AMPLE: an anytime planning and execution framework for dynamic and uncertain problems in robotics

Caroline Ponzoni Carvalho Chanel, Alexandre Albore, Jorrit T'hooft, Charles Lesire, Florent Teichteil-Königsbuch

AMPLE: an anytime planning and execution framework for dynamic and uncertain problems in robotics

Caroline P. Carvalho Chanel*[1], Alexandre Albore†[2], Jorrit T'Hooft‡[3], Charles Lesire§[3], and Florent Teichteil-Knigsbuch¶[4]

[1] *Université de Toulouse, ISAE-SUPAERO, Toulouse, France*
[2] *IRT Saint-Éxupéry, Toulouse, France*
[3] *ONERA, Toulouse, France*
[4] *Airbus Central Research & Technology, Toulouse, France*

Received: 13 April 2016 / Accepted: 20 January 2018

Abstract

Acting in robotics is driven by reactive and deliberative reasonings which take place in the competition between execution and planning processes. Properly balancing reactivity and deliberation is still an open question for harmonious execution of deliberative plans in complex robotic applications. We propose a flexible algorithmic framework to allow continuous real-time planning of complex tasks in parallel of their executions. Our framework, named AMPLE, is oriented towards robotic modular architectures in the sense that it turns planning algorithms into services that must be generic, reactive, and valuable. Services are optimized actions that are delivered at precise time points following requests from other modules that include states and dates at which actions are needed. To this end, our framework is divided in two concurrent processes: a planning thread which receives planning requests and delegates action selection to embedded planning softwares in compliance with the queue of internal requests, and an execution thread which orchestrates these planning requests as well as action execution and state monitoring. We show how the behavior of the execution thread can be parametrized to achieve various strategies which can differ, for instance, depending on the distribution of internal planning requests over possible future execution states in anticipation of the uncertain evolution of the system, or over different underlying planners to take several levels into account. We demonstrate the flexibility and the relevance of our framework on various robotic benchmarks and real experiments that involve complex planning problems of different natures which could not be properly tackled by existing dedicated planning approaches which rely on the standard plan-then-execute loop.

Keywords: automated planning, planning and execution, anytime framework, autonomous robots

*email: caroline.chanel@isae-supaero.fr
†email: alexandre.albore@irt-saintexupery.com
‡email: jorrit.thooft@onera.fr
§email: charles.lesire@onera.fr
¶email: florent.teichteil-koenigsbuch@airbus.com

1 Introduction

While most robots can nowadays autonomously adapt to their local environment (e.g. avoiding obstacles, reacting to wind gusts, stabilizing flight) or complete simple missions in static environments (e.g. joining waypoints given by hands), few of them can achieve higher-level missions in unknown or highly dynamic environments, which requires to reason globally in terms of both space and time. For instance, a fully autonomous search and rescue mission would require to continuously map the surrounding and distant areas, allocate and schedule the various tasks that are needed to search for survivors and help them while avoiding moving obstacles, by anticipating the long-term dynamics of the environment which is itself impacted by the effects of the robot's current actions.

One of the key ingredients in making fully autonomous robots is the ability to select and organize actions over time in order to fulfill some high level objectives (e.g. maximizing reward functions, reaching symbolic system states that aggregate topological robot locations and mission status). This implies controlling the course of potentially uncertain events that may occur due to the execution of the actions: their effects on the environment, as well as the imperfection of sensors and actuators onboard the robot, are indeed various sources of exogenous events impacting the mission (Ingrand and Ghallab, 2014).

Artificial Intelligence planning (Nau et al., 2004) is a model-based and theorem-proving approach to this problem. Given a dynamical model of the actions of the robot, which includes environmental conditions on their applicability and descriptions of their effects on the world, AI planning aims at formally proving whether a sequence of actions named *plan* can fulfill a given objective. Whereas AI planning does not provide any means to actually generate the high-level mission objectives, it allows the robot to autonomously select relevant actions and organize them over time in order to achieve these objectives. Many variants of AI planning have been studied, ranging from single-agent deterministic "classical" planning, where effects of actions and environmental observations of a single agent are deterministic, to multi-agent probabilistic planning with concurrent durative actions, where many agents execute in parallel time-dependent actions with probabilistic action effects and observations of the world, and including many intermediate models of variable computational tractability or practical usability.

However, planning has, until recent years, been seen under a purely theoretical prism with relatively little consideration for the real test cases, especially applications that involve time-constrained interleaving of planning and acting, as is typically the case in robotics. In such situations, as time goes on, the actions that were planned few seconds ago might be no more suitable nor feasible in the current situation because of the increasing discrepancy between the expected situation and the observed one. Therefore, a robot might have to continuously change its plan, and an ill thought out autonomous system might permanently chase after the current changing world state without being able to execute its planned actions. Such poorly-designed systems are generally unable to achieve complex missions due to their inability to properly execute the plans that are needed to achieve high-level mission objectives. Thus, an important aspect of real-time planning in robotics is the necessary tight coordination between the predicted evolution of the environment and the time required to generate plans so as to maximize the chance that planned actions are valid in the states where they were expected to be executed.

