16
Angular resolution of robot orientation	$22.5°$
Number of basic motion primitives per direction	7
ϵ_{init}	3.0
ϵ_{step}	2.0
A* Algorithm (GGS: level 2)	
Search direction	forward
Search space	2D 16-connected grid
Heuristic	$h(q, q_{goal}) = \frac{\|q_{goal} - q\|}{v_{max}}$, where $q = (x, y) \in \mathbb{R}^2$

Table 8.2: Experimental results for three-level motion planning among four moving obstacles.

#Run	Search Time per Re-plan. [ms]		#Search Steps per Re-plan.	Total Time [s]	Traveled distance [m]
	Median	Max.			
1	200.04	5811.22	1953	195.20	53.52
2	230.94	5795.22	1761	183.20	53.72
3	200.07	5865.70	1893	184.30	52.77
4	200.03	6015.99	2067	194.00	53.22
5	284.95	5847.56	1421	190.80	52.93
6	280.35	5815.11	1445	189.30	52.44
7	320.56	5767.72	1264	189.70	53.67
8	200.04	6065.45	1766	209.30	53.24
9	200.06	5877.71	1374	205.30	53.23
10	311.90	5906.54	1615	181.70	53.76
Total	242.89	5876.82	1656	192.28	53.25

8.4 Conclusions

In this chapter, the system utilizing the three-level hierarchical algorithm for mobile robot motion planning among moving obstacles was presented. The algorithm is based on the proposed Real-time Switchback (Sec. 7.4), and consists of local A*, AD*-Cut and A* algorithms, which are running in real time. The results of the simulation in Stage suggest that the proposed algorithms and their hierarchical composition can solve the problem of mobile robot motion planning in a dynamic environment. Although the algorithms in their current forms were not tested on a real robot, it is a common approach to run comprehensive benchmarks and simulations before a system can be deployed on a robot.

