Navigation and guidance strategy online planning and execution for autonomous UAV

Ana Raquel Araújo Simão do Carmo

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor(s): Caroline Ponzoni Carvalho Chanel
Prof. Rodrigo Martins de Matos Ventura

Examination Committee

Chairperson: Prof. José Fernando Alves da Silva
Supervisor: Prof. Rodrigo Martins de Matos Ventura
Member of the Committee: Prof. Manuel Fernando Cabido Peres Lopes

November 2019

Para os meus pais.

Acknowledgments

I would like to express my appreciation to Caroline Chanel, Yoko Watanabe, Jean Alexis and Corentin Chauffaut for the valuable support given to me throughout my work in Toulouse in the scope of this dissertation. Specially to Caroline Chanel, I would like to thank all the effort and kindness, for mentoring me, sharing her knowledge and conceding an amazing working place in ISAE-SUPAERO. I'd also like to thank my supervisor Professor Rodrigo Ventura for accepting to mentor me in this project as well and whose advice and support were highly appreciated. Additionally to the Erasmus+ Program and all the administrative staff involved, I would like to thank for the scholarship provided to help me finance my stay in France.

On a more personal note, I'm grateful for all the support my parents and brother have given me throughout my education years, in my darkest and brightest moments, and for always standing by my side in the choices I've made. A heartfelt thank you to my group in the DCAS department, for making my experience in Toulouse so worthwhile. To the most amazing group back in my bachelor degree in Mechanical Engineering: Ângela, Catarina, Fernando, João and Tomás, thank you for remaining the truest of friends, always creating the best memories and stories that I will carry for life. And last but not least, to Adriana, Mafalda and Hugo, thank you for all the late nights studying, sharing the stress and doubts, the support and strength, during this journey in IST.

Resumo

O planeamento eficiente de trajetórias de Veículos Aéreos Não Tripulados (VANTs) autónomos em ambientes estocásticos é um problema de tomada de decisões desafiante. Recentemente, Processos de Decisão de Markov com Observação Mista (MOMDPs) foram propostos para o resolver, reduzindo o esforço computacional. Neste trabalho, um problema de planeamento da trajetória mais curta em ambiente estocástico parcialmente observável (PO-SSP) é modelado como um MOMDP. O modelo resultante permite ao planeador lidar *a priori* com a disponibilidade probabilística do sensor e a propagação de erros de execução da trajetória, que dependem da solução de navegação e guiamento usada. Esta abordagem resulta num problema pesado e complexo a ser resolvido. Para o enfrentar, é utilizada uma variante orientada a objetivos do *Partially Observable Monte-Carlo Planning* (POMCP), um dos algoritmos de pesquisa em árvore mais rápidos para ambientes parcialmente observáveis em tempo real. Este usa *Upper Confidence Bounds* (UCB1) como estratégia de seleção da ação, a qual depende de uma constante tipicamente ajustada manualmente que requer uma pesquisa exaustiva para encontrar o valor mais adequado. Esta pesquisa exaustiva aplicada num problema tão complexo como este pode consumir muito tempo. Assim, são propostos coeficientes dinâmicos como primeira contribuição desta dissertação. A segunda contribuição passa por incorporar esta solução de navegação e guiamento numa arquitetura de planeamento e execução em simultâneo. A estrutura resultante fornece soluções estritamente *anytime*, enquanto explora o tempo de execução da ação para antecipar estados futuros, minimizando a duração total da missão. Simulações realizadas em Gazebo permitem avaliar o desempenho desta estrutura.

Palavras-chave: Planeamento de Trajetórias sob Incerteza, Estratégia de Navegação e Guiamento, Processo de Decisão de Markov com Observação Mista, Planeamento e Execução em simultâneo.

Abstract

Efficient path planning for autonomous Unmanned Aerial Vehicles (UAVs) in cluttered environments is a challenging decision-making problem under uncertainty. Recently, Mixed-Observability Markov Decision Processes (MOMDPs) have been proposed to solve such navigation problems with the goal of reducing computational effort. In this work, a Partially Observable Stochastic Shortest Path (PO-SSP) planning problem is modelled as a MOMDP. The resulting model enables the planner to deal with a priori probabilistic sensor availability and path execution error propagation, which depend on the navigation and guidance solution used. This approach results in a large and complex path planning problem to be solved. To address this issue, a goal-oriented variant of the Partially Observable Monte-Carlo Planning (POMCP), one of the fastest online state-of-the-art Monte-Carlo Tree Search algorithms for partially observable environments, is used. It relies on the Upper Confidence Bounds (UCB1) as an action selection strategy, which depends on a coefficient typically adjusted manually that requires an exhaustive search to find the most suitable value. This exhaustive search applied to such a complex planning problem may be extremely time consuming. Therefore, dynamic coefficients are proposed as a first contribution of this dissertation. A second contribution is to incorporate this navigation and guidance strategy in a planning-while-executing framework. The resulting structure provides strictly anytime solutions, while exploiting action execution time to anticipate and plan future states, minimizing the overall duration of the mission. Simulations performed in Gazebo allow to evaluate the performance of this structure.

Keywords: Path Planning under Uncertainty, Navigation and Guidance Strategy, Mixed-Observability Markov Decision Process, Concurrent Planning and Execution.

Contents

Acknowledgments . . . v
Resumo . . . vii
Abstract . . . ix
List of Tables . . . xv
List of Figures . . . xvii
Nomenclature . . . xix
Glossary . . . xxiii

1 Introduction 1
1.1 Motivation . . . 1
1.2 Problem Statement . . . 3
1.3 State-of-the-art . . . 4
1.3.1 Motion planning . . . 4
1.3.2 Offline and Online Planning . . . 7
1.4 Objectives and contributions . . . 8
1.5 Thesis Outline . . . 10

2 Background 11
2.1 Markov Decision Processes . . . 11
2.2 Stochastic Shortest-Path MDP . . . 15
2.3 Partially Observable Markov Decision Processes . . . 16
2.3.1 Belief State . . . 17
2.3.2 Policy and value function . . . 18
2.3.3 Extension to continuous state space . . . 19
2.4 Solving POMDPs . . . 19
2.4.1 Offline solvers . . . 20
2.4.2 Online solvers . . . 21
2.4.3 Mixed-Observability Markov Decision Processes . . . 22
2.5 Monte Carlo methods in POMDPs . . . 23
2.5.1 Monte Carlo Rollouts . . . 24
2.5.2 Monte Carlo Tree Search . . . 25

2.5.3 Tree and Simulation Policies . . . 26
2.5.4 Upper Confidence Bounds applied to Trees Algorithm . . . 29
2.5.5 Partially Observable UCT Algorithm . . . 29
2.5.6 Monte Carlo Belief State Updates . . . 30
2.5.7 Partially Observable Monte Carlo Planning . . . 31

3 Problem formulation **35**
3.1 System architecture . . . 35
3.2 GNC transition model . . . 37
3.2.1 State transition model . . . 37
3.2.2 State estimation in the navigation module . . . 38
3.3 MOMDP model . . . 41

4 Offline Approach **47**
4.1 Offline POMCP-GO Algorithm . . . 47
4.1.1 Heuristic function . . . 49
4.2 Proposed approach for parameter tuning . . . 49
4.2.1 Action selection strategies for planning . . . 49
4.2.2 Back-propagation strategies . . . 52
4.3 Evaluation based on offline planning and simulations . . . 53
4.3.1 Configuration . . . 53
4.3.2 Results . . . 56
4.3.3 Summary . . . 61

5 Online Approach **63**
5.1 Online POMCP-GO Algorithm . . . 63
5.2 AMPLE planning-while-executing framework . . . 64
5.2.1 AMPLE architecture . . . 65
5.2.2 AMPLE planning thread . . . 66
5.2.3 AMPLE execution thread . . . 68
5.2.4 Integration of POMCP-GO in AMPLE . . . 69
5.3 Simulations . . . 69
5.3.1 Configuration . . . 70
5.3.2 Results . . . 71
5.4 Application in Gazebo . . . 76

6 Conclusions **79**
6.1 Achievements . . . 79
6.2 Future Work . . . 80

Bibliography **81**

A Additional results in parameter tuning stage **87**

A.1 Comparison between EBC and the extensive search using UCB1 87

A.2 Results during the optimization process 89

List of Tables

4.1 Performance comparison between strategies in the *WallBaffle* map. The data is organized as Average (Standard deviation). In **bold** are represented the best values of the three metrics (success rate, flight time and computational time) for each back-propagation strategy. 59

4.2 Performance comparison between strategies in the *CubeBaffle* map. The data is organized as Average (Standard deviation). In **bold** are represented the best values of the three metrics (success rate, flight time and computational time) for each back-propagation strategy. 61

5.1 Parameters used for the online simulations. 71

List of Figures

1.1 Generalized robotic path planning problem shown in a 2D configuration space. We wish to plan a path from x_{start} to x_{goal} through the free space C_{free} [10]. 4
1.2 Four strategies for interleaving periods of planning (in black) and execution (in red) [25]. . 7
1.3 Organization of the NavPlan research project. 9

2.1 Transition model of an MDP, in which an agent in state s_t performs an action a_t. After doing so, he finds himself in state s_{t+1} with a reward $\mathcal{R}(s_t, a_t)$. 12
2.2 Transition model of a POMDP. 16
2.3 Factored model of state and observation transitions of a MOMDP for an action a_t. 23
2.4 Monte Carlo Tree Search. 25
2.5 Rejection sampling process. 31
2.6 The search tree constructed by POMCP for a problem with 2 actions and 2 observations, with history nodes represented in white and sequence nodes in black. 32
2.7 An illustration of POMCP in an environment with 2 actions and 2 observations. 33

3.1 System architecture diagram, in which the GNC closed-loop vehicle model is incorporated into the MOMDP transition function [5]. 36
3.2 Difference between the functioning frequency of the GNC system and the planner. 36
3.3 MOMDP model considered [8]. 42

4.1 Decay of the coefficient c_{DWD} over the depth of the search tree. 51
4.2 Obstacle maps proposed on the test benchmark [75] as well as examples of probability maps of GPS availability with different precision thresholds, given in $meters$. 54
4.3 Representation of the probability map on GPS availability and the environment map. . . . 55
4.4 Set of directions considered. 55
4.5 *WallBaffle* map with initial position in $(10, 25, 5)m$, using different combinations of action selection and back-propagation strategies. 57
4.6 *WallBaffle* map with initial position in $(50, 25, 5)m$, using different combinations of action selection and back-propagation strategies. 58
4.7 *CubeBaffle* map with initial position in $(35, 20, 5)m$, using different combinations of action selection and back-propagation strategies. 60

4.8 *CubeBaffle* map with initial position in $(65, 20, 5)m$, using different combinations of action selection and back-propagation strategies. 60

5.1 AMPLE architecture: connections between the execution and the planning threads [27]. . 65
5.2 *WallBaffle* map with initial position in $(10, 25, 5)m$, using UCB*. 71
5.3 *WallBaffle* map with initial position in $(10, 25, 5)m$, using EBC. 72
5.4 *WallBaffle* map with initial position in $(50, 25, 5)m$, using UCB*. 72
5.5 *WallBaffle* map with initial position in $(50, 25, 5)m$, using EBC. 73
5.6 *CubeBaffle* map with initial position in $(35, 20, 5)m$, using UCB*. 73
5.7 *CubeBaffle* map with initial position in $(35, 20, 5)m$, using EBC. 74
5.8 *CubeBaffle* map with initial position in $(65, 20, 5)m$, using UCB*. 74
5.9 *CubeBaffle* map with initial position in $(65, 20, 5)m$, using EBC. 75
5.10 Five trajectories executed in the *WallBaffle* map for two distinct initial positions, using EBC in the action selection strategy in POMCP-GO and a bootstrap time of $15s$ for AMPLE. Projections of the trajectories are illustrated over the three planes xy, xz and yz. 77
5.11 Five trajectories executed in the *CubeBaffle* map for two distinct initial positions, using EBC in the action selection strategy in POMCP-GO and a bootstrap time of $15s$ for AMPLE. Projections of the trajectories are illustrated over the three planes xy, xz and yz. . . 78

A.1 *WallBaffle* map with initial position in $(10, 25, 5)m$. 87
A.2 *WallBaffle* map with initial position in $(50, 25, 5)m$. 88
A.3 *CubeBaffle* map with initial position in $(35, 20, 5)m$. 88
A.4 *CubeBaffle* map with initial position in $(65, 20, 5)m$. 88
A.5 Best (UCB$_{\sqrt{(\cdot)}}$ – up) and worst (EBC – down) strategies for *WallBaffle* map with initial position in $(10, 25, 5)m$ and $10m$ precision on GPS availability. 89
A.6 Best (EBC$_{\text{MP}}$ – up) and worst (EBC – down) strategies for *WallBaffle* map with initial position in $(50, 25, 5)m$ and $10m$ precision on GPS availability. 90
A.7 Best (UCB$_{\text{MP}}$ – up) and worst (EBC – down) strategies for *WallBaffle* map with initial position in $(10, 25, 5)m$ and $2m$ precision on GPS availability. 90
A.8 Best (UCB$_{\text{MP}}$ – up) and worst (DWD – down) strategies for *WallBaffle* map with initial position in $(50, 25, 5)m$ and $2m$ precision on GPS availability. 91
A.9 Best (DWD – up) and worst (EBC – down) strategies for *CubeBaffle* map with initial position in $(35, 20, 5)m$ and $2m$ precision on GPS availability. 91
A.10 Best (EBC$_{\text{MP}}$ – up) and worst (DWD – down) strategies for *CubeBaffle* map with initial position in $(65, 20, 5)m$ and $2m$ precision on GPS availability. 92
A.11 Best (DWD – up) and worst (UCB$_{\sqrt{(\cdot)}}$ – down) strategies for *CubeBaffle* map with initial position in $(35, 20, 5)m$ and $1m$ precision on GPS availability. 92
A.12 Best (EBC$_{\text{MP}}$ – up) and worst (DWD – down) strategies for *CubeBaffle* map with initial position in $(65, 20, 5)m$ and $1m$ precision on GPS availability. 93

Nomenclature

Physics Constants

g Gravitational Constant.

Number Sets

$\mathbb{R}$ Real Numbers.

GNC Transition Symbols

β_a Accelerometer bias.

$\hat{\mathbf{x}}_k^-$ Predicted state at time k.

$\hat{\mathbf{x}}_k$ Estimated state at time k.

$\mathcal{V}$ Vehicle velocity vector.

$\mathcal{X}$ Vehicle position vector.

$\tilde{\mathbf{x}}_k^- = \mathbf{x}_k - \hat{\mathbf{x}}_k^-$ State prediction error.

$\tilde{\mathbf{x}}_k = \mathbf{x}_k - \hat{\mathbf{x}}_k$ State estimation error.

ξ Measurement error.

k GNC unit of time.

v Process noise.

$\mathbf{P}_k^- = \mathbb{E}[\tilde{\mathbf{x}}_k^- \tilde{\mathbf{x}}_k^{-T}]$ Prediction covariance error.

$\mathbf{P}_k = \mathbb{E}[\tilde{\mathbf{x}}_k \tilde{\mathbf{x}}_k^T]$ Estimation covariance error.

$\mathbf{R}_{BI}$ Rotation matrix from the inertial to the UAV body frames.

$\mathbf{x}$ Vehicle state vector.

MOMDP Model Symbols

γ Discount factor.

$\mathcal{A}$ Action space.

$\mathcal{B}$ Belief state space.

$\mathcal{C}$ Cost function.

$\mathcal{G}$ Goal state space.

$\mathcal{O}$ Observation function.

$\mathcal{R}$ Reward function.

$\mathcal{S}$ State space.

$\mathcal{T}$ Transition function.

Ω Observation space.

π Policy.

a Action.

b Belief state.

f_t Flight time.

F_{Col} Collision flag.

F_{S_i} Availability flag of sensor S_i.

K Collision cost.

o Observation.

s State.

s_h Hidden state.

s_p Partially observable state.

s_v Visible state.

t Planning epoch.

Other Symbols

$\mathcal{L}$ Bellman's dynamic operator.

c Exploration factor.

G Black-box simulator.

h History node.

L Leaf node.

M Set of particles.

T Tree node.

Subscripts

∞ Infinite condition.

i, j, k Computational indexes.

x, y, z Cartesian components.

ref Reference condition.

Supercripts

* Optimal.

T Transpose.

Glossary

ABT	Adaptive Belief Tree
AEMS	Anytime Error Minimization Search
AMPLE	Anytime Meta PLannEr
CR	Cumulative Regret
DESPOT	DEterminized Sparse Partially Observable Tree
DWD	Decay With Depth
EBC	Entropy-Based Coefficient
GNC	Guidance, Navigation and Control
GPS	Global Positioning System
HSVI	Heuristic Search Value Iteration
IMU	Inertial Measurement Unit
INS	Inertial Navigation System
MAB	Multi-Armed Bandit
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process
MOMDP	Mixed-Observability Markov Decision Process
PBVI	Point-Based Value Iteration
PDOP	Position Dilution Of Precision
POMCP	Partially Observable Monte Carlo Planning
POMDP	Partially Observable Markov Decision Process
PRM	Probabilistic RoadMap
PUMA	Planning under Uncertainty with Macro-Actions
PWLC	PieceWise Linear and Convex
RBSR	Randomized Belief-Space Replanning
RL	Reinforcement Learning
ROS	Robotic Operating System
RRT	Rapidly-exploring Random Tree
RTDP	Real-Time Dynamic Programming
SARSOP	Successive Approximations of the Reachable Space under Optimal Policies

SLAM	Simultaneous Localization And Mapping
SR	Simple Regret
SSP	Stochastic Shortest-Path
UAV	Unmanned Aerial Vehicle
UCB	Upper Confidence Bound
UCT	Upper Confidence Bounds applied to Trees

Chapter 1

Introduction

This chapter aims to briefly explain the relevance of an online navigation and guidance strategy for planning-while-executing in autonomous Unmanned Aerial Vehicles (UAVs). A short review of the state-of-the-art on motion planning and existing frameworks for offline and online planning is conducted in Section 1.3. The thesis' objectives and contributions are exposed in Section 1.4, as well as its outline in Section 1.5.

1.1 Motivation

Unmanned Aerial Vehicles (UAVs) constitute a research field that has been extensively explored in the last decade. Until now, UAVs have been almost exclusively operated by humans. However, recent advances in sensing technology, computing power, mapping and data processing have made it possible for an UAV to begin developing autonomous capabilities, that is, to perform tasks without requiring a human to control them. As a result, UAVs can nowadays be employed in a variety of operations, both civil and military, ranging from search and rescue, surveillance and inspection missions (e.g. landmine detection or inspection of towers for corrosion and other defects) [1], to navigation missions in cluttered environments (e.g. urban areas) [2].

Autonomous navigation capabilities include environment mapping, localization and guidance functionalities embedded in a Guidance, Navigation and Control system (GNC) onboard the UAV system. Navigation refers to the determination, at a given time, of the vehicle's state (localization, velocity, etc.). Guidance uses the state estimation to compute the desired trajectory from the vehicle's current location to a designated target, as well as desired changes in the state, for following that path. Finally, control is responsible for manipulating and giving the actuators commands to be executed to carry out the reference trajectory.

The navigation module estimates the state of the vehicle based on the measurements of the onboard sensors. Besides the classical solution combining Inertial Navigation Systems (INS) with Global Positioning Systems (GPS) as a self-localization system, one can also rely on solutions based on 2D or 3D vision sensors: visual odometry, visual Simultaneous Localization and Mapping (SLAM), etc. [1] [2].

However, it is important to notice that all of these solutions have their own limitations, depending on how the environment is perceived. An INS-only navigation solution quickly diverges due to accumulation of inertial measurement bias, so it is usually coupled with GPS. Yet, GPS signal can easily be masked or degraded due to occlusion in cluttered environments. Consequently, a UAV cannot sustain autonomous flight in the absence of GPS signal, if it is not backed up with alternative sensors that do not rely on GPS information. Nevertheless, visual SLAM and optical flow techniques require rich texture on the image and are influenced by the visibility of the points of interest. For these reasons and in order to guarantee autonomous capabilities and the safety of the mission, it is of crucial importance to embed, in the UAV onboard flight system, different navigation solutions and their use in each phase of the mission to be decided according to the surrounding environment.

Furthermore, real life scenarios are often stochastic, inheriting uncertainty from various sources, such as state evolution and perception. Uncertainty in state evolution is considered if the outcome of an action is not fully predictable, due to unexpected environment changes (e.g. winds or impacts) or faulty actuators onboard the UAV. On the other hand, uncertainty in perception occurs if the current state is not precisely known, due to noisy measurements from imperfect sensors. Reasons for non-reliable sensors are numerous, namely signal occlusion, noise, electronic failure or range limits. Consequently, not accounting for uncertainty when designing an autonomous vehicle can cause failure of actions or deviation from the trajectories, which may result in unfeasible trajectories and lead to disastrous outcomes, such as collisions.

Fortunately, the performance of some sensors can be predicted by a priori knowledge about the environment. For instance, the GPS accuracy depends on the visibility of the constellation of satellites, which itself depends on geolocation, time and the environment. Given a known satellite constellation and a position, it is possible to obtain information about the accuracy of GPS measurement in the form of Dilution of Precision (DOP) [3]. DOP is a metric allowing to estimate a priori the degradation of the precision of the GPS. In particular, PDOP (Position Dilution of Precision) is the standard deviation of the position error calculated by a GPS receiver and it is possible to calculate the PDOP for each given point in a 3D georeferenced map [4]. This information could be very useful for path planning, as this would allow the planner to anticipate an eventual degradation of the sensors' accuracy and its influence on the path execution error along the way, thereby minimizing the risk of collision [5].

Although extensive research exists on applications of these UAV localization and guidance approaches, only recent developments can be found on the decision of switching navigation modes among the ones available onboard as a function of the availability of the sensors that depend on the environment. Previous work [6] [7] dealt with the problem of deterministic and discrete path planning by considering the uncertainty of the location of the vehicle and propagating it along a path calculated according to the environment. Delamer et al. have explored the complete model of closed-loop vehicle motion with the functions of GNC in the decision-making process, while taking into account the sensors' availability, in a significant body of work, including ([3], [8], [5]). Incorporating the GNC transition module, which can be considered as a "low level" system, in a "high level" POMDP model, allowed to propagate the influence of sensor availability on the uncertainty of the trajectory executed. Nevertheless, such

developments were achieved under an offline configuration of the planning algorithm, which is not appropriate for applications in real scenarios with environments as dynamic as the ones found in urban areas.

The work developed in this dissertation is integrated in an ongoing research project named NavPlan, result of a collaboration between the French Aerospace Lab, ONERA, and ISAE-SUPAERO. This project aims to create solutions regarding the subject of decision on navigation and guidance strategies for autonomous UAVs in cluttered environments.

1.2 Problem Statement

This dissertation deals with the problem of planning safe (avoiding obstacles) and efficient (minimum distance or time) trajectories towards a goal under uncertainty, taking into account the availability of sensors that depend on the environment and, with that, deciding on the navigation and guidance strategy to be used by the UAV.

Such trajectory planning problem is a sequential decision-making problem under partial observability. In fact, the autonomous vehicle must perform a sequence of actions in order to carry out its mission with uncertainty about its position, in an environment itself uncertain, because the availability of sensors is probabilistic. In this context, the agent can only estimate the world state by propagating a probability distribution over the set of possible states he could be in, in other words, a belief state. An even bigger challenge arises when considering that the state estimation (localization) and path execution (guidance) results are defined over a continuous state space in a dynamic environment.

Ultimately, the planning approach will have to handle new perceptions of the environment or possible evolutions of the UAV state, adapting the model and replanning when necessary in a real-time configuration. Considering only when the model and the goal do not change and for a finite-dimension discrete state, action and observation spaces, it is possible to compute a policy for every reachable belief state before the mission. However, it becomes a challenge to do so when having a continuous state space in a dynamic environment, due to the exponential complexity of the model.

Having that in consideration, the main objective of this work is to apply a generic planning-while-executing framework to deal with such a continuous state space problem in an real-time configuration onboard a UAV. In particular, to handle possible evolutions of the environment and of the UAV state during the mission execution, providing strictly anytime solutions. In such framework, a GNC transition model will be integrated in the decision-making process of the online planner, thereby allowing the propagation of the belief state according to the probabilistic availability of the onboard sensors.

1.3 State-of-the-art

1.3.1 Motion planning

The problem of motion planning has been widely discussed in the literature, and there are several studies to tackle the same problem from different perspectives. A number of classic text books describe motion planning with many details and under different conditions and constraints ([9], [10]). A key common concept in motion planning is a *configuration*. A configuration specifies positions of all the points in a system at a given time. Consequently, configuration space refers to the set of all possible configurations and is split into two subsets: *free space*, C_{free}, and *obstacle space*, C_{obs}. C_{obs} contains all the configurations where the robot intersects any obstacle, while C_{free} represents the set of configurations where the robot does not collide with any obstacle.

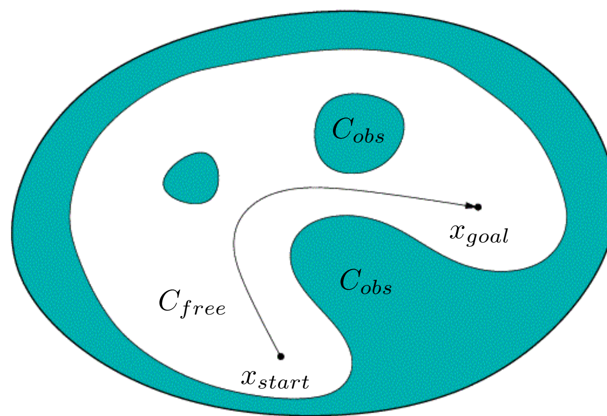


Figure 1.1: Generalized robotic path planning problem shown in a 2D configuration space. We wish to plan a path from x_{start} to x_{goal} through the free space C_{free} [10].

Many algorithms have been developed to fulfill the objective of finding the optimal motion plan. The way to achieve this objective varies greatly on how the environment is perceived and what is enclosed by the plan, which implies distinct characteristics to each algorithm [11]. Motion planners can be divided into two main categories depending on the input given: path planning or trajectory planning algorithms [12]. Path planning algorithms have the purpose of generating a geometric path (continuous curve), given an initial and a goal state, passing through a sequence of locations. On the other hand, trajectory planning algorithms take the solution from the robot path planning algorithm and determine how to move along such path, endowing it with time information.

Path planning

Path planning algorithms are usually divided according to the methodologies used to generate the geometric path. Most literature works focus only on 2D path planning algorithms [10], thus limiting the behavior of the vehicles to surface. Since the work here addressed considers aerial vechiles, the extensive survey on 3D path planning algorithms by Yang et al. [13] provide a more interesting view on the current literature. Following the taxonomy they proposed, five distinct categories are considered to divide path planning algorithms: sampling based, node based, mathematical model based, bioinspired

and multifusion based algorithms.

Sampling based algorithms require a general representation of the whole workspace and act by continuously sampling nodes from a continuous space within its frontiers. They rely on a sufficiently large number of samples to eventually find a solution (if there is one), avoiding building a complete configuration space. As such, their main concern lies with feasibility rather than optimality. These methods are further divided into active and passive planners. Passive planners generate a roadmap connecting the start and goal, but require a search algorithm to operate in the generated network to find the best feasible path, while active planners can achieve the best feasible path to the goal independently. Some of the most implemented families include passive algorithms like the Probabilistic RoadMap (PRM) and 3D Voronoi Diagrams, as well as active algorithms such as Rapidly-exploring Random Tree (RRT) and the Artificial Potential Field.

PRM [14] is a multi-query method that builds a roadmap by iteratively sampling random configurations in C_{free} and attempting to connect them. Once the roadmap is constructed, a node based algorithm is applied to compute the least cost path from the initial to the goal states. RRT, first developed by LaValle and Kuffner [15], aimed at solving path planning problems for nonholonomic (each state depending on the path taken to achieve it) and kinodynamic (considering both cinematic and dynamic constraints) systems. It explores the configuration space by rapidly growing a tree to generate a path connecting the start node and the goal node. Several extensions to the basic algorithm were developed, namely RRT*, which computes a better path than RRT by undergoing through a graph search when a new vertex is added to the tree, making it provably asymptotically optimal [16].

Artificial Potential methods, first introduced by Khatib [17], are based on the idea of assigning a potential function (relationship between C_{free} and C_{obs}) to C_{free} and simulating the vehicle as a particle subject to a force defined by a potential field. It computes the goal attraction and obstacle repulsion simultaneously, guiding the vehicle to move always along the total force gradient. However, such methods are incomplete because they are prone to drop into local minima area.

Node based optimal algorithms convert the problem of finding a path into a graph search problem. They work with a predetermined set of nodes that represent the environment and apply search algorithms, guaranteeing optimality and completeness with the information they are given.

One of the first approaches to solving search problems was the Dijkstra algorithm [18]. This method focuses on finding the shortest path in a graph where edges' weights are already known. Initially classified as uninformed, due to using no heuristic function, this algorithm was further improved in [19] to include such feature. Another approach, arguably the most widely used one, is the A* algorithm. This algorithm is an extension of Dijkstra's and reduces the total number of states by introducing a heuristic estimation of the cost from the current state to the goal state. The heuristic function can be designed to obtain the constraints, while the estimation must never overestimate the actual cost to reach the nearest goal node.

Mathematical model based algorithms compute models of the environment (kinematic constraints) and the system itself (dynamics constraints) and then bound the models with the cost function to achieve an optimal solution. As an example, [20] proposed Mixed-Integer Linear Programming (MILP) for path planning and obstacle avoidance problems for a wall-climbing robot in 3D environments.

Bioinspired algorithms mimic biological behaviours to solve path planning problems. These methods avoid computing complex environment models, choosing to apply stochastic approaches to search a near optimal path. For instance, Ant Colony Optimization (ACO) [21] imitates the behaviour of ants in finding the shortest path by using pheromone information.

Multifusion based algorithms combine several classes of algorithms together to achieve a better performance than when they are implemented on their own.

Optimal Path Planning

Optimal path planning algorithms are developed considering some optimality parameters or criteria such as minimum state error with or without uncertainty, execution time, energy (actuator effort), minimum jerk or hybrid approaches [12].

The methods presented in the previous sections can incorporate some kind of feature that measures costs or have some heuristic policy which, when associated to a cost function, constitute an optimization tool that must be minimized or maximized (depending on the problem) attaining an optimal path plan. For instance, applying the concept of cost to a certain action or state generates cost-based algorithms.

One of the first numerical methods for solving optimal control problems was the work of Bellman [22], in the 1950s. A major pillar of dynamic programming method is the Principle of Optimality defined by Bellman, which enunciates that an optimal path has the property that whatever the initial conditions and control variables (choices) over some initial period, the remaining decisions variables must constitute an optimal policy with regard to the state resulting from the first decision.

Planning under Uncertainty

When uncertainty is added to path planning, the problem changes, and many extra considerations need to be taken into account. As explained earlier in Section 1.2, there are various sources of non-determinism. Therefore, it is very beneficial to have planners that include uncertainty while solving problems.

Markov Decision Processes (MDPs) are one of the most popular planning tools in non-deterministic environments where the state of the system is observable. Even though background information about MDPs is provided in Chapter 2, we refer the readers to [23], [24] for comprehensive references.

A more complex approach uses the Partially Observable Markov Decision Processes (POMDPs) framework. POMDPs extend MDPs to include incomplete sensing of the state. This approach makes up the majority of the background for this dissertation and, as such, is thoroughly described in Chapter 2.

1.3.2 Offline and Online Planning

One of the key ingredients in making fully autonomous robots to act in real-time scenarios is the ability to select and organize actions over time in order to fulfil some high level objectives (e.g. maximizing reward functions, reaching symbolic system states). This implies reacting to potentially uncertain events that may occur due to the execution of the actions.

There are several approaches to face real-time scenarios in which an agent has to plan and execute a sequence of actions. Figure 1.2 illustrates four different existing paradigms. While the first two methods are usually implemented offline, the last two are implemented online, i.e. in real-time. Black line segments represent planning duration, while red line segments represent plan execution duration.

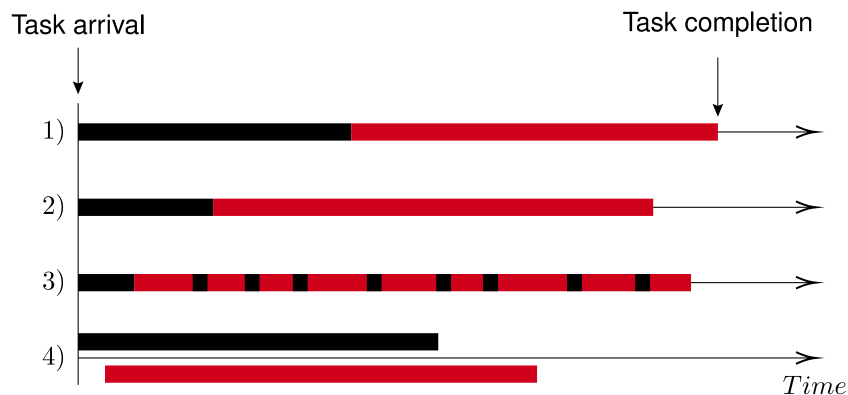


Figure 1.2: Four strategies for interleaving periods of planning (in black) and execution (in red) [25].

In method $1)$, the agent computes an optimal plan first and then executes it. While the execution time is minimized, the combined time of the planning and execution phases is substantial. In method $2)$, the agent computes a feasible but non-optimal plan first and then executes it. In this case, plan execution takes more time, but the combined cost is reduced. In method $3)$, the agent performs a limited amount of search that leads to an incomplete plan and then executes it until further planning is required, repeating this cycle of interleaving planning and execution phases until the task is completed. The resulting combined cost can be lower or higher compared to the previous ones, depending on the application domain and online planning technique.

Finally, method $4)$ proposes to perform planning and execution concurrently. As such, it requires a shared access to the current partial plan with proper synchronization mechanisms to guarantee the correctness and integrity of each process. The benefit of such a continuous planning paradigm is that it can fully exploit execution time for planning, reducing the overall cost of the solution [25].

Several applications in robotics involve time-constrained interleaving of planning and execution phases (method $3)$) [1] [2] [26]. In these cases, as time goes on, the actions that were planned few seconds ago might no longer be suitable nor feasible in the current situation, due to the increasing discrepancy between the expected situation and the one observed, forcing the agent to continuously change its plan. Therefore, real-time planning demand a tight coordination between the predicted evolution of the environment and the time required to generate plans as to maximize the chance that planned actions are valid in the states where they were expected to be executed. Thus continuous planning methods emerge

as an interesting option to tackle the problem [25] [27].

Moreover, when facing real-time critical applications, the autonomous system is required to provide some basic guarantees to ensure the safety of the mission, such as *anytime* solutions, i.e. guarantee an executable action for any time threshold specified. Anytime algorithms quickly build an applicable sub-optimal (feasible) solution and then refine it if more time is available. A particular case of anytime planning algorithms consist in the online planning algorithms [28] [29] [26]. However, the authors in [27] claim that most online approaches are not strictly anytime in the sense that the execution of their continually improved policy is not guaranteed to succeed under strict time constraints. Indeed, the replanning requests are usually performed when a planned action is not applicable in the current state and most of these algorithms do not take into account if the replanning episode will end before the given deadline is reached nor do they account for their own computational time as a cost to optimize along with the action costs.

Recent theoretical advances have been made to intimately reason about the evolution of the environment and the computational time of the planning algorithms [30]. [25] also propose a planning-while-executing framework for stochastic shortest path problems, where they exploit prior knowledge of the time it takes to execute any action directly in the cost function. In this way, the combined cost of planning and plan execution is minimized. However, they still cannot ensure valid executable actions when the execution engine requests them.

To address this issue, Chanel et al. [27] proposed an anytime generic planner for execution-driven planning in robotics, called AMPLE – Anytime Meta PLannEr. This framework will be implemented in the present dissertation and as such is thoroughly described in Chapter 5.

While plan execution frameworks in literature exploit individual and specific methods and languages that are not widely adopted standards, [31] proposes ROSplan, a framework for linking task-planning with an execution interface provided by ROS. However, the drawback with this architecture is that only temporal planners able to interpret PDDL2.1, a planning domain description language, can be used.

Many successful planning frameworks for autonomous systems also explore scheduling a set of parallel actions, i.e. concurrently executed, so as to minimize the total execution time or to maximize the number of executed actions. For instance, [32] propose EUROPA, which is an open-source framework for planning and scheduling within a constraint-based temporal planning paradigm.

1.4 Objectives and contributions

As stated previously, the work elaborated within this dissertation is integrated in a research project, called NavPlan. This project aims to study in the continuous state space models to address the planning problem of the navigation and the guidance strategy for autonomous UAVs that evolve in cluttered environments. The resulting online planning framework will run onboard the UAV and will have to adapt and replan when necessary, for instance when considering a new goal state, or upon new perception of the environment, or when the UAV state estimation during its flight no longer matches with the model.

Figure 1.3 illustrates a diagram concerning two different phases of the project: the previous devel-

oped work (in black) and the objectives set out for the present dissertation (in red).