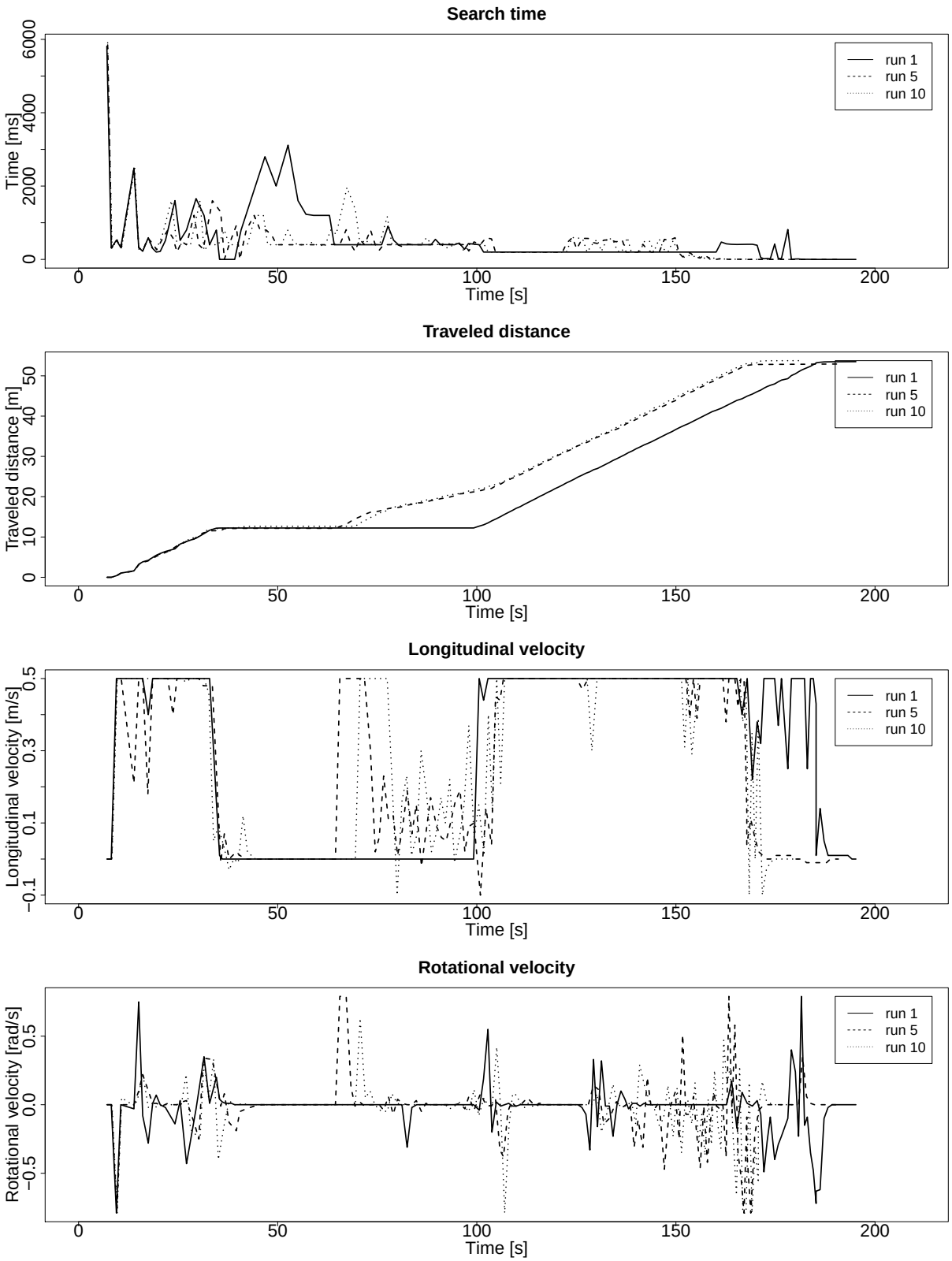


Figure 8.10: Data recorded for the three sample runs of the three-level motion planning algorithm in the simulated scenario.

9. Summary

In this thesis, the problem of mobile robot motion planning in a dynamic environment was addressed. The term *dynamic environment* used herein involves all types of environmental variability, such as static obstacle appearance and disappearance and changes induced by moving obstacles. It is important to note that the presence of moving obstacles introduces time dependency, which significantly complicates the problem.

The main objective of the thesis was to develop new motion planning algorithms that are suitable for real-world usage and provide a good trade-off between optimality (i.e., the least-cost path, either the minimum time or minimum distance) and computation time. This has been achieved using hierarchically composed heuristic search algorithms within the proposed Real-time Switchback algorithm. The algorithms were applied to planning in a time-dependent search space with an event-based description of moving obstacles. Finally, as a proof of concept, the system for a mobile robot motion planning in a dynamic environment utilizing the new algorithms was proposed and tested.

Although single elements of the proposed system can be identified across systems developed over the decades, the hierarchical composition of the algorithms, a compact event-based search-space description, and the two new incremental search algorithms, D* Extra Lite and AD*-Cut are the author's novel contributions to the field of robot motion planning and graph search.

The conducted work was divided into the following stages: development of the new quick incremental search algorithms, development of the theoretical description of a search space based on events for planning among moving obstacles, development of the hierarchical composition of heuristic search algorithms, and development of the system for a mobile robot motion planning in a dynamic environment.

9.1 Main Contributions

New incremental search algorithms. In Chapters 4 and 5, two new incremental heuristic search algorithms, D* Extra Lite and AD*-Cut were presented. While D* Extra Lite is an optimal algorithm (i.e., it provides the least-cost path within the current knowledge of an

environment), AD*-Cut works in an anytime manner, quickly providing a sub-optimal path and improving it in the remaining time allocated for searching. Both algorithms utilize a search-tree cutting technique.

This technique allows reinitializing search-tree branches affected by obstacle appearance and disappearance, which is reflected by search graph edge-cost changes. Although search-tree branch cutting was used in the past for incremental planning [61, 60], this technique was enhanced by the author and utilized in the D* Extra Lite and AD*-Cut algorithms that outperformed the state-of-the-art algorithms in the 2D grid path-planning benchmark. The benchmark results (Sec. 4.6 and Sec. 5.3) suggest that D* Extra Lite is from 1.08 to 1.94 times faster than D* Lite [10] and that AD*-Cut provides the first solution 1.47 times faster than AD* [32], which supports the thesis stated in this dissertation (Proposition 2.2) that search-tree brunch cutting can be used to speed up an incremental search.

Event-based search-space description. As planning among moving obstacles requires time as an additional variable (i.e., planning is held in a state-time space), a compact representation of a search space is desirable. A state-time space can be decomposed like a non-temporal state space; for example, it is possible to embed a regular grid in it. However, to reduce the size of a search space, time can be decomposed into cells representing qualitative situations that represent time before and after the presense of the moving obstacle in a place. The application of such an approach can be found in [33, 22] and in the author's former work [23].

In this thesis, a generalized description of such a qualitative time decomposition is proposed, namely, an event-based state-time space decomposition (Sec. 6.3). In this description, events are state-time points describing a moment of entering or leaving a map cell by a moving obstacle. Hence, for n moving obstacles that visit the same map cell at most once, a search space requires only $n + 1$ time layers, which supports one of the stated theses (Proposition 2.3). Then, motion planning among moving obstacles can be considered an action-event synchronization. In this thesis (Sec. 6.6), an action-event synchronization is investigated for applicability to robot motion planning (including kinematic, dynamic, and problem-specific constraints). Next, it has been shown that an event-based description can be used for minimum-time path planning among moving obstacles.