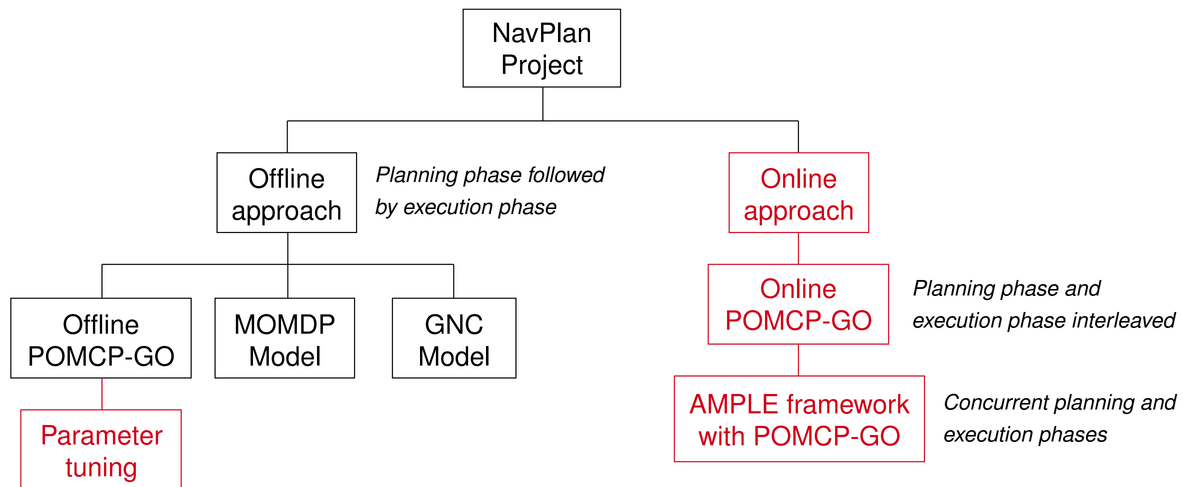


Figure 1.3: Organization of the NavPlan research project.

The previous work comprises the problem formulation based on a Mixed-Observability Markov Decision Process (MOMDP) model [33] [34], which is an extension of the POMDP framework, in order to compute a safe and efficient trajectory from an initial to a goal state, in a partial sensing environment with motion uncertainty; a GNC model to be incorporated in the planning algorithm to allow the propagation of the sensor's availability in the state transition and an offline planning algorithm, POMCP-GO, to solve the MOMDP planning problem.

In order to achieve the main objective of the project, this dissertation has secondary goals:

1. develop strategies for parameter tuning in the offline planning algorithm proposed in the previous work;

2. convert the offline planning algorithm into an online configuration, which is an approach more adaptable to real flight application cases;

3. integrate the online planning algorithm in a concurrent planning-while-executing framework, to be developed in Robotic Operating System (ROS);

4. apply the resulting framework in a robot simulator, Gazebo, in order to evaluate its performance.

Contributions

This dissertation's contribution consists in the implementation of a state-of-the-art online path planning algorithm in a planning-while-executing framework, thereby enhancing the state-of-the-art of path planning frameworks applied to autonomous navigation problems.

A paper submission to the ECAI Conference 2020, in the Prestigious Applications of Intelligence Systems (PAIS) track, is currently under preparation.

1.5 Thesis Outline

Seeking the achievements of the stated objectives, this dissertation entails the mathematical context and the merging process of different algorithms in 6 chapters, as follows:

Chapter 1 Introduction – presents the need for developing an algorithm which computes a path for a UAV facing environmental uncertainty and partial sensing while simultaneously considering a priori knowledge on sensor availability. It also comprises a short review of the state-of-the-art on motion planning and existing frameworks;

Chapter 2 Background – is devoted to introduce Markov Decision Processes and some of its extensions, namely POMDPs and MOMDPs. State-of-the-art algorithms to solve problems using the POMDP framework are mentioned, along with their advantages and disadvantages. Additionally, Monte Carlo methods and their application to POMDPs are elaborated, namely the POMCP algorithm;

Chapter 3 Problem formulation – focuses in previous work developed in the scope of the NavPlan problem. It comprises the GNC model proposed and formalizes the problem addressed as a MOMDP;

Chapter 4 Offline approach – presents the planning algorithm to solve the MOMDP problem in an offline configuration and our contribution on strategies for parameter tuning;

Chapter 5 Online approach – leads the reader through the online approach, by proposing an online version of the planning algorithm and explaining the planning-while-executing framework in which the planner is integrated. Simulation results illustrate the advantages and disadvantages of the proposed approach. Furthermore, the performance of this framework is also tested in the robot simulator Gazebo.

Chapter 6 Conclusions – addresses the final remarks and further work directions.

Chapter 2

Background

This chapter leads the reader from the challenge of decision-making under uncertainty in fully observable environments to partially observable ones, through the presentation of approximate solving methods that derive near optimal behaviour for an agent, both offline and online.

Section 2.1 introduces mathematical notations for environments with complete observability as Markov Decision Processes (MDPs), while Section 2.2 describes a particular subclass of goal-oriented planning problems in fully observable environments. Afterwards, partially observable environments are introduced in Section 2.3, with Partially Observable Markov Decision Processes (POMDPs) constituting the mathematical model for such environments. There are two classes of algorithms that construct policies for POMDPs: offline and online planning algorithms. Both classes are addressed in Section 2.4, with the addition of a subclass of POMDPs, the Mixed-Observability Markov Decision Processes (MOMDPs) that eases the process of solving POMDPs. Monte Carlo methods, a family of online planning algorithms, range from simple rollout methods to tree search methods such as Monte Carlo Tree Search (MCTS), and are elaborated on in Section 2.5. The extensions of MCTS to POMDPs are also discussed in that section.

2.1 Markov Decision Processes

The task of controlling an autonomous agent can be seen as a sequential decision-making problem under uncertainty. That is, the system must decide which actions to execute to achieve a desired performance, even when there is uncertainty about the world around it and what will happen in the future. Consider now an agent that perceives its environment through a number of onboard sensors and tries to reach a goal which may require several sequential actions to accomplish. As long as the sensors provide the complete state of the environment, such sequential decision problems can be approached by a number of reinforcement learning and planning algorithms [35]. Moreover, these planning problems are typically formulated as an instance of a Markov Decision Process or an MDP.

The Markov Decision Process (MDP) [23] is a mathematical formalism that can represent a wide range of sequential decision-making problems, in which the environment is fully observable. Such for-

malism is able to represent outcome uncertainty with a stochastic state transition model.

Formally, an MDP is defined by a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, s_0)$, where:

- $\mathcal{S}$ is the state space, representing the set of all possible states of the system;
- $\mathcal{A}$ denotes the action space, i.e. the set of all actions available to the agent;
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, represents the transition function between states.

$$\mathcal{T}(s, a, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a), \forall t, \forall a_t \in \mathcal{A}, \forall s_t \in \mathcal{S} \text{ and } \forall s_{t+1} \in \mathcal{S}$$

  is defined as the probability that the system will transition to state s', given that action a is performed in state s. Since $\mathcal{T}$ is a conditional probability defined for every triplet (s, a, s'), $\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') = 1, \forall (s, a)$;
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, giving the expected immediate reward gained by the agent for taking each action in each state. In particular, $\mathcal{R}(s, a)$ represents the expected reward for executing action a in state s;
- s_0 is the initial state of the system.

In this model, the mappings $\mathcal{T}$ and $\mathcal{R}$ depend only on the previous state and the action taken; even if additional previous states were to be considered, the transition probabilities and the expected rewards would remain the same. This independence assumption is known as the *first order Markov property* i.e., the state and reward at time $t + 1$ is dependent only on the state and the action at time t [35]. In the graph represented in Figure 2.1, the nodes correspond to state and action variables distributed along two successive instances in time, and the arrows represent probabilistic dependencies between the variables.

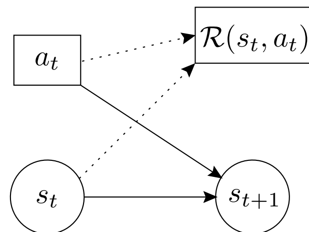


Figure 2.1: Transition model of an MDP, in which an agent in state s_t performs an action a_t. After doing so, he finds himself in state s_{t+1} with a reward $\mathcal{R}(s_t, a_t)$.

Due to the uncertainty on the actions' outcomes, an agent is not able to control the evolution of the environment. Instead, he can only influence the decision-making process using a set of actions, for which he collects rewards (or pays costs). For this reason, the aim of solving an MDP is to develop a strategy that allows to choose the action to be performed for all possible situations (states) it could encounter, in order to guarantee robustness in the face of uncertainty. Such strategy is called *policy* $\pi : \mathcal{S} \to \mathcal{A}$, assigning an action $a \in \mathcal{A}$ to each state $s \in \mathcal{S}$.

This strategy should optimize some numerical criterion, generally defined as the total amount of rewards gathered (or costs payed) when successfully applying the policy $\pi(s)$ from the beginning of the

decision process over a finite or infinite discrete time horizon. Considering a *finite-horizon* optimality criterion,

$$\mathbb{E}\left[\sum_{t=0}^{k-1} \mathcal{R}_t\right],$$

where $\mathcal{R}_t$ is the reward received at the time step t and $\mathbb{E}(\cdot)$ denotes expectation, the agent's goal is to maximize this expected long-term reward over the horizon length k. This criterion induces a non-stationary policy, i.e., the choice of action depends not only on the state, but also on the time step.

On the other hand, the *infinite-horizon* optimality criterion,

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}_t\right],$$

applies a discount factor $\gamma \in [0, 1)$ in order to guarantee convergence over the horizon. As the value of this parameter decreases, the significance of rewards received at a later point is gradually reduced, thereby reducing their influence on the choice of actions at an earlier stage. Here, the optimality criterion becomes the maximization of the expected sum of discounted rewards (also called the return or discounted return). As no time horizon is imposed, this criterion induces a stationary policy.

The cumulative expected discounted rewards obtained by following a specific policy π, from a certain state s, is defined by the value function equation $V^\pi(s)$, as follows:

$$V^\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t \, \mathcal{R}(s_t, \pi(s_t)) \middle| \, s_0 = s\right] \tag{2.1}$$

By opening the sum in Equation 2.1, a Bellman's equation can be defined over the value function [23], such as:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi\left[\mathcal{R}(s_0, \pi(s_0)) | s_0 = s\right] + \mathbb{E}_\pi\left[\sum_{t=1}^{\infty} \gamma^t \mathcal{R}(s_t, \pi(s_t)) \middle| \, s_1 = s\right] \\ &= \sum_{s' \in \mathcal{S}} \Pr(s'|s, \pi(s)) \left(\mathcal{R}(s, \pi(s)) + \gamma \, \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, \pi(s_t)) \middle| \, s_0 = s'\right]\right) \\ &= \sum_{s' \in \mathcal{S}} \Pr(s'|s, \pi(s)) \left[\mathcal{R}(s, \pi(s)) + \gamma \, V^\pi(s')\right] \end{aligned} \tag{2.2}$$

A policy π that maximizes the value function $V^\pi(s)$ is called *optimal policy* π^*. It specifies, for each state s, the optimal action to execute, assuming that the optimal policy is followed at every step after this action is taken. It has been proven that, for a policy of any structure, e.g. that is stochastic or that depends on more than just the state, there exists a Markov policy that can attain the same objective value. Consequently, it is always possible to find a deterministic Markov policy that achieves the optimal objective value [36].

The optimal value function $V^*(s)$ is defined as follows:

$$\begin{aligned} V^*(s) &= \max_{\pi} V^{\pi}(s) \\ &= \max_{\pi} \left\{ \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, \pi(s_t)) \middle| s_0 = s \right] \right\} \end{aligned} \tag{2.3}$$

It can be demonstrated that the optimal value function satisfies the Bellman's equation, $V^* = \mathcal{L}V^*$, where the operator $\mathcal{L}$ is the Bellman's dynamic operator [23] [35]. Since the convergence of the sums at an infinite horizon is ensured by the discount factor $\gamma \in [0, 1)$, one has, for all policies $\pi = (a, \pi')$:

$$\begin{aligned} V^*(s) &= \max_{\pi} \left\{ \sum_{s' \in \mathcal{S}} \Pr(s'|s, \pi(s)) \left[\mathcal{R}(s, \pi(s)) + \gamma V^{\pi}(s') \right] \right\} \\ &= \max_{(a,\pi')} \left\{ \sum_{s' \in \mathcal{S}} \Pr(s'|s, a) \left[\mathcal{R}(s, a) + \gamma V^{\pi'}(s') \right] \right\} \\ &= \max_{a \in \mathcal{A}} \left\{ \sum_{s' \in \mathcal{S}} \Pr(s'|s, a) \left[\mathcal{R}(s, a) + \gamma \max_{\pi'} V^{\pi'}(s') \right] \right\} \\ &= \max_{a \in \mathcal{A}} \left\{ \sum_{s' \in \mathcal{S}} \Pr(s'|s, a) \left[\mathcal{R}(s, a) + \gamma V^*(s') \right] \right\} \end{aligned} \tag{2.4}$$

When Equation 2.4 converges for all $s \in \mathcal{S}$, then the solution is guaranteed to be optimal.

In order to calculate the optimal policy, a method based on dynamic programming is applied [37], which relies on the Bellman equation that characterizes the optimal value function and consists of converging the value progressively towards the fixed point of Equation 2.4 for each state, through repeated application of the Bellman operator $\mathcal{L}$.

Let V be a value function with the following initialization:

$$V_0(s) = \max_{a \in \mathcal{A}} \{ \mathcal{R}(s, a) \} \tag{2.5}$$

Then the value function for each instant t, $V_t(s)$, can be computed based on the value function for successor values from the previous iteration V_{t-1}, according to its recursive propriety [35]. Therefore, $V_t(s)$ can be given as:

$$V_t(s) \leftarrow \max_{a \in \mathcal{A}} \left[\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \Pr(s'|s, a) V_{t-1}(s') \right] \tag{2.6}$$

When $\mathcal{L}$ is applied iteratively, i.e. $V_{n+1} = \mathcal{L}[V_n]$, the value function will eventually reach a fixed point when $V_{n+1} = V_n$. This point is the optimal value function and the process of finding the value function in this manner is known as Value Iteration [23] [24].

Once the optimal value function has been found, the optimal policy π^* may be easily extracted according to:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \left[\mathcal{R}(s_t, \pi(s_t) = a) + \gamma \sum_{s' \in \mathcal{S}} \Pr(s'|s, a) V^*(s') \right] \tag{2.7}$$

2.2 Stochastic Shortest-Path MDP

Finite-horizon and infinite-horizon discounted MDPs are located at the opposite ends of a spectrum $-$ the former assume the number of time steps to be limited and known; the latter assume it to be infinite. Naturally a third possibility comes along: problems with finite but unknown horizon, i.e. indefinite-horizon optimality criterion [35] [38].

The subclass of decision-making problems that assumes this third possibility is called Stochastic Shortest-Path MDP (SSP MDP). In this case, there is a set of target states and the goal is to minimize the expected total cost until the target set is reached. The optimality criterion is not discounted-sum, but a total sum without discounts. As there is no a priori bound to reach the target set (a target can be reached at different times along different paths), it is also not a finite-horizon objective.

A SSP MDP is defined by a 6-tuple $(\mathcal{S}, \mathcal{G}, \mathcal{A}, \mathcal{T}, \mathcal{C}, s_0)$, where:

- $\mathcal{S}$ is the state space, representing the set of all possible discrete states of the system;
- $\mathcal{G}$ is a non-empty set of discrete target (or goal) states , where $\mathcal{G} \subseteq \mathcal{S}$;
- $\mathcal{A}$ denotes the action space, i.e. the set of all actions available to the agent;
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, represents the transition function between states, similarly to an MDP. In addition, $\Pr(s, a, s) = 1$ for each $s \in \mathcal{G}$ and all $a \in \mathcal{A}$, meaning target states are absorbing;
- $\mathcal{C} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_0^+$ is the cost function that returns the immediate positive cost $\mathcal{C}(s, a) > 0$ payed by the agent for taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S} \setminus \mathcal{G}$ and $\mathcal{C}(s, a) = 0$ for each $s \in \mathcal{G}$ and all $a \in \mathcal{A}$, i.e., target states are cost-free;
- s_0 is the initial state of the system.

Taking this information in consideration, solving a SSP MDP means finding an expected cost-minimizing, as opposed to a reward-maximizing, policy [35]. The value function equation $V^{\pi}(b)$ is given as:

$$V^{\pi}(s) = \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \mathcal{C}(s_t, \pi(s_t)) \middle| s_0 = s \right] \tag{2.8}$$

And the optimal value function $V^*(s)$ satisfies, for all $s \in \mathcal{S}$:

$$\begin{aligned} V^*(s) &= \min_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \Pr(s'|s, a) \left[\mathcal{C}(s, a) + V^*(s') \right] \right] \\ &= \min_{a \in \mathcal{A}} \left[\mathcal{C}(s, a) + \sum_{s' \in \mathcal{S}} \Pr(s'|s, a) V^*(s') \right] \end{aligned} \tag{2.9}$$

The largest differences from the classical version of the MDP (Equations 2.1 and 2.3) are the replacement of a maximization problem with a minimization problem, the replacement of a reward function with a cost function, the definition of goal states as being absorbing states, and the omitted γ factor, since it is fixed at 1.

To be noted that both MDP and SSP MDP assume the environment to be fully observable, i.e., the onboard sensors are perfect. In many domains, however, sensors do not receive complete information

about the state of the environment and its perception becomes partial and imperfect. In this sense, an agent found in an environment, which is not fully observable and the outcome of its actions is not completely certain, will have to rely on the observations it receives from the environment in order to make decisions. This agent-environment interaction can be modelled in terms of a Partially Observable Markov Decision Process (POMDP), which is introduced in the next section.

2.3 Partially Observable Markov Decision Processes

A Partially Observable Markov Decision Process (POMDP) is similar to an MDP, except that the agent cannot directly observe the state. Instead, it only has access to observations that give incomplete information about the current states. In this way, a POMDP is able to represent state uncertainty in addition to the outcome uncertainty that can be encoded in an MDP.

A POMDP can be defined as a 7-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, b_0)$, where:

– $\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}$ have the same meaning as in an MDP;

– Ω is the set of all possible observations;

– $\mathcal{O} : \mathcal{S} \times \mathcal{A} \times \Omega \to [0, 1]$ is the observation function.

$$\mathcal{O}(s', a, o) = \Pr(o_{t+1} = o | s_{t+1} = s', a_t = a), \forall t, \forall o_{t+1} \in \Omega, \forall a_t \in \mathcal{A} \text{ and } \forall s_{t+1} \in \mathcal{S}$$

is defined as the probability of observing o if action a is performed and the resulting state is s'. This conditional probability is defined for every triplet (s', a, o) and, therefore, $\sum_{o \in \Omega} \mathcal{O}(s', a, o) = 1, \forall (s', a)$;

– b_0 represents the agent's initial probability distribution over the states, also known as belief state.

The transition model of a POMDP is represented by Figure 2.2, where the nodes correspond to state s, action a and observation o variables distributed along two successive instances in time, and the arrows represent probabilistic dependencies between the variables.

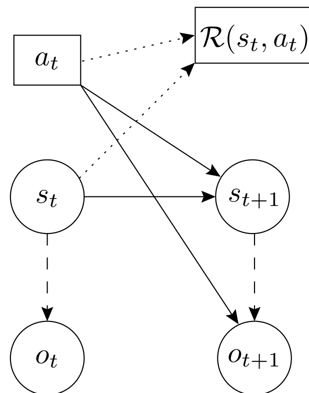


Figure 2.2: Transition model of a POMDP.

2.3.1 Belief State

In environments where the state is not directly observable, the agent needs to reason over the complete history of actions and observations up to the current time, in order to decide which action to perform. Such history, at time step t, is defined as

$$h_t = \{a_0, o_1, ..., o_{t-1}, a_{t-1}, o_t\}.$$

As the action to execute is among $|\mathcal{A}|$ actions and the observation to be perceived in the new state is in the set of $|\Omega|$ observations, the number of possible histories grows exponentially with the size t of the history as $(|\mathcal{A}||\Omega|)^t$. Therefore, rather than resorting to this explicit and memory expensive representation, one may summarize all relevant past information in a probability distribution over the state space $\mathcal{S}$, which is denoted as the belief state b. It is defined such that $\sum_s b(s) = 1$, where $b(s)$ is the probability assigned to the particular state s in the belief state b.

The agent can compute its current belief state as the posterior probability distribution of being in each state, given the complete history:

$$b_t \in \mathcal{B}, \qquad b_t(s) = \Pr(s_t = s | h_t = h), \forall s \in \mathcal{S} \tag{2.10}$$

where $\mathcal{B}$ forms the set of all possible belief states.

An important feature of the belief state b_t is that it respects the Markov property, making it a sufficient statistic for the complete past history of the process h_t [37]. Therefore, the agent can choose its actions based on the current belief state $b(s)$ and no additional past information can improve the knowledge about the current state of the world.

At any time t, given the current belief state $b(s)$, an action a executed and an observation o perceived by the agent, the belief state at $t+1$ can be updated according to Bayes' rule [37] as:

$$\begin{aligned} b_o^a(s') &= \Pr(s'|b,a,o) \\ &= \frac{\Pr(o,s'|b,a)}{\Pr(o|b,a)} \\ &= \frac{\Pr(o|s',a)\Pr(s'|b,a)}{\Pr(o|b,a)} \\ &= \frac{\Pr(o|s',a)\Pr(s'|b,a)}{\sum_{s'\in\mathcal{S}}\sum_{s\in\mathcal{S}}\Pr(o|b,a,s,s')\Pr(s,s'|b,a)} \\ &= \frac{\Pr(o|s',a)\sum_{s\in\mathcal{S}}\Pr(s'|s,a)b(s)}{\sum_{s'\in\mathcal{S}}\Pr(o|s',a)\sum_{s\in\mathcal{S}}\Pr(s'|s,a)b(s)} \end{aligned} \tag{2.11}$$

The size of the belief space is illustrated best by an example. Consider a 2×2 grid world with its 4 discrete states. The belief is a 4-dimensional probability mass function and to update it the agent needs

to consider the transition probabilities of all possible states and the probability of being in each of these states. While the computations necessary to update the belief state can still be performed in a relatively small amount of time for problems with a small state space, computing the update is hard for problems with thousands or millions of states. In addition, it might not even be possible to represent the transition or observation probabilities in a compact form. This problem will be further explored in Section 2.4.

2.3.2 Policy and value function

As in an MDP, the aim of solving a POMDP is to find a policy π^* that optimizes the value function V^π. Since the environment in a POMDP is not fully observable, the policy is no longer defined over the states, but rather over the belief state b as $\pi : \mathcal{B} \to \mathcal{A}$, assigning an action $a \in \mathcal{A}$ to each belief $b \in \mathcal{B}$.

The value function $V^\pi(b)$ of a discounted reward-based POMDP can be defined as the cumulative expected discounted rewards obtained by following a specific policy π, from a certain belief state b, as follows:

$$V^\pi(b) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \, \mathcal{R}(b_t, \pi(b_t)) \middle| \, b_0 = b \right] \tag{2.12}$$

where $\mathcal{R}(b_t, \pi(b_t) = a) = \sum_{s \in \mathcal{S}} \mathcal{R}(s, a) b_t(s)$ is the expected reward of executing an action a in belief state b for the discrete state space.

Similarly to what has been done in Section 2.1, by opening the sum in Equation 2.12, a Bellman's equation can be defined over the value function, such as:

$$\begin{aligned} V^\pi(b) &= \mathbb{E}_\pi \left[\mathcal{R}(b_0, \pi(b_0)) | b_0 = b \right] + \mathbb{E}_\pi \left[\sum_{t=1}^{\infty} \gamma^t \mathcal{R}(b_t, \pi(b_t)) \middle| \, b_1 = b_\pi \right] \\ &= \sum_{s \in \mathcal{S}} \mathcal{R}(s, \pi(b)) b(s) + \gamma \sum_{o \in \Omega} \Pr(o | b, \pi) \, \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(b_t, \pi(b_t)) \middle| \, b_0 = b_\pi^o \right] \\ &= \sum_{s \in \mathcal{S}} \mathcal{R}(s, \pi(b)) b(s) + \gamma \sum_{o \in \Omega} \Pr(o | b, \pi) V^\pi(b_\pi^o) \end{aligned} \tag{2.13}$$

The optimal value function $V^*(b)$ becomes:

$$V^*(b) = \max_{a \in \mathcal{A}} \left\{ \sum_{s \in \mathcal{S}} \mathcal{R}(s, a) b(s) + \gamma \sum_{o \in \Omega} \Pr(o | b, a) V^*(b_a^o) \right\} \tag{2.14}$$

where b_a^o is the belief state reached by performing action a and perceiving observation o.

And the optimal policy is extracted according to:

$$\pi^*(b) = \arg\max_{a \in \mathcal{A}} \left[\sum_{s \in \mathcal{S}} \mathcal{R}(s, a) b(s) + \gamma \sum_{o \in \Omega} \Pr(o | b, a) V^*(b_a^o) \right] \tag{2.15}$$

Equation 2.7 associates an action to a specific belief state, i.e. $\pi : b \to a$, and therefore must be computed for all possible belief states in order to define a full policy, which can be an intractable problem.

Another important notion inherent to POMDPs is the action-value function, or Q-value function

$Q(b, a)$, defined by:

$$Q^{\pi}(b, a) = \sum_{s \in \mathcal{S}} \mathcal{R}(s, a) b(s) + \gamma \sum_{o \in \Omega} \Pr(o|a, b) V^{\pi}(b_a^o) \tag{2.16}$$

Equation 2.16 determines the expected value when executing action a in belief state b, assuming that the optimal policy is followed at every step after this action is taken. The first term specifies the expected immediate reward of action a in belief state b and the second term specifies the summed value of all successor belief states, weighted by the probability of observing o in those belief states.

2.3.3 Extension to continuous state space

Several planning problems, especially navigation problems, are modelled as POMDPs with a continuous state space [39], [40]. In this case, the belief state update, considered initially in Equation 2.11, uses integrals instead of sums, as follows:

$$b_o^a(s') = \frac{\Pr(o|s', a) \int_{s \in \mathcal{S}} \Pr(s'|s, a) b(s)\, ds}{\int_{s' \in \mathcal{S}} \Pr(o|s', a) \int_{s \in \mathcal{S}} \Pr(s'|s, a) b(s)\, ds\, ds'} \tag{2.17}$$

Similarly, the optimal value function $V^*(b)$ becomes:

$$V^*(b) = \max_{a \in \mathcal{A}} \left\{ \int_{s \in \mathcal{S}} \mathcal{R}(s, a) b(s)\, ds + \gamma \sum_{o \in \Omega} \Pr(o|b, a) V^*(b_a^o)\, do \right\} \tag{2.18}$$

Equation 2.18 considers a continuous state space, while the observation space remains discrete. Recent work has shown the POMDP definition for continuous action spaces [41] and continuous observation spaces [40], including algorithms to solve it.

2.4 Solving POMDPs

POMDPs provide a general framework for planning in partially observable stochastic environments. Nevertheless, only few POMDP planning problems can be solved exactly due to their computational complexity. In fact, POMDP planning is computationally intractable in the worst case: infinite-horizon POMDPs [42]. The challenges arise from three main sources:

1. Most real world applications have extremely large state and observation spaces;

2. As the state is not fully observable, the agent must reason with beliefs, which are probability distributions over the states. Since a belief state has to be maintained for an estimate of the agent's state and has to be updated every time the agent performs an action, the update process becomes computationally expensive as the size of the state space increases. Consider that there are N unique possible states of the system, then the belief state will be an N dimensional vector of continuous values. As the value N gets larger, the planning problem becomes cumbersome.

3. The number of action-observation histories that must be considered for POMDP planning grows exponentially with the planning horizon, as well as with the state and observation spaces. A common approach followed by POMDP solvers is a tree search procedure to explore combinations of actions and observations. Exploring all these combinations causes the search tree to grow wider as a consequence of a large branching factor. The planning algorithm, thus, has to evaluate all nodes in the search tree so as to derive an optimal solution and a large branching factor in trees demands voluminous computations.

The first two difficulties are usually referred to as the *curse of dimensionality*, and the last one, the *curse of history* [35] [43].

Despite this issue, the value function denotes a mathematical property that simplifies the computation of the optimal value function for a finite-horizon POMDP. In fact, a key result by [37] shows that the value function is piecewise linear and convex (PWLC), which allows to represent it as a set of values over the belief space, often called α-vectors. A number of exact value function algorithms exploit this property, namely Policy or Value Iteration algorithms. The remaining problem with these approaches is that the number of α-vectors needed to represent the value function grows exponentially in the number of observations at each iteration (inherit the curse of dimensionality), turning out to be computationally very expensive.

Due to the high complexity of exact solving approaches, most of the recent research in the area has focused on developing approximate algorithms that can be applied to larger POMDPs. Based on time budget, these approaches can be classified into two categories: offline solvers and online solvers.

2.4.1 Offline solvers

Offline algorithms attempt to find approximate solutions for POMDP problems offline, in the sense that they specify, prior to the execution, the best action to execute for all possible situations. This approach has achieved dramatic progress in computing near-optimal policies, particularly point-based algorithms.

The idea behind point-based algorithms is to sample a representative finite set of attainable belief states Δ, where $\Delta \subset \mathcal{B}$, and then update the value function and its gradient over the sampled subset, rather than the entire belief state space $\mathcal{B}$, achieving belief space dimensionality reduction. Several algorithms exploit this approach, namely PBVI [44], PERSEUS [45], HSVI [46] and SARSOP [47], differing slightly in how they choose belief states and how they update the value function with respect to the sampled belief states.

Nevertheless, successful application of offline POMDP solvers to real world problems has been limited in scalability, because the number of future events grows exponentially with the planning horizon and, consequently, the computation time required to calculate a policy over every possible belief state increases significantly. Moreover, small changes in the environment's dynamics require recomputing the full policy and when the environment is largely unknown, this approach constructs POMDP models too heavy to be solved by even the best offline solver today. To overcome this matter, online POMDP solvers

were developed and are explored in the next section.

2.4.2 Online solvers

Online planning algorithms approach POMDPs from a completely different angle than offline algorithms. Whereas an offline search would compute an exponentially large contingency plan considering all possible happenings, an online search plans only for the current belief state, limiting the set of belief states reachable from the current belief state to a time computational budget.

In this setting, a planning phase and an execution phase are applied alternately at each time step. In the planning phase, the agent's current belief state is passed on to the algorithm and it searches for a single best action to execute in that belief only. In the execution phase, this action is executed in the environment and the agent's current belief state is updated. Interleaving these two steps enables online algorithms to adapt faster to a dynamic environment, given the smaller number of reachable belief states considered due to the time budget.

One drawback of online planning is that it generally needs to meet hard real-time constraints, thus greatly reducing the available planning time, compared to offline approaches. Nevertheless, further developments [29], [48], [28] show that online search and offline policy computation are complementary and can be combined. For instance, using approximate or partial policies computed offline as heuristic functions to guide the online search algorithm. This combination enables online search algorithms to plan on shorter horizons, thereby respecting online real-time constraints and retaining a good precision.

Ross et al. [28] provide an extensive survey on many existing online planning methods and a general framework for online algorithms that use an AND/OR-tree during the planning phase. The authors relate this framework to a set of existing online planning algorithms which apply Branch-and-Bound pruning [49], Heuristic Search (e.g., AEMS [29], [50]) and Monte Carlo sampling [26] to improve the search.

In the Branch-and-Bound pruning method, upper and lower bounds on the value $Q^*(b, a)$ of each action a, for every belief b in the tree, are computed offline for the fringe nodes, and are propagated up the tree during search. Then, nodes that have upper bounds lower than lower bounds of other nodes are pruned out, thus limiting the tree width in the context of actions.

Heuristic Search, instead of reducing the branching factor, try to focus the search on the most relevant reachable beliefs by using heuristics to select the best fringe belief node to expand. Thereby allowing the search algorithm to make good decisions by expanding as few nodes as possible, improving the performance of the agent.

Finally, Monte Carlo sampling methods break both curses mentioned in Section 2.4, by randomly sampling states from the belief state and by sampling histories using a black-box simulator. These methods are introduced and further explored in Section 2.5, as the main algorithm used to solve the POMDP problem in the present work, Partially Observable Monte Carlo Planning (POMCP), derives from this concept (Section 2.5.7).

Determinized Sparse Partially Observable Tree (DESPOT) is a similar approach to the POMCP algorithm, that attempts to achieve better performance by analysing only a small number of random out-

comes in the tree [48]. Planning under Uncertainty with Macro-Actions (PUMA) [51] and Randomized Belief-Space Replanning (RBSR) [52] perform best first search in the belief space, just as POMCP does. However, instead of solving a Multi-Armed Bandit problem, these approaches sample action sequences using heuristics in the state space, assuming the states are fully observable after an action is performed.

Additionally, an online algorithm called Adaptive Belief Tree (ABT) was designed specifically to accommodate changes in the environment without having to replan from scratch [53]. This algorithm addresses the issue of the significant waste of computational resources inherent to replanning in most POMDP online solvers, by proposing to reuse and improve an existing policy at each time step, updating it as needed whenever the POMDP model changes. Some very recent methods, e.g. QMDP-Net [54], have attempted to solve POMDPs by training recurrent neural networks.

2.4.3 Mixed-Observability Markov Decision Processes

An extension of the POMDP model, proposed by [33] and [34], is a model with mixed observability designated as Mixed-Observability Markov Decision Process (MOMDP). MOMDPs explore a particular structure where certain state variables are fully known at each epoch. Indeed MOMDPs propose a middle-ground scenario between MDPs and POMDPs, considering state variables that can be directly observed, namely *visible variables*, as well as *partially observable variables*, in the same model. This approach follows a similar idea to factored POMDPs [55] [35], that present the state s as a vector of variables in a view to exploit the underlying structure of the problem.

Formally a MOMDP model is specified as an 9-tuple $(\mathcal{S}_v, \mathcal{S}_p, \mathcal{A}, \mathcal{T}_{\mathcal{S}_v}, \mathcal{T}_{\mathcal{S}_p}, \mathcal{R}, \Omega, \mathcal{O}, \left(s_v^0, b_{\mathcal{S}_p}^0\right))$. $\mathcal{S}_v$ represent the set of fully observable state variables s_v, while $\mathcal{S}_p$ represent the set of partially observable state variables s_p. Thus the complete state space is described by (s_v, s_p) such that $|\mathcal{S}| = |\mathcal{S}_v| \times |\mathcal{S}_p|$ represents the complete domain of the state space. The observation and reward functions remain the same as in POMDP, but can be expressed in terms of the two sets of state variables: $\mathcal{O}(s', a, o) = \mathcal{O}(s'_v, s'_p, a, o)$ and $\mathcal{R}(s, a) = \mathcal{R}(s_v, s_p, a)$, respectively.

Instead of a single transition probability $\mathcal{T}$, the MOMDP considers the transition probabilities for each component separately:

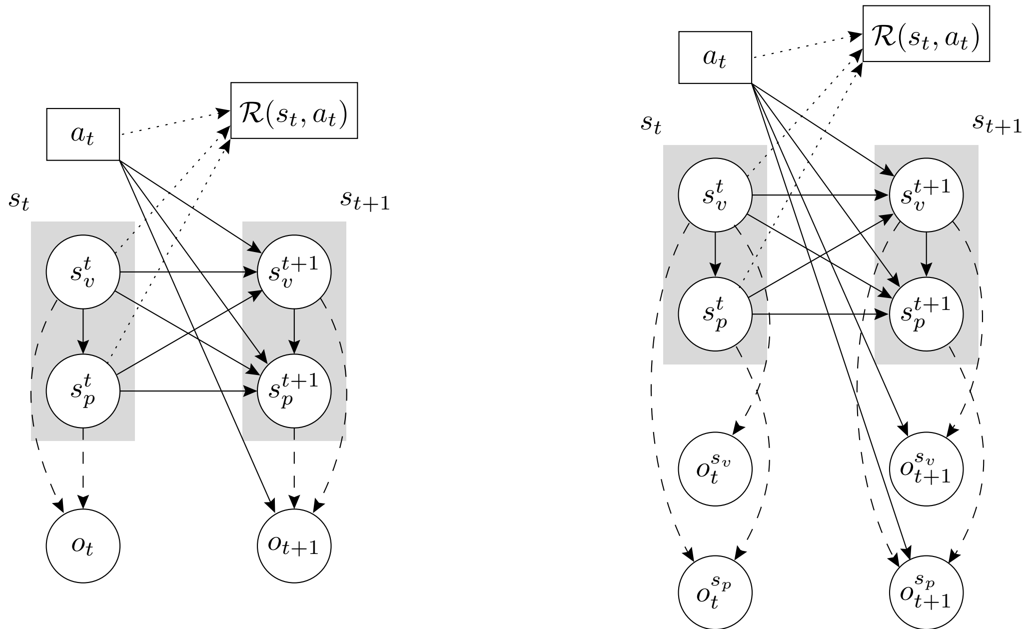
– $\mathcal{T}_{\mathcal{S}_v} : \mathcal{S}_v \times \mathcal{S}_p \times \mathcal{A} \times \mathcal{S}_v \to [0, 1]$ represents a transition function such that $\mathcal{T}_{\mathcal{S}_v}(s_v, s_p, a, s'_v) = \Pr(s'_v | s_v, s_p, a)$ gives the probability that the fully observable state variable has value s'_v if the agent takes action a in state (s_v, s_p);

– $\mathcal{T}_{\mathcal{S}_p} : \mathcal{S}_v \times \mathcal{S}_p \times \mathcal{A} \times \mathcal{S}_v \times \mathcal{S}_p \to [0, 1]$ represents a transition function such that $\mathcal{T}_{\mathcal{S}_p}(s_v, s_p, a, s'_v, s'_p) = \Pr(s'_p | s_v, s_p, a, s'_v)$ gives the probability that the partially observable state variable has value s'_p if action a is taken in state (s_v, s_p) and the fully observable state variable has a resulting value of s'_v.