Hierarchical heuristic search algorithms. Hierarchical planning is a natural approach to solving complex problems. A complex planning problem can be divided into several hierarchy levels called abstractions. Among algorithms for hierarchical planning discussed in this thesis (Sec. 7.1), algorithms that use abstraction-based heuristics calculated by alternating search directions have been found especially useful for time-dependent motion planning.

As discussed in Section 7.2, a non-temporal search space consisting only of static obstacles is a quotient of the state-time space obtained by discarding a dimension of time; thus, it can be used to compute abstraction-based heuristics. Such a heuristic does not overestimate the true cost; hence, it is an admissible heuristic that can be used for the minimum-time search in a state-time space. This claim is supported by the experiments (Sec. 7.3), in which two algorithms utilizing alternating search directions, one running bottom-up (Switchback [16]) and one running top-down (Sec. 7.1.5), were over 1.5 times quicker than A* using the Euclidean distance as a heuristic.

To make use of hierarchical planning advantages in robot motion planning, a new Real-time Switchback algorithm has been proposed. Real-time Switchback is a combination of techniques utilized in real-time algorithms (discussed in Sec. 6.7.1) and the Switchback algorithm. Furthermore, in this thesis (Sec. 7.4.2), it has been proposed (and supported by the appropriate proof) that, under certain conditions, a heuristic learning step that is typical for real-time algorithms is not necessary for real-time time-dependent planning. This proposition is important, as it allows reducing the space and time complexities of the Real-time Switchback algorithm.

As Switchback and Real-time Switchback can be considered meta-algorithms, it was possible to combine algorithms of different classes. Thus, local A* running at the base level has been combined with the novel D* Extra Lite and AD*-Cut algorithms, resulting in the hierarchical real-time incremental algorithms (Sec. 7.4.2). These new algorithms allow performing a real-time hierarchical search that provides a trade-off between optimality and computation time, supporting Proposition 2.1 of this dissertation.

System for mobile robot motion planning in a dynamic environment. In Chapter 8, the system for mobile robot motion planning in a dynamic environment, which is based on a three-level hierarchical real-time incremental planning algorithm, has been proposed.

The entire system in the proposed form was tested only in a Stage simulator. (The main difficulty with testing on a real robot is that the system requires an algorithm for moving-obstacle detection, which is a non-trivial problem and is out of the scope of this thesis.) However, the hierarchical planning algorithm was implemented using the ROS framework [118]; hence, it is ready for deployment on a real robot. Furthermore, the use of a Stage simulator within a ROS framework made the tests more realistic. The simulation tests (Sec. 8.3) strongly suggest that the proposed system with the three-level planning algorithm at its core can be used for mobile robot motion planning in real dynamic environments.

9.2 Conclusions and Future Work

The main objective of the thesis has been achieved; the algorithms for mobile robot motion planning in a dynamic environment have been developed and thoroughly tested. To summarize, the following new algorithms have been proposed: D* Extra Lite, AD*-Cut, and its combinations with real-time search in accordance with the Real-time Switchback meta-algorithm.

With the experience gained from the work, it can be proposed that search-tree branch cutting used for an incremental search and the alternating search directions used for abstraction-based heuristic computations are crucial to the performance of the proposed system for mobile robot motion planning. In the future, the proposed system will be deployed and tested on real robots (also robotic arms); however, this will require the development of a moving-obstacle detection algorithm.

Although this thesis is devoted in particular to mobile robot motion planning, the algorithms and event-based description of dynamic environments are general purpose. Thus, all the proposed algorithms apply to any path-searching problem that can be represented as a graph. Indeed, much of the work in this field was originally for use in video games. A qualitative event-based description is an important step toward the integration of motion planning and action planning, which is typically based on the qualitative description of planning problems [121]. Furthermore, as robots sharing a common workspace can consider each other moving obstacles, the methods presented in this thesis, such as action-event synchronization, can be applied to multi-vehicle motion planning [122, 123] and robot cooperation [124].

References

[1] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT Press, 2011.

[2] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[4] Cezary Zieliński, Wojciech Szynkiewicz, Tomasz Winiarski, and Tomasz Kornuta. MRROC++ based system description. Technical Report Technical Report 06-9, IAIS, Warsaw, 2006.

[5] David L Poole and Alan K Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[6] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems*, pages 767–774, 2004.

[7] Eric A Hansen and Rong Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 28:267–297, 2007.

[8] Jur Van Den Berg, Rajat Shah, Arthur Huang, and Ken Goldberg. ANA*: anytime nonparametric A*. In *Proceedings of Twenty-fifth AAAI Conference on Artificial Intelligence (AAAI'11)*, pages 105–111, 2011.

[9] Anthony Stentz. The Focussed D* algorithm for real-time replanning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2, Montreal, Quebec, Canada*, IJCAI'95, pages 1652–1659, 1995.