Finally, $\left(s_v^0, b_{\mathcal{S}_p}^0\right)$ represents the initial belief state $b_{\mathcal{S}_p}^0$ conditioned by s_v^0. All other aspects remain as they are in the POMDP. The authors in [56] describe how any system modelled as a POMDP, can be equivalently constructed as a MOMDP by reparameterizing $\mathcal{S}$.

A difference remains between the MOMDP models proposed by [33] and [34]. The latter approach considers an additional factorization of the observation space in accordance with the division of the

state space, resulting in $\Omega = \Omega_v \times \Omega_p$, where Ω is the complete set of observations composed by the observation set for the fully observable states $o^{s_v} \in \Omega_v$ and the observation set for the partially observable states $o^{s_p} \in \Omega_p$. As a result of this partition, the visible counterpart of the observation is equal to the set of visible states, $\Omega_v = \mathcal{S}_v$, i.e. $\Pr(o^{s_v}|s_v) = 1$. Figure 2.3 illustrates this difference.



(a) Transition model of the MOMDP proposed by [33].

(b) Transition model of the MOMDP proposed by [34].

Figure 2.3: Factored model of state and observation transitions of a MOMDP for an action a_t.

The main benefit of the MOMDP over the POMDP is that the belief state probability distribution can be defined over a smaller belief state space $\mathcal{B}_p$ (which refers to the $\mathcal{S}_p$ space, instead of the complete $\mathcal{S}$ space) and is represented as $b = (s_v, b_{\mathcal{S}_p})$. This reduces the problem dimension from $|\mathcal{S}_v| \times |\mathcal{S}_p|$ to simply $|\mathcal{S}_p|$, leading to a vast improvement in computational efficiency for POMDP solvers.

2.5 Monte Carlo methods in POMDPs

Monte Carlo methods have been widely used in areas such as challenging games [57], engineering and computational biology [58].

Monte Carlo methods use a very different paradigm for online planning in POMDPs, when compared to the strategies mentioned in the previous section. In this approach, the algorithm uses a black-box simulator G as a *generative model* of the POMDP. This simulator is a randomized algorithm that employs the same dynamics as the POMDP. Once given a state $s_t \in \mathcal{S}$ and an action $a_t \in \mathcal{A}$ as input, it provides a sample of a successor state s_{t+1}, an observation o_{t+1} and a reward $\mathcal{R}_{t+1}$ as output, $(s_{t+1}, o_{t+1}, \mathcal{R}_{t+1}) \sim G(s_t, a_t)$, where the operator $\sim$ denotes sampling, e.g., $y \sim w$ means y is sampled from w.

The models contained in the underlying POMDP provide the successor variables: the state $s_{t+1} \in \mathcal{S}$ is given by the transition model, $s_{t+1} \sim \mathcal{T}(s, a)$, the observation $o_{t+1} \in \Omega$ is given by the observation model, $o_{t+1} \sim \mathcal{O}(s, a)$, and the reward $\mathcal{R}_{t+1}$ is drawn from the reward function, $\mathcal{R}_{t+1} \sim \mathcal{R}(s, a)$. Instead

of directly considering the model's dynamics, the value function is updated using the sequences of states, observations and rewards that are generated by the simulator, given an initial state s_0 sampled from b_0. Besides this, a generative model may often be available when explicit next-state distributions are not.

A key idea behind this approach is to perform several simulations to derive a successful strategy, since not much can be learned from a single random simulation. Therefore, all Monte Carlo methods keep track of the value of an action a in a state s, denoted by $Q(s, a)$, and the number of times action a has been selected in state s, denoted by $N(s, a)$, throughout the simulations.

In addition, Kearns et al. [59] have shown that the sample complexity of Monte Carlo methods, i.e. the amount of samples required to create good policies, is determined only by the difficulty of the POMDP, rather than the size of the state space or observation space.

The development of Monte Carlo planners for POMDPs has evolved from simple rollout methods that do not construct a search tree [60] over methods that perform depth-first search with a fixed horizon [59] to a Monte Carlo Tree Search method [26] that is not limited by a fixed horizon and can compete with offline full-width planners in problems with large state spaces. These algorithms are addressed in detail in the next subsections.

2.5.1 Monte Carlo Rollouts

Monte Carlo rollouts, or Monte Carlo simulations, is the most basic Monte Carlo method. Each iteration consists of two steps. The first step, called **Simulation**, selects actions according to some probability distribution, usually a uniform distribution, until a terminal condition is reached. Each action is given to the black-box simulator to obtain a sequence of rewards $(\mathcal{R}_{d=0}, \mathcal{R}_{d=1}, ..., \mathcal{R}_{d_n})$, where d_n denotes the depth of the simulation phase, i.e., the number of steps until the terminal condition is reached. From this sequence, the simulated return $\hat{R}$ is computed as follows:

$$\hat{R} = \sum_{d=0}^{d_n} \gamma^d \mathcal{R}_d \tag{2.19}$$

The second step, called **Back-propagation**, updates the values of the action a selected during the Simulation step. The visitation count $N(s, a)$ is increased by the unit and the value $Q(s, a)$ is estimated by the mean return from s of all simulations in which a was selected from state s, according to the following set of equations:

$$N(s, a) = N(s, a) + 1 \tag{2.20}$$

$$Q(s, a) = Q(s, a) + \frac{\hat{R} - Q(s, a)}{N(s, a)} \tag{2.21}$$

To be noted that $Q(s, a)$ is not defined as introduced in Equation 2.16. The algorithm then repeats these steps until some terminal condition is reached. At this point, the action with the highest average value $Q(s, a)$ is chosen.

2.5.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a best-first search algorithm that combines Monte Carlo rollouts with tree search, and was introduced in different versions by Chaslot [57] and Coulom [61] in 2006. Although MCTS has been mainly applied to the domain of game theory in classical board games such as Go [57], it can be applied to any domain that can be expressed in terms of state-action pairs, such as decision theory and bandit-based methods. In fact, over the last few years, MCTS has also achieved great success with complex real-world planning, optimization and control problems.

MCTS rests on two fundamental concepts: that the true value of an action may be approximated using random sampling; and that these values may be used efficiently to guide the exploration of the tree. Its aim is to select the best move to execute by exploring the search space pseudo-randomly. In the beginning of the search, the exploration is performed at random. Then by using the results of previous explorations, the algorithm becomes able to predict more accurately the most promising moves, and thus, the exploration becomes narrower.

In MCTS, the nodes can be called state nodes, because they represent states in the problem, and directed links to child nodes represent actions leading to subsequent states. The method iteratively builds a search tree, one node after the other, based on the results of random simulations, until it is interrupted or a computational budget is reached. This budget can be defined in many ways: for instance, in terms of time or number of simulations performed.

Four steps are applied per search iteration: **Selection**, **Expansion**, **Simulation** and **Back-propagation**. The construction process of the MCTS tree is illustrated in Figure 2.4.

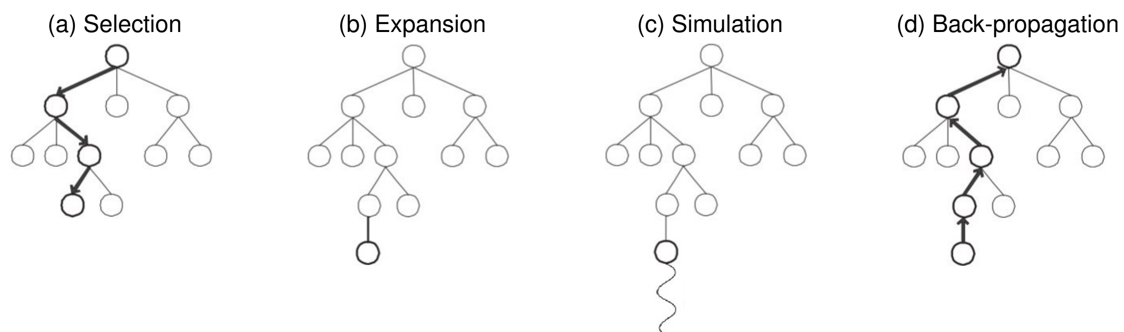


Figure 2.4: Monte Carlo Tree Search.

The algorithm starts with only the root node. A **Selection** strategy (Figure 2.4a) is used to recursively choose actions, according to the statistics stored in the nodes of the search tree, descending through the tree until a leaf node L is reached. This process is done using a Tree Policy, which will be explained in detail in Section 2.5.3.

Once this happens, the **Expansion** phase (Figure 2.4b) is called to store one or multiple children of L in the tree, according to the available actions. The simplest rule, proposed by Coulom [61], is to expand one node per simulation. The node expanded corresponds to the first position encountered that was not stored yet.

From this point forward, the **Simulation** step (Figure 2.4c) begins: if L does not correspond to a

terminal state in the problem, a simulation is performed using a Simulation Policy (also explained in Section 2.5.3). This simulation starts with the action selected at L and ends when a terminal state is reached (either a goal was achieved or constraints were violated). At the end, a reward is computed to assess the quality of the reached state.

Finally, a **Back-propagation** (Figure 2.4d) is performed on the statistics of each node traversed with the outcome of the Simulation step (using Equations 2.20 and 2.21), while backtracking through the internal nodes up to the root of the tree.

When the algorithm reaches a computational budget or is interrupted, the search terminates and an action a is selected at the root node by one of the following criteria, introduced by [62]:

– *Max child*: Select the action with the highest reward.

– *Robust child*: Select the most visited action.

– *Max-Robust child*: Select the action with both the highest visitation count and the highest reward.

– *Secure child*: Select the action which maximizes a lower confidence bound.

MCTS provides three important advantages [62]. (1) *Asymmetric* growth of the tree developed during the MCTS search. State nodes which appear to be more valuable in the problem are visited more often, biasing the search towards more promising areas of exploration. (2) *Anytime* property that allows the execution of the algorithm to be stopped at any time to return the action that is currently estimated to be optimal. (3) *Aheuristic* property, i.e. the lack of need for domain-specific knowledge, making it readily applicable to any domain that may be modelled using a tree. However, significant improvements in performance may often be achieved using heuristics.

2.5.3 Tree and Simulation Policies

There are two main aspects of the Monte Carlo Tree Search that can be changed for algorithm variability: the *Tree or Selection Policy* and the *Simulation or Rollout Policy*.

The *Tree Policy* defines the selection procedure of the action-nodes already contained on the tree. When traversing it, they can be chosen at random or confidence bounds can be applied to estimate the probability of the outcome being the optimal solution. By definition, an action is sub-optimal for a given state, if its value is less than the best of the action-values for the same state. Since action-values depend on the values of successor states, the problem boils down to getting the estimation error of the state-values for such states to decay fast [43]. In order to achieve this, an efficient algorithm must face the exploration-exploitation dilemma, which can be described as the search for a balance between exploring the environment to find profitable actions while taking the empirically best action as often as possible. The simplest instance of this dilemma is the Multi-Armed Bandit (MAB) problem [63], which is analogous to the node selection problem encountered in MCTS.

In the most basic formulation of a K-armed bandit problem, an agent (gambler) is presented with a set of K slot machines ("arms" of the bandit) and has to decide which one of them to play, how many times to do so for each machine and in which order, knowing that, when he pulls an arm i, it provides

a random payoff $X_{i,n}$, where $i = 1, ..., K$ and $n \geq 1$. Successive plays of machine i yield payoffs $X_{i,1}, X_{i,2}, ...$ which are independent and drawn from a fixed but unknown distribution over $[0, 1]$, with unknown expectation $\mu_i = \mathbb{E}[X_{i,n}]$, associated with that specific arm.

The goal of the gambler is to maximize the total payoff earned through iterative plays. In this way, a dilemma arises: the gambler can select the machine that gives the current highest expected payoff based on the knowledge already acquired (exploitation), or attempt sub-optimal arms so as to further increase knowledge (exploration) and to ensure that no good alternatives are missed because of early estimation errors.

Therefore, the MAB problem relies on the elaboration of an allocation strategy to decide what is the next machine to play in order to optimize the expected total reward and minimize the *cumulative regret*, R_n, which is the difference between the sum of rewards associated with an optimal strategy $\mu^* n$ and the sum of the actual collected rewards after n plays [64]. In other words, the cumulative regret is the expected loss due to not playing the best option in hindsight. This type of regret is accumulated during execution of the algorithm and each time a non-optimal arm is sampled, the cumulative regret increases.

Let $T_i(n)$ be the number of times machine i has been played during the first n plays. Then the cumulative regret after n plays can be expressed as:

$$R_n = \mu^* n - \mu_j \sum_{j=1}^{K} \mathbb{E}[T_j(n)] \tag{2.22}$$

where $\mu^* = \max\{\mu_1, \mu_2, ..., \mu_K\}$. It is important to highlight the necessity of attaching non-zero probabilities to all arms at all times, in order to ensure that the optimal arm is not missed due to temporarily promising rewards from a sub-optimal arm. Thus, an upper confidence bound ought to be defined as a function of the rewards obtained so far.

Several effective strategies based on the Multi-Armed Bandit problem have been proposed in literature (and detailed in [63]). Some of these methods are described below:

- **Random**: The nodes are selected at random from the parent, not depending on any reward estimate.

- **UCB1**: The simplest Upper Confidence Bound (UCB) strategy, which minimizes cumulative regret, while converging to the empirically best arm without any prior knowledge on the reward distributions. This strategy dictates to play arm j that maximizes Equation 2.23, as:

$$j^* = \arg\max_j \left\{ \overline{X}_j + c\sqrt{\frac{\ln n}{T_j(n)}} \right\} \tag{2.23}$$

  in which $\overline{X}_j$ is the average reward obtained from machine j, $T_j(n)$ is the number of times machine j has been played so far, n is the overall number of selections done so far and $c \geq 0$ is a parameter that enables to control the exploration-exploitation trade-off. Although the theory suggest a default value of $c = \sqrt{2}$, this parameter is usually experimentally tuned to increase performance.

  The first term in Equation 2.23 vouches for the exploitation of the previously visited choices with the

highest reward values, while the second encourages the exploration of undiscovered nodes. Once the empirical best arm is found by exploring the available options, UCB1 exploits it by repeatedly sampling it more often, minimizing the overall cumulative regret.

The authors show that if strategy UCB1 is run on K machines having arbitrary reward distributions with support in $[0, 1]$, then the expected cumulative regret after any number n of plays has logarithmic growth uniformly over n and is bounded by:

$$\mathbb{E}R_{n_{\text{UCB1}}} \leq \left[8 \sum_{i:\mu_i<\mu^*} \frac{\ln n}{\Delta_i} \right] + \left(1 + \frac{\pi^2}{3} \right) \sum_{i=1}^{K} \Delta_i \tag{2.24}$$

where $\Delta_i = \mu^* - \mu_i$.

The MCTS instance using this action selection strategy is usually referred to, in literature, as Upper Confidence Bounds applied to Trees (UCT) [43] and will be further explained in Section 2.5.4.

– **UCB1-Tuned**: This strategy relies on the tuning of the UCB1 formula accounting also for the variance of each arm to obtain a refined upper bound on rewards expectation. In this case, Equation 2.23 is replaced by:

$$j^* = \arg\max_j \left\{ \overline{X}_j + \sqrt{\frac{\ln n}{T_j(n)} \min\left\{ \frac{1}{4}, V_j(T_j(n)) \right\}} \right\} \tag{2.25}$$

where

$$V_j(T_j(n)) = \sigma_j^2 + \sqrt{\frac{2\ln n}{T_j(n)}}$$

is an upper confidence bound on the variance of machine j computed from samples observed so far,

$$\sigma_j^2 = \left(\frac{1}{2} \sum_{m=1}^{T_j(n)} X_{j,m}^2 \right) - \overline{X}_{j,T_j(n)}^2$$

and the factor $\frac{1}{4}$ is an upper bound on the variance of a Bernoulli random variable. The authors claim this approach performs better in practice than the UCB1, but comes without theoretical guarantees.

– **ε-Greedy**: The node is selected so as to maximize the value of the following formula:

$$j^* = \arg\max_j \left\{ \overline{X}_j + \frac{\varepsilon_n}{T_j(n)} \right\} \tag{2.26}$$

This approach requires the computation of $\varepsilon_n = \min\left\{1, \frac{cK}{d^2 n}\right\}$, where K is the number of child nodes, n is the child node being considered, $c > 0$ and $0 < d < 1$. This calculation determines that the probability to play the node with the highest average reward is equal to $1 - \varepsilon_n$, whereas a random node will be selected with probability ε_n.

The *Simulation Policy* deals with how the domain is played out in the simulations that are performed from the leaf node onwards, in order to produce a value estimate. The simplest and regular version of

Monte Carlo Tree Search uses a uniform random rollout policy, $\pi_{rollout}$.

2.5.4 Upper Confidence Bounds applied to Trees Algorithm

Kocsis and Szepesvári (2006) introduced an instance of a MCTS algorithm, the Upper Confidence Bounds applied to Trees (UCT) algorithm, that applies the UCB1 strategy (Equation 2.23) to guide selective sampling of actions in rollout-based planning problems [43]. Their aim was to design a MCTS-based algorithm that had a small error probability if stopped prematurely and that converged to game-theoretic optimum given sufficient time.

In UCT, the action selection problem is treated as a separate multi-armed bandit for every explored internal node. The arms correspond to actions and the payoffs to the cumulative (discounted) rewards of the paths originating at the nodes. In particular, in state s, the action that maximizes Equation 2.27 is selected:

$$\bar{a}_{\text{UCT}} = \arg\max_{a \in \mathcal{A}} \left\{ Q(s,a) + 2c\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \tag{2.27}$$

where $Q(s,a)$ is computed as in Equation 2.21, $N(s) = \sum_a N(s,a)$ is an overall count of visits payed to state s and $N(s,a)$ is the number of times action a was selected when state s has been visited. It is generally understood that $N(s,a) = 0$ yields a UCT value of ∞, so that previously unvisited children are assigned the largest possible value, to ensure that all children of a node are considered at least once before any child is expanded further.

Similarly to UCB1, an exploration factor c is introduced for a better tuning of the exploration-exploitation trade-off. A higher value of this parameter implies that the investigation of the state space close to the root is done more thoroughly, while with $c = 0$ the algorithm acts greedily within the tree. For rewards in the range $[0,1]$, [43] found a choice of $c = 1/\sqrt{2}$ to satisfy the Hoeffding inequality. With rewards outside this range, the exploration factor needs to be set through extensive experimentation because the need for exploration varies from domain to domain.

2.5.5 Partially Observable UCT Algorithm

The UCT algorithm introduced in the previous section is applied to fully observable environments. In order to extend it to partial observability, a search tree of histories instead of states has to be considered. The resulting algorithm is denominated Partially Observable Upper Confidence Bound applied to Trees algorithm (PO-UCT) [26] and applies a modified version of Equation 2.27 to execute action selection:

$$\bar{a}_{\text{PO-UCT}} = \arg\max_{a \in \mathcal{A}} \left\{ Q(h,a) + c\sqrt{\frac{\log N(h)}{N(h,a)}} \right\} \tag{2.28}$$

where $Q(h,a)$ is computed as in Equation 2.21 considering history h instead of state s, $N(h)$ counts the number of times history h has been visited and $N(h,a)$ is the number of times the algorithm selected action a upon visiting history h.

Related Algorithms

Motivated by some prolific achievements in the challenging trade-off between exploration and exploitation, researchers have since been striving to attain a better understanding of when and why the UCT algorithm succeeds and fails, and to extending and refining it beyond its basic form. These developments are greatly increasing the range of applications for which UCT is a tool of choice and pushing its performance to ever higher levels.

Browne et al. [62] present a comprehensive survey on variations and enhancements of the UCT algorithm, as well as of other MCTS techniques. For instance, [65] use Progressive Bias to linearly combine the standard UCT evaluation with an heuristic evaluation with the weight proportional to the number of simulations. The more simulations are performed, the more statistical confidence, and therefore, the higher weight is assigned to the standard UCT formula. Another approach [66] explores the use of simple regret minimizing bandit algorithms at the root, while using UCT throughout the tree, which has shown the potential to overcome some weaknesses of the UCT algorithm. A variant of the UCT algorithm, used in deep Reinforcement Learning (RL) tool such as the AlphaGo, exploits the neural network to predict the next action and is called the PUCT algorithm [67] [68]. In [69], the authors integrate a task hierarchy into the MCTS framework, specifically leading to a hierarchical version of UCT (H-UCT).

Moreover, [70] propose the trial-based heuristic tree-search framework, which incorporates ingredients from MCTS, Dynamic Programming and Heuristic Search. Within their framework, they derive three new novel algorithms: MaxUCT, that merges action-value Monte-Carlo backup function and state-value Full Bellman backup function; DP-UCT, which considers probabilities in the backups of action-value estimates; and UCT*, that incorporates trial length in DP-UCT. Such variants of UCT proved to perform significantly better and in less time than the standard UCT algorithm. Finally, [71] attempted to dynamically tune online the exploration constant from the UCT formula, without however being able to successfully enhance the performance relatively to the fixed value.

2.5.6 Monte Carlo Belief State Updates

Once applied a Tree Policy strategy and selected an action a, the agent transits into the execution phase to perform the action in the real world. At this point, the agent receives an observation o and has to update its belief state. In small state spaces, the belief state can be updated exactly by Bayes' theorem (Equation 2.17). However, in large state spaces, even a single Bayes update may be computationally hard. Moreover, a compact representation of the transition or observation probabilities may not be available.

To tackle this problem, the belief state can be approximated by a set of random state samples, called particles, and rejection sampling can be used to update the particles based on a sampled observation, rewards and state transitions [26]. The process of rejection sampling is depicted in Figure 2.5.

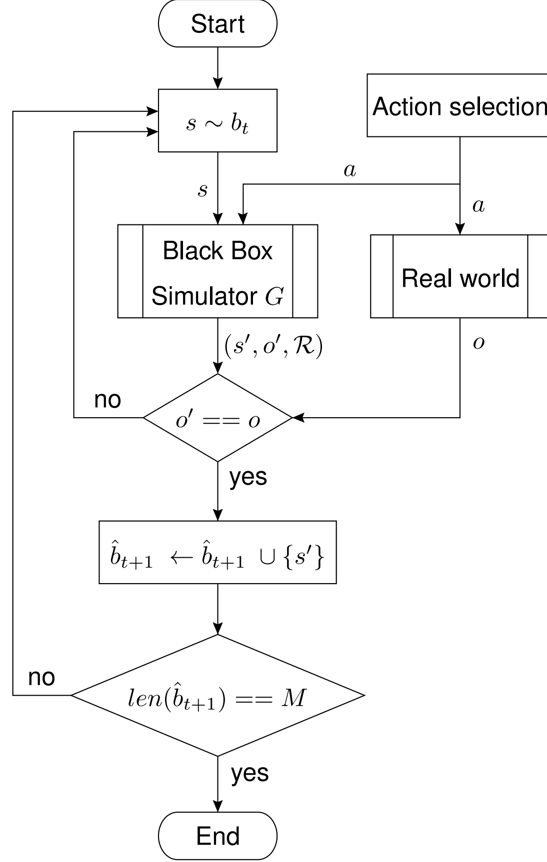


Figure 2.5: Rejection sampling process.

The process begins by sampling a state s from the current time step belief state b_t, i.e. by selecting a particle at random. This particle is passed into the black-box simulator, $(s', o', \mathcal{R}) \sim G(s, a)$, to return a successor state s' and an observation o'. If the sampled observation o' matches the one received from the real world o, then the next state s' is added to the updated belief state $\hat{b}_{t+1}$. The process is repeated until M particles have been added to $\hat{b}_{t+1}$. With sufficient particles, the approximated belief state approaches the true belief state, $\lim_{M \to \infty} \hat{b}(h) = b(h)$.

2.5.7 Partially Observable Monte Carlo Planning

The Partially Observable Monte Carlo Planning (POMCP) algorithm, developed by Silver and Veness [26], is an online algorithm that combines the PO-UCT algorithm (Section 2.5.5) with Monte Carlo belief state updates (Section 2.5.6). By incorporating the observations inherited from the partially observable domain, the search tree changes from a tree of states to a tree of histories, as illustrated in Figure 2.6.

More specifically, there are two variations of nodes in the POMCP tree: *history nodes* and *sequence nodes*. Each history node represents a possible history $h^k = \{a^0, o^0, a^1, o^1, ..., a^k, o^k\}$. The edges emerging from the history node correspond to actions. Each sequence node represent a sequence $h^k = \{h^{k-1}, a^k\}$ that includes the previous history h^{k-1} and the action a^k selected at the preceding history node. The edges emerging from a sequence node correspond to observations, leading to subsequent history nodes. Figure 2.6 illustrates the search tree built by POMCP, while Figure 2.7 represents its

evolution. The complete POMCP algorithm is described in Algorithm 1.

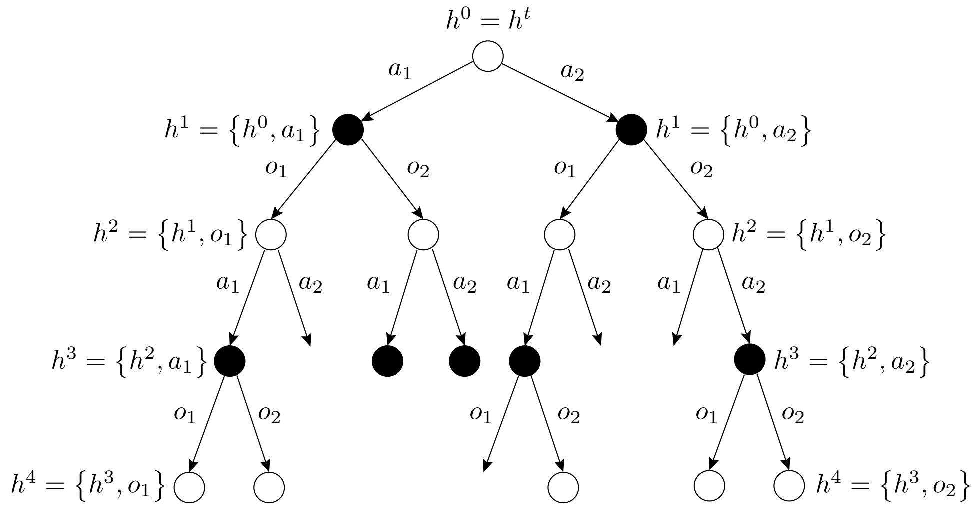


Figure 2.6: The search tree constructed by POMCP for a problem with 2 actions and 2 observations, with history nodes represented in white and sequence nodes in black.

The tree contains a node $T(h) = \langle N(h), V(h), b(h)\rangle$ for each represented history h, where $N(h)$ counts the number of times history h has been visited, $V(h)$ is the value of history h and a set of particles $b(h)$ that approximate the belief state for the represented history h. For each sequence node, the tree stores $T(ha) = \langle N(ha), Q(h, a)\rangle$, where $N(ha)$ is the number of times a given action a has been chosen after history h was visited and $Q(h, a)$ is the action-value estimate, or Q-value, computed from the mean return of all simulations started when action a was selected in history h.

The search procedure is called from the current history h_t and begins from a start state s, that is sampled from the belief state $b(h_t)$ (line 3). The **Selection** strategy alternates between the selection of actions at history nodes and observations at sequence nodes. At a history node, an action a is selected according to the PO-UCT algorithm mentioned in Section 2.5.5 (line 19). This action a is then submitted to the black-box simulator to generate an observation o (line 20). At the succeeding sequence node, o decides which edge the algorithm needs to follow.

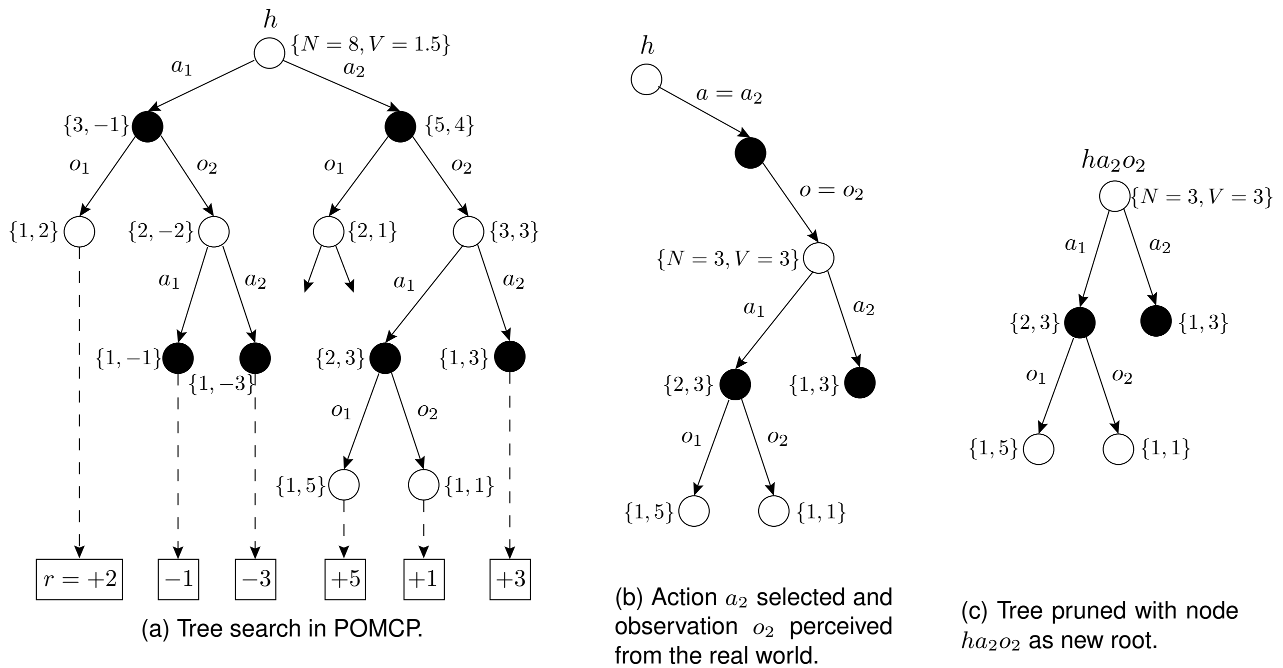
In the **Expansion** step, each new sequence node ha that is added to the tree corresponds to the first new history encountered during the Selection step and is initialized with $\langle N_{init}(ha), Q_{init}(h, a)\rangle$ (line 17). This Q-value initialization is performed with a rollout method starting from this node (line 18). Each **Simulation** phase then progresses by simulating action-observation sequences, allowing to build an asymmetric tree (line 21).

The **Back-propagation** step (lines 22 to 24) updates the statistics of each node traversed according to:

$$\begin{cases} N(h) \leftarrow N(h) + 1 \\ N(ha) \leftarrow N(ha) + 1 \\ Q(h,a) \leftarrow Q(h,a) + \dfrac{Q(h,a)' - Q(h,a)}{N(ha)} \end{cases} \tag{2.29}$$

where $Q(h,a)'$ is the simulated return computed recursively over the trajectories.

When the algorithm reaches a computational budget (line 2) or is interrupted, the search terminates and the agent selects the action with the greatest Q-value (line 5), for which it receives an observation o_t from the world (Figure 2.7b). At this point, the node $T(h_t a_t o_t)$ becomes the root of the new search tree and the belief state $b(h_t a_t o_t)$ determines the agent's new belief state. In case the new belief state is not yet composed of M particles, the algorithm performs rejection sampling (as described in Section 2.5.6) until $len(b(h_t a_t o_t)) == M$. The remainder of the tree is pruned (Figure 2.7c), as all other histories are now impossible.



(a) Tree search in POMCP.

(b) Action a_2 selected and observation o_2 perceived from the real world.

(c) Tree pruned with node ha_2o_2 as new root.

Figure 2.7: An illustration of POMCP in an environment with 2 actions and 2 observations.

Algorithm 1: Partially Observable Monte-Carlo Planning

```
1  Function SEARCH(h):
2      while not timeout do
3          s ~ b(h)
4          SIMULATE(s, h, 0)
5      return a* ← argmax_a Q(h, a)
6  Function ROLLOUT(s, h, depth):
7      if γ^depth < ε then
8          return 0
9      a ~ π_rollout(h)
10     (s', o, R(s, a)) ~ G(s, a)                          /* Black-box simulator */
11     return R(s, a) + γ ROLLOUT(s', hao, depth + 1)
12 Function SIMULATE(s, h, depth):
13     if γ^depth < ε then
14         return 0
15     if h ∉ T then
16         for a ∈ A do
17             T(ha) ← (N_init(ha), Q_init(h, a))
18         return ROLLOUT(s, h, depth)
19     ā ← argmax_a { Q(h, a) + c sqrt( log N(h) / N(ha) ) }
20     (s', o, R(s, ā)) ~ G(s, ā)                          /* Black-box simulator */
21     Q(h, ā)' ← R(s, ā) + γ SIMULATE(s', hao, depth + 1)
22     N(h) ← N(h) + 1
23     N(hā) ← N(hā) + 1
24     Q(h, ā) ← Q(h, ā) + (Q(h, ā)' − Q(h, ā)) / N(hā)
```

Chapter 3

Problem formulation

This chapter focuses in previous work developed in the NavPlan project. Section 3.2 presents the GNC model proposed by [5] to be integrated in the planning algorithm. Section 3.3 formalizes the Partially Observable Stochastic Shortest Path MDP (PO-SSP) addressed in this work as a MOMDP, to take advantage of the factorization of the state space for a reduction in policy computation time.

3.1 System architecture

This section follows the work developed in [5] and [8], concerning the definition of the path planning problem. The problem addressed aims at finding a navigation and guidance strategy for making UAVs reach a given destination safely and efficiently in a cluttered environment under uncertainty. Therefore, it combines the partial observability property of the POMDP model with the objective of reaching a goal as in the SSP MDP, thus making it a Partially Observable Stochastic Shortest Path (PO-SSP) planning problem. It considers a priori probabilistic availability of the vehicle onboard sensors and execution error propagation, which depends on the navigation solution being used. Given the nature of the state variables being considered, [5] make use of the MOMDP representation to model this PO-SSP problem.

In the path planning problem addressed, a target state or a set of target states is considered and, therefore, the optimization objective becomes the minimization of the expected total cost until the target state is reached.

Let us suppose a vehicle equipped with N different onboard navigation sensors, such as inertial sensors, GPS and vision sensors, which are used by the GNC system to execute a path. The navigation filter estimates the vehicle state $\mathbf{x}$ and its error covariance matrix $\mathbf{P}$ by using measurements from a set of selected and available sensors, which define a navigation mode. The guidance and control module executes a selected path segment (or action) by using the navigation solution. A priori knowledge on the environment is assumed to be given by a set of probability grid maps of obstacles and availability of each of the N onboard navigation sensors. These maps are used during the planning task to propagate the path execution uncertainty given the probabilistic sensor's availability, and then to evaluate obstacle collision risk.

The architecture of the overall planning system considers the (closed-loop) GNC model as part of the MOMDP planning problem. The GNC sub-system considered includes the vehicle motion model, the onboard sensor's models in the navigation module and the guidance law. As mentioned previously, the goal of using this closed-loop configuration between the vehicle motion model with the GNC module and the planner is to be able to propagate, during planning, the influence of sensor availability and uncertainty on the execution error of the trajectory.

The planner will then consider GNC's closed-loop transitions to compute the possible evolutions of the system's state. The policy computed by the planner take as inputs the probability distribution over the current state, b_{s_h}, and a vector of booleans on sensors' availability, s_v, and returns an action a, which contains information on a reference velocity to pursue, $\mathcal{V}_{ref}$, and on the navigation mode to use. Afterwards, a new vector of sensors' availability s'_v is observed from the environment and the belief state update is performed. The new belief $b_a^{s'_h}$ is then used by the policy to define the next action to execute. Figure 3.1 illustrates the system's architecture described.

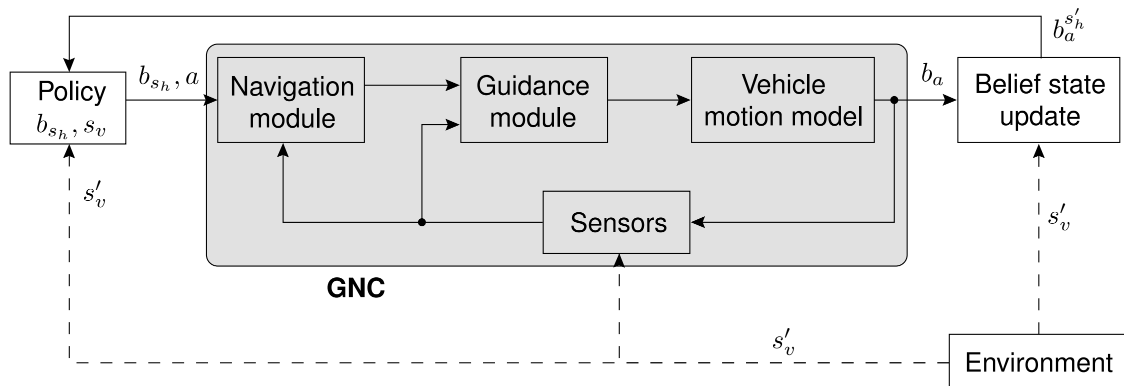


Figure 3.1: System architecture diagram, in which the GNC closed-loop vehicle model is incorporated into the MOMDP transition function [5].