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

[11] Leszek Podsędkowski, Jacek Nowakowski, Marek Idzikowski, and Istvan Vizvary. A new solution for path planning in partially known or unknown environment for nonholonomic mobile robots. *Robotics and Autonomous Systems*, 34(2):145–152, 2001.

[12] Carlos Hernández, Roberto Asín, and Jorge A Baier. Reusing previously found A* paths for fast goal-directed navigation in dynamic terrain. In *Twenty-Ninth AAAI Conference on Artificial Intelligence, Austin, Texas, USA*, AAAI'15, pages 1158–1164, 2015.

[13] Maciej Przybylski and Barbara Putz. D* Extra Lite: a Dynamic A* with search-tree cutting and frontier-gap repairing. *International Journal of Applied Mathematics and Computer Science (AMCS)*, 27(2):273–290, 2017.

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

[15] R.C. Holte, T. Mkadmi, R.M. Zimmer, and A. J. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence*, 85(1-2):321–361, 1996.

[16] Bradford John Larsen, Ethan Burns, Wheeler Ruml, and Robert Holte. Searching without a heuristic: Efficient use of abstraction. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI'10)*, pages 114–120, 2010.

[17] Michael J Leighton, Wheeler Ruml, and Robert C Holte. Faster optimal and suboptimal hierarchical search. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search (SoCS-2011)*, pages 92–99, 2011.

[18] Maxim Likhachev and Dave Ferguson. Planning long dynamically–feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research*, 28(8):933–945, 2009.

[19] Oliver Brock and Oussama Khatib. High-speed navigation using the global dynamic window approach. In *Robotics and Automation (ICRA), Proceedings. 1999 IEEE International Conference on*, volume 1, pages 341–346. IEEE, 1999.

[20] Robert C Holte, Chris Drummond, Maria B Perez, Robert M Zimmer, and Alan J MacDonald. Searching with abstractions: A unifying framework and new high-performance algorithm. In *Proceedings of The Tenth Conference of the Canadian Society for Computational Studies of Intelligence*, pages 263–270, 1994.

[21] Thierry Fraichard. Trajectory planning in a dynamic workspace: a 'state-time space' approach. *Advanced Robotics*, 13(1):75–94, 1998.

[22] Mike Phillips and Maxim Likhachev. SIPP: Safe interval path planning for dynamic environments. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 5628–5635. IEEE, 2011.

[23] Maciej Przybylski and Barbara Siemiątkowska. A new CNN-based method of path planning in dynamic environment. In Rutkowski L., Korytkowski M., Scherer R., Tadeusiewicz R., Zadeh L.A., and Zurada J. M., editors, *Artificial Intelligence and Soft Computing. ICAISC 2012*, volume 7268 of *Lecture Notes in Computer Science*, pages 484–492. Springer, Berlin, Heidelberg, 2012.

[24] Maciej Przybylski, Piotr Węclewski, and Mateusz Wiśniowski. Moduł planowania ścieżki w środowisku dynamicznym dla robota Kurier. *Prace Naukowe Politechniki Warszawskiej*, pages 235–245, 2012.

[25] Steven M LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[26] Benjamin Cohen, Sachin Chitta, and Maxim Likhachev. Single- and dual-arm motion planning with heuristic search. *The International Journal of Robotics Research*, 33(2):305–320, 2014.

[27] Christos Katrakazas, Mohammed Quddus, Wen-Hua Chen, and Lipika Deka. Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions. *Transportation Research Part C: Emerging Technologies*, 60:416–442, 2015.

[28] David González, Joshué Pérez, Vicente Milanés, and Fawzi Nashashibi. A review of motion planning techniques for automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 17(4):1135–1145, 2016.

[29] Michael Hoy, Alexey S Matveev, and Andrey V Savkin. Algorithms for collision-free navigation of mobile robots in complex cluttered environments: a survey. *Robotica*, 33(03):463–497, 2015.

[30] Johann Borenstein and Yoram Koren. The vector field histogram-fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288, 1991.

[31] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997.

[32] Maxim Likhachev, David I Ferguson, Geoffrey J Gordon, Anthony Stentz, and Sebastian Thrun. Anytime Dynamic A*: An anytime, replanning algorithm. In *Proceedings of the Fifteenth International Conference on International Conference on Automated Planning and Scheduling, Monterey, California, USA*, ICAPS'05, pages 262–271, 2005.

[33] Jur P Van Den Berg and Mark H Overmars. Roadmap-based motion planning in dynamic environments. *Robotics, IEEE Transactions on*, 21(5):885–897, 2005.

[34] Maciej Przybylski. Hierarchiczne planowanie akcji robota usługowego w środowisku dynamicznym. *Prace Naukowe Politechniki Warszawskiej. Elektronika*, 2(194):471–480, 2014.