The GNC module proposed works with a high frequency. However, for a POMDP-based planner, it might not be desirable for it to work in a high operating frequency, as this would involve increasing the number of operations over the indefinite horizon and to further confront the problem of the curse of dimensionality. Therefore, it is counterproductive to operate the planner at the same frequency as the GNC module. As such, two distinct functioning frequencies are considered: a unit of time from GNC, k, and a planning epoch, t. The planner's time represents several units of time of the GNC transition system, as illustrated in Figure 3.2.

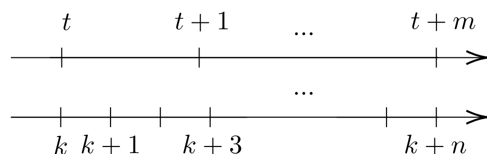


Figure 3.2: Difference between the functioning frequency of the GNC system and the planner.

In the following sections, it is considered the presence of at least an onboard Inertial Measurement Unit (IMU), whose availability and accuracy do not depend on the environment. However, its acceleration

and angular velocity measurements are biased, causing a drift in the estimation of the velocity and the position, when used alone in the navigation module during state estimation. Therefore, other sensors, such as GPS, can be fused with IMU in order to limit or even correct such drift.

3.2 GNC transition model

This section presents the GNC transition model proposed by [5], to be integrated in the MOMDP planning problem.

3.2.1 State transition model

The state $\mathbf{x}$ of the UAV is defined by its position $\mathcal{X}^T = [x \quad y \quad z]$, its velocity in the inertial local frame $\mathcal{V}^T = [v_x \quad v_y \quad v_z]$, and the accelerometer bias in the UAV frame $\beta_a^T = \begin{bmatrix}\beta_{a_x} & \beta_{a_y} & \beta_{a_z}\end{bmatrix}$, such as:

$$\mathbf{x} = \begin{bmatrix} \mathcal{X} \\ \mathcal{V} \\ \beta_a \end{bmatrix} \tag{3.1}$$

Then, considering the GNC closed-loop model with an acceleration a as input, the state transition can be defined as:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\mathcal{X}} \\ \dot{\mathcal{V}} \\ \dot{\beta}_a \end{bmatrix} = \begin{bmatrix} \mathcal{V} \\ a \\ 0 \end{bmatrix} + \begin{bmatrix} v_{\mathcal{X}} \\ v_{\mathcal{V}} \\ v_{\beta} \end{bmatrix} = \begin{bmatrix} 0 & \mathbf{I} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ \mathbf{I} \\ 0 \end{bmatrix} a + v = \mathbf{A}\,\mathbf{x} + \mathbf{B}\,a + v \tag{3.2}$$

where $v \sim \mathcal{N}(0, \tilde{\mathbf{Q}})$ is the transition process noise. According to this model, the state transition from $\mathbf{x}(t_k)$ to $\mathbf{x}(t_{k+1} = t_k + \Delta t) = \mathbf{x}_{k+1}$ can be derived as:

$$\mathbf{x}_{k+1} = \mathbf{\Phi}\,\mathbf{x}_k + \mathbf{B}\,a_k + v_{k+1} \tag{3.3}$$

where $v_{k+1} \sim \mathcal{N}(0, \mathbf{Q})$ is the discretized transition process noise and

$$\mathbf{\Phi} = \begin{bmatrix} \mathbf{I} & \Delta t\mathbf{I} & 0 \\ 0 & \mathbf{I} & 0 \\ 0 & 0 & \mathbf{I} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \frac{\Delta t^2}{2}\mathbf{I} \\ \Delta t\mathbf{I} \\ 0 \end{bmatrix}, \quad \mathbf{Q} \simeq \Delta t\tilde{\mathbf{Q}}$$

Based on Equation 3.3, one can define the probability density $\Pr(\mathbf{x}_{k+1}|\mathbf{x}_k, a_k)$ that the vehicle is in state $\mathbf{x}_{k+1}$ after receiving the vector of control (acceleration) a_k in state $\mathbf{x}_k$. Such probability density is obtained as a multidimensional normal distribution:

$$\mathbf{x}_{k+1} \sim \mathcal{N}(\mathbf{\Phi}\mathbf{x}_k + \mathbf{B}a_k, \mathbf{Q}) \tag{3.4}$$

3.2.2 State estimation in the navigation module

The UAV state $\mathbf{x}$ is not accessible in reality and hence, it is estimated by the navigation module using sensor measurements available onboard. The state estimator is based on an Extended Kalman Filter (EKF) [72], which includes two steps: (1) *Prediction*, by the IMU acceleration measurements, and (2) *Correction* by the other navigation sensor $S_n, n = \{1, ..., N\}$ measurements , if available.

Prediction

The IMU acceleration measurement a_{IMU_k} is used to propagate the estimated state from k to $k+1$. It measures the biased, non-gravitational UAV body acceleration and is given as:

$$a_{IMU_k} = \mathbf{R}_{BI_k}(a_k - g) + \beta_{a_k} + \xi_{IMU_k} \tag{3.5}$$

where $\mathbf{R}_{BI_k}$ is a rotation matrix from the inertial to the UAV body frames (assumed to be known), g is the gravity vector and $\xi_{IMU_k} \sim \mathcal{N}(0, \mathbf{R}_{IMU})$ is the IMU acceleration measurement error.

From Equation 3.5, the real acceleration in the local inertial frame is described as:

$$a_k = \mathbf{R}_{BI_k}^T(a_{IMU_k} - \beta_{a_k} - \xi_{IMU_k}) + g \tag{3.6}$$

Whereas the estimated acceleration $\hat{a}_k$ is computed based on the measure a_{IMU_k} and on the estimated bias $\hat{\beta}_{a_k}$:

$$\hat{a}_k = \mathbf{R}_{BI_k}^T(a_{IMU_k} - \hat{\beta}_{a_k}) + g \tag{3.7}$$

According to the process model (Equation 3.3), the estimated state $\hat{\mathbf{x}}_k$ is propagated as:

$$\begin{aligned}\hat{\mathbf{x}}_{k+1}^- &= \mathbf{\Phi}\hat{\mathbf{x}}_k + \mathbf{B}\hat{a}_k \\ &= \mathbf{\Phi}\hat{\mathbf{x}}_k + \mathbf{B}\left(\mathbf{R}_{BI_k}^T\left(a_{IMU_k} - \hat{\beta}_{a_k}\right) + g\right)\end{aligned} \tag{3.8}$$

Then the state prediction error $\tilde{\mathbf{x}}_{k+1}^-$ becomes:

$$\begin{aligned}\tilde{\mathbf{x}}_{k+1}^- &= \mathbf{x}_{k+1} - \hat{\mathbf{x}}_{k+1}^- \\ &= (\mathbf{\Phi}\,\mathbf{x}_k + \mathbf{B}\,a_k + v_{k+1}) - (\mathbf{\Phi}\,\hat{\mathbf{x}}_k + \mathbf{B}\,\hat{a}_k) \\ &= \mathbf{\Phi}\,(\mathbf{x}_k - \hat{\mathbf{x}}_k) + \mathbf{B}\,(a_k - \hat{a}_k) + v_{k+1} \\ &= \mathbf{\Phi}\,\tilde{\mathbf{x}}_k - \mathbf{B}\,\mathbf{R}_{BI_k}^T\left(\tilde{\beta}_{a_k} + \xi_{IMU_k}\right) + v_{k+1} \\ &= (\mathbf{\Phi} - \Delta\mathbf{\Phi}_k^a)\,\tilde{\mathbf{x}}_k + v_{k+1} - \mathbf{B}\,\mathbf{R}_{BI_k}^T\xi_{IMU_k}\end{aligned} \tag{3.9}$$

where $\Delta\mathbf{\Phi}_k^a = \mathbf{B}\,\mathbf{R}_{BI_k}^T\begin{bmatrix}0 & 0 & \mathbf{I}\end{bmatrix}$. And the associated covariance error is then given by:

$$\mathbf{P}_{k+1}^- = (\mathbf{\Phi} - \Delta\mathbf{\Phi}_k^a)\,\mathbf{P}_k\,(\mathbf{\Phi} - \Delta\mathbf{\Phi}_k^a)^T + \mathbf{Q} + \tilde{\mathbf{R}}_{IMU_k} \tag{3.10}$$

where $\tilde{\mathbf{R}}_{IMU_k} = \mathbf{B}\,\mathbf{R}_{BI_k}^T\,\mathbf{R}_{IMU}\,\mathbf{R}_{BI_k}\,\mathbf{B}^T$. For simplicity, the case of $\mathbf{R}_{IMU} = \sigma_{IMU}^2\mathbf{I}$ was considered and hence $\tilde{\mathbf{R}}_{IMU_k} = \mathbf{B}\,\mathbf{R}_{IMU}\,\mathbf{B}^T$ remains constant for all k.

Sensor correction

When the n-th onboard sensor is available at t_{k+1}, the predicted state (Equation 3.3) can be corrected by using its measurement $z_{S_{n_{k+1}}}$, according to:

$$z_{S_{n_{k+1}}} = h_{S_n}\left(\mathbf{x}_{k+1}\right) + \xi_{S_{n_{k+1}}} \tag{3.11}$$

where $\xi_{S_n} \sim \mathcal{N}\left(0, \mathbf{R}_{S_n}\right)$ is a measurement noise of the n-th sensor. For instance, in the case of a GPS sensor, $z_{\mathsf{GPS}_{k+1}}$ would be a position and velocity measurement, whereas for a vision-based sensor, $z_{\mathsf{LM}_{k+1}}$ would be the pixel-coordinates information of a visible landmark. Then, the predicted state $\hat{\mathbf{x}}_{k+1}^-$ is corrected such as:

$$\begin{aligned}
\hat{\mathbf{x}}_{k+1} &= \hat{\mathbf{x}}_{k+1}^- + K_{S_{n_{k+1}}}\left(z_{S_{n_{k+1}}} - h_{S_n}\left(\hat{\mathbf{x}}_{k+1}^-\right)\right) \\
&= \hat{\mathbf{x}}_{k+1}^- + K_{S_{n_{k+1}}}\left(h_{S_n}\left(\mathbf{x}_{k+1}\right) - h_{S_n}\left(\hat{\mathbf{x}}_{k+1}^-\right) + \xi_{S_{n_{k+1}}}\right) \\
&\simeq \hat{\mathbf{x}}_{k+1}^- + K_{S_{n_{k+1}}}\left(\mathbf{H}_{S_{n_{k+1}}}\tilde{\mathbf{x}}_{k+1}^- + \xi_{S_{n_{k+1}}}\right)
\end{aligned} \tag{3.12}$$

where $\mathbf{H}_{S_{n_{k+1}}}$ is the measurement matrix such that:

$$\mathbf{H}_{S_{n_{k+1}}} = \left.\frac{\partial h_{S_n}(\mathbf{x})}{\partial \mathbf{x}}\right|_{\mathbf{x}=\hat{\mathbf{x}}_{k+1}^-} \tag{3.13}$$

and $K_{S_{n_{k+1}}} = \mathbf{P}_{k+1}^-\mathbf{H}_{S_{n_{k+1}}}^T\left(\mathbf{H}_{S_{n_{k+1}}}\mathbf{P}_{k+1}^-\mathbf{H}_{S_{n_{k+1}}}^T + \mathbf{R}_{S_n}\right)^{-1}$ is the gain that minimizes the estimation covariance error $\mathbf{P}_{k+1}$, designated Kalman gain. Then, the estimated error and its covariance are updated as:

$$\tilde{\mathbf{x}}_{k+1} = \left(\mathbf{I} - K_{S_{n_{k+1}}}\mathbf{H}_{S_{n_{k+1}}}\right)\tilde{\mathbf{x}}_{k+1}^- - K_{S_{n_{k+1}}}\xi_{S_{n_{k+1}}} \tag{3.14}$$

$$\mathbf{P}_{k+1} = \left(\mathbf{I} - K_{S_{n_{k+1}}}\mathbf{H}_{S_{n_{k+1}}}\right)\mathbf{P}_{k+1}^- \tag{3.15}$$

If there is no onboard sensor available for the Correction step, the estimation error and its covariance remain as those from the Prediction step (Equations 3.9 and 3.10).

Guidance module

A set of actions is defined in the planning problem (Section 3.3) such that each action comprises a desired velocity vector $\mathcal{V}_{ref}$ for the UAV. Given such desired velocity $\mathcal{V}_{ref}$, the following linear guidance law is applied to execute an action:

$$a_k = \hat{K}_p\mathcal{V}_{ref} - K_d\left(\hat{\mathcal{V}}_k - \mathcal{V}_{ref}\right) = K_p\mathcal{V}_{ref} - K_d\hat{\mathcal{V}}_k \tag{3.16}$$

where $K_p, K_d > 0$ are control gains and $\hat{\mathcal{V}}_k$ is the estimated UAV velocity at instant t_k, i.e. $\hat{\mathcal{V}}_k = \begin{bmatrix} 0 & \mathbf{I} & 0 \end{bmatrix} \hat{\mathbf{x}}_k$. Then, $\mathbf{x}_{k+1}$ can be obtained by substituting this guidance law into the discrete process model (Equation 3.3):

$$
\begin{aligned}
\mathbf{x}_{k+1} &= \mathbf{\Phi}\,\mathbf{x}_k + \mathbf{B}\left(K_p \mathcal{V}_{ref} - K_d \begin{bmatrix} 0 & \mathbf{I} & 0 \end{bmatrix} \hat{\mathbf{x}}_k\right) + v_{k+1} \\
&= \mathbf{\Phi}\,\mathbf{x}_k + \mathbf{B}\,K_p \mathcal{V}_{ref} - \mathbf{B}\,K_d \begin{bmatrix} 0 & \mathbf{I} & 0 \end{bmatrix} (\mathbf{x}_k - \tilde{\mathbf{x}}_k) + v_{k+1} \\
&= \left(\mathbf{\Phi} - \Delta\mathbf{\Phi}^{\mathcal{V}}\right) \mathbf{x}_k + \mathbf{B}\,K_p \mathcal{V}_{ref} + \Delta\mathbf{\Phi}^{\mathcal{V}} \tilde{\mathbf{x}}_k + v_{k+1}
\end{aligned} \tag{3.17}
$$

where $\Delta\mathbf{\Phi}^{\mathcal{V}} = \mathbf{B}\,K_d \begin{bmatrix} 0 & \mathbf{I} & 0 \end{bmatrix}$. Hence, given the current state $\mathbf{x}_k$, the state $\mathbf{x}_{k+1}$ follows the Gaussian distribution as described below:

$$
\begin{aligned}
\mathbf{x}_{k+1} &\sim \mathcal{N}\left(\left(\mathbf{\Phi} - \Delta\mathbf{\Phi}^{\mathcal{V}}\right) \mathbf{x}_k + \mathbf{B}\,K_p \mathcal{V}_{ref}, \Delta\mathbf{\Phi}^{\mathcal{V}}\,\mathbf{P}_k\,\Delta\mathbf{\Phi}^{\mathcal{V}^T} + \mathbf{Q}\right) \\
&= \mathcal{N}\left(\bar{\mathbf{x}}_{k+1|k}, \tilde{\mathbf{Q}}^a_{k+1}\right)
\end{aligned} \tag{3.18}
$$

where the covariance $\tilde{\mathbf{Q}}^a_{k+1}$ becomes a function of the estimation error covariance $\mathbf{P}_k$ given by the navigation system (Equation 3.10 or 3.15). Note that, this normal distribution defines the execution error for a time step $k|k+1$ of the path segment. Recalling an action effect at the decision time of the planning problem, the step t (epoch) $-$ see Figure 3.2 $-$ is the result of a path segment that considers several steps $k|k+1$. In the following section this relation is formalized.

State probability density function

Consider an initial state $\mathbf{x}(t_0) = \mathbf{x}_0$ and an initial error covariance $\mathbf{P}_0$, such that $\tilde{\mathbf{x}}_0 = \mathcal{N}(0, \mathbf{P}_0)$. For a given action a, it is possible to compute the distribution of the next state $\mathbf{x}_1 = \mathbf{x}(t_0 + \Delta t)$, given that Δt is considered n times k steps, as:

$$
f_X(\mathbf{x}_1|\mathbf{x}_0) \sim \mathcal{N}(\bar{\mathbf{x}}_{1|0}, \tilde{\mathbf{Q}}^a_1) \tag{3.19}
$$

Simultaneously, the state estimation error covariance is updated to $\mathbf{P}_1$ according to the selected navigation mode (Equation 3.10 or 3.15). As mentioned in Section 3.1, the GNC transition model and the planning model work at different rates, meaning that a single state transition from s_0 to s_1 with action a in the planner, makes up several state transitions from $\mathbf{x}_0$ to $\mathbf{x}_n$ in the system's model. Therefore, state transitions in the system must continue further up to $n > 1$ with the same action a. The conditional state transition at t_k, knowing $\mathbf{x}_0$, can be obtained sequentially as follows:

$$
f_X(\mathbf{x}_k|\mathbf{x}_0) \sim \int f_X(\mathbf{x}_k|\mathbf{x}_{k-1}) f_X(\mathbf{x}_{k-1}|\mathbf{x}_0)\, d\mathbf{x}_{k-1} \tag{3.20}
$$

where $f_X(\mathbf{x}_k|\mathbf{x}_{k-1}) \sim \mathcal{N}(\bar{\mathbf{x}}_{k|k-1}, \tilde{\mathbf{Q}}^a_{k-1})$. In parallel, the Kalman filter process is repeated k times to obtain $\mathbf{P}_k$. Assuming, as simplifications, that $\tilde{\mathbf{Q}}_k$ and $\mathbf{P}_k$ do not depend on the state $\mathbf{x}_{k-1}$ and that the sensor is available during the entire action, than the state transition function in the planning model can

be given as in Equation 3.21 when $k = n$.

$$f_X(\mathbf{x}_k|\mathbf{x}_0) \sim \mathcal{N}(\bar{\mathbf{x}}_{k|0}, \tilde{\Sigma}_k), \quad k > 1 \tag{3.21}$$

where $\tilde{\Sigma}_k$ is the execution error covariance.

The state transition function from s_0 to s_1 can be re-written with a notation from the planning model as shown below.

$$f_S(s_1|s_0) = f_X(\mathbf{x}_n|\mathbf{x}_0) \sim \mathcal{N}(\bar{\mathbf{x}}_{n|0}, \tilde{\Sigma}_n) = \mathcal{N}(\bar{s}_1, \Sigma_1) \tag{3.22}$$

Equation 3.22 constitutes the link between the GNC's closed-loop transition model and the MOMDP planning model.

3.3 MOMDP model

Following [5], the PO-SSP path planning problem addressed is modelled as a MOMDP problem, described in Section 2.4.3. As mentioned previously, in MOMDPs the state space is factorized into partially observable state variables and fully observable state variables. In this way, the belief state space (distribution probability over states) has a smaller dimension compared to the classical POMDP framework, which allows to decrease the processing time for value and policy computation.

It is assumed that the vehicle always knows which sensors are available at each decision time step. Thus the sensor's availability is considered a fully observable state variable of the model. On the other hand, the vehicle state vector $\mathbf{x}$ is taken as a hidden (non observable) state from the planning model point of view. Given the GNC transition model described in Section 3.2, the only output considered is the execution error distribution (bounded by the covariance matrix $\tilde{\mathbf{Q}}$) and so, neither partial nor direct symbolic observation is possible for it.

The MOMDP considered, represented in Figure 3.3, is defined as a tuple ($\mathcal{S}_v, \mathcal{S}_h, \mathcal{G}, \mathcal{A}, \Omega, \mathcal{T}, \mathcal{O}, \mathcal{C}, b_0$), such that:

- $\mathcal{S}_v$ is the set of fully observable states;
- $\mathcal{S}_h$ is the set of hidden continuous states;
- $\mathcal{G}$ is the set of fully observable target states, whose states $s_g \in \mathcal{G}$ are composed only by the 3D position;
- $\mathcal{A}$ is the set of discrete actions;
- Ω is the set of discrete observations;
- $\mathcal{T}(s_v, s_h, a, s'_v, s'_h) \rightarrow [0, 1]$ is the state transition function;
- $\mathcal{O}$ is the observation function, such that:

$$\mathcal{O}(s'_v, s'_h, a, o) = \Pr(o|s'_v, s'_h, a) = \begin{cases} 1, \text{if } o = s'_v \\ \\ 0, \text{otherwise} \end{cases} \tag{3.23}$$

– $\mathcal{C} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}_0^+$ is the cost function, where $\mathcal{S} = \mathcal{S}_v \times \mathcal{S}_h$;

– $b_0 = (s_v^0, b_{\mathcal{S}_h}^0)$, where $b_{\mathcal{S}_h}^0 \in \mathcal{B}_h$ is the initial probability distribution over the initial hidden continuous state, conditioned to $s_v^0 \in \mathcal{S}_v$, the initial fully observable state.

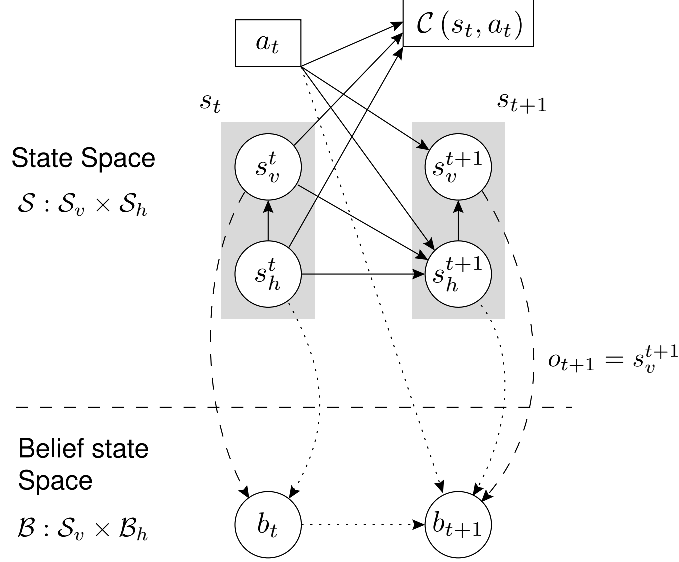


Figure 3.3: MOMDP model considered [8].

State Space

The complete state space is described by (s_v, s_h) such that $|\mathcal{S}| = |\mathcal{S}_v| \times |\mathcal{S}_h|$ represents the complete domain of the state space.

The visible state $s_v \in \mathcal{S}_v$ is defined as a tuple $s_v = (F_{S_1}, F_{S_2}, ..., F_{S_N}, F_{Col}, \mathbf{P})$, where F_{S_i} defines the fully observable boolean state variable for the availability of sensor S_i, $i = 1, ..., N$, and F_{Col} represents a boolean variable for a collision flag. It is assumed that the latter is also fully observable, for instance, by measuring or estimating a force of contact. Finally, $\mathbf{P}$ is the localization error covariance matrix computed by the Kalman Filter (Section 3.2.2). It is necessary to include this matrix in the state definition, as it is propagated by the navigation module in function of the selected action in each decision step.

The hidden continuous state $s_h \in \mathcal{S}_h$ is defined as $s_h = \mathbf{x}$, recalling that $\mathbf{x}$ is the continuous vehicle state vector defined by $\mathbf{x} = \begin{bmatrix} \mathcal{X}^T & \mathcal{V}^T & \beta_a^T \end{bmatrix}^T$. Apart from the vehicle position $\mathcal{X}$, it is necessary to consider the velocity $\mathcal{V}$ and the accelerometer bias β_a in the state s_h, as they are considered in the transition function to estimate the next state.

Action Space

An action $a \in \mathcal{A}$ is defined as a tuple $a = (\{\mathcal{V}_{ref}\}, m_n)$, where $\mathcal{V}_{ref}$ is the reference velocity in the guidance module (see Equation 3.16), $\{\mathcal{V}_{ref}\}$ defines a finite set of possible $\mathcal{V}_{ref}$ and $m_n \in \{S_1, ..., S_N\}$ is the navigation mode to be considered at each planning epoch t, depending on the sensors' availability.

Observation Space and Observation Function

As a result of the factorization of the observation space and the specificity of the model chosen, the set of observations Ω is equal to $\mathcal{S}_v$, and therefore the observation function $\mathcal{O}$ is deterministic, because $o = s'_v$ (Equation 3.23) regardless of s'_v, s'_h and a. However, it is important to be noted that there is still uncertainty attached to the hidden state, because in the MOMDP model used (Figure 3.3) the visible state s'_v depends on s'_h, which itself depends on the previous states s_v and s_h, thus $\Pr(s'_v|s'_h, a)$ is not deterministic.

Although the agent receives no direct or even imprecise observation on the state s_h, the propagation dynamics of the error execution depending on the choice of navigation mode are known. This modelling choice was made to avoid considering sensor measurements, which are not accessible at the moment of planning, and therefore avoid working with a continuous observation space, recalling that POMDPs have an exponential complexity with the size t of the history, being $(|\mathcal{A}||\Omega|)^t$ the branching factor for tree search solving methods. This allows to have a number of observations equal to the number of visible states (available sensors and associated matrix $\mathbf{P}$), reducing the complexity of the problem.

Transition function

The transition function $\mathcal{T}(s_v, s'_v, a, s'_h, s_h)$ is factorized according to the state space and, thereby, is composed of two functions:

– a transition function $\mathcal{T}_{\mathcal{S}_h}$ such as:

$$\mathcal{T}_{\mathcal{S}_h}(s_h, s_v, a, s'_h) = \Pr(s'_h|s_h, s_v, a) \sim \mathcal{N}(\bar{s}'_h, \tilde{\Sigma}'(s_v)) \tag{3.24}$$

which is based on the GNC closed-loop model, given that the probability distribution of a predicted state s'_h follows a normal distribution $\mathcal{N}(\bar{s}'_h, \tilde{\Sigma}'(s_v))$ (see Equation 3.22), which, in turn, is a function of the previous hidden state s_h, the previous visible state s_v and the action a.

– a transition function $\mathcal{T}_{\mathcal{S}_v}$ such that $\mathcal{T}_{\mathcal{S}_v}(s'_h, s'_v) = \Pr(s'_v|s'_h)$, which represents the transition function for s'_v and depends only on the probabilistic sensors availability maps and the obstacle map, and thereby only on the next state s'_h. Concretely, $\Pr(s'_v|s'_h)$ is the product between the probabilities of the availability of each onboard sensor,

$$\Pr(s'_v|s'_h) = \Pr\left(F'_{Col}|s'_h\right) \prod_{i=1}^{|\mathcal{S}_v \setminus \{\mathbf{P}, F_{Col}\}|} \Pr(F'_{S_i}|s'_h) \tag{3.25}$$

where $|\mathcal{S}_v \setminus \{\mathbf{P}, F_{Col}\}|$ is the number of on-board sensors. $\mathbf{P}$ is not considered in the transition function $\mathcal{T}_{\mathcal{S}_v}$ because the transition of $\mathbf{P}$ is deterministic for a given navigation mode.

Thus, the complete transition function becomes:

$$\begin{aligned}\mathcal{T}(s_v, s'_v, a, s'_h, s_h) &= \mathcal{T}_{\mathcal{S}_v}(s'_h, s'_v)\, \mathcal{T}_{\mathcal{S}_h}(s_h, s_v, a, s'_h) \\ &= \Pr(s'_v|s'_h)\, \Pr(s'_h|s_h, s_v, a) \\ &= \Pr(s'_v, s'_h|s_h, s_v, a) \end{aligned} \tag{3.26}$$

Belief State

The belief state b represents the probability distribution over states. In this MOMDP model, the belief state can be factorized into a probability distribution over the hidden state space $\mathcal{S}_h$ conditioned on the fully observable state s_v, such as $b^{\mathcal{S}_v}_{\mathcal{S}_h} = (s_v, b_{\mathcal{S}_h})$. The current belief state is updated after each action a and each perceived visible state s'_v, in accordance with the transition and observation functions, by using the Bayes' rule (see Equation 2.17). Therefore, the belief state update is decomposed into two sub-functions:

1. Given a belief state b_{s_h} and an action a, the GNC transition function propagates the belief state to the next one, b_a, thus defining a belief state transition related with the action execution error propagation:

$$b_a(s'_h) = \Pr(s'_h|b, a) = \int_{\mathcal{S}_h} \Pr(s'_h|s_h, s_v, a)\; b(s_h)\; ds_h \tag{3.27}$$

2. The second function is related with the probability of observing s'_v, given by the probability grid maps, based on the belief b_a (see [3] for more details):

$$\Pr(s'_v|b, a) = \sum_{i=0}^{|G|} \Pr(s'_v|s'_h \in c_i)\; \Pr(s'_h \in c_i|b, a) \tag{3.28}$$

where c_i corresponds to the i-th cell of the probability map and $|G|$ is the number of cells in the map.

Finally, the complete new belief state $b^{s'_h}_a$ can be written as:

$$b^{s'_h}_a(s'_h) = \Pr(s'_h|b, a, s'_v) = \frac{\Pr(s'_h, s'_v|b, a)}{\Pr(s'_v|b, a)} = \frac{\Pr(s'_v|s'_h)\Pr(s'_h|b, a)}{\Pr(s'_v|b, a)} \tag{3.29}$$

Cost function

The cost function to be minimized is defined accounting for the UAV flight time and a cost of collision. It is expected that by minimizing the cost, the algorithm will minimize the flight time (for efficiency) and the probability of collision (for safety) at the same time. More precisely, the cost function is defined as:

$$\mathcal{C}(s_t, a_t) = \begin{cases} 0, & \text{if } s_t \in \mathcal{G} \\ K - \sum_{k=0}^{t-1} \mathcal{C}(s_k, a_k), \forall a_k \in \mathcal{A} & \text{if } s_t \text{ is in collision} \\ f_t, & \text{otherwise} \end{cases} \tag{3.30}$$

where f_t is the flight time for a given action a at decision step t and K is a fixed cost in case of collision. When a collision occurs, the cost of any action is a fixed penalty subtracted with the total flight time since the initial belief state until the collision state. This trick avoids penalizing more if the collision occurs after a longer flight time or near the goal.

For simplicity, it is assumed that all action durations coincide with the constant planning epoch, which generates a constant f_t for any action a_t and any state s_t at any depth t. Furthermore, this assumption removes the dependency of the cost function on the action, consequently leading to the following simplification in Equation 3.30, in case of collision:

$$\mathcal{C}(s_t, a_t) = \mathcal{C}(s_t) = K - t\, f_t. \tag{3.31}$$

Value function

As mentioned in Section 2.3, the aim of solving a POMDP problem is to find a policy $\pi : \mathcal{B} \rightarrow \mathcal{A}$, where $\mathcal{B}$ define the belief state space, which optimizes a given criterion usually defined by a value function. In the MOMDP problem addressed, the value function $V^\pi(b)$ is defined as the expected total cost payed from starting in a certain belief state b and following policy π. Therefore, Equations 2.8 and 2.12 are combined into the following form:

$$V^\pi(b) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \mathcal{C}(b_t, \pi(b_t)) \middle| b_0 = b \right] \tag{3.32}$$

where $\mathcal{C}(b_t, \pi(b_t) = a)$ is the expected cost of executing an action a in belief state b.

Then the optimal value function $V^*(b)$ is given as:

$$V^*(b) = \min_{a \in \mathcal{A}} \left[\mathcal{C}(b, a) + \sum_{s_v \in \mathcal{S}_v} \Pr(s_v | b, a) V^*(b_a^{s_v}) \right] \tag{3.33}$$

When the value (Equation 3.33) converges for all belief states, it is possible to extract the optimal policy π^*, given as:

$$\pi^*(b) = \arg\min_{a \in \mathcal{A}} \left[\mathcal{C}(b, a) + \sum_{s_v \in \mathcal{S}_v} \Pr(s_v | b, a) V^*(b_a^{s_v}) \right] \tag{3.34}$$

Chapter 4

Offline Approach

This chapter presents the offline approach. Section 4.1 describes the planning algorithm proposed by [5] to solve the MOMDP problem addressed in this work, in an offline configuration. This allows a better understanding of the performance and properties of the algorithm, which gives way to a parameter tuning section (Section 4.2). This section constitutes the first contribution of this dissertation, in which different action selection policies and back-propagation strategies are explored in the planning algorithm and whose results will then be used for the online approach in Chapter 5. In particular, a novel adaptive coefficient for action selection is proposed, which avoids extensive parameter tuning, while guaranteeing a given level of performance for further online planning.

4.1 Offline POMCP-GO Algorithm

Maintaining and updating belief states can be an extremely difficult and expensive step. Given the specificity of the MOMDP model chosen, comprising a continuous hidden state space and a discrete fully observable state space, the calculation of the probability distribution on $\mathcal{S}_h$ and posterior correction by s'_v (Equation 3.29) would require an expensive computational effort. [8] explored the use of a Gaussian Mixture Model (GMM) to approximate the new belief, but this technique still emerged as being computationally heavy. To tackle this problem, an interesting solution is to apply algorithms that do not need to maintain an explicit representation of the belief state at each decision stage, such as the POMCP algorithm presented in Section 2.5.7.

Nevertheless, the classical POMCP algorithm does not completely meet the needs for path planning towards a goal state, therefore a new variant was proposed. Such algorithm is called POMCP-GO as it is a goal oriented version of the POMCP algorithm. The main differences between the classic version of the POMCP (Algorithm 1) and the one presented are hereafter discussed. Algorithm 2 fully describes the implementation of POMCP-GO.

Firstly, a computational budget as a number of trials to perform is used (line 2). In an offline configuration, all trials start from the initial belief state $V(b_0)$, that is, the root of the exploration tree (line 3), rather than at the current belief state as done in the online configuration. Then, belief states are represented

Algorithm 2: POMCP-GO Offline

```
1  Function POMCP-GO Offline(h, b_0):
2      while nbTrial < nb_max do
3          s_h ~ b_0
4          Trial(h, s_h, s_v, 0)
5          nbTrial += 1
6      return a* ← argmin_{a∈A} Q(b_0, a)
7  Function Trial(h, s_h, s_v, d):
8      if s_h ∈ G then
9          return 0
10     if F_Col == 1 then
11         return K − t f_t
12     if h ∉ T then
13         for a ∈ A do
14             T(ha) ← (N_init(ha), Q_init(h, a))
15         V_init(h) ← min_{a∈A} Q(h, a)
16     ā ← argmin_{a∈A} { Q(h, a) − c sqrt(log N(h) / N(ha)) }
17     (s'_h, s'_v, C(s_h, ā)) ~ G(s_h, ā)
18     Q(h, ā)' ← C(s_h, ā) + Trial(hao, s'_h, s'_v, d + 1)
19     N(h) ← N(h) + 1
20     N(hā) ← N(hā) + 1
21     Q(h, ā) ← Q(h, ā) + (Q(h, ā)' − Q(h, ā)) / N(ha)
22     V(h) ← min_{a∈A} Q(h, a)
```

indirectly through the POMCP property of generating sequences of history nodes and sequence nodes. Since it is a goal-oriented problem, [5] proposed that each trial should end up only in a terminal state (goal or collision), instead of being stopped every time a new node is initialized. In the case of reaching a goal, a cost of 0 is returned (line 9); whereas if collision is detected, a collision cost is returned (line 11), as defined by Equation 3.30.

Recalling the POMCP algorithm (Algorithm 1), when a new history node h is explored, a sequence node ha is created and initialized for each action a. This initialization is done with a rollout method to estimate the Q-value of each new node ha, by simulating sequences of random action-observation pairs starting from this new node. At each step, the action a is chosen with a UCB1-based selection strategy (line 16) and the next new state s' and observation o are generated using the black-box simulator. The algorithm then continues to explore from the new node $hao = \{h, a, o\}$.

Differently from POMCP, in POMCP-GO (Algorithm 2), if a new node ha is created, its value is estimated using an initial heuristic value and not a rollout method. Such informative heuristic initialization is described in Section 4.1.1. As the Q-value is progressively updated throughout the simulations, it allows to compute the value of $V(h)$ as $V(h) = \min_{a \in \mathcal{A}} Q(h, a)$ (line 15). At each step, the action a is

chosen with a UCB1-based selection strategy (line 16), the next new state s'_h is calculated using the GNC transition function, rather than the black-box simulator, and the visible state s'_v is randomly drawn according to the probabilities given by the sensor's availability maps. The algorithm then continues to explore from the new node $hao = \{h, a, s'_v\}$.

4.1.1 Heuristic function

Following the work proposed in [73], instead of applying a rollout method to each new node ha created, as done in the classical version of POMCP [26], a deterministic shortest path algorithm is used to initialize the Q-values and thereby to guide the search for UCB1 in the first explorations, until the goal is reached. This choice was inspired by the RTDP-Bel algorithm [38], which proposes to solve mainly goal-oriented POMDPs and that assigns values based on a heuristic function to belief states that have not been visited yet.

Given the planning problem addressed, the Dijkstra algorithm [18] (presented in Section 1.3) was chosen to estimate the distance between a certain position and the goal. In order to do so, the deterministic obstacle grid map was turned into a graph, in which each cell corresponds to a node. The cell containing the goal is the goal node and the edges of the graph are weighted by the estimated distance to move from one cell to another.