[35] Stephane Petti and Thierry Fraichard. Safe motion planning in dynamic environments. In *Intelligent Robots and Systems (IROS), Proceedings. 2005 IEEE/RSJ International Conference on*, pages 2210–2215. IEEE, 2005.

[36] Ming Lin and Stefan Gottschalk. Collision detection between geometric models: A survey. In *Proc. of IMA conference on mathematics of surfaces*, volume 1, pages 602–608, 1998.

[37] Pablo Jiménez, Federico Thomas, and Carme Torras. 3D collision detection: a survey. *Computers & Graphics*, 25(2):269–285, 2001.

[38] Sinan Kockara, Tansel Halic, K Iqbal, Coskun Bayrak, and Richard Rowe. Collision detection: A survey. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pages 4046–4051. IEEE, 2007.

[39] Robert Bohlin and Lydia E Kavraki. Path planning using lazy PRM. In *Robotics and Automation (ICRA), Proceedings.2000 IEEE International Conference on*, volume 1, pages 521–528. IEEE, 2000.

[40] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications. Third Edition*. Springer-Verlag, Berlin Heidelberg, 2008.

[41] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[42] Sebastian Thrun and John J Leonard. Simultaneous localization and mapping. In Siciliano B. and Khatib O., editors, *Springer Handbook of Robotics*, chapter 37, pages 871–889. Springer, 2008.

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

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

[45] Frank Lingelbach. Path planning using probabilistic cell decomposition. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 1, pages 467–472. IEEE, 2004.

[46] Peng Cheng and Steven M LaValle. Resolution completeness for sampling-based motion planning with differential constraints. *International Journal of Robotics Research*, pages 1–37, 2004.

[47] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical Report Technical Report 98-11, Computer Science Dept., Iowa State University, 1998.

[48] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Practical search techniques in path planning for autonomous driving. In *Proceedings of the First International Symposium on Search Techniques In Artificial Intelligence and Robotics (STAIR-08)*, pages 1–6, 2008.

[49] Thomas M Howard and Alonzo Kelly. Optimal rough terrain trajectory generation for wheeled mobile robots. *The International Journal of Robotics Research*, 26(2):141–166, 2007.

[50] Mihail Pivtoraiko, Ross A Knepper, and Alonzo Kelly. Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics*, 26(3):308–333, 2009.

[51] Search-Based Planning Library (SBPL). `http://sbpl.net`.

[52] Traverso P. Nau D., Ghallab M. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

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

[54] Stefan Edelkamp and Stefan Schrödl. *Heuristic Search - Theory and Applications.* Academic Press, 2012.

[55] Mohamed Elbanhawi and Milan Simic. Sampling-based robot motion planning: A review. *IEEE Access*, 2:56–77, 2014.

[56] Steven M LaValle and James J Kuffner Jr. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001.

[57] Tobias Kunz and Mike Stilman. Kinodynamic RRTs with fixed time step and best-input extension are not probabilistically complete. In *Algorithmic Foundations of Robotics XI*, pages 233–244. Springer, 2015.

[58] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The international journal of robotics research*, 5(1):90–98, 1986.

[59] Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *Robotics and Automation (ICRA), Proceedings. 1994 IEEE International Conference on*, volume 4, pages 3310–3317, 1994.

[60] Leszek Podsędkowski. Path planner for nonholonomic mobile robot with fast replanning procedure. In *Robotics and Automation (ICRA), Proceedings. 1998 IEEE International Conference on*, volume 4, pages 3588–3593, 1998.

[61] Karen I Trovato. Differential A*: An adaptive search method illustrated with robot path planning for moving obstacles and goals, and an uncertain environment. *International Journal of Pattern Recognition and Artificial Intelligence*, 4(2):245–268, 1990.

[62] Karen I Trovato and Leo Dorst. Differential A*. *IEEE Transaction on Knowledge and Data Engineering*, 14(6):1218–1229, 2002.

[63] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning A*. *Artificial Intelligence*, 155(1):93–146, 2004.

[64] Xiaoxun Sun and Sven Koenig. The Fringe-Saving A* search algorithm—a feasibility study. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence, Hyderabad, India*, IJCAI'07, pages 2391–2397, 2007.

[65] Sven Koenig and Maxim Likhachev. Adaptive A*. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, Utrecht, Netherlands*, AAMAS '05, pages 1311–1312, 2005.

[66] Carlos Hernández, Pedro Meseguer, Xiaoxun Sun, and Sven Koenig. Path-adaptive A* for incremental heuristic search in unknown terrain. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling, Thessaloniki, Greece*, ICAPS'09, pages 358–361, 2009.

[67] Carlos Hernández, Jorge A Baier, and Roberto Asín. Making A* run faster than D* Lite for path-planning in partially known terrain. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, Portsmouth, New Hampshire, USA*, ICAPS'14, pages 504–508, 2014.

[68] Carlos Hernández, Xiaoxun Sun, Sven Koenig, and Pedro Meseguer. Tree Adaptive A*. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1, Taipei, Taiwan*, AAMAS '11, pages 123–130, 2011.

[69] Xiaoxun Sun, Sven Koenig, and William Yeoh. Generalized Adaptive A*. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1, Estoril, Portugal*, AAMAS '08, pages 469–476, 2008.

[70] Sven Koenig and Xiaoxun Sun. Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems*, 18(3):313–341, 2009.

[71] Sven Koenig and Maxim Likhachev. Improved fast replanning for robot navigation in unknown terrain. Technical Report Tech. Rep. GIT-COGSCI-2002/3, College of Computing, Georgia Institute of Technology, Atlanta (Georgia), 2001.

[72] Nathan R Sturtevant. Benchmarks for grid-based pathfinding. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(2):144–148, 2012.

[73] Rong Zhou and Eric A Hansen. Multiple sequence alignment using anytime A*. In *American Association for Artificial Intelligence (AAAI-02) Proceedings*, pages 975–977, 2002.

[74] Sandip Aine and Maxim Likhachev. Anytime Truncated D*: Anytime replanning with truncation. In *Sixth Annual Symposium on Combinatorial Search*, pages 2–10, 2013.

[75] Kalin Gochev, Alla Safonova, and Maxim Likhachev. Anytime tree-restoring weighted A* graph search. In *Seventh Annual Symposium on Combinatorial Search*, pages 80–88, 2014.

[76] Christopher Wilt and Wheeler Ruml. When does weighted A* fail? In *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SOCS 2012)*, pages 137–144, 2012.

[77] Michael Erdmann and Tomas Lozano-Perez. On multiple moving objects. In *Robotics and Automation (ICRA), Proceedings. 1986 IEEE International Conference on*, volume 3, pages 1419–1424. IEEE, 1986.

[78] Thierry Fraichard and Hajime Asama. Inevitable collision states—a step towards safer robots? *Advanced Robotics*, 18(10):1001–1024, 2004.

[79] Jur Pieter van den Berg. *Path planning in dynamic environments*. PhD thesis, Utrecht University, 2007.

[80] Juan P Gonzalez, Andrew Dornbush, and Maxim Likhachev. Using state dominance for path planning in dynamic environments with moving obstacles. In *Robotics and Automation (ICRA), Proceedings, 2012 IEEE International Conference on*, pages 4009–4015. IEEE, 2012.

[81] John Reif and Micha Sharir. Motion planning in the presence of moving obstacles. Technical Report TR-06-85, Center for Research in Computing Technology, Harvard University, 1985.

[82] Kikuo Fujimura. Time-minimum routes in time-dependent networks. *IEEE Transactions on Robotics and Automation*, 11(3):343–351, 1995.

[83] Robert Kindel, David Hsu, J-C Latombe, and Stephen Rock. Kinodynamic motion planning amidst moving obstacles. In *Robotics and Automation (ICRA), Proceedings. 2000 IEEE International Conference on*, volume 1, pages 537–543. IEEE, 2000.

[84] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. *The International Journal of Robotics Research*, 21(3):233–255, 2002.

[85] Léonard Jaillet and Thierry Siméon. A PRM-based motion planner for dynamically changing environments. In *Intelligent Robots and Systems (IROS), Proceedings. 2004 IEEE/RSJ International Conference on*, volume 2, pages 1606–1611. IEEE, 2004.

[86] Mikael Svenstrup, Thomas Bak, and Hans Jørgen Andersen. Trajectory planning for robots in dynamic human environments. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 4293–4298. IEEE, 2010.

[87] Jur van den Berg, Dave Ferguson, and James Kuffner. Anytime path planning and replanning in dynamic environments. In *Robotics and Automation (ICRA), Proceedings. 2006 IEEE International Conference on*, pages 2366–2371. IEEE, 2006.

[88] Chao Chen, Markus Rickert, and Alois Knoll. Kinodynamic motion planning with space-time exploration guided heuristic search for car-like robots in dynamic environments. In *Intelligent Robots and Systems (IROS), Proceedings. 2015 IEEE/RSJ International Conference on*, pages 2666–2671. IEEE, 2015.

[89] Chonhyon Park, Jia Pan, and Dinesh Manocha. Itomp: Incremental trajectory optimization for real-time replanning in dynamic environments. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, pages 207–215, 2012.

[90] Takeshi Ohki, Keiji Nagatani, and Kazuya Yoshida. Local path planner for mobile robot in dynamic environment based on distance time transform method. *Advanced Robotics*, 26(14):1623–1647, 2012.