Once this graph is created, the Dijkstra algorithm computes the minimum distance for each node, which is then divided by the reference velocity to get the estimated flight time to reach the goal. These calculations are made a priori and the result is stored as a map, in order to be consulted by the algorithm whenever a new node is initialized. When a node h in the tree is explored for each action $a \in \mathcal{A}$, a new node ha is created. Using the GNC transition function, it is possible to compute the average state $\overline{s}'_h$ from the normal law defined by Equation 3.18. The cell containing this state will then be the node of the Dijkstra graph to be used to initialize the value $Q_{init}(h, a)$.

4.2 Proposed approach for parameter tuning

This section serves the purpose of applying the various Action selection strategies proposed in Section 4.2.1, as well as the Back-propagation strategies described in Section 4.2.2, for the POMCP-GO algorithm described in the previous section, in order to evaluate its performance, to verify the convergence of the algorithm over the trials and to find the most suitable approach to apply in an online configuration.

4.2.1 Action selection strategies for planning

As mentioned in sections 2.5.3 and 2.5.7, the Selection Policy, applied to the classic POMCP algorithm, defines the way to select the actions at history nodes. As an action selection strategy, a UCB1-based formula (Equation 2.28) is applied. Adapting this formula to the minimization problem here

addressed, one gets the following:

$$\bar{a}_{\text{UCB}} = \arg\min_{a\in\mathcal{A}} \left\{ Q(h,a) - c\sqrt{\frac{\log N(h)}{N(h,a)}} \right\} \tag{4.1}$$

Equation 4.1, similarly to Equation 2.28, depends on an exploration factor c, which is a constant value typically adjusted manually. This parameter varies significantly between planning domains, requiring an exhaustive search to find the most suitable value. This exhaustive search, applied to the complex path planning problem addressed in this work, may be extremely time consuming.

Therefore, besides applying this method as a Selection strategy, the performance of three other selection strategies is examined: decay with depth (DWD), entropy-based coefficient (EBC) and a two-steps sampling scheme based on simple regret and cumulative regret. While the first two techniques explore the use of adaptive coefficients in Equation 4.1 to avoid the exhaustive search inherent to the use of a fixed exploration factor c, the last approach introduces the notion of simple regret minimization into the selection procedure. These three strategies are explained hereafter.

Decay with depth

In online path planning, it is often more interesting to explore at the beginning or near obstacles than at the end, when the vehicle is close to the goal. Indeed, it is during the first periods that it is crucial to explore alternative paths, which anticipate likely collisions. For this reason, using a variable coefficient that decreases as time goes by is expected to promote the breadth-first search.

Using the following model proposed by [73], the coefficient starts with a higher value in the beginning, to support the exploration of available actions, and then as the depth of the tree increases, lower values of the coefficient are considered in order to give way to the exploitation. The exact formula of the Decay with Depth (DWD) coefficient for our application case is as follows:

$$c_{DWD} = \frac{C_k}{t}\,(K - t\,f_t) \tag{4.2}$$

where C_k is a constant value, K is the collision penalty and t is the depth of the tree since the beginning of the flight, i.e. $t = t_0$. Figure 4.1 depicts the evolution of the DWD coefficient over the depth of the tree.

The action selection formula in this case becomes:

$$\bar{a}_{\text{DWD}} = \arg\min_{a\in\mathcal{A}} \left\{ Q(h,a) - c_{DWD}\sqrt{\frac{\log N(h)}{N(h,a)}} \right\} \tag{4.3}$$

Entropy-based Coefficient

In this case, the suggested coefficient is based on a score related to a measure of the uncertainty about the sensors' availability. More specifically in our application case, the coefficient adapts according to the entropy of the probability grid map of the navigation sensors' availability. Then, during the planning process, the value of the coefficient lies within a specific user-defined interval $[c_{min}, c_{max}]$, encoding, on

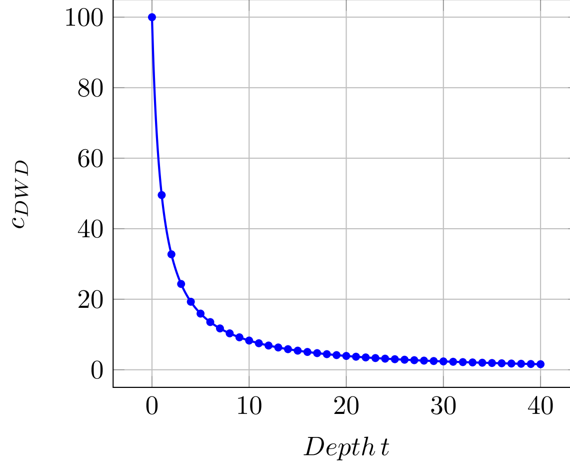


Figure 4.1: Decay of the coefficient c_{DWD} over the depth of the search tree.

the action selection strategy, the need to explore more thoroughly in the areas where the uncertainty is higher.

The Entropy-based Coefficient (EBC) is proportional to the maximum possible cost value as follows:

$$c_{EBC}(s_v, s_h) = e_n(s_v, s_h) \max_{\substack{s \in \mathcal{S} \\ a \in \mathcal{A}}} |\mathcal{C}(s, a)| \tag{4.4}$$

where $e_n(s_v, s_h) = (c_{max} - c_{min})\, e(s_v, s_h) + c_{min}$ is the normalized entropy to fit in $[c_{min}, c_{max}]$, and

$$e(s_v, s_h) = - \sum_{s_v \in \int_v} \Pr(s_v|s_h) \log_2(\Pr(s_v|s_h)) \tag{4.5}$$

is the value of the entropy according to the probability grid map of the navigation sensors' availability. In this case, $\int_v = (F_{S_1}, F_{S_2}, ..., F_{S_N})$ is a subset of $\mathcal{S}_v$ that only considers the fully observable boolean state variables F_{S_i}, i.e. neither $\mathbf{P}$ nor F_{Col} are accounted for during the computation of this coefficient.

The action selection formula in this case becomes:

$$\bar{a}_{\text{EBC}} = \arg\min_{a \in \mathcal{A}} \left\{ Q(h, a) - c_{EBC} \sqrt{\frac{\log N(h)}{N(h, a)}} \right\} \tag{4.6}$$

Simple Regret (SR) + Cumulative Regret (CR) MCTS Sampling Scheme

Recently, *simple regret* has been proposed as a new criterion for assessing the performance of both MAB [74] and MCTS [66] [64] algorithms. In order to introduce this notion, first consider the MAB problem described in Section 2.5.3. In a cumulative regret setting, every choice made in each play has a direct effect on the agent's reward. As such, the agent wants to minimize the number of sub-optimal arms pulled in order to achieve a reward as high as possible. Now suppose that the agent can perform a number of trials without consequence, in a simulated environment, after which it must make a recommendation (i.e. must choose the final move). Based only on its recommendation, the agent is rewarded. In this case, the performance of the agent is measured by the simple regret of its recommendation. A low simple regret implies that the recommendation is close to the actual best option.

In this way, simple regret, r_n, is defined as the expected difference between the reward μ_j of the arm with the greatest sample mean, $j = \arg\max_i \overline{X}_i$, and the optimal reward, μ^*:

$$r_n = \mu^* - \mu_j \tag{4.7}$$

In [66], authors argue that, at the root node, the sampling in MCTS is usually aimed at finding the first move to perform. For this, one needs to promote exploration. Once one move is shown to be the best choice with high confidence, the value of information of additional samples of the best move is low. Therefore, simple regret becomes a naturally fitting quantity to optimize at the root node, rather than cumulative regret.

For this purpose, they propose a simple regret minimization sampling scheme similar to UCB1, but samples the current best arm less often. This can be achieved by replacing the $\log(\cdot)$ operator in Equation 2.28, with a faster growing sub-linear function, such as $\sqrt{(\cdot)}$:

$$\bar{a}_{\text{UCB}_{\sqrt{(\cdot)}}} = \arg\min_{a \in \mathcal{A}} \left\{ Q(h,a) - c\sqrt{\frac{\sqrt{N(h)}}{N(h,a)}} \right\} \tag{4.8}$$

In order to support an optimal move choice at the root, it is beneficial in many cases to find a more precise estimate of the value of the state in the nodes deeper in the tree, i.e. to encourage exploitation. Thus a cumulative regret minimization scheme becomes a better option at this point. Taking this into consideration, [66] propose a two-stage sampling scheme, SR+CR MCTS, which selects an action at the current root node according to a scheme suitable for minimizing the simple regret (SR), such as $\text{UCB}_{\sqrt{(\cdot)}}$, and then at non-root nodes selects actions according to UCB1, which approximately minimizes the cumulative regret (CR).

In this case, the exploration factor c in Equation 4.8 has to be tuned in the same manner as in the UCB1. However, the authors argue that this approach is expected to be significantly less sensitive to the tuning of this parameter and also to outperform the classic UCB1 in terms of convergence rate of the value function.

4.2.2 Back-propagation strategies

The backup function defines how the knowledge on state-value estimates $V(h)$ and action-value estimates $Q(h,a)$ that is gathered in the trials is propagated through the tree [70]. Depending on the algorithm, any number of additional parameters might be updated, for instance the visitation counter, as done in POMCP, or a solve label that indicates that a node's estimate has converged.

In this work, two different Q-value approximations are tested, resulting in two distinct back-propagation strategies. For both approaches, the state-value estimates are updated based on the best successor.

Classical POMCP

In this strategy, the action-value estimates are updated based on the mean return from all trajectories started when action a was selected in history h. Let $Q(h,a)'$ be the simulated return computed recursively over the trajectories, then the state-value and action-value estimates are calculated as:

$$\begin{cases} V(h) \leftarrow \min_{a \in \mathcal{A}} Q(h,a) \\ Q(h,a) \leftarrow Q(h,a) + \dfrac{Q(h,a)' - Q(h,a)}{N(ha)} \end{cases} \tag{4.9}$$

This strategy is the one commonly adopted in the POMCP algorithm. As the action-value estimate $Q(h,a)$ averages over all trajectories started in that action a, and not over trials starting with a and following the optimal policy, this might cause a potential pitfall: if a trajectory yields a very high cost compared to an optimal one, a single trial over said course can bias $Q(h,a)$ disproportionately over many trials [70].

MinPOMCP

This strategy derives from the MaxUCT algorithm proposed in [70] and constitutes part of the contributions of this dissertation. In this case, the estimation of the action-values is based on the value of its best successor, rather than on all trajectories, as follows:

$$\begin{cases} V(h) \leftarrow \min_{a \in \mathcal{A}} Q(h,a) \\ Q(h,a) \leftarrow C(h,a) + \dfrac{\sum_{hao} N(hao)\, V(hao)}{N(ha)} \end{cases} \tag{4.10}$$

where

$$C(h,a) \leftarrow C(h,a) + \frac{\mathcal{C}(s_h,a) - C(h,a)}{N(ha)} \tag{4.11}$$

is the mean immediate cost of executing action a in history h. As a result of applying this strategy, the contributing sub-tree in the back-propagation step is identical to the best partial solution tree and, therefore, the pitfall discussed in the Classical POMCP strategy no longer applies.

4.3 Evaluation based on offline planning and simulations

4.3.1 Configuration

A total of 8 case studies are examined: 2 environment maps, for which 2 distinct sensor's availability maps, with 2 different initial states each, are considered. A total of 10 value and policy optimizations are performed for each case study. The total number of trials in each optimization process is set to 50000. For each 5000 trials, 1000 simulations are performed to evaluate the policy being currently optimized.

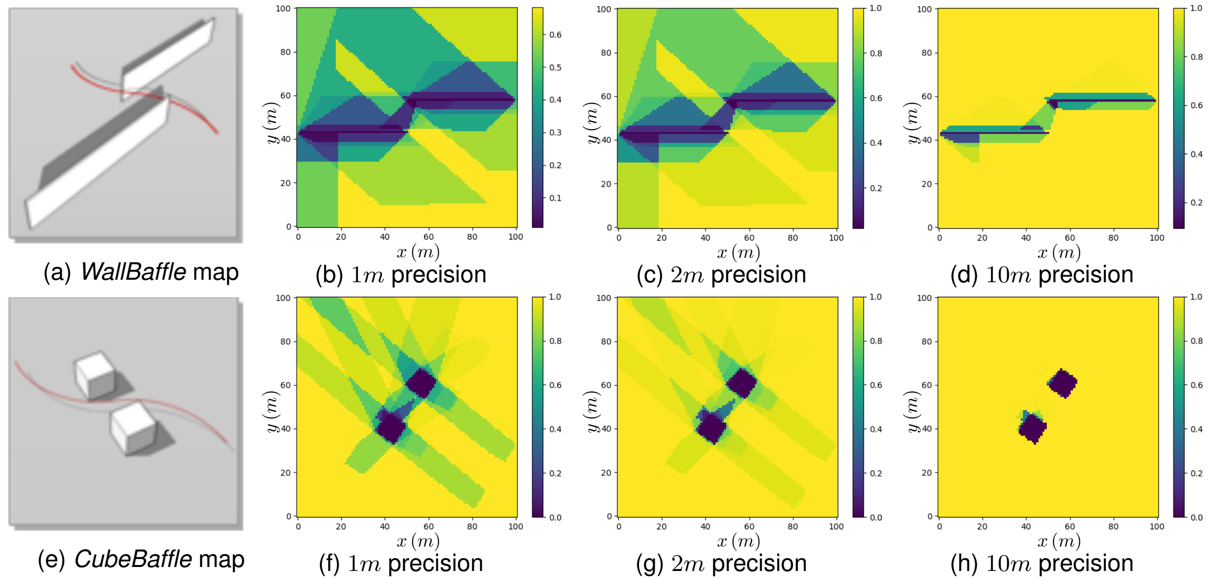
Sensors: The vehicle model used is the one defined in Section 3.2 of Chapter 3, considering only two onboard sensors: INS, available all the time, and GPS, available according to the probability maps.

Environments: Simulations are conducted on a benchmarking framework for UAV obstacle field navigation proposed in [75], which provides environments with different obstacle configurations. Two benchmarks are selected:

WallBaffle: map containing two walls as obstacles (Figure 4.2a);

CubeBaffle: map containing two cubes as obstacles (Figure 4.2e);

with a grid size of $100 \times 100 \times 20$ cells, where each grid cell has the size of $2m \times 2m \times 2m$.



(a) *WallBaffle* map (b) $1m$ precision (c) $2m$ precision (d) $10m$ precision

(e) *CubeBaffle* map (f) $1m$ precision (g) $2m$ precision (h) $10m$ precision

Figure 4.2: Obstacle maps proposed on the test benchmark [75] as well as examples of probability maps of GPS availability with different precision thresholds, given in $meters$.

A priori knowledge on the GPS availability is supposed. As done in [5], probability grid maps for GPS availability are used for different GPS precision thresholds, given in meters. These thresholds yield different probabilistic availability maps. For instance, if the $1\,meter$ precision GPS availability map indicates a probability of 60% in a given cell, it means that there is a 60% chance that GPS will be available with $1\,meter$ of precision in this cell. Moreover, these precision thresholds are only used to generate different GPS availability maps, and not used in the Kalman filter.

For the *WallBaffle* map, $2m$ and $10m$ GPS precision thresholds are examined, while in the *CubeBaffle* map, $1m$ and $2m$ GPS precisions are chosen. To be noted that when less GPS precision is required, more likely the GPS availability is. Figure 4.3 illustrates the set of maps considered, which is composed by the probability map on GPS availability as well as a map of the environment (obstacles).

Initial conditions: The initial belief state is defined as $b_0 = (s_v^0, b_{\mathcal{S}_h^0} = (\bar{s}_h^0, \tilde{\Sigma}_0))$, where:

- $s_v^0 = [1, 1, 0, \mathbf{P}]$ is the initial visible state, where $\mathbf{P} = \tilde{\Sigma}_0$;
- $\tilde{\Sigma}_0 = diag(1, 1, 1, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01)$;
- Two initial states s_h^0 are considered for each map:

  WallBaffle: $s_h^0 = [10, 25, 5, 0, 0, 0, 0, 0, 0]$ and $s_h^0 = [50, 25, 5, 0, 0, 0, 0, 0, 0]$, representing the initial positions $(10, 25, 5)m$ and $(50, 25, 5)m$, respectively;

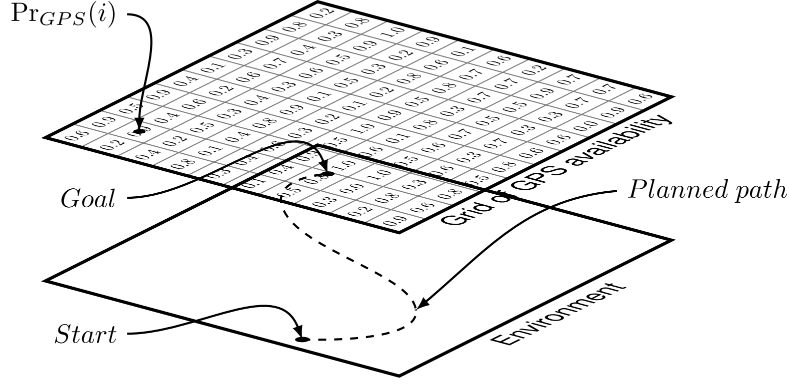


Figure 4.3: Representation of the probability map on GPS availability and the environment map.

CubeBaffle: $s_h^0 = [35, 20, 5, 0, 0, 0, 0, 0, 0]$ and $s_h^0 = [65, 20, 5, 0, 0, 0, 0, 0, 0]$, representing the initial positions $(35, 20, 5)m$ and $(65, 20, 5)m$, respectively;

The target state is defined at the position $s_g = (50, 80, 5)m$ and is considered reached if the position comprised in s_h is within a margin of $3\,meters$ from s_g.

Actions: An action is composed by a reference speed and a navigation mode depending on the choice of the sensor. The set of $\mathcal{V}_{ref}$ is composed by 26 reference speeds, which can be represented as directions (Figure 4.4), for each navigation sensor, thus comprising a total of 52 possible actions. Parameters of the GNC model are configured to ensure an action duration of 4 seconds.

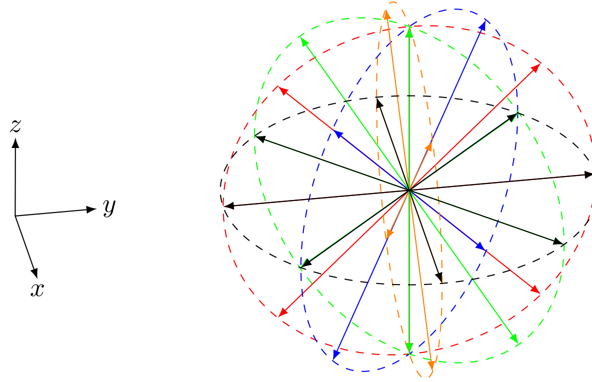


Figure 4.4: Set of directions considered.

Parameters: The collision cost is fixed at $K = 450$. Moreover, for the implementation of the proposed selection strategies, extra parameters need to be configured:

- **UCB1** (UCB(c)): extensive search is performed for the values in $c = (0.01, 0.05, 0.1, 0.5, 1, 5, 10)$;
- **Decay with Depth** (DWD): $C_k = 0.2222$, so that the first coefficient c_{DWD} has the value of $c_{DWD} = 0.2222 \times K = 100$ with the evolution depicted in Figure 4.1;
- **Entropy-based Coefficient** (EBC): $[c_{min}, c_{max}] = [0, 0.0222]$ so that c_{EBC} lies within an interval of $[0, 10]$, to be comparable with the values examined in UCB1;
- **SR+CR MTCS** (UCB$_{\sqrt{(\cdot)}}(c)$): only the coefficient that showed the best results in UCB1 is used in this approach.

Metrics: To evaluate the quality of the policy computed, the following metrics are examined:

- The **value of the initial belief state $V(b_0)$ during the optimization**. This metric is used to check the convergence in value of the policy being optimized for the initial state of belief;
- The **value of the initial belief state $V(b_0)$ executed** or simulated. The value of b_0 during optimization is biased because it is the average cost of all trials and therefore does not represent the actual value of b_0. The value of executed b_0 is based on a formula derived by [5] and is computed with the results of the simulations, as follows:

$$V(b_0)_{exe} = p_c K + (1 - p_c)T \tag{4.12}$$

  where p_c is the collision probability at the initial belief state b_0 when following the optimized policy during the simulations, K is the collision penalty and T is the average total flight time to reach the goal, starting from b_0 knowing that no collision occurred, i.e. $F_{Col_k} = 0, \forall k \geq 0$;
- The **success rate**, which is the ratio between the number of simulations that ended up in the goal state and the total number of simulations performed (in %);
- The **average flight time** T for the successful simulations (without collision), which allows to verify the minimization of the flight time;
- The **computational time per optimization** is an indicator of the computational complexity inherent to the strategies.

Strategies: Two back-propagation strategies are tested in this step: Classical POMCP and MinPOMCP. Classical POMCP is tested for each of the 4 selection strategies, while MinPOMCP is performed only for the coefficient that showed the best results in UCB1 and for the EBC, making it a total of 6 different combinations used. Selection strategies using the MinPOMCP back-propagation approach are identified with an additional subscript MP.

4.3.2 Results

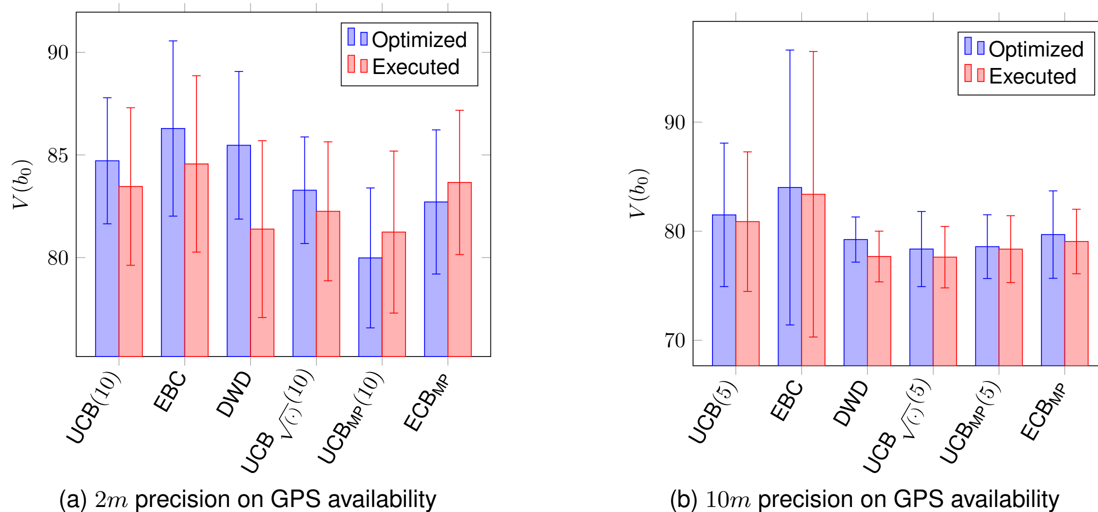
The results showed in this section reflect only the values obtained after the total number of 50000 trials were reached. Figures 4.5, 4.6, 4.7 and 4.8 represent the results of the metrics $V(b_0)$ optimized and $V(b_0)$ simulated for all the approaches. The data is organized in the form of *bars*, being the tip of the filled bar the average, while the extremes of the vertical lines represent the standard deviation. Tables 4.1 and 4.2 register the average and standard deviation values for the success rate, average flight time and computation time per optimization.

Appendix A.1 depicts the results of the metrics $V(b_0)$ optimized and $V(b_0)$ simulated for the extensive search process using the UCB1 formula, along with the results of the EBC for comparison. Appendix A.2 shows the evolution of the metrics $V(b_0)$ optimized and $V(b_0)$ simulated during the optimization process, i.e. for each 5000 trials, for the best and the worst strategy in each case study. Furthermore, examples of the 1000 trajectories executed after the 50000 trials are also shown in Appendix A.2.

Regarding the initial step of the exhaustive hand-tuning process using the UCB1 formula, Figures A.1, A.2, A.3 and A.4 show that the UCB1's best fixed coefficient varies significantly, not only across the different GPS precision probabilistic availability maps considered, but also when changing initial positions in the same map. For this reason, the use of a coefficient that can be dynamically tuned to fit all the different planning domains becomes a very promising solution to avoid the previous extensive parameter tuning, being therefore more appropriate for an online planning configuration. Moreover, comparing the UCB1 fixed coefficients and the EBC results, one can verify that both $V(b_0)$ optimized and simulated computed for EBC lie within the values obtained for the fixed coefficients for every case study, although never reaching the values of the best fixed coefficient. These results were expected, since the application of EBC implies the use of values within the user specified interval, which comprises not only the optimal coefficient, but also others that do not generate results as satisfying. In this case, using $c_{EBC} \in [0, 10]$, values close to 10 were used where the uncertainty on GPS availability was higher, encoding the need for more exploration in these areas, otherwise lower values close to 0 were applied. Therefore, a compromise must be made between the desired quality of the results and the time one is willing to spend on the exhaustive search for the optimal coefficient value before online planning.

WallBaffle **map**

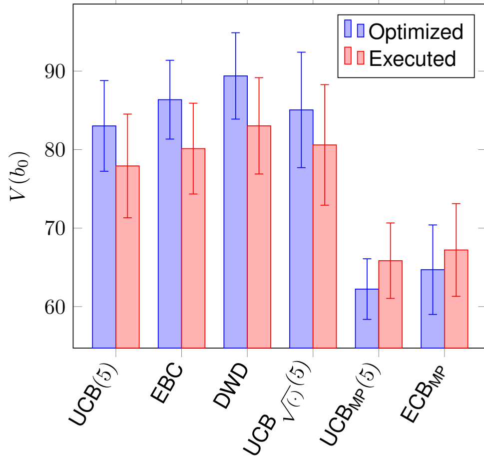
Figures 4.5 and 4.6 and Table 4.1 comprise the results for the *WallBaffle* map. The GPS availability maps chosen allow to detect a difference in the exploration factor c of the UCB1 formula that showed the best results: the lower the GPS precision required, more likely the GPS availability is and less exploration is needed, resulting in a lower factor c. Consequently, the average computation time per optimization is also lower for a lower c, as can be seen in Table 4.1.



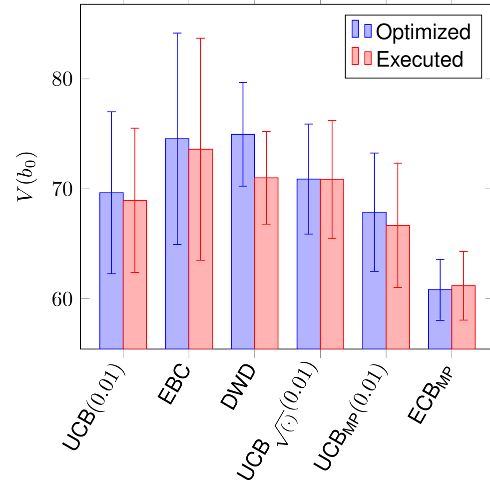
(a) $2m$ precision on GPS availability (b) $10m$ precision on GPS availability

Figure 4.5: *WallBaffle* map with initial position in $(10, 25, 5)m$, using different combinations of action selection and back-propagation strategies.

Focusing on the results obtained when applying the Classical POMCP back-propagation strategy, it can be immediately verified that the $V(b_0)$ optimized is always higher or close to the simulated $V(b_0)$, for all the selection policies. As explained in Section 4.2.2, the action-value estimates $Q(h, a)$ in the POMCP



(a) $2m$ precision on GPS availability



(b) $10m$ precision on GPS availability

Figure 4.6: *WallBaffle* map with initial position in $(50, 25, 5)m$, using different combinations of action selection and back-propagation strategies.

algorithm are averages of all trials starting with a, regardless of following the optimal policy afterwards. If trajectories holding a very high cost (collision cost) are encountered, $Q(h, a)$ will retain a higher value over many trials. Thus, the value of optimized b_0 will often be higher than the simulated value.

Regarding the selection strategy Decay with Depth (DWD), it shows the highest difference between the optimized $V(b_0)$ and the executed $V(b_0)$, which may indicate that the optimized $V(b_0)$ needed more trials to converge to a lower value. For the initial position $(10, 25, 5)m$ in both GPS precision maps, it yielded the lowest simulated $V(b_0)$, influenced by the success rate being the highest. However, this approach does not offer consistency, reaching the highest $V(b_0)$ values, along with the worst success rates, for the initial position $(50, 25, 5)m$ in the $2m$ GPS precision map. This can be explained by observing the lower graph of the first column of Figure A.8, in which it is possible to verify that 50000 trials were not sufficient to guarantee the convergence of the optimized $V(b_0)$, as the tendency of its evolution is to keep decreasing over the trials. Furthermore, this strategy requires on average the highest computation time per optimization.

The UCB$_{\sqrt{(\cdot)}}$ strategy also does not present a consistent behaviour. According to what was expected, this two-sampling scheme strategy should accelerate the convergence of the algorithm, leading to lower values of both optimized and executed $V(b_0)$ and higher success rates as compared to the ones obtained by the UCB1 formula using the same coefficient c. However, this is only the case for the initial position $(10, 25, 5)m$ in the $10m$ GPS precision map.

When applying the MinPOMCP back-propagation strategy, a substantial improvement on the convergence of the initial belief state value is verified for both selection policies UCB1 and EBC. The values registered in Table 4.1 follow this outcome, as the success rate also increases. Moreover, for both selection policies used, the simulated $V(b_0)$ is now higher than the optimized $V(b_0)$ in some cases. This is explained because this back-propagation strategy considers only the best successors in the tree, leading often to an optimistic value of b_0. Additionally, the EBC$_{MP}$ approach achieves even better success rates and lower values in both $V(b_0)$, when compared with the best UCB1 coefficient, for most of the

cases here considered. On the other hand, this back-propagation strategy requires more computation time per optimization.

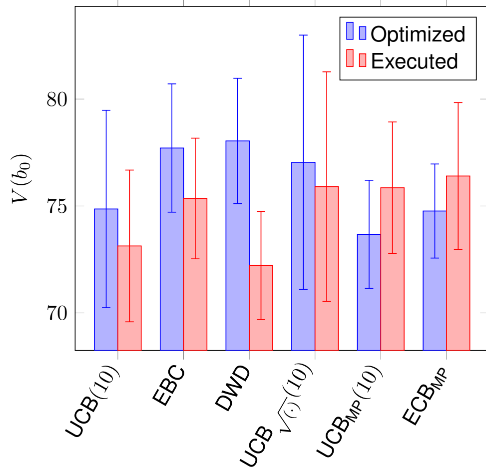
	WallBaffle, $2m$ GPS precision		*WallBaffle*, $10m$ GPS precision	
Initial position	$(10, 25, 5)m$	$(50, 25, 5)m$	$(10, 25, 5)m$	$(50, 25, 5)m$
UCB1*				
Success $(\%)$	$97.42\,(1.09)$	$\mathbf{97.35\,(1.45)}$	$98.45\,(1.83)$	$98.54\,(1.12)$
$T\,(s)$	$\mathbf{73.75\,(0.81)}$	$\mathbf{67.80\,(3.33)}$	$75.05\,(2.52)$	$63.32\,(3.98)$
Time per optimization (min)	$153.03\,(4.50)$	$\mathbf{80.80\,(9.23)}$	$88.33\,(3.47)$	$65.30\,(8.08)$
EBC				
Success $(\%)$	$97.59\,(1.02)$	$97.04\,(0.85)$	$97.19\,(3.28)$	$97.64\,(2.51)$
$T\,(s)$	$75.54\,(1.42)$	$68.87\,(2.95)$	$\mathbf{72.81\,(1.08)}$	$64.51\,(2.55)$
Time per optimization (min)	$\mathbf{113.80\,(3.34)}$	$119.60\,(4.29)$	$75.45\,(4.63)$	$79.33\,(2.05)$
DWD				
Success $(\%)$	$\mathbf{98.68\,(1.04)}$	$96.52\,(1.26)$	$\mathbf{99.91\,(0.08)}$	$\mathbf{99.69\,(0.45)}$
$T\,(s)$	$76.46\,(0.88)$	$69.80\,(3.88)$	$77.34\,(2.31)$	$69.82\,(3.50)$
Time per optimization (min)	$196.00\,(28.26)$	$266.5\,(29.06)$	$101.67\,(4.71)$	$164.9\,(17.55)$
UCB$_{\sqrt{(\cdot)}}$				
Success $(\%)$	$97.99\,(0.90)$	$96.69\,(1.42)$	$99.00\,(0.78)$	$97.83\,(1.50)$
$T\,(s)$	$74.71\,(1.58)$	$67.97\,(3.93)$	$73.85\,(1.06)$	$\mathbf{62.40\,(3.03)}$
Time per optimization (min)	$203.30\,(14.73)$	$144.50\,(28.32)$	$\mathbf{66.33\,(2.62)}$	$\mathbf{55.56\,(3.82)}$
UCB1$^*_{\text{MP}}$				
Success $(\%)$	$\mathbf{98.25\,(0.94)}$	$98.28\,(1.03)$	$99.00\,(0.89)$	$98.35\,(1.25)$
$T\,(s)$	$\mathbf{74.68\,(1.24)}$	$\mathbf{59.13\,(1.33)}$	$\mathbf{74.59\,(1.52)}$	$60.25\,(1.72)$
Time per optimization (min)	$277.67\,(9.67)$	$161.00\,(17.47)$	$124.30\,(4.61)$	$86.34\,(3.47)$
EBC$_{\text{MP}}$				
Success $(\%)$	$97.81\,(0.80)$	$\mathbf{98.29\,(1.19)}$	$\mathbf{99.07\,(0.92)}$	$\mathbf{99.75\,(0.31)}$
$T\,(s)$	$75.46\,(1.71)$	$60.57\,(2.07)$	$75.56\,(2.15)$	$\mathbf{60.21\,(2.31)}$
Time per optimization (min)	$\mathbf{119.01\,(6.31)}$	$\mathbf{98.30\,(4.63)}$	$\mathbf{101.20\,(5.35)}$	$\mathbf{85.67\,(2.87)}$

Table 4.1: Performance comparison between strategies in the *WallBaffle* map. The data is organized as Average (Standard deviation). In **bold** are represented the best values of the three metrics (success rate, flight time and computational time) for each back-propagation strategy.

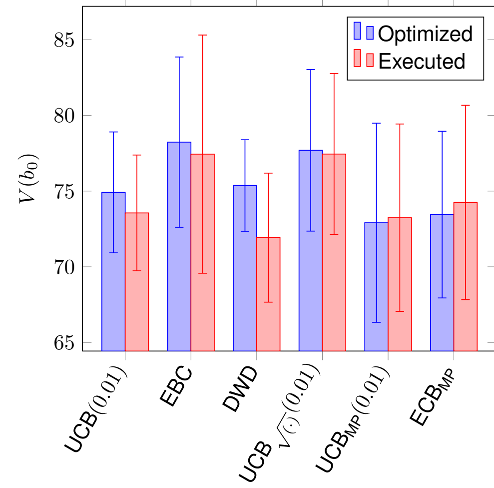
***CubeBaffle* map**

Figures 4.7 and 4.8 and Table 4.2 comprise the results for the *CubeBaffle* map. The GPS availability maps chosen are so close in terms of precision that it does not allow to draw the same clear conclusion for the exploration factor c with the best results in the UCB1, as done with the *WallBaffle* map. However, the average computation time for each optimization remains lower for the map with the lowest GPS precision required (Table 4.2).

Once again, the DWD strategy does not appear to be consistent, achieving the best results for the initial position $(35, 20, 5)m$ in both GPS precision maps, with the lowest simulated $V(b_0)$ along with the highest success rates, while the initial position $(65, 20, 5)m$ scored the worst success rates and highest simulated $V(b_0)$ in both GPS precision maps. The lower graph in the first column of Figure A.10 indicates

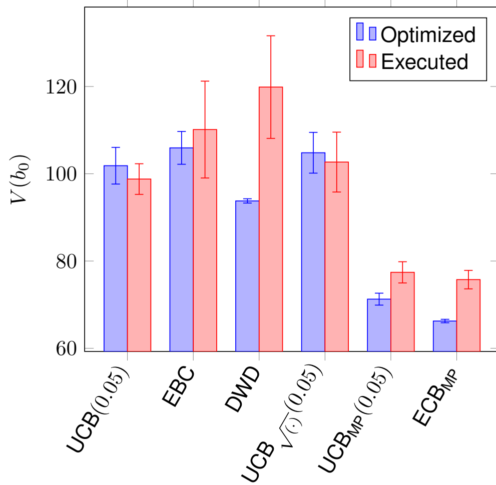


(a) $1m$ precision on GPS availability

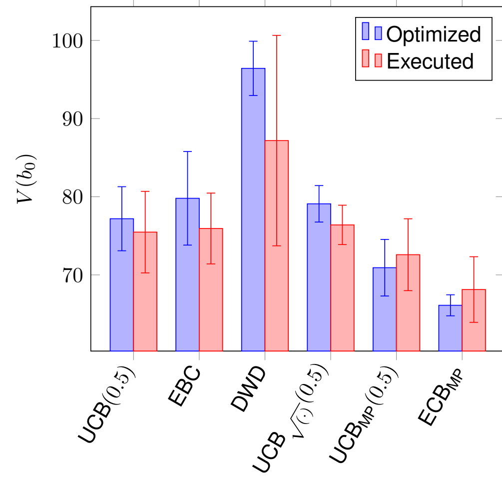


(b) $2m$ precision on GPS availability

Figure 4.7: *CubeBaffle* map with initial position in $(35, 20, 5)m$, using different combinations of action selection and back-propagation strategies.



(a) $1m$ precision on GPS availability



(b) $2m$ precision on GPS availability

Figure 4.8: *CubeBaffle* map with initial position in $(65, 20, 5)m$, using different combinations of action selection and back-propagation strategies.

that, although convergence on the optimized value function was ensured over the optimization process, the 50000 trials were not sufficient to converge the policy, which keeps changing and leading to different success rates and total fight times, as can be seen by the behaviour of the simulated $V(b_0)$ and its variability. Additionally, the lower graph in the first column of Figure A.12 denotes a convergence of the value function begin optimized, while the executed $V(b_0)$ remains comparatively high. This can be explained by the choice of executing the trajectory between the obstacles, rather than going around, as illustrated in the example of trajectories shown for this case. Indeed, the uncertainty on GPS availability in the area between the obstacles is higher, which may lead the navigation solution to be predominantly INS, meaning higher path execution errors and an increase on the collision risk, hence why the success rates are lower. The computation time remains, on average, the highest for this approach.

The performance of the UCB$_{\sqrt{(\cdot)}}$ strategy is worse in all 4 case studies, when compared to the UCB1 formula using the same coefficient c. Particularly for both initial positions in the $1m$ GPS precision map,

this approach was computationally the most expensive.

Finally, the combination of the MinPOMCP back-propagation strategy with the best UCB1 coefficient and the EBC shows an enhancement of the performance of such selection strategies, once again accelerating the value convergence, except in the initial position $(35, 20, 5)m$ of the $1m$ GPS precision map. Particularly in the initial position $(65, 20, 5)m$ of the $1m$ GPS precision map, the success rates increased significantly, with both simulated and optimized $V(b_0)$ decreasing dramatically. On the other hand, this back-propagation strategy requires on average more computation time per optimization.

	CubeBaffle, $1m$ GPS precision		*CubeBaffle*, $2m$ GPS precision	
Initial position	$(35, 20, 5)m$	$(65, 20, 5)m$	$(35, 20, 5)m$	$(65, 20, 5)m$
UCB1*				
Success $(\%)$	$98.65\,(0.54)$	$\mathbf{92.44\,(0.81)}$	$97.59\,(0.85)$	$97.76\,(1.48)$
$T\,(s)$	$67.97\,(3.66)$	$70.06\,(0.96)$	$64.77\,(1.38)$	$66.88\,(1.56)$
Time per optimization (min)	$155.33\,(1.25)$	$\mathbf{94.34\,(8.29)}$	$66.34\,(3.82)$	$173.33\,(2.36)$
EBC				
Success $(\%)$	$98.02\,(0.58)$	$89.34\,(2.89)$	$96.82\,(1.42)$	$\mathbf{98.35\,(0.67)}$
$T\,(s)$	$\mathbf{67.78\,(1.64)}$	$\mathbf{69.58\,(0.91)}$	$65.23\,(3.58)$	$69.67\,(2.46)$
Time per optimization (min)	$\mathbf{110.45\,(17.39)}$	$177.30\,(5.09)$	$98.67\,(3.94)$	$86.67\,(9.42)$
DWD				
Success $(\%)$	$\mathbf{99.57\,(0.67)}$	$86.91\,(3.32)$	$\mathbf{99.66\,(0.52)}$	$95.39\,(3.36)$
$T\,(s)$	$70.58\,(1.64)$	$70.10\,(2.10)$	$70.64\,(3.48)$	$69.66\,(3.67)$
Time per optimization (min)	$233.75\,(25.34)$	$356.34\,(4.32)$	$199.30\,(34.32)$	$285.76\,(18.90)$
UCB$_{\sqrt{(\cdot)}}$				
Success $(\%)$	$97.96\,(0.82)$	$91.51\,(1.77)$	$96.69\,(1.30)$	$97.43\,(0.74)$
$T\,(s)$	$68.12\,(3.41)$	$70.46\,(0.89)$	$\mathbf{64.69\,(2.61)}$	$\mathbf{66.54\,(1.07)}$
Time per optimization (min)	$333.67\,(10.53)$	$407.53\,(8.99)$	$\mathbf{51.33\,(6.18)}$	$\mathbf{53.36\,(1.70)}$
UCB1$^*_{\text{MP}}$				
Success $(\%)$	$\mathbf{97.20\,(0.66)}$	$97.11\,(0.43)$	$\mathbf{97.70\,(1.64)}$	$98.09\,(1.26)$
$T\,(s)$	$65.08\,(0.74)$	$66.32\,(0.98)$	$64.37\,(1.60)$	$\mathbf{65.23\,(0.61)}$
Time per optimization (min)	$199.33\,(7.59)$	$\mathbf{113.75\,(4.97)}$	$106.67\,(1.25)$	$215.42\,(4.08)$
EBC$_{\text{MP}}$				
Success $(\%)$	$96.89\,(0.87)$	$\mathbf{97.49\,(0.53)}$	$97.27\,(1.76)$	$\mathbf{99.36\,(0.95)}$
$T\,(s)$	$\mathbf{64.41\,(0.80)}$	$\mathbf{66.10\,(0.21)}$	$\mathbf{63.70\,(0.81)}$	$65.67\,(0.82)$
Time per optimization (min)	$\mathbf{107.66\,(6.13)}$	$121.63\,(1.25)$	$\mathbf{97.68\,(3.09)}$	$\mathbf{100.03\,(9.27)}$

Table 4.2: Performance comparison between strategies in the *CubeBaffle* map. The data is organized as Average (Standard deviation). In **bold** are represented the best values of the three metrics (success rate, flight time and computational time) for each back-propagation strategy.

4.3.3 Summary

From the results obtained in the previous section, it is possible to draw some conclusions regarding the different action selection and back-propagation strategies tested.

Firstly, the EBC strategy avoids the extensive parameter tuning inherent to the use of the fixed exploration factor c in the UCB1 formula, while guaranteeing a satisfying level of performance for all domains

considered. On the other hand, both DWD and UCB$_{\sqrt{(\cdot)}}$ do not show a consistent behaviour, yielding the best results for particular case studies, while the worst results for others. Additionally, the DWD strategy takes longer to ensure the convergence of the value function. Concerning an online planning configuration, where an extensive parameter tuning is not desirable, the EBC strategy is more adaptable to different planning domains and therefore will be the one used in Chapter 5, along with the UCB1 best coefficient for comparison purposes.

Furthermore, the combination of ECB with the MinPOMCP back-propagation strategy is a promising approach for the online planning configuration, since it offers no need for extensive search of the best UCB1 coefficient and the MinPOMCP helps to accelerate value convergence.

Chapter 5

Online Approach

This chapter leads the reader through the online approach. Section 5.1 proposes an online version of the POMCP-GO planning algorithm. Section 5.2 explains the planning-while-executing framework in which the online POMCP-GO algorithm is integrated to provide strictly anytime and continuous solutions during the mission execution. In Section 5.3, simulations are performed on the resulting framework, whose implementation is done in Robot Operating System (ROS), and comparison remarks are made with the POMCP-GO algorithm implemented alone. Finally, Section 5.4 explores the performance of this framework in a robot simulator, Gazebo.

5.1 Online POMCP-GO Algorithm

This section introduces the online algorithm to solve the PO-SSP planning problem. This algorithm is an online configuration of the POMCP-GO algorithm presented in Chapter 4 and is fully described in Algorithm 3. The main differences between the online and the offline approach include the computational budget, the belief state representation and the trial length.

Regarding the computational budget, a timeout is applied (line 2), instead of a fixed number of trials, in order to submit the planner to time-constrained environments. Being an online approach and as mentioned in Chapter 2, the trials begin from the current belief state b, rather than starting always in the initial belief. As such, rejection sampling (Section 2.5.6) is performed to represent the new belief state, after each action a is executed and the respective observation s_v is perceived from the real world.

Similarly to the classic POMCP [26], each trial finishes after a new node is created (line 11). Furthermore, an additional trial length condition is established in Algorithm 3, ending the trial if the current depth reached a maximum depth D (line 18). This condition reinforces the need to investigate the action space closer to the root of the tree more thoroughly (i.e. promotes breadth-first search), which might improve action selection with short time budgets.

This online approach acts by interleaving planning and execution phases. Because the ultimate goal of these guidance and navigation strategies is to apply them onboard the UAV in a real-time configuration, it becomes unfeasible to have the UAV constantly stopping mid-air while the planning phase takes

Algorithm 3: Online POMCP-GO

```
1  Function Search(h):
2      while not timeout do
3          s_h ~ b(h)
4          Trial(s_h, h, 0)
5      return a* ← argmin_{a∈A} Q(h, a)
6  Function Trial(s_h, h, depth) :
7      if s_h ∈ G then
8          return 0
9      if F_Col == 1 then
10         return K − t × f_t
11     if h ∉ T then
12         for a ∈ A do
13             T(ha) ← (N_init(ha), Q_init(h, a))
14         V_init(h) ← min_{a∈A} Q(h, a)
15         return V_init(h)
16     ā ← SelectionPolicy(h, ha, c)
17     (s'_h, s'_v, C(s_h, ā)) ~ G(s_h, ā)
18     if depth < D then
19         Q(h, ā)' ← C(s_h, ā) + Trial(s'_h, hao, depth + 1)
20     N(h) ← N(h) + 1
21     N(hā) ← N(hā) + 1
22     V(h), Q(h, ā) ← Back-propagation(C(s_h, ā), Q(h, ā)')
```

place, regardless of its duration. For this reason and for those mentioned in Section 1.3.2, the POMCP-GO is incorporated in a planning-while-executing framework, which is described in the following section.

5.2 AMPLE planning-while-executing framework

AMPLE [27] follows the direction of reactive or continuous planning, in which a plan is initially proposed and then works by iteratively fixing flaws in such plan when an anomaly occurs at execution-time, and on continuously updating goals, state and planning horizons.

This framework is **strictly anytime** in the sense of policy execution under time constraints, as it ensures the return of an applicable action in any possible execution state at a precise time point, exactly when required by the execution engine; **reactive** to environment changes, which are incorporated in the planning requests being managed in parallel to the action execution and **conditional** by prioritizing future execution states and computing a partial (incomplete) but applicable policy for each action.

AMPLE has been successfully applied to solve UAV's missions such as target detection [1] and autonomous landing [76], using classical planning frameworks or (PO) MDPs models.

5.2.1 `AMPLE` architecture

The AMPLE framework, depicted in Figure 5.1 is composed by two threads: a *planning thread*, which can also be seen as a server thread, that manages plan optimization while answering to client requests; and an *execution thread*, or client thread, that adds and removes planning requests according to the system's and environment's evolution, in order to get the action to execute in the current state, from the optimized policy or from the default one.

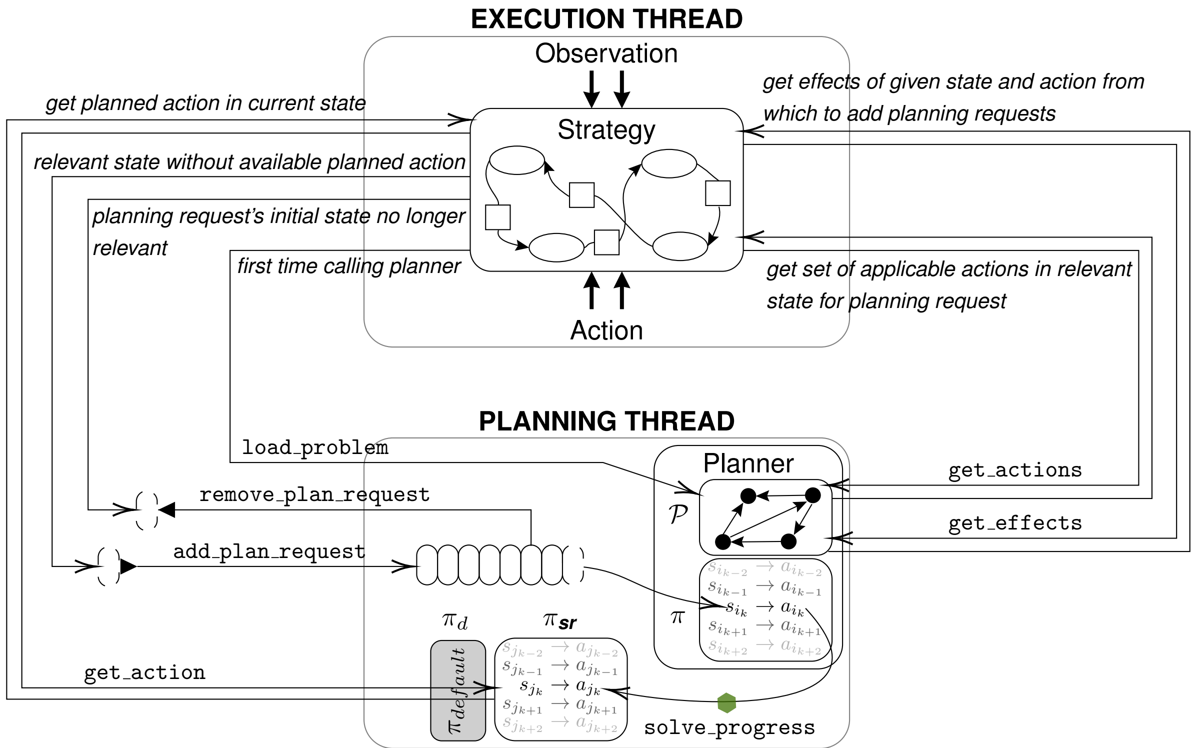


Figure 5.1: AMPLE architecture: connections between the execution and the planning threads [27].

The planning thread contains an instance of the planning algorithm called *Planner*. Most of the requests addressed to the planning thread from the execution thread concern the management of *planning requests*, while others correspond to getting information about the planning problem being solved.

When the planning thread receives the planning requests, it must compute and update the current policy depending on the information included in the request. A planning request r is a 4-tuple $(I, \Delta, \alpha, \alpha_p)$, where I is a (belief) state for which a policy must be computed, Δ is the (continuous) maximum allowed duration for the policy optimization, α is an algorithm variant of the planner (some planners can only provide one variant) and finally, α_p comprises the parameters of α. This last variable is generic enough to take into account any parameter required by the planner to handle requests.

For the explanations that follow, some notation needs to be clarified (***italic bold*** data identify data structures shared by both threads):

- $\mathcal{P}$ represents the planning problem;
- psm is the state machine that formalizes the interaction between both threads;
- π_d is the default policy. This policy is the ”rescue” policy, which can be either computed offline or

quickly computed online before the optimization of the policy;

– π_{sr} is the backup policy to solve requests. This policy is a copy of the policy being optimized that is updated during the optimization process;

– ***pr*** forms a queue of planning requests managed by the execution thread and solved by the planning thread;

– ***stopCurrentRequest*** is a boolean indicating whether the current request being solved in the planning thread must be interrupted;

– ***stopPlannerRequest*** is a boolean indicating whether AMPLE must be stopped.

As seen in Figure 5.1, the execution thread can interact with the planning thread through the use of several methods, namely:

– `load_problem` to load an initial problem $\mathcal{P}$ in the planner;

– `add_plan_request` to add a request r to the queue of pending planning requests ***pr***;

– `remove_plan_request` to remove a request r from ***pr***, if it is not being solved by the planning thread; otherwise, it sets the boolean ***stopCurrentRequest*** to true to inform the planning thread to stop solving this request and to remove it;

– `get_action` to read the optimized action to be executed in the current state, if it is included in the backup policy π_{sr}; otherwise an action defined by the default policy π_d is read instead;

– `get_actions` to get the set of actions applicable in state s;

– `get_effects` to get the set of states reachable by performing action a in state s.

5.2.2 AMPLE planning thread

The planning thread automatically manages the queue of planning requests and locally updates the policy in bounded time for each planning request. Algorithm 4 illustrates the pseudocode of the planning thread. The algorithm is conceived as an endless loop that continuously reacts to the current mode of the state machine psm.

At first, if the mode of the state machine is `LOADING_PROBLEM` (after an activation of the `load_problem` command by the execution thread), then it loads the planning problem $\mathcal{P}$ and changes the mode to `PROBLEM_LOADED` (lines 3 to 5). If the mode is `PLANNING` (after an `add_plan_request` command in the execution thread), it checks if the front planning request in the queue is already being solved (line 7). If not, it launches its optimization (lines 25 to 28), which mainly consists in recording the current time and calling the `solve_initialize` planner procedure. Otherwise, it means that the planning thread was already solving this request from a previous iteration of its endless loops. In this case, it tests if the optimization of this request must end (line 11), which can happen if the planner procedure has converged, if the time allocated to solve the request has been consumed, if requested by the execution thread via the `remove_plan_request` command or if the AMPLE planner has to be stopped.

If all these conditions are false, the optimization of the request continues by calling the `solve_progress` planner procedure and the backup policy $\pi_{\textbf{\textit{sr}}}$ is updated for the (belief) state queried by the current planning request (lines 22 to 23). Otherwise, the `solve_end` planner procedure is called, the backup policy is updated and the current planning request is removed from the queue (lines 12 to 15). Then, the next planning request in the queue, if any, is launched or the mode of the state machine is changed to `PROBLEM_SOLVED` (line 19).

Algorithm 4: `AMPLE` Planning

```
1  solvingRequest ← false ;
2  while true do
3      if state == LOADING_PROBLEM then
4          pln.load_problem (P) ;
5          state = PROBLEM_LOADED ;
6      else if state == PLANNING then
7          if solvingRequest == false then
8              launch_front_request() ;
9          else
10             t ← get current time ;
11             if pln.solve_converged (pr.front.α_p) or t − requestStartTime > pr.front.Δ or
                  stopCurrentRequest == true or stopPlannerRequest == true then
12                 pln.solve_end() ;
13                 π_sr ← π_sr ∪ pln.policy (pr.front.I) ;
14                 pr.pop_front() ;
15                 stopCurrentRequest ← false ;
16                 if pr not empty then
17                     launch_front_request() ;
18                 else
19                     state = PROBLEM_SOLVED ;
20                     solvingRequest ← false ;
21             else
22                 pln.solve_progress (pr.front.α_p) ;
23                 π_sr ← π_sr ∪ pln.policy (pr.front.I) ;
24 Procedure launch_front_request():
25     solvingRequest ← true ;
26     pln.set_algorithm (pr.front.α, pr.front.α_p) ;
27     requestStartTime ← get current time ;
28     pln.solve_initialize (pr.front.I) ;
```

5.2.3 AMPLE execution thread

The way the execution thread manages the planning requests, by adding or removing them according to the current execution state and to the future probable evolution of the system, and the action execution can follow different planning strategies, depending on the instantiation of the state machine. AMPLE gives to the planner the capability of switching between such planning strategies, with the guarantee to provide a plan anytime, with respect to time constraints.

Chanel et al. [27] propose three different strategies instantiated within the execution thread:

- AMPLE-NEXT that provides a short-term prediction of the system's evolution. As the system begins to execute an action, all the next possible (belief) states coming from this action are computed and plan requests are added for each of them, allowing for the planning thread to provide an optimized action on time as soon as the current action terminates;
- AMPLE-PATH that provides long-term reasoning about the most probable execution path of the system, computed by applying the current optimized policy (or the default one, if necessary) from the current execution state via successive calls to the get_action command;
- AMPLE-PORTFOLIO that uses a set of distinct planners to solve the current problem, in order to have more chance of finding a solution.

The AMPLE-PATH strategy is not appropriate for problems with high uncertainty in action effects, since the chance of leaving the most probable path is very high, leading to an invalid strategy at almost every action and the consequent use of only the default policy. Furthermore, the AMPLE-PATH strategy is not suitable for random problems because the various portfolio planners should be logically chosen to be efficient at solving a specific problem. Therefore, the AMPLE-NEXT strategy will be the one chosen to be implemented in this work and will be further explored in the next section. For more information on the remaining strategies, please refer to [27].

AMPLE-NEXT: predicting the evolution of the system one step ahead

The AMPLE-NEXT strategy is formalized in Algorithm 5. Prior to the planning-while-executing loop, the AMPLE framework requires an initial bootstrap in which the planner computes a first optimized action in the initial (belief) state for a given period of time $\Delta_{bootstrap}$ (line 1). Once an action has been completed, the next one to be executed, action a, is gathered by calling the get_action command for the current execution state (line 3), its expected duration Δ_a is computed (line 5) and its execution begins. Then, plan requests are added for each possible next (belief) state of the system I', with a maximum computation time proportional to Δ_a and to the probability of getting I' as an effect of executing a in the current (belief) state I (lines 7 to 11). The system waits for action a to complete (line 12), while added planning requests are being solved in the planning thread. Once the execution is terminated, the system removes the previous planning requests in case they have not yet been solved by the planning thread (line 14), the current execution state is observed and the algorithm goes back to the beginning of the execution loop.

Algorithm 5: `AMPLE-NEXT`

```
1  I ← bootstrap_execution (P, Δ_bootstrap) ;
2  while stopPlannerRequest == false do
3      a ← get_action (psm, π_d, P, π_sr, b) ;
4      Start execution of action a ;
5      Δ_a ← expected duration of action a ;
6      prNext ← empty list of planning request pointers ;
7      for I' ∈ get_effects (I, a) do
8          r.I ← I' ;
9          r.Δ ← Pr(I'|a, I) Δ_a ;
10         add_plan_request (psm, pr, r) ;
11         prNext.pushback(r) ;
12     Wait until action a has completed ;
13     for r ∈ prNext do
14         remove_plan_request (psm, pr, r) ;
15         stopCurrentRequest ← true ;
16     I ← observe and update current belief state ;
```

5.2.4 Integration of POMCP-GO in `AMPLE`

The AMPLE framework is developed in the Robotic Operating System (ROS). As such, three nodes are established for the planning thread, the execution thread and the planner (online POMCP-GO). The several methods described in Section 5.2.1 and illustrated in Figure 5.1 constitute services that connect these nodes. The planning thread node is implemented as an intermediary in all of these services, meaning that the execution thread node and the planner node are not directly connected.

In this application case, the planning time for each future state s'_v is computed proportionally to the probability of getting such effect after the execution of an action a in the current belief state b, i.e. $\Pr(s'_v|b, a)$. However, the MOMDP model is not declarative on these probabilities, so an approximation is done with the following formula:

$$\Pr(s'_v|b, a) = \frac{N(hao)}{N(ha)} \tag{5.1}$$

As the values of the visitation counters $N(hao)$ and $N(ha)$ are iteratively updated in the back-propagation step of the POMCP-GO algorithm during the trials, this probability becomes more precise as the mission progresses. Furthermore, both `solve_progress` and `solve_initialize` planner procedures constitute calls to the Trial function (line 6) in the POMCP-GO algorithm (Algorithm 3).

5.3 Simulations

This section presents the simulations performed online using the AMPLE framework implemented in ROS. The performance of three approaches is compared: POMCP-GO driven in AMPLE; POMCP-GO implemented using the classical online paradigm, that is, with planning and execution phases inter-

leaved; and an algorithm that consecutively applies default actions with no planning time associated. The latter serves as a baseline indicator of the quality of the default policy π_d used.

5.3.1 Configuration

There are a total of 4 application cases examined in these simulations: 2 environment maps, using 1 sensor's availability map with 2 different initial states. A total of 100 simulations are performed for each application case.

Sensors: The vehicle model used is the one defined in Section 3.2, considering only two onboard sensors: INS, available all the time, and GPS, available according to the probability maps.

Environments: Two benchmarks are selected:

WallBaffle: map containing two wall as obstacles (Figure 4.2a), with a $2\,meter$ precision on GPS availability (Figure 4.2c);

CubeBaffle: map containing two cubes as obstacles (Figure 4.2e), with a $2\,meter$ precision on GPS availability (Figure 4.2g);

with a grid size of $100 \times 100 \times 20$ cells, where each grid cell has the size of $2m \times 2m \times 2m$.

Initial conditions: Similar to the initial conditions described in the simulations performed for parameter tuning (Section 4.2).

Actions: Similar to the actions considered in the simulations performed for parameter tuning (Section 4.2). Parameters in the GNC model are configured as to ensure an action duration of 4 seconds.

Parameters: Table 5.1 summarises the parameters used in these simulations. For the AMPLE framework, three different bootstrap durations are examined $\Delta_{bootstrap} = 5, 10, 15s$ for each application case. The online POMCP-GO algorithm also requires the specification of extra parameters: the maximum depth is $D = 10$, the number of particles for the belief state representation is set at $M = 300$, the collision cost is fixed at $K = 450$ and the timeout given to the planning phase for the classical online configuration is tested using $timeout = 2, 3, 4s$ for each application case.

Furthermore, two selection policies are used for both configurations of POMCP-GO (classical online paradigm and the integration in AMPLE): the UCB1 coefficient that showed the best results in each application case (denoted with $*$) and the EBC with the same setup as the one used in the simulations for parameter tuning. The back-propagation strategy applied is the Classical POMCP.

Default policy π_d: This policy consists on the result obtained when applying the Dijkstra's algorithm to the obstacle grid map (see Section 4.1.1).

Metrics: To evaluate the performance of the algorithm, the following metrics are examined:

- The **total mission duration**, from the moment the problem is loaded until a terminal state is reached (either a goal or a collision);
- The **total planning time** during the mission, taking into consideration the initial bootstrap time in the case of AMPLE;

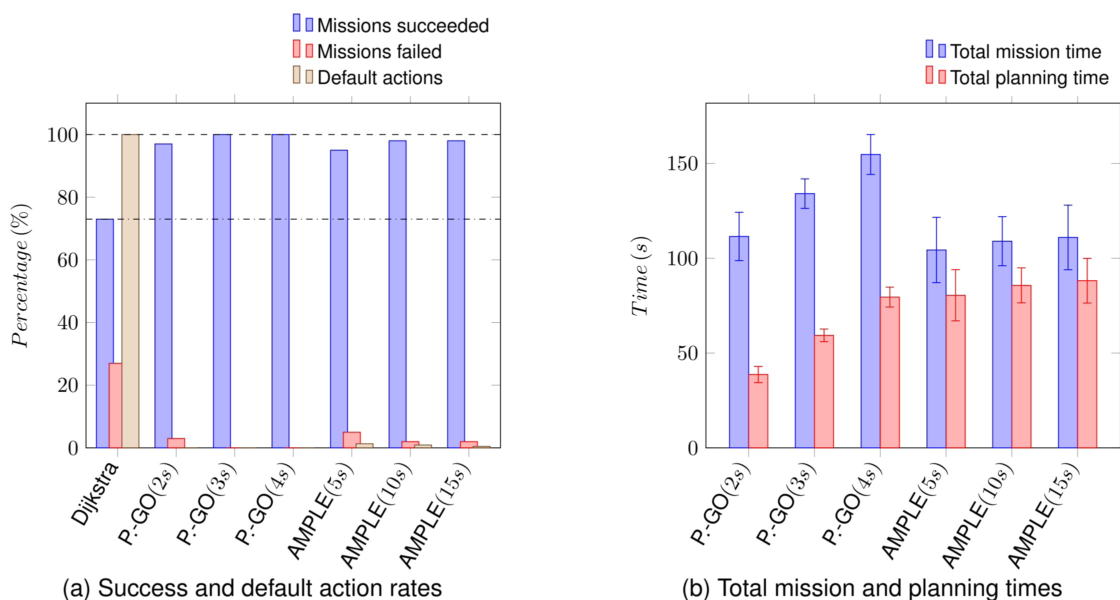
	POMCP-GO	POMCP-GO driven in AMPLE
Paradigm	Planning and execution phases interleaved	Concurrent planning and execution phases
Planning duration ($timeout$)	$2s$, $3s$, $4s$	$4s$ (action duration)
Initial bootstrap ($\Delta_{bootstrap}$)	–	$5s$, $10s$, $15s$
Action selection strategy	UCB1* and EBC	UCB1* and EBC
Back-propagation strategy	Classical POMCP	Classical POMCP
Maximum depth D	10	10
Collision cost	450	450
Number of particles to represent belief state	300	300

Table 5.1: Parameters used for the online simulations.

- The **success rate**, which is the ratio between the number of simulations that ended up in the goal state and the total number of simulations performed (in %).
- The **default actions rate**, which is the ratio between the number of default actions, computed from the default policy π_d, used during the missions and the total number of actions performed (in %).

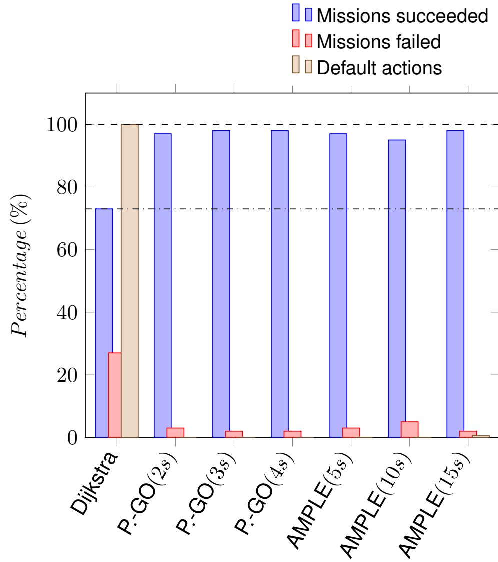
5.3.2 Results

Figures 5.2, 5.3, 5.4 and 5.5 depict the results obtained for the *WallBaffle* map. Figures 5.2 and 5.3 are relative to the simulations with the best UCB1 coefficient and the EBC, respectively, for the initial position $(10, 25, 5)m$, while Figures 5.4 and 5.5 represent the simulations with the best UCB1 coefficient and the EBC, respectively, for the initial position $(50, 25, 5)m$.

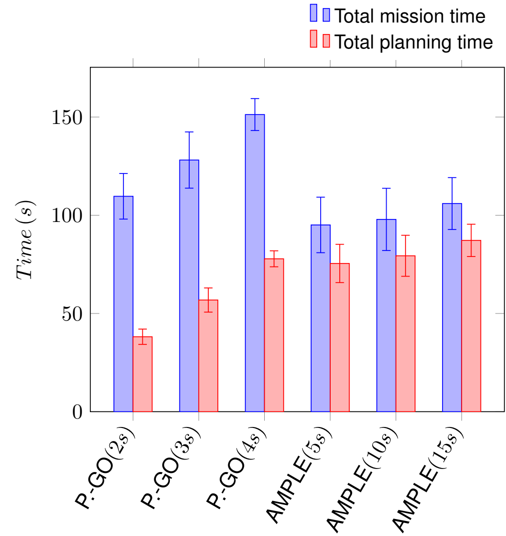


(a) Success and default action rates (b) Total mission and planning times

Figure 5.2: *WallBaffle* map with initial position in $(10, 25, 5)m$, using UCB*.

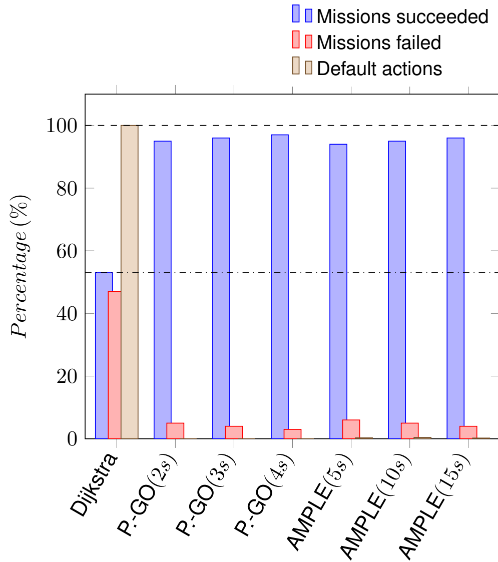


(a) Success and default action rates

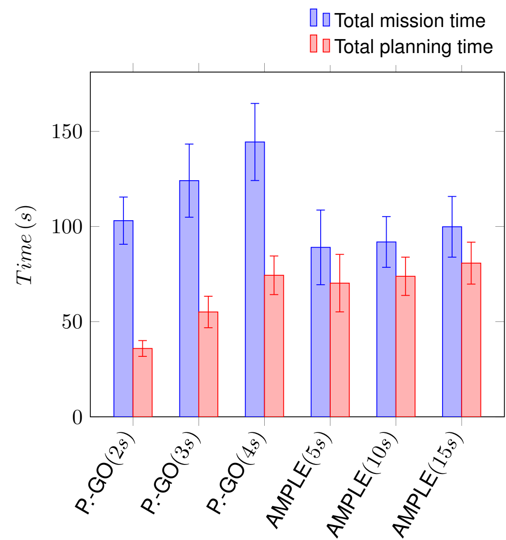


(b) Total mission and planning times

Figure 5.3: *WallBaffle* map with initial position in $(10, 25, 5)m$, using EBC.



(a) Success and default action rates

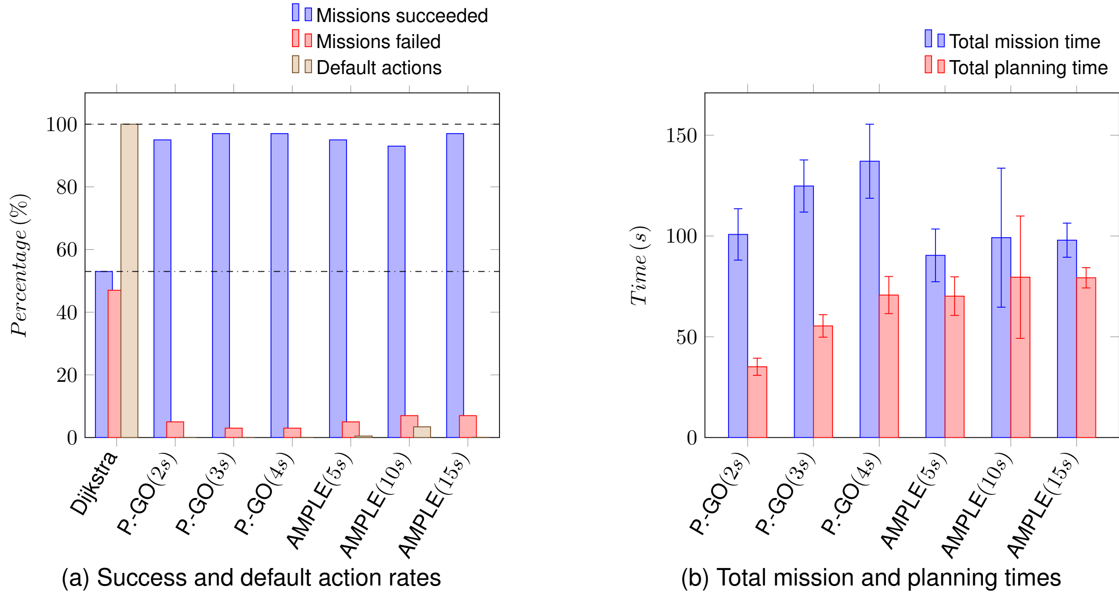


(b) Total mission and planning times

Figure 5.4: *WallBaffle* map with initial position in $(50, 25, 5)m$, using UCB*.

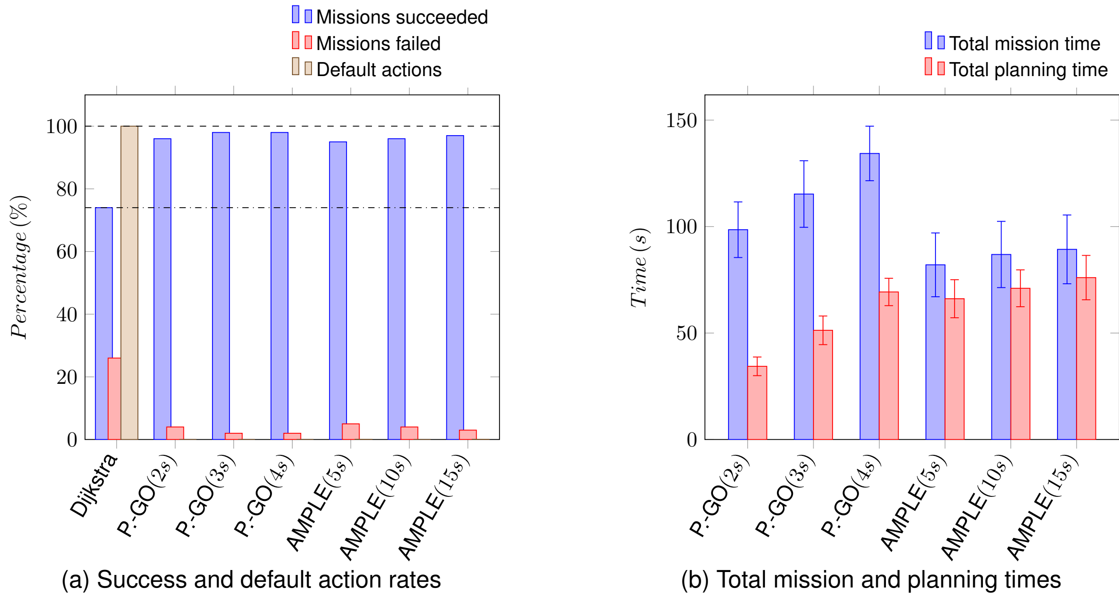
Figures 5.6, 5.7, 5.8 and 5.9 depict the results obtained for the *CubeBaffle* map. Figures 5.6 and 5.7 are relative to the simulations with the best UCB1 coefficient and the EBC, respectively, for the initial position $(35, 20, 5)m$, while Figures 5.8 and 5.9 represent the simulations with the best UCB1 coefficient and the EBC, respectively, for the initial position $(65, 20, 5)m$.