[91] Aleksandr Kushleyev and Maxim Likhachev. Time-bounded lattice for efficient planning in dynamic environments. In *Robotics and Automation (ICRA), Proceedings. 2009 IEEE International Conference on*, pages 1662–1668. IEEE, 2009.

[92] Martin Rufli and Roland Siegwart. On the application of the D* search algorithm to time-based planning on lattice graphs. *Proceedings of The 4th European Conference on Mobile Robots (ECMR)*, 9:105–110, 2009.

[93] Julius Ziegler and Christoph Stiller. Spatiotemporal state lattices for fast trajectory planning in dynamic on-road driving scenarios. In *Intelligent Robots and Systems (IROS), Proceedings. 2009 IEEE/RSJ International Conference on*, pages 1879–1884. IEEE, 2009.

[94] Kamal Kant and Steven W Zucker. Planning collision-free trajectories in time-varying environments: a two-level hierarchy. *The Visual Computer*, 3(5):304–313, 1988.

[95] Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using velocity obstacles. *The International Journal of Robotics Research*, 17(7):760–772, 1998.

[96] Jur van den Berg, Ming Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Robotics and Automation (ICRA), Proceedings. 2008 IEEE International Conference on*, pages 1928–1935. IEEE, 2008.

[97] Brian C Dean. Shortest paths in fifo time-dependent networks: Theory and algorithms. *Rapport technique, Massachusetts Institute of Technology*, 2004.

[98] Luca Foschini, John Hershberger, and Subhash Suri. On the complexity of time-dependent shortest paths. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 327–341. SIAM, 2011.

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

[100] Sven Koenig. A comparison of fast search methods for real-time situated agents. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 864–871. IEEE Computer Society, 2004.

[101] Sven Koenig and Maxim Likhachev. Real-time adaptive A*. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 281–288. ACM, 2006.

[102] Nathan R Sturtevant and Vadim Bulitko. Scrubbing during learning in real-time heuristic search. *Journal of Artificial Intelligence Research*, 57:307–343, 2016.

[103] Venkatraman Narayanan, Mike Phillips, and Maxim Likhachev. Anytime safe interval path planning for dynamic environments. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4708–4715. IEEE, 2012.

[104] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1(1):7–28, 2004.

[105] Nathan Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *Proceedings of the 20th national conference on Artificial Intelligence (AAAI'05)*, volume 3, pages 1392–1397, 2005.

[106] Nathan Sturtevant and Renee Jansen. An analysis of map-based abstraction and refinement. In *International Symposium on Abstraction, Reformulation, and Approximation*, pages 344–358. Springer, 2007.

[107] Robert C Holte, Maria B Perez, Robert M Zimmer, and Alan J MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings of the thirtinth national conference on Artificial Intelligence – Volume 1 (AAAI'96)*, pages 530–535, 1996.

[108] Ariel Felner, Nathan R Sturtevant, and Jonathan Schaeffer. Abstraction-based heuristics with true distance computations. In *Symposium on Abstraction, Reformulation and Approximation (SARA-09)*, pages 1–8, 2009.

[109] Geňa Hahn and Claude Tardif. Graph homomorphisms: structure and symmetry. In *Graph symmetry*, pages 107–166. Springer, 1997.

[110] Vadim Bulitko, Nathan Sturtevant, Jiesan Lu, and Timothy Yau. Graph abstraction in real-time heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 30:51–100, 2007.

[111] Vadim Bulitko and Greg Lee. Learning in real-time search: A unifying framework. *Journal of Artificial Intelligence Research (JAIR)*, 25:119–157, 2006.

[112] Joseph C Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[113] Ning Jing, Yun wu Huang, and Elke A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10:409–432, 1998.

[114] Hector Gonzalez, Jiawei Han, Xiaolei Li, Margaret Myslinska, and John Paul Sondag. Adaptive fastest path computation on a road network: a traffic mining approach. In *Proceedings of the 33rd international conference on Very large data bases*, pages 794–805. VLDB Endowment, 2007.

[115] Robert C Holte, Jeffery Grajkowski, and Brian Tanner. Hierarchical heuristic search revisited. In *Abstraction, Reformulation and Approximation*, pages 121–133. Springer, 2005.

[116] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1):191–246, 2006.

[117] Adam Niewola and Leszek Podsedkowski. L* algorithm–a linear computational complexity graph searching algorithm for path planning. *Journal of Intelligent & Robotic Systems*, pages 1–20, Dec 2017.

[118] Robot Operating System (ROS). `http://ros.org`.

[119] Gregory Dudek and Michael Jenkin. *Computational principles of mobile robotics*. Cambridge University Press, 2010.

[120] C Samson and K Ait-Abderrahim. Mobile robot control, part 1: Feedback control of a non-holonomic wheeled cart in cartesian space. *INRIA Report*, 1288, 1990.

[121] Maciej Przybylski, Daniel Koguciuk, Barbara Siemiątkowska, Bogdan Harasymowicz-Boggio, and Łukasz Chechliński. Integration of qualitative and quantitative spatial data within a semantic map for service robots. In Szewczyk R., Zieliński C., and Kaliczyńska M., editors, *Progress in Automation, Robotics and Measuring Techniques. Advances In Intelligent Systems and Computing, vol. 351*, pages 223–232. Springer, Cham, 2015.

[122] Elżbieta Roszkowska. High-level motion control for workspace sharing mobile robots. In Kozłowski K., editor, *Robot Motion and Control 2007. Lecture Notes in Control and Information Sciences, vol. 360*, pages 427–436. Springer, London, 2007.

[123] Igor Wojnicki, Sebastian Ernst, and Wojciech Turek. A robust planning algorithm for groups of entities in discrete spaces. *Entropy*, 17(8):5422–5436, 2015.

[124] Włodzimierz Kasprzak, Wojciech Szynkiewicz, Dimiter Zlatanov, and Teresa Zielińska. A hierarchical CSP search for path planning of cooperating self-reconfigurable mobile fixtures. *Engineering Applications of Artificial Intelligence*, 34:85–98, 2014.

List of Algorithms

3.1 Forward search. 33
3.2 Backward search. 33
3.3 Roadmap construction. 34
3.4 RRT construction. 35
3.5 Kinodynamic RRT. 36
4.1 Procedures common for the D* Lite and D* Extra Lite algorithms. 40
4.2 D* Lite (optimized version) procedures. 45
4.3 D* Extra Lite procedures. 47
4.4 CUTBRANCH() procedure of the D* Extra Lite algorithm for domains in which $Succ(s) \equiv Pred(s)$. 54
5.1 Anytime repairing A* (ARA*) 68
5.2 Anytime D* (AD*) 70
5.3 AD*-Cut. Required parameters: $\epsilon_{init}, \epsilon_{step}$. 73
6.1 Main procedure common for real-time algorithms. 98
6.2 LRTA* heuristic value update. 99
7.1 Switchback algorithm, where i denotes the abstraction level and $i = 0$ is the base level. 115
7.2 Real-time Switchback algorithm; functions modified with respect to Algorithm 7.1, where i denotes the abstraction level and $i = 0$ is the base level. 123
7.3 Real-time Switchback main function. 124

Index

A*, 17, **32**
abstraction transformation, **107**
action, **28**, 29, 77, 78, 137
action space, 28, **77**
action-event synchronization, **91**, 94, 136
AD*, 67, **70**
AD*-Cut, **73**
admissible heuristic, 17, 32
AltO, 111
anytime search, 17, **67**
ARA*, **68**, 71, 102

classical refinement, **110**
collision, 24
configuration, **23**
configuration space, **23**, 77
configuration-time space, **78**
cost function, **30**, 81, 91, 138
cost-map, **81**

D* Extra Lite, **47**
D* Lite, 39, **45**, 69
differential-drive robot, 36, **131**
dispersion, 26
dynamic constraints, 27
dynamic environment, **15**

event, **84**
exhaustive search, 32

graph, **29**, 107
grid, **26**

heuristic function, **32**
heuristic search, 17, **32**
homomorphism, **107**

incremental search, 15, **39**
inverse transition function, 31

kinematic constraints, 27, 132
kinodynamic motion planning, 109
kinodynamic planning, **28**

label (graph), 109
local motion planner, 34
local search, 18, 98
LRTA*, 98

monotonic refinement, **110**

narrow passages, 26
non-holonomic constraints, **27**, 132

optimality, 17, 32

parent, **31**
path-marking, 111
planning in an unknown (or partially-known) environment, 15
potential field, **37**
predecessors, **31**
principle of optimality, 88
probabilistic roadmap (PRM), **26**

probabilistically complete, **26**, 36

quotient, **108**

Rapidly Exploring Random Tree, **35**
real-time search, 98, 121
Real-Time Switchback, **122**
refinement, 109
region of an inevitable collision (RIC), 81
regular lattice, **26**
road-map, **26**
robot, 17, 23

safe interval, **85, 86**
safe state, 122
sampling-based planning, 26
search-tree, **32**
SLAM, 25
state, **28**, 77
state lattice, **28**
state space, 77
state-lattice search, **29**
state-space, **28**
state-time space, **78**, 78
static environment, **15**
successors**,** 31
Switchback, 112, **115**

time-consistency, **88**
time-dependent networks, 88
time-dependent planning, 15, 109
trajectory tracking, 140
transition function, **31**

visibility graph, 26
Voronoi diagram, 26
Voronoi field, 37

workspace, **23**