Regarding the default policy π_d, it is possible to observe that the baseline heuristic policy computed with the Dijkstra algorithm provides satisfying initial solutions, with no planning time associated, for most of the application cases, except for the *WallBaffle* map with starting position in $(50, 25, 5)m$, for which the success rate registered was 53%. Moreover, this heuristic policy affects the performance of both configurations being tested, as they achieve better success rates when the heuristic policy itself performs better. Nevertheless, it can be noticed that the number of default actions used during the



(a) Success and default action rates (b) Total mission and planning times

Figure 5.5: *WallBaffle* map with initial position in $(50, 25, 5)m$, using EBC.



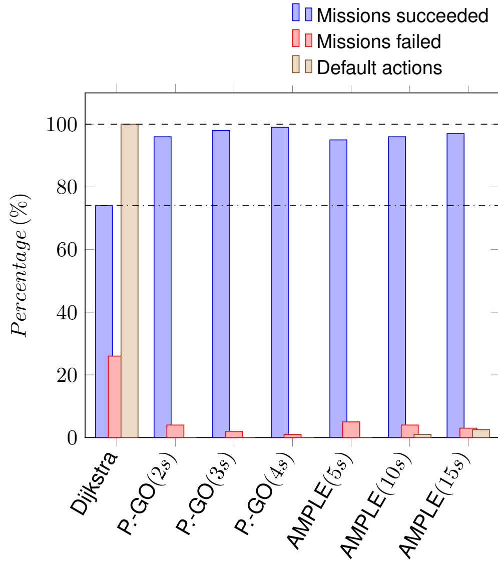
(a) Success and default action rates (b) Total mission and planning times

Figure 5.6: *CubeBaffle* map with initial position in $(35, 20, 5)m$, using UCB*.

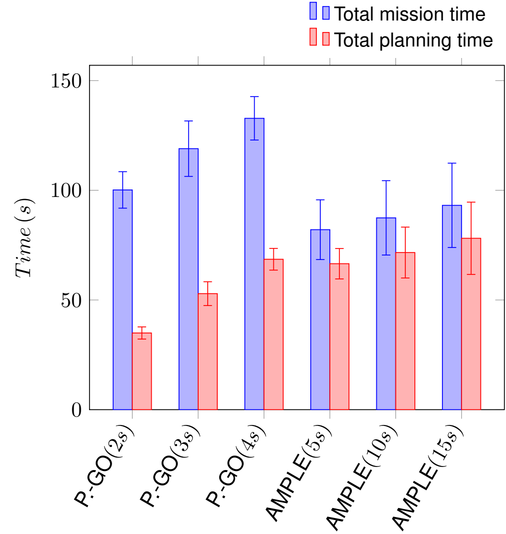
missions is quite low, meaning that the planner was able to compute policies for the corresponding belief states.

As expected, increasing the time budget in the POMCP-GO algorithm leads to better success rates, since providing more time for planning allows the algorithm to search deeper in the tree and thus to improve the quality of the policy being optimized. Furthermore, the improvement on the quality of the results achieved when going from $timeout = 2s$ to $timeout = 4s$ lies in most cases between 1% and 3%, meaning that those 2 seconds of additional planning time allows the algorithm to further exploit the best moves and their values. On the other hand, such 2 extra seconds increases significantly the total duration of the mission, by more than 35% in most cases.

Similarly in the AMPLE framework, increasing the initial bootstrap time improves the performance of

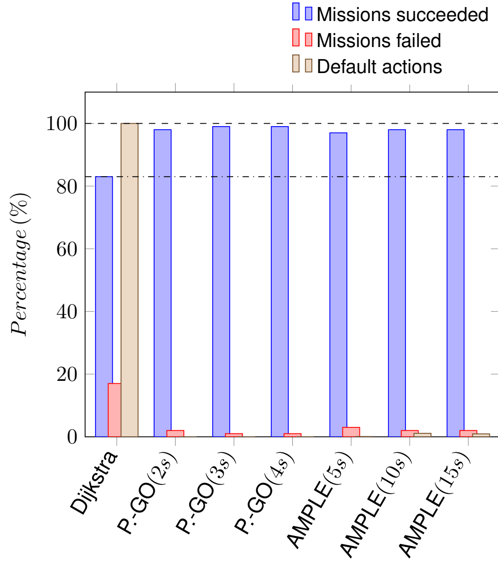


(a) Success and default action rates

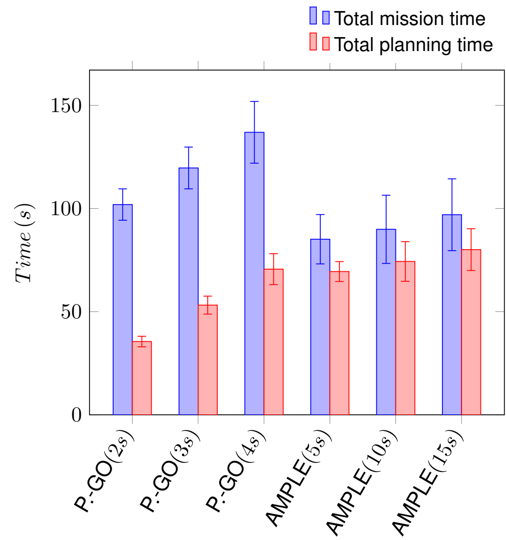


(b) Total mission and planning times

Figure 5.7: *CubeBaffle* map with initial position in $(35, 20, 5)m$, using EBC.



(a) Success and default action rates

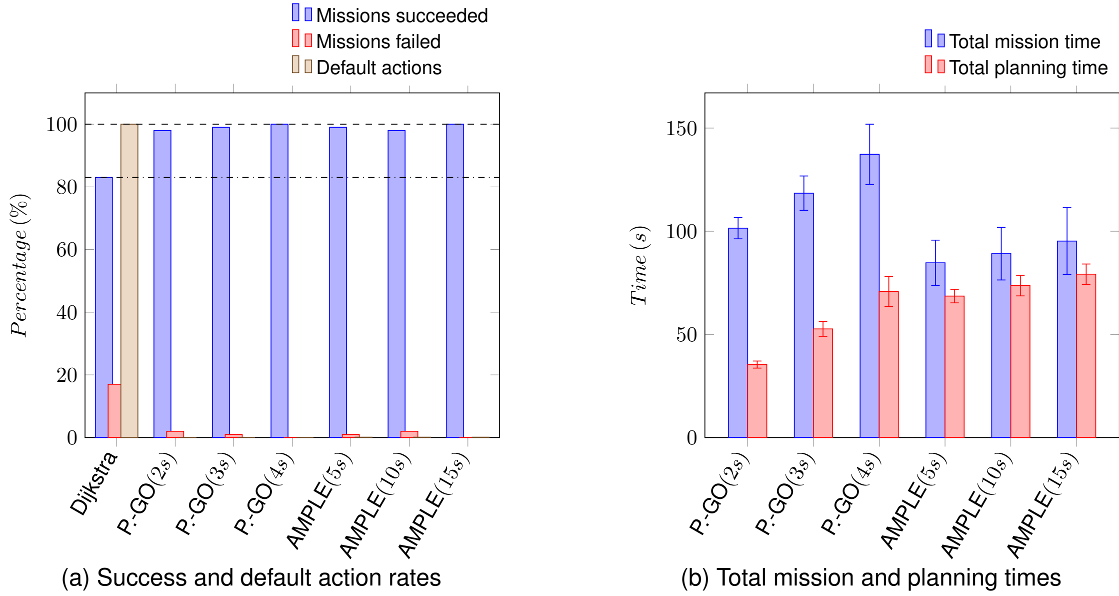


(b) Total mission and planning times

Figure 5.8: *CubeBaffle* map with initial position in $(65, 20, 5)m$, using UCB*.

the algorithm. This happens because providing more time to the planner to optimize the action for the first belief state also allows it to explore more thoroughly the state space near the root, improving the policy for the initial belief states. However, the success rates achieved by AMPLE are in general worse than when using POMCP-GO in the classical online paradigm, except when comparing POMCP-GO with a timeout of $2s$ and AMPLE with bootstrap of $\Delta_{bootstrap} = 15s$. This stems from three factors.

Firstly, in the POMCP-GO the time budget is used entirely to compute the next optimized action, without limiting the tree to search in a specific subtree. On the other hand, the AMPLE framework distributes unevenly the planning time over the expected future effects of the current action being executed. Indeed, this planning time given is proportional to the probability of such effect/observation being the one to actually be perceived after the action execution, i.e. is proportional to $\Pr(s'_v|b,a)$, and consequently during



(a) Success and default action rates

(b) Total mission and planning times

Figure 5.9: *CubeBaffle* map with initial position in $(65, 20, 5)m$, using EBC.

such duration the tree is limited to search in the subtree correspondent to that effect. Furthermore, the MOMDP model is not declarative on these probabilities and so the algorithm has to learn them as the mission progresses (Equation 5.1). As a consequence, it becomes fairly possible for the planner to assign the lowest planning time to the effect that was posteriorly perceived from the environment. One way to solve this problem is to explore a parallelization of the planning requests for all the possible future effects, creating parallel trees for each effect and assign the same planning time for all, as opposed to the current sequentialization of planning requests being used.

Secondly, the POMCP-GO is an online algorithm with anytime properties. However, it does not account for its own computation time during the optimization process and, consequently, if the $timeout$ is reached before the end of a trial, the algorithm will only provide an action after the back-propagation step of said trial is performed, allowing to further improve the quality of the policy. Whereas in the AMPLE framework, the algorithm is strictly anytime, meaning it does not wait for the end of the trial and automatically provides the action already present in the backup policy $\pi_{\mathbf{sr}}$.

Thirdly, while the POMCP-GO is implemented in Python, the AMPLE framework runs in ROS, having the planning thread, the execution thread and the planner in three distinct nodes connected through services. Planning requests sent by the execution thread are first managed by the planning thread and are posteriorly sent to the planner through these services. This process causes them to naturally inherent a lag, meaning the actual planning time budget given to an effect is lower than the one included in the planning request. This can have as a consequence that no actual planning is done for that effect, leading to a default action being used instead.

Apart from the loss of optimality found in AMPLE, this approach allows the reduction of the overall time of the mission, exploiting action execution time to anticipate and plan future states.

Regarding the action selection strategies used, there is no clear distinction as to which performs best, although it is expected that the performance of both strategies is enhanced if combined with the

MinPOMCP back-propagation strategy, as seen in Chapter 4. Nevertheless, the EBC selection strategy, being dynamically tuned, does not require the exhaustive search that is associated with the UCB1 formula and achieves approximately the same quality results for both application cases, making it more suited to an online planning configuration. Thus its implementation is considered successful and further work on its scalability to other planning domains would result in promising developments.

5.4 Application in Gazebo

As the AMPLE framework using the POMCP-GO algorithm is implemented in ROS, it makes possible testing it in a robot simulator called Gazebo. In order to do so, the hector quadrotor package [77] is used to simulate a quadrotor UAV system in an environment that replicates the obstacle maps used throughout this dissertation.

Figures 5.10 and 5.11 represent 5 trajectories planned for the *WallBaffle* map and the *CubeBaffle* map, respectively. For each map, a GPS availability map of $2\,meter$ precision is used (Figures 4.2c and 4.2g) and the same 2 distinct initial states used in the previous simulations are considered. A bootstrap duration of $\Delta_{bootstrap} = 15s$ is used for AMPLE. Regarding the POMCP-GO algorithm, the EBC is applied in the action selection strategy and Classical POMCP is the back-propagation strategy used.

The two red spheres represent the initial and the goal positions, whose size relies approximately on the covariance error considered for each position, and the light grey structures illustrated in the graphs constitute the obstacles for each map. More specifically, in Figure 5.10 the obstacles include 2 walls, while in Figure 5.11 these become 2 cubes. Furthermore, projections of the trajectories executed are illustrated over the three planes xy, xz and yz in a lighter shade.

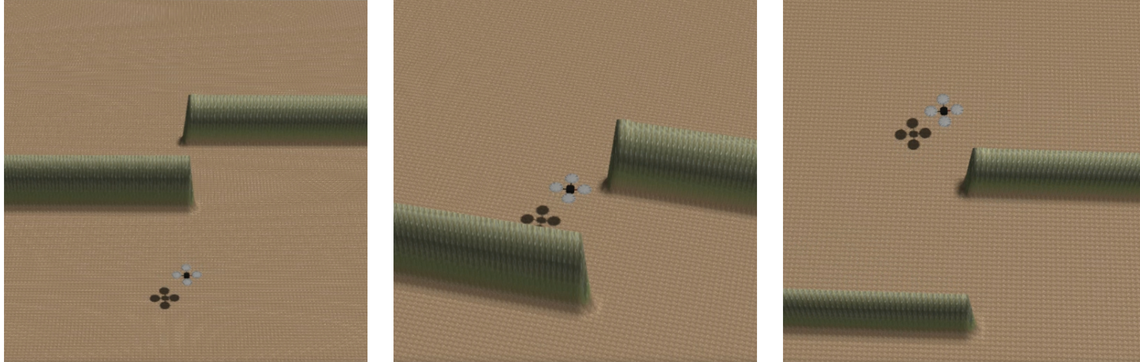
To be noted that the dynamics of the quadrotor UAV system were not modified to fit the vehicle motion model considered in the GNC model, as such process would lie outside the scope of this dissertation. For this reason, these symbolic simulations are not as realistic as they could have been if the real dynamic model had been used.

***WallBaffle* map**

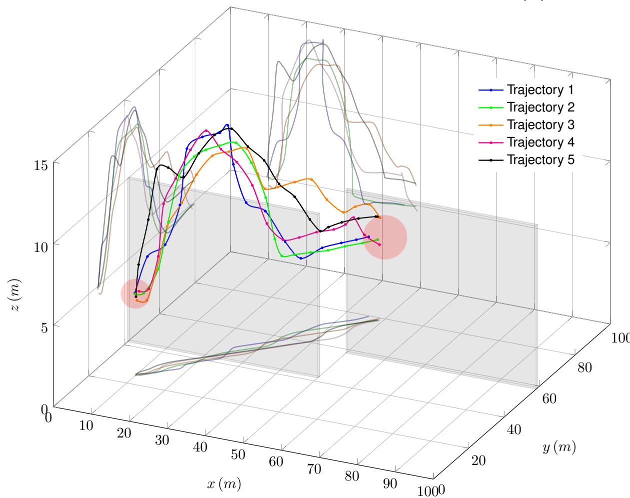
Regarding the trajectories performed in the *WallBaffle* map in the initial position $(10, 25, 5)m$ (Figure 5.10b), the UAV avoids the wall obstacles by flying over and then descending again. Getting high allows it to have a greater probability of GPS availability and thus reduce its uncertainty and the risk of collision (see Figure 4.2c). In this case, bypassing obstacles does not avoid areas where the GPS has a low probability of being available and the flight time may be longer.

On the other hand, with an initial position in $(50, 25, 5)m$, the UAV chooses to go between the obstacles in order to minimize the flight time, instead of flying over them. This choice leads the UAV through areas where the availability of the GPS is less likely, causing the navigation solution chosen to be predominantly INS. This has an impact on the width of the simulated trajectory, which reflects the execution error inherent to the INS. Particularly in trajectory 3 (in Orange) of Figure 5.10c, it can be observed an initial safe policy with an intention of rising to fly over the obstacles, but then the observation received

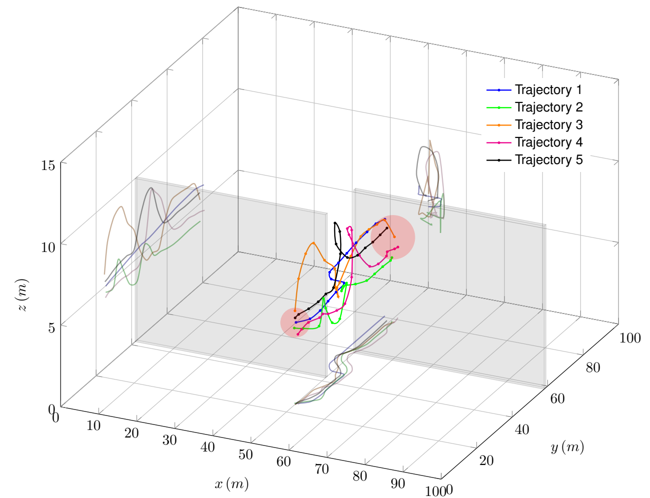
led it to descend again and to fly between the two obstacles. This can be explained on the basis of online planning. In POMCP-GO, similarly to POMCP, there is no guarantee of optimality. The optimal policy would fly over obstacles, but the most promising one, following the heuristic initial value, is to pass between them, yielding a lower flight time. Thus online planning algorithms would have to optimize a lot to be able to change this policy.



(a) Environment in Gazebo.



(b) Initial position at $(10, 25, 5)m$ and goal position at $(50, 80, 5)m$.



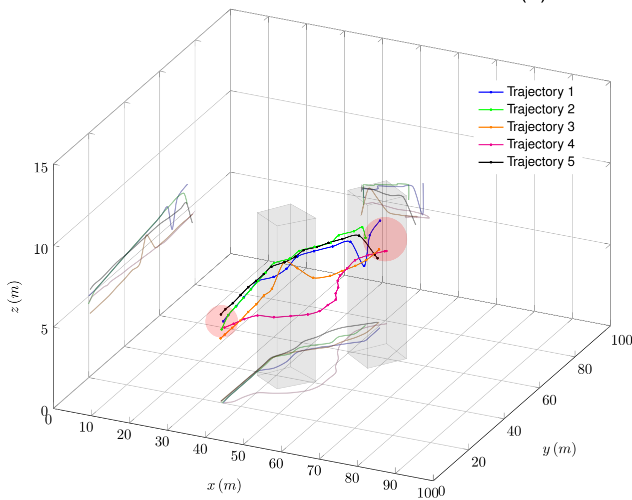
(c) Initial position at $(50, 25, 5)m$ and goal position at $(50, 80, 5)m$.

Figure 5.10: Five trajectories executed in the *WallBaffle* map for two distinct initial positions, using EBC in the action selection strategy in POMCP-GO and a bootstrap time of $15s$ for AMPLE. Projections of the trajectories are illustrated over the three planes xy, xz and yz.

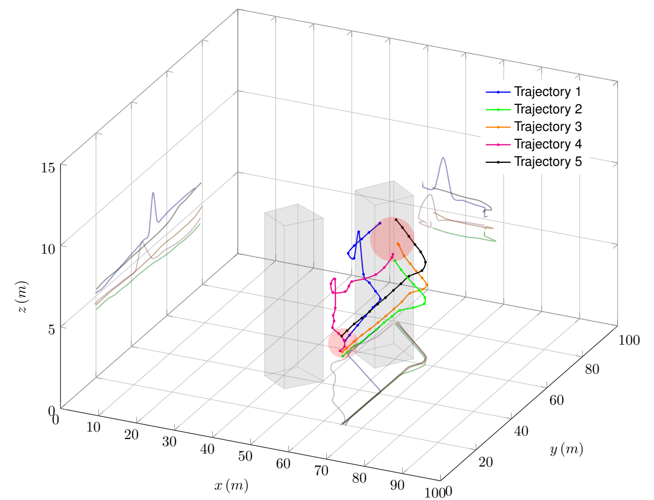
CubeBaffle **map**

Analysing the trajectories performed in the *CubeBaffle* map for both initial positions, there is a clear preference by the UAV to bypass the obstacles on the outside instead of flying between them, as in the latter option the probability of GPS availability is lower (see Figure 4.2g), yielding a higher risk of collision. Furthermore, no trajectory opted to fly over the obstacles, which would be the safest policy, but would also achieve a higher flight time. Particularly in trajectories 1 (in Blue) and 4 (in Magenta) for an initial position at $(65, 20, 5)m$, there is a clear distortion in the executed path, unseen in the remaining trajectories, resulting from the uncertainty on the GPS availability and consequent choice of INS as a navigation solution in the area between the obstacles.

(a) Environment in Gazebo.



(b) Initial position at $(35, 20, 5)m$ and goal position at $(50, 80, 5)m$.



(c) Initial position at $(65, 20, 5)m$ and goal position at $(50, 80, 5)m$.

Figure 5.11: Five trajectories executed in the *CubeBaffle* map for two distinct initial positions, using EBC in the action selection strategy in POMCP-GO and a bootstrap time of $15s$ for AMPLE. Projections of the trajectories are illustrated over the three planes xy, xz and yz.

Chapter 6

Conclusions

This chapter makes an overview of the fulfilment of this thesis objectives, previously stated in Chapter 1. Future work that could improve or complement this dissertation is proposed in Section 6.2.

Building upon previous work in the scope of the NavPlan project, this dissertation deals with the problem of planning safe and efficient trajectories towards a goal under uncertainty, taking into account the availability of sensors that depend on the environment and, with that, deciding on the navigation and guidance strategy to be used by the UAV. Given the specificity of the planning problem, a GNC transition module was integrated in the planning algorithm to propagate the influence of sensor availability on the trajectory being executed.

The formulation of the planning problem as a MOMDP was addressed in Chapter 3, along with a description of the GNC model to be incorporated in the planning algorithm. The resulting MOMDP problem comprised continuous hidden states and discrete fully observable states, which required an expensive computational effort to update the belief states.

6.1 Achievements

Over the developed work addressed in this dissertation, the main objective was to apply a generic planning-while-executing framework to an online planning algorithm able to solve the planning problem under consideration. In order to accomplish this goal, further secondary objectives were fulfilled.

After recalling why the POMCP algorithm was chosen to address this MOMDP planning problem, Chapter 4 was dedicated to the first contribution of this dissertation: a parameter tuning stage for the planning algorithm. As the final aim was to perform online planning, extensive parameter tuning was not a good option. Therefore, adaptive coefficients were proposed to be integrated in the action selection step. Because an offline configuration of the planner allowed for a better understanding of its performance and properties, four distinct selection strategies (UCB1, EBC, DWD and $UCB_{\sqrt{\cdot}}$) and two back-propagation approaches (Classical POMCP and MinPOMCP) were combined and tested in an offline goal-oriented variant of POMCP (POMCP-GO). Comparisons between these approaches allowed to conclude that while DWD and $UCB_{\sqrt{\cdot}}$ did not show a consistent behaviour over the different scenarios,

the EBC results always lied within the values obtained for the fixed coefficients in UCB1, never reaching the values of the best fixed coefficient but still yielding satisfying results for all application cases, thus making it more suited to an online planning configuration. Therefore, a compromise must be made between the desired quality of the results and the time one is willing to spend on the exhaustive search for the optimal value as done in UCB1. Furthermore, the implementation of the back-propagation MinPOMCP generally improved the performance for both UCB1 and EBC selection strategies, accelerating the convergence of the value function as expected, following the work of [70].

Chapter 5 was dedicated to the online approach. Firstly, the POMCP-GO algorithm was changed to an online configuration, with the necessary modifications. Then a generic framework named AMPLE, that allows concurrent planning and execution phases, was described, implemented in ROS, and used to drive the POMCP-GO algorithm. To compare the performance of AMPLE and POMCP-GO combined, online simulations were performed with the POMCP-GO algorithm implemented using the classical paradigm of planning and execution phases interleaved, as well as simulations with an algorithm that consecutively applied default actions. Results showed that although the AMPLE framework did not achieve in general success rates as good as the POMCP-GO algorithm, the overall time of the mission was reduced and actions were ensured to be provided strictly anytime. A final implementation of the resulting framework in a robot simulator, Gazebo, was successfully achieved.

6.2 Future Work

Even though the proposed objectives were achieved, further studies could improve or complement this dissertation. Therefore, the following topics are suggested as future work:

- A study over the scalability of the EBC coefficient to other POMDP planning problems could be interesting for further characterization of this approach;
- Test the proposed combination of the AMPLE framework and the POMCP-GO algorithm with the MinPOMCP back-propagation strategy, which is expected to achieve better results than the ones obtained in Section 5.3;
- Test the proposed combination of the AMPLE framework and the POMCP-GO algorithm in more complex and realistic scenarios with different dynamics that truly reflect an urban area;
- Explore a parallelization of planning requests in the AMPLE framework, rather than the current sequentialization strategy being used, in order to provide the same planning duration for each request;
- Create the dynamic model of the autonomous UAV that is going to be used in test flights in the Gazebo simulation, in order to obtain simulation results as realistic as possible.

Bibliography

[1] C. P. Chanel, F. Teichteil-Königsbuch, and C. Lesire. Multi-target detection and recognition by UAVs using online POMDPs. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence, AAAI 2013*, pages 1381–1387, 2013.

[2] F. Vanegas, D. Campbell, N. Roy, K. J. Gaston, and F. Gonzalez. UAV tracking and following a ground target under motion and localisation uncertainty. In *2017 IEEE Aerospace Conference*, pages 1–10. IEEE, 2017.

[3] J.-A. Delamer, Y. Watanabe, and C. Ponzoni Carvalho Chanel. MOMDP modeling for UAV safe path planning in an urban environment. In *Journées Francophones sur la Planification, la Décision et l'Apprentissage pour la conduite de systèmes (JFPDA 2017)*, 2017.

[4] F. Kleijer, D. Odijk, and E. Verbree. Prediction of GNSS Availability and Accuracy in Urban Environments Case Study Schiphol Airport. In *Location Based Services and TeleCartography II*, Lecture Notes in Geoinformation and Cartography, pages 387–406. Springer, Berlin, Heidelberg, 2009.

[5] J.-A. Delamer, Y. Watanabe, and C. P. Carvalho Chanel. Solving path planning problems in urban environments based on a priori sensors availability and execution error propagation. In *AIAA Scitech 2019 Forum*, 2019.

[6] Y. Watanabe, S. Dessus, and P. Fabiani. Safe path planning with localization uncertainty for urban operation of VTOL UAV. In *AHS 70th Annual Forum*, 2014.

[7] M. W. Achtelik and S. Lynen. Motion- and Uncertainty-aware Path Planning for Micro Aerial Vehicles. *Journal of Field Robotics*, 31(4):676–687, 2014.

[8] J.-A. Delamer, Y. Watanabe, and C. Ponzoni Carvalho Chanel. MOMDP solving algorithms comparison for safe path planning problems in urban environments. In *10th Workshop on Planning, Perception and Navigation for Intelligent Vehicles*, 2018.

[9] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991.

[10] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[11] A. Brooks, A. Makarenko, S. Williams, and H. Durrant-Whyte. Parametric POMDPs for planning in continuous state spaces. *Robotics and Autonomous Systems*, 54(11):887–897, 2006.

[12] A. Gasparetto, P. Boscariol, A. Lanzutti, and R. Vidoni. Path Planning and Trajectory Planning Algorithms: a General Overview. In *Motion and Operation Planning of Robotic Systems*, pages 3–27. Springer, Cham, 2015.

[13] L. Yang, J. Qi, D. Song, J. Xiao, J. Han, and Y. Xia. Survey of Robot 3D Path Planning Algorithms. *Journal of Control Science and Engineering*, 2016(1):1–22, 2016.

[14] L. E. Kavraki, M. N. Kolountzakis, and J. C. Latombe. Analysis of probabilistic roadmaps for path planning. *IEEE Transactions on Robotics and Automation*, 14(1):166–171, 1998.

[15] S. M. LaValle. Rapidly-Exploring Random Trees: A New Tool for Path Planning. Technical report, Computer Science Department, Iowa State University, 1998.

[16] S. Karaman and E. Frazzoli. Sampling-based Algorithms for Optimal Motion Planning. *Special Issue on Robotics: Science and Systems*, 30(7):846–894, 2011.

[17] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *IEEE International Conference on Robotics and Automation*. IEEE, 1985.

[18] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematlk*, 1(1): 269–271, 1959.

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

[20] R. Yue, J. Xiao, S. Wang, and S. L. Joseph. Modeling and path planning of the City-Climber robot part II: 3D path planning using mixed integer linear programming. In *IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE, 2009.

[21] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[22] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1st edition, 1957.

[23] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New Jersey, 1994.

[24] D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, Massachusetts, 3rd edition, 1995.

[25] L. Pineda and S. Zilberstein. Realtime Concurrent Planning and Plan Execution in Stochastic Domains. Technical report, School of Computer Science, University of Massachusetts, 2014.

[26] D. Silver and J. Veness. Monte-Carlo Planning in Large POMDPs. In *Advances in Neural Information Processing Systems 23 (NIPS 2010)*, pages 2164–2172, 2010.

[27] C. Ponzoni Carvalho Chanel, A. Albore, J. T'Hooft, C. Lesire, and F. Teichteil-Königsbuch. AMPLE: an anytime planning and execution framework for dynamic and uncertain problems in robotics. *Autonomous Robots*, 43(1):37–62, 2019.

[28] S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa. Online Planning Algorithms for POMDPs. *Journal of Artificial Intelligence Research*, 32(1):663–704, 2008.

[29] S. Ross and B. Chaib-Draa. AEMS: An Anytime Online Search Algorithm for Approximate Policy Refinement in Large POMDPs. In *Twentieth International Joint Conference on Artificial Intelligence*, pages 2592–2598, 2007.

[30] C. H. Lin, A. Kolobov, E. Kamar, and E. Horvitz. Metareasoning for Planning Under Uncertainty. In *24th International Conference on Artificial Intelligence (IJCAI'15)*, pages 1601–1609, 2015.

[31] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtós, and M. Carreras. ROSPlan: Planning in the Robot Operating System. In *Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS'15)*, pages 333–341, 2015.

[32] J. Barreiro, M. Boyce, M. Do, J. Frank, M. Iatauro, T. Kichkaylo, P. Morris, J. Ong, E. Remolina, T. Smith, and D. Smith. EUROPA: A Platform for AI Planning, Scheduling, Constraint Programming, and Optimization. In *Fourth International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*, 2012.

[33] S. C. W. Ong, S. W. Png, D. Hsu, and W. S. Lee. Planning under Uncertainty for Robotic Tasks with Mixed Observability. *International Journal of Robotics Research*, 29(8):1053–1068, 2010.

[34] M. Araya-López, V. Thomas, O. Buffet, and F. Charpillet. A closer look at MOMDPs. In *22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 197–204. IEEE, 2010.

[35] Mausam and A. Kolobov. *Planning with Markov Decision Processes: An AI Perspective*, volume 6. Morgan & Claypool, 2012.

[36] Eitan Altman. *Constrained Markov Decision Processes*. Chapman and Hall/CRC, 1999.

[37] R. D. Smallwood and E. J. Sondik. The Optimal Control of Partially Observable Markov Processes over a Finite Horizon. *Operations Research*, 21(5):1071–1088, 1973.

[38] B. Bonet and H. Geffner. Solving POMDPs: RTDP-Bel vs. Point-based Algorithms. In *Twenty First International Joint Conference on Artificial Intelligence (IJCAI'09)*, 2009.

[39] H. Bai, D. H. Wee, and S. Lee. Integrated Perception and Planning in the Continuous Space: A POMDP Approach. *International Journal of Robotics Research*, 33(9):1288–1302, 2014.

[40] Z. Sunberg and M. Kochenderfer. Online algorithms for POMDPs with continuous state, action, and observation spaces. In *Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 2017.

[41] K. M. Seiler, H. Kurniawati, and S. P. Singh. An online and approximate solver for POMDPs with continuous action space. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2290–2297. IEEE, 2015.

[42] O. Madani, S. Hanks, and A. Condon. On the Undecidability of Probabilistic Planning and Infinite-Horizon Partially Observable Markov Decision Problems. In *Sixteenth National Conference on Artificial Intelligence*, 1999.

[43] L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, Berlin, Heidelberg, 2006.

[44] J. Pineau, G. Gordon, and S. Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In *International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1025–1030, 2003.

[45] M. T. J. Spaan and N. Vlassis. Perseus: Randomized Point-based Value Iteration for POMDPs. *Journal of Artificial Intelligence Research*, 24:195–220, 2005.

[46] T. Smith and R. Simmons. Heuristic Search Value Iteration for POMDPs. In *Twentieth Conference on Uncertainty in Artificial Intelligence*, pages 520–527, 2004.

[47] H. Kurniawati, D. Hsu, and W. Sun Lee. SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces. In *Robotics: Science and Systems IV*, 2008.

[48] A. Somani, N. Ye, D. Hsu, and W. S. Lee. DESPOT: Online POMDP planning with regularization. In *Advances in Neural Information Processing Systems 26 (NIPS)*, pages 1772–1780. Curran Associates, Inc., 2013.

[49] S. Paquet, L. Tobin, and B. Chaib-draa. Real-Time Decision Making for Large POMDPs. In *18th Conference of the Canadian Society for Computational Studies of Intelligence*, pages 450–455. Springer, 2005.

[50] J. K. Satia and R. E. Lave. Markovian Decision Processes with Probabilistic Observation of States. *Management Science*, 20(1):1–13, 1973.

[51] R. He, E. Brunskill, and N. Roy. PUMA: Planning under Uncertainty with Macro-Actions. In *Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI'10)*, 2010.

[52] K. Hauser. Randomized Belief-Space Replanning in Partially-Observable Continuous Spaces. In *Algorithmic Foundations of Robotics IX*, volume 68, pages 193–209. Springer, Berlin, Heidelberg, 2010.

[53] H. Kurniawati and V. Yadav. An Online POMDP Solver for Uncertainty Planning in Dynamic Environment. In *Robotics Research*, pages 611–629. Springer, Cham, 2016.

[54] P. Karkus, D. Hsu, and W. S. Lee. QMDP-Net: Deep Learning for Planning under Partial Observability. In *Neural Information Processing Systems (NIPS)*, 2017.

[55] E. A. Hansen and Z. Feng. Dynamic Programming for POMDPs using a Factored State Representation. In *Fifth International Conference on AI Planning Systems*, pages 130–139, 2000.

[56] S. C. W. Ong, S. W. Png, D. Hsu, and W. S. Lee. POMDPs for robotic tasks with mixed observability. In *Robotics: Science and Systems V*, 2009.

[57] G. Chaslot, J. W. H. M. Uiterwijk, J.-T. Saito, B. Bouzy, and H. Jaap Van Den Herik. Monte-Carlo Strategies for Computer Go. In *Benelux Artificial Intelligence Conference*, pages 83–91, 2006.

[58] S. Edelkamp and Z. Tang. Monte-Carlo Tree Search for the Multiple Sequence Alignment Problem. In *Eighth International Symposium on Combinatorial Search (SoCS'15)*, 2015.

[59] M. Kearns, Y. Mansour, and A. Y. Ng. Approximate Planning in Large POMDPs via Reusable Trajectories. In *12th International Conference on Neural Information Processing Systems (NIPS'99)*, pages 1001–1007, 1999.

[60] D. Bertsekas and D. Castanon. Rollout Algorithms for Stochastic Scheduling Problems. *Journal of Heuristics*, 5(1):89–108, 1999.

[61] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computer and Games*, pages 72–83. Springer, Berlin, Heidelberg, 2006.

[62] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[63] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. In *Machine Learning*, volume 47, pages 235–256. Kluwer Academic Publishers, 2002.

[64] T. Pepels, T. Cazenave, M. H. M. Winands, and M. Lanctot. Minimizing Simple and Cumulative Regret in Monte-Carlo Tree Search. In *Computer Games*, Communications in Computer and Information Science, pages 1–15. Springer, Cham, 2014.

[65] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. V. D. Herik, J. W. H. M. Uiterwijk, and B. Bouzy. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008.

[66] D. Tolpin and S. E. Shimony. MCTS Based on Simple Regret. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[67] C. D. Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[68] David Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[69] N. A. Vien and M. Toussaint. Hierarchical Monte-Carlo Planning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 3613–3619, 2015.

[70] T. Keller and M. Helmert. Trial-based Heuristic Tree Search for Finite Horizon MDPs. In *Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS'13)*, pages 135–143, 2013.

[71] Tomáš Kozelek. *Methods of MCTS and the game Arimaa*. PhD thesis, Charles University, Prague, 2009.

[72] H. W. Sorenson. *Kalman Filtering: Theory and Application*. IEEE Press, New York, 1985.

[73] J.-A. Delamer. *Planification de stratégies de navigation et de guidance pour des drones autonomes dans des milieux encombrés*. Unpublished, Institut Supérieur de l'Aéronautique et de l'Espace (ISAE), 2019.

[74] S. Bubeck, R. Munos, and G. Stoltz. Pure exploration in finitely-armed and continuous-armed bandits. *Theoretical Computer Science*, 412(19):1832–1852, 2011.

[75] B. Mettler, Z. Kong, C. Goerzen, and M. Whalley. Benchmarking of obstacle field navigation algorithms for autonomous helicopters. *Annual Forum Proceedings - AHS International*, 3:1936–1953, 2010.

[76] F. Teichteil-Konigsbuch, C. Lesire, and G. Infantes. A generic framework for anytime execution-driven planning in robotics. In *IEEE International Conference on Robotics and Automation*, pages 299–304. IEEE, 2011.

[77] J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. von Stryk. Comprehensive Simulation of Quadrotor UAVs Using ROS and Gazebo. In *Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 400–411. Springer, Berlin, Heidelberg, 2012. URL `https://github.com/tu-darmstadt-ros-pkg/hector_quadrotor`. (accessed: 2019-10-30).

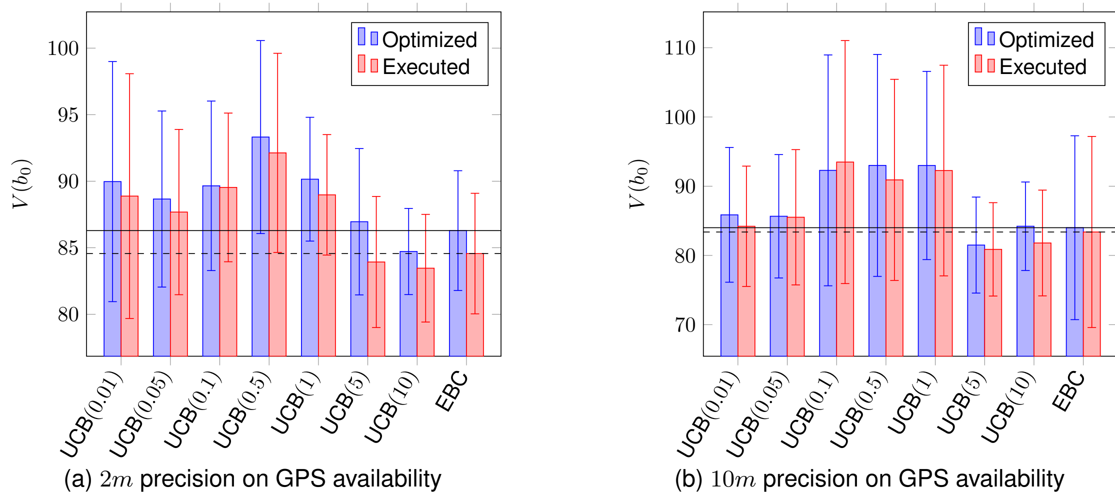
Appendix A

Additional results in parameter tuning stage

A.1 Comparison between EBC and the extensive search using UCB1

The following Figures A.1, A.2, A.3 and A.4 depict the results of the metrics $V(b_0)$ optimized and $V(b_0)$ simulated for the extensive search process using the UCB1 formula, along with the results of the EBC for comparison, for the eight case studies considered. The data is organized in the form of *bars*, being the tip of the filled bar the average, while the extremes of the vertical lines represent the standard deviation.

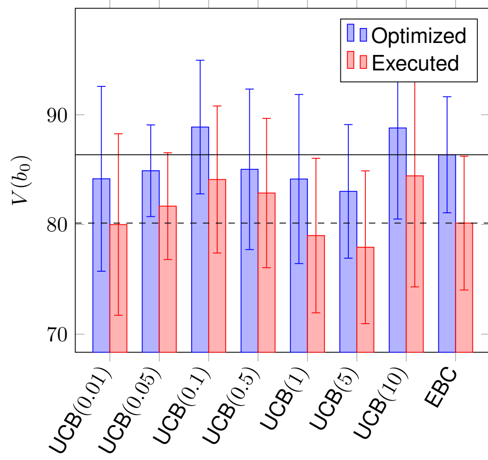
Comparing the UCB1 fixed coefficients and the EBC results, one can verify that both $V(b_0)$ optimized and simulated computed for EBC lie within the values obtained for the fixed coefficients for every case study, although never reaching the values of the best fixed coefficient. Therefore, a compromise must be made between the desired quality of the results and the time one is willing to spend on the exhaustive search for the optimal coefficient value before online planning.



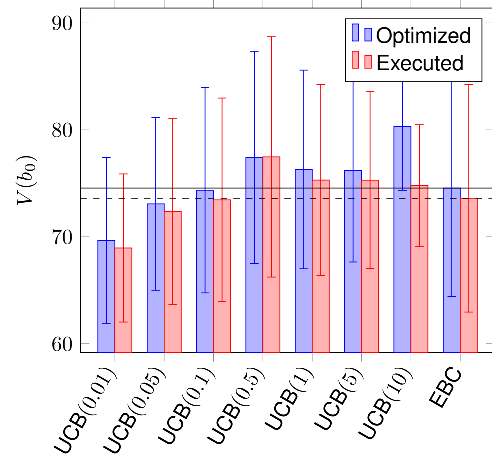
(a) $2m$ precision on GPS availability

(b) $10m$ precision on GPS availability

Figure A.1: *WallBaffle* map with initial position in $(10, 25, 5)m$.

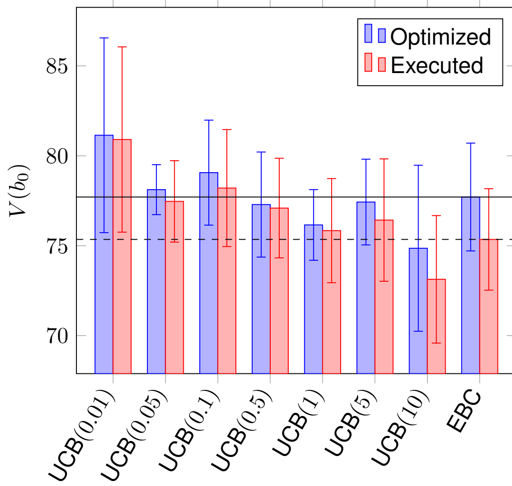


(a) $2m$ precision on GPS availability

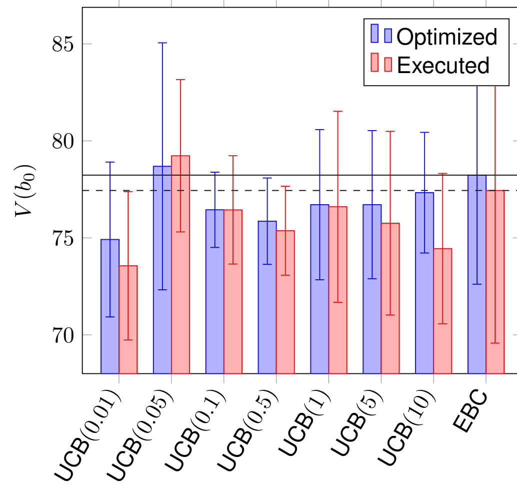


(b) $10m$ precision on GPS availability

Figure A.2: *WallBaffle* map with initial position in $(50, 25, 5)m$.

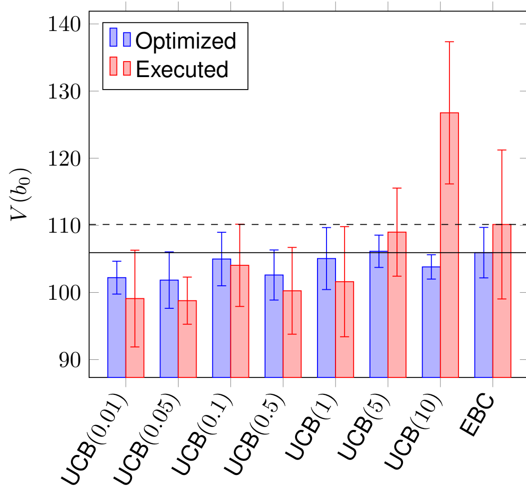


(a) $1m$ precision on GPS availability

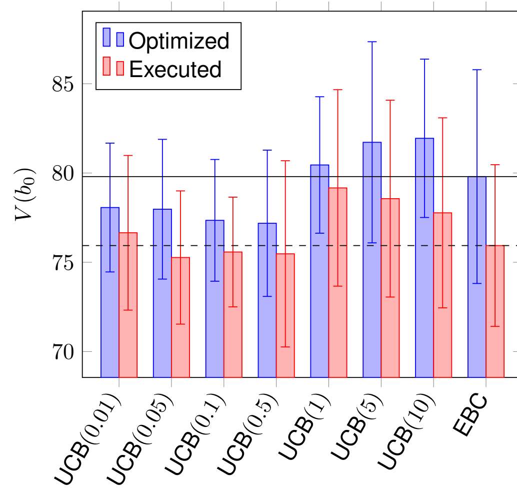


(b) $2m$ precision on GPS availability

Figure A.3: *CubeBaffle* map with initial position in $(35, 20, 5)m$.



(a) $1m$ precision on GPS availability



(b) $2m$ precision on GPS availability

Figure A.4: *CubeBaffle* map with initial position in $(65, 20, 5)m$.

A.2 Results during the optimization process

In the following figures, the strategies among the ones described in Section 4.2 that achieved the best and the worst overall performances in the simulations regarding the parameter tuning are presented for each of the eight case studies considered. The first column graphs represent the evolution of the value of the initial belief state $V(b_0)$ optimized and executed along the trials. The data is organized as the average being the mark, while the extremes of the vertical lines represent the standard deviation. Additionally, the second and third columns are plots of 1000 trajectories simulated at the end of the 50000 trials in the xy plane and yz plane, respectively.

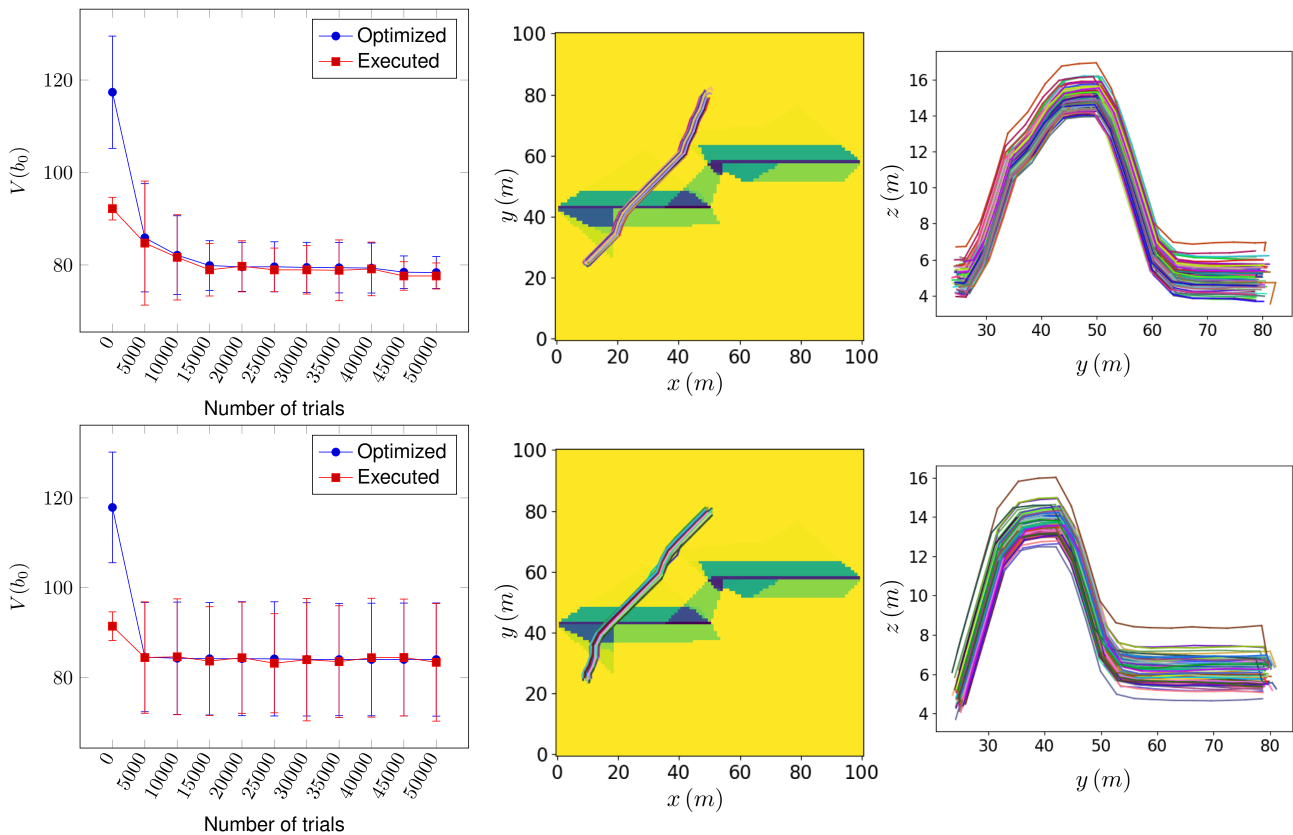


Figure A.5: Best (UCB$_{\sqrt{(\cdot)}}$ – up) and worst (EBC – down) strategies for *WallBaffle* map with initial position in $(10, 25, 5)m$ and $10m$ precision on GPS availability.

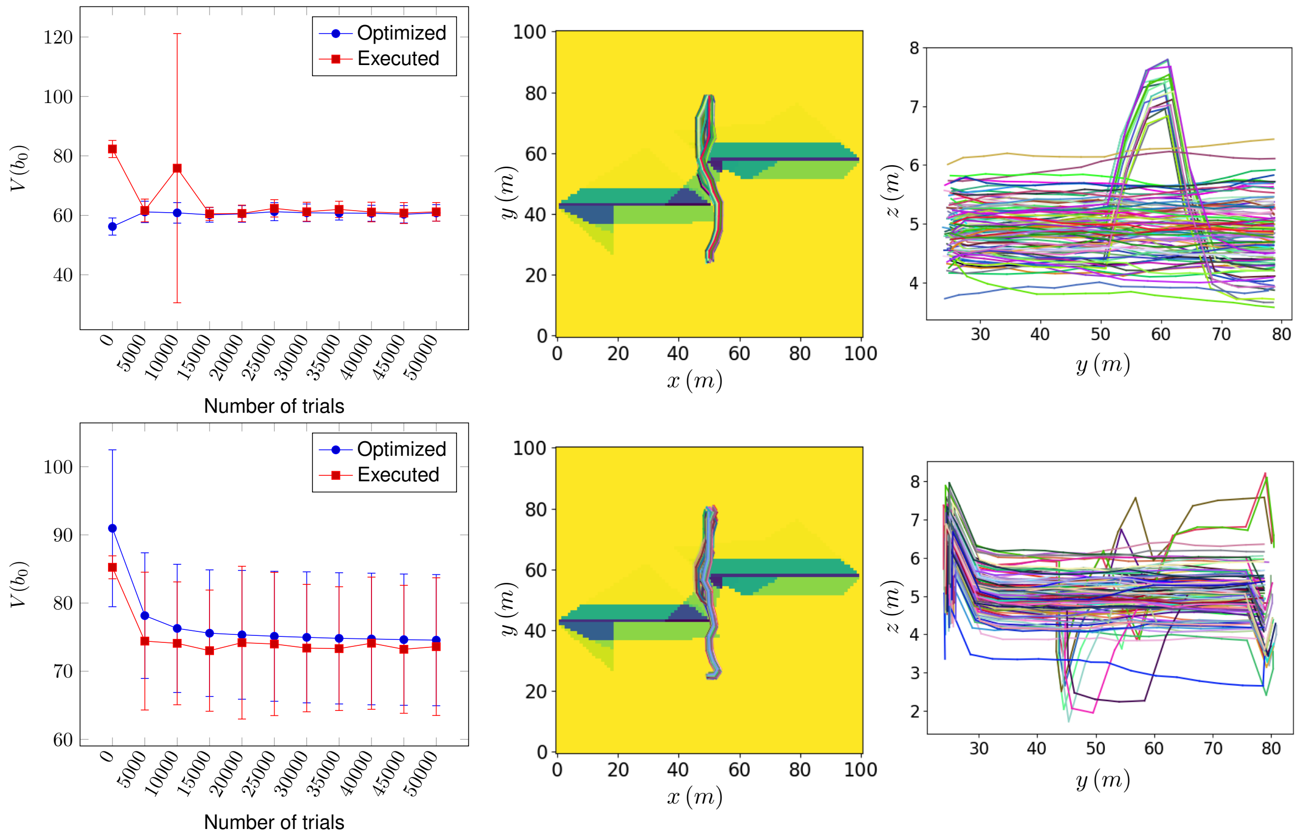


Figure A.6: Best (EBC$_{\text{MP}}$ – up) and worst (EBC – down) strategies for *WallBaffle* map with initial position in $(50, 25, 5)m$ and $10m$ precision on GPS availability.

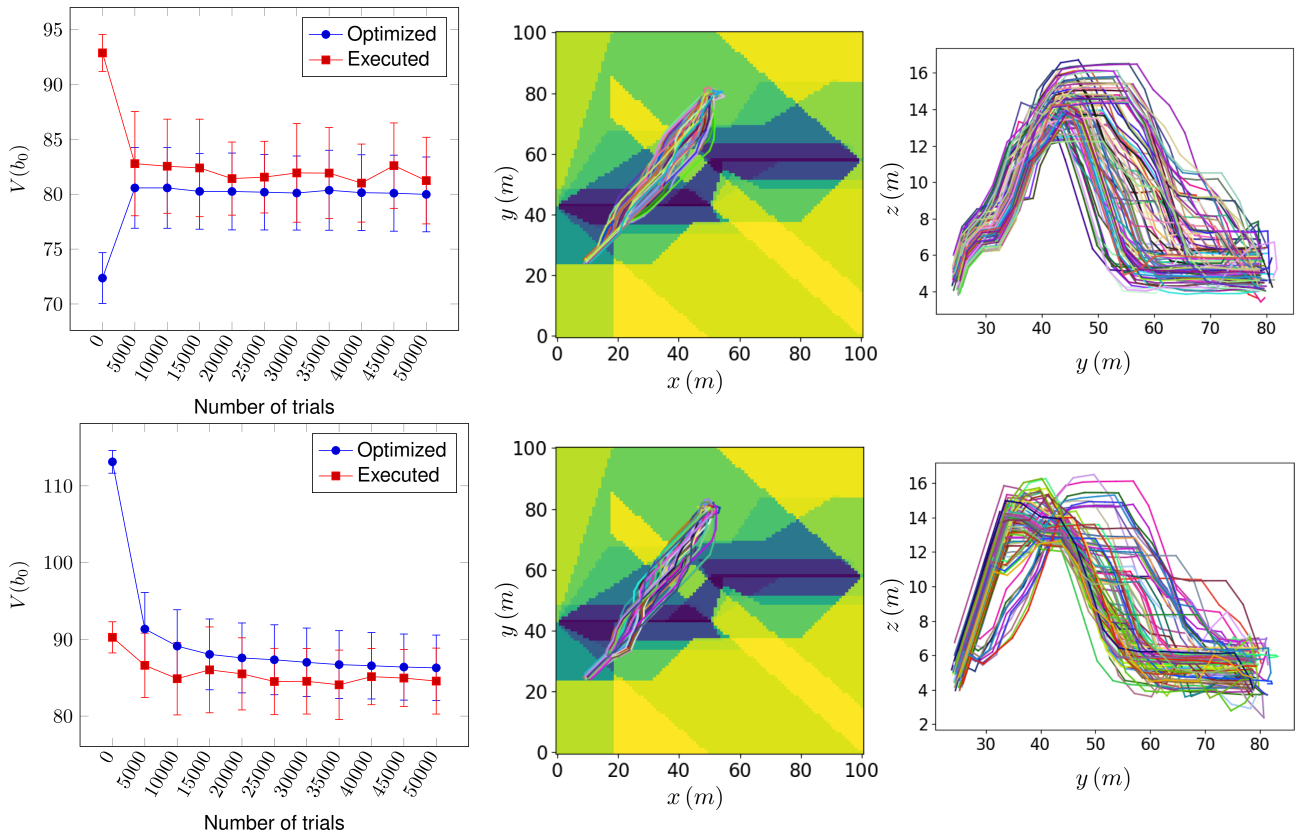


Figure A.7: Best (UCB$_{\text{MP}}$ – up) and worst (EBC – down) strategies for *WallBaffle* map with initial position in $(10, 25, 5)m$ and $2m$ precision on GPS availability.

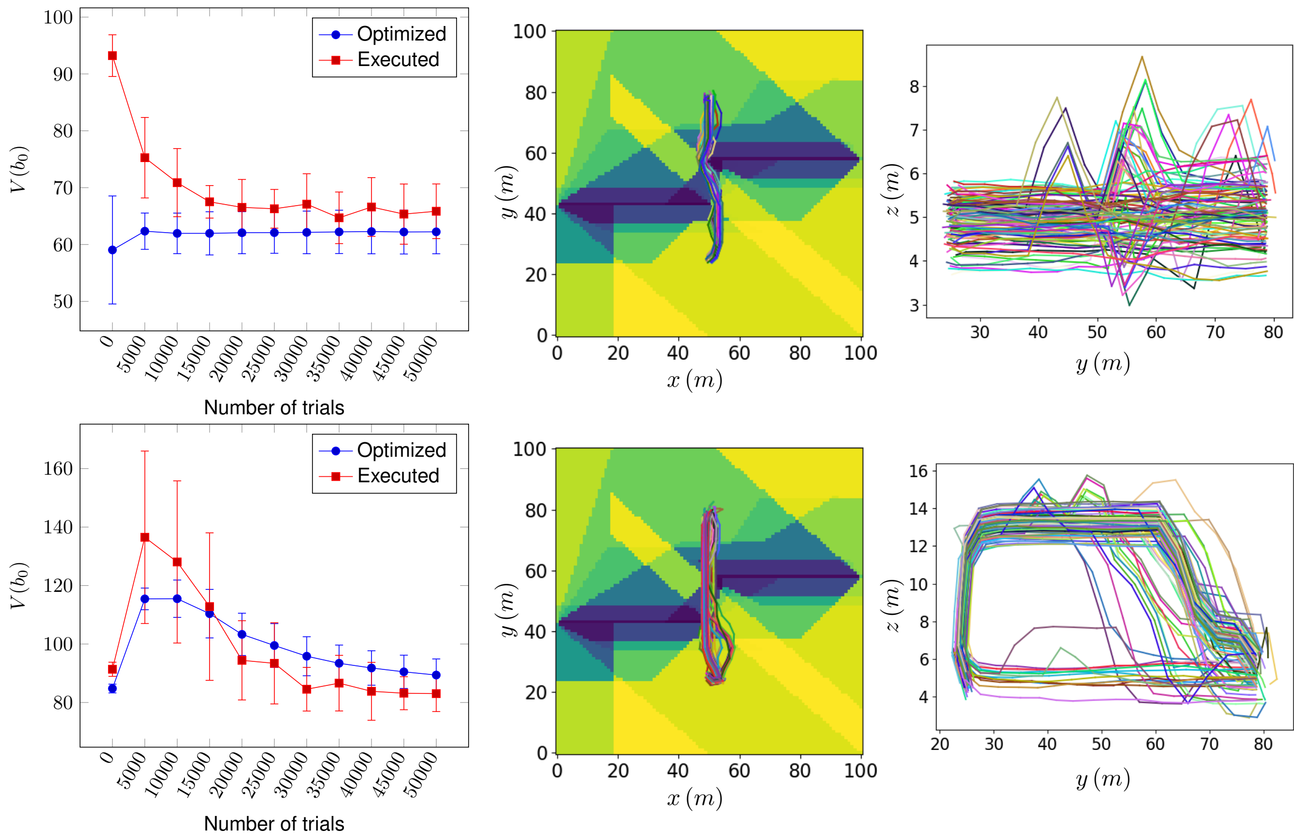


Figure A.8: Best (UCB$_{\text{MP}}$ – up) and worst (DWD – down) strategies for *WallBaffle* map with initial position in $(50, 25, 5)m$ and $2m$ precision on GPS availability.

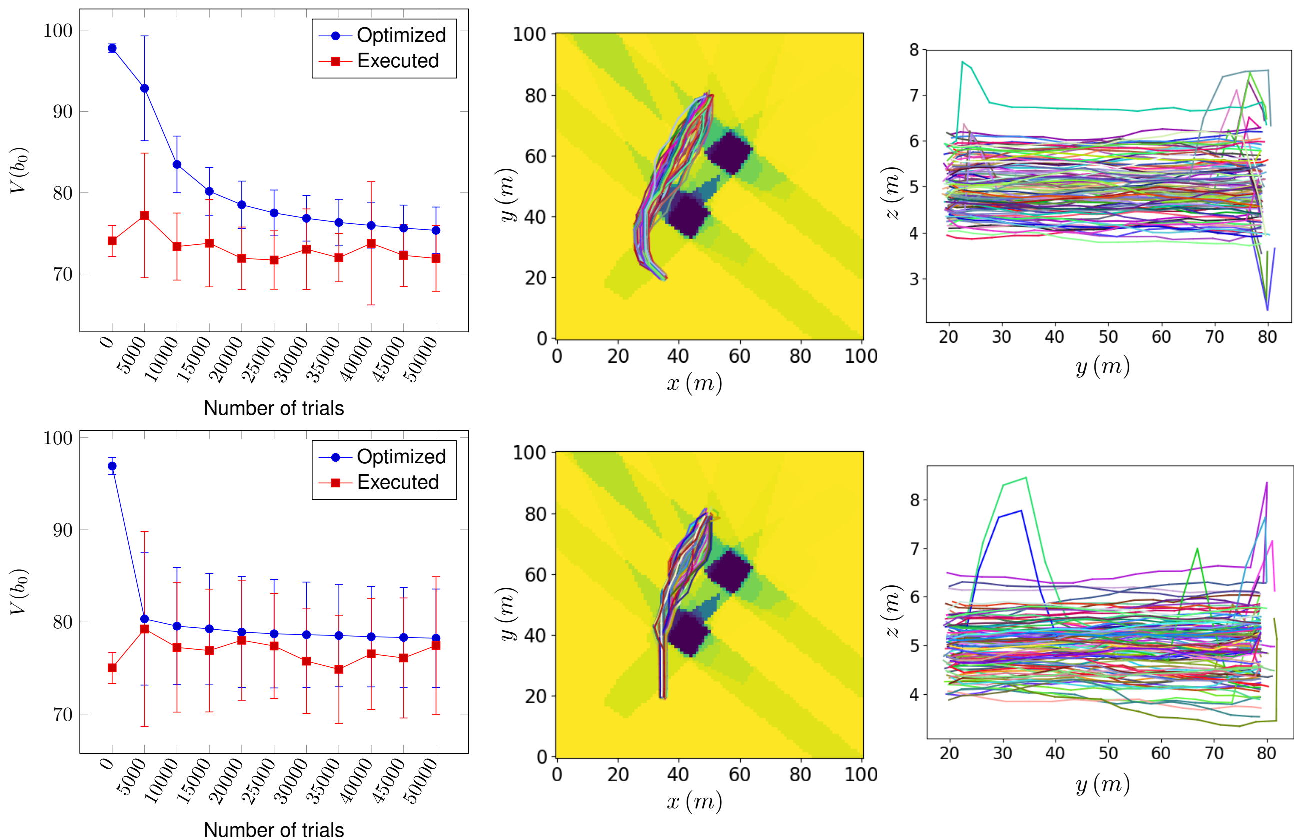


Figure A.9: Best (DWD – up) and worst (EBC – down) strategies for *CubeBaffle* map with initial position in $(35, 20, 5)m$ and $2m$ precision on GPS availability.

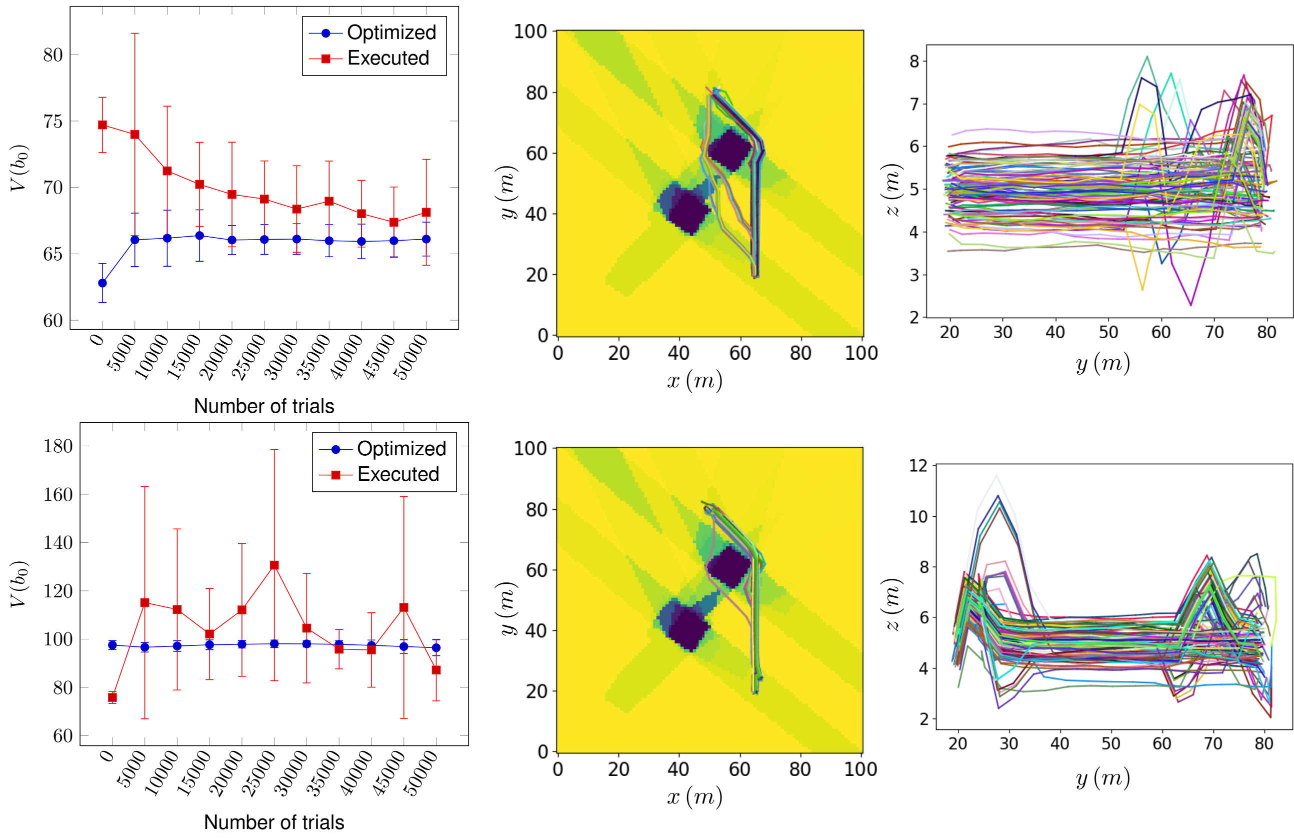


Figure A.10: Best (EBC$_{\text{MP}}$ – up) and worst (DWD – down) strategies for *CubeBaffle* map with initial position in $(65, 20, 5)m$ and $2m$ precision on GPS availability.

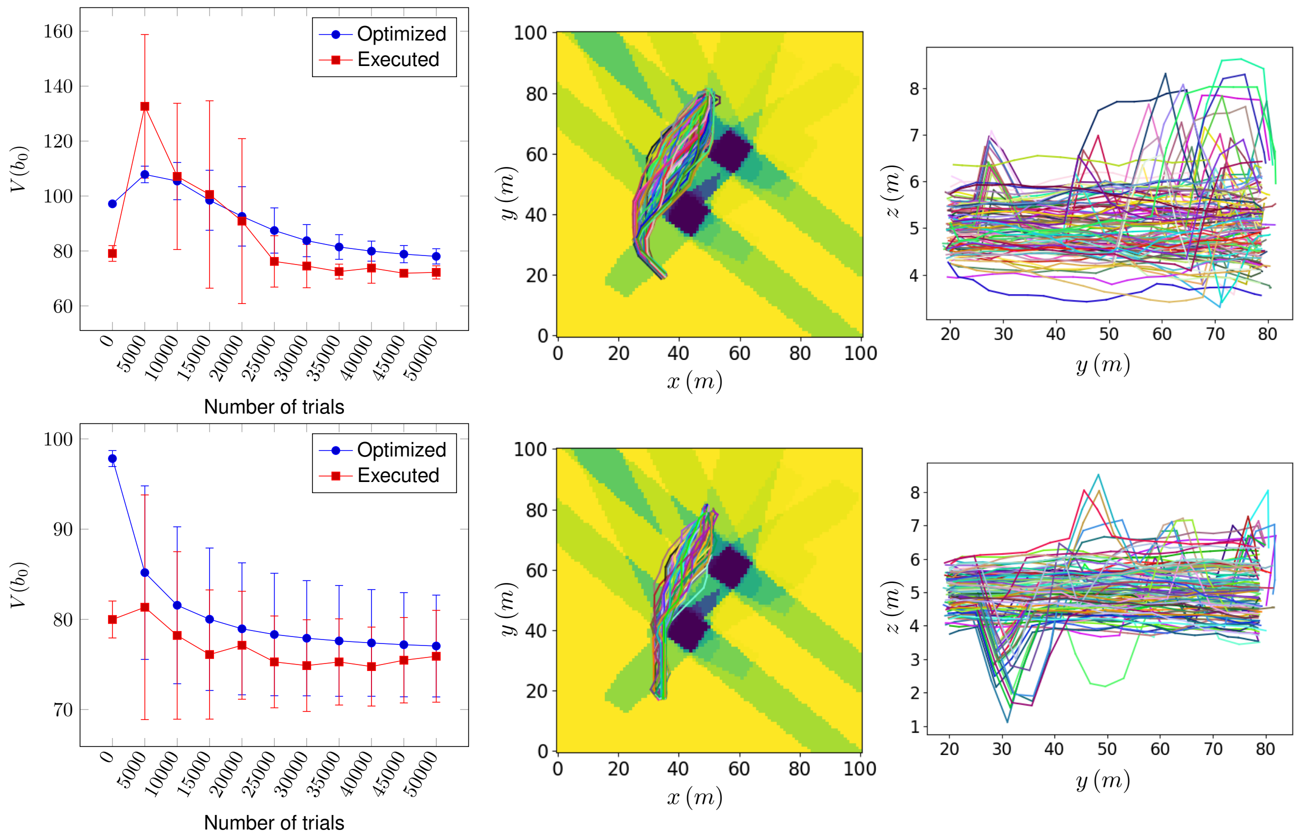


Figure A.11: Best (DWD – up) and worst (UCB$_{\sqrt{(\cdot)}}$ – down) strategies for *CubeBaffle* map with initial position in $(35, 20, 5)m$ and $1m$ precision on GPS availability.

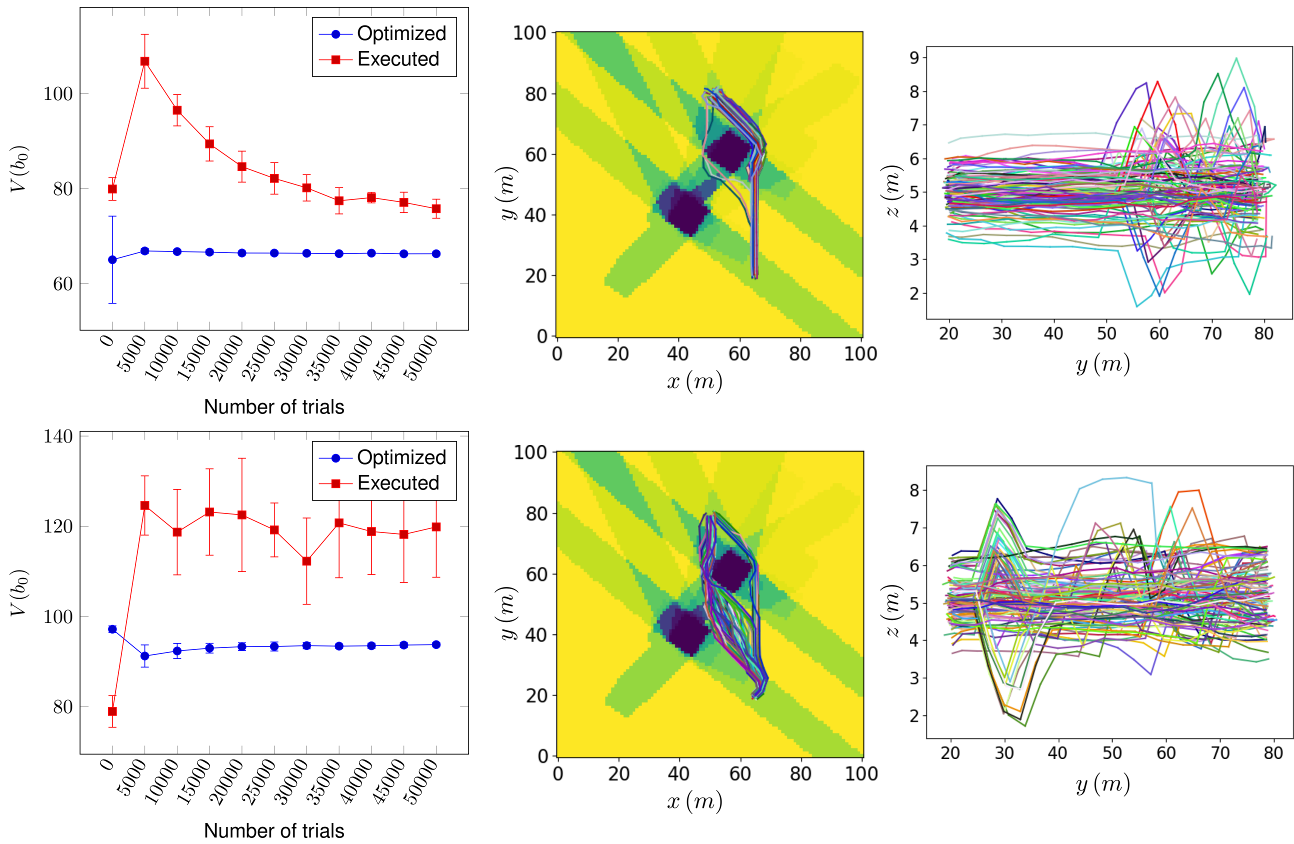


Figure A.12: Best (EBC$_{\text{MP}}$ – up) and worst (DWD – down) strategies for *CubeBaffle* map with initial position in $(65, 20, 5)m$ and $1m$ precision on GPS availability.