Many successful planning frameworks for autonomous systems have been de-

signed in the specific case of scheduling which consists in ordering 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, whith an action selection mechanism that uses the strong assumption that the environment is deterministic: e.g. EUROPA (Barreiro et al., 2012), IxTeT and IxTet-ExEc (Lemai and Ingrand, 2004) or T-REX (McGann et al., 2008). Such approaches are particularly useful in the field of robotics, because multiple concurrent subsystems (e.g. arms, cameras, wheels, etc.) have to be concurrently controlled and activated in parallel. This is even more true for synchronous languages as well, where languages as PLEXIL (Verma et al., 2006) assume that the only source of the non-determinism comes from the environment, so they specify the plan as sets of conditional actions organized in a tree structure and relying on a reactive platform for execution, similarly to how conditional planners do. Our approach, however, follows the direction of reactive or continuous planning (e.g. CASPER (Knight et al., 2001)) where an "iterative planning" procedure focuses on fixing flaws in the proposed plans, when an anomaly occurs at execution-time, and on continuously updating goals, state, and planning horizon. Even with this approach, continuous planners do not guarantee to converge toward a solution within the time and resource constraints given; we improve this approach by giving to the planning framework, the capability to switch between planning strategies, with the guarantee to provide a plan anytime, with respect to time constraints.

The need for improving deliberative solutions is also highlighted by Nau et al. (2015) and Ghallab et al. (2014). They focus on two interconnected principles: a hierarchical structure to integrate the actor's deliberation functions, and *continual online planning and reasoning throughout the acting process*. Their argument is that the importance and difficulty of deliberative acting has been underestimated so far, and they consequently call for more research on the problems that arise when endeavoring to integrate acting with continuous planning and deliberation.

A system interacting with a real environment would need, 1) to move toward a more expressive and general planning model, built upon the classical deterministic one, to extend the scope of the solver, and 2) the ability to recover from execution errors, and to actively interact with the environment. This second point aims at a general framework for plan execution that goes further the single action of planning or simply an integration of planning and acting. To address this issue, we decided to tackle planning in real-world applications by developping a "planning framework" that actively supports the planning process through execution. This is necessary because, during the execution, the planner could need to reactively adapt to the changes in the environment, to revise its preferences, and to simply provide a revised solution to the problem.

In this sense, we have worked in developing ***AMPLE*** - *Anytime Meta PLannEr* (Teichteil-Konigsbuch et al., 2011; Carvalho Chanel et al., 2013, 2014), which is a general framework for continuous and anticipatory planning while executing. AMPLE is *generic* with regards to planning algorithms, strictly *anytime* in the sense of policy execution under time constraints, *conditional* by prioritizing future execution states and *reactive* to environment changes (and to other components' requests). AMPLE has been successfully applied to solve UAV's missions such as : target detection (Carvalho Chanel et al., 2013, 2014) and autonomous landing (Teichteil-Konigsbuch et al., 2011), using (Partially Observable) Markov Decision Process planning algorithms.

The contribution of this paper is first a detailed description of planning paradigms for managing several kind of uncertainty (Sect. 2), that leads to a general schema

for solving such problems. This schema generalizes the AMPLE framework as we show that we can encompass any planning paradigm (not only (PO)MDP as in (Teichteil-Konigsbuch et al., 2011; Carvalho Chanel et al., 2013, 2014)). We then describe AMPLE (Sect. 3), better justify the variants, and present a new variant for integrating several deterministic planning algorithms to manage uncertainty by solving several sub-problems at the same time. Finally, we present several experiments (some already published in (Teichteil-Konigsbuch et al., 2011; Carvalho Chanel et al., 2013, 2014)) in a unified way, showing the complementarity and the interest of each evaluation. These experiments also include a new application to an indoor mobile robot navigation problem (Sect. 4). These experiments emphasize that AMPLE is an effective framework for anytime planning and executing in dynamic and uncertain domains: it can manage several planning paradigms in a unified framework, it has been successfully applied to solving probabilistic problems onboard robots in real-time, and allows to implement execution strategies to adapt to specific missions.

2 Planning models

Real world applications, as robotics, are very demanding in terms of performance and reactivity in *a priori* unknown environments. Dynamic domains are challenging as they imply several complex aspects that participate to the decision making process, namely, non-determinism, exogenous events, partial observability, etc.

Automated Planning is the model-based approach to autonomous behavior, the branch of Artificial Intelligence dedicated to decision making and acting. Methods for classical planning have had great success, and are in continual development and subject of research.

Yet, the classical planning model has been extended in several directions to embrace more general problems that deterministic planning cannot handle. These extensions go in the direction of more elaborate models of planning to include incomplete information and sensing, motivated by realistic applications when the planning agent can be immersed in a non perfectly known environment, where the effects of acting can be non-deterministic or encoded as probabilities of success. In the domain of robotics, it is common to consider continuous (or hybrid) state spaces, rather than discrete state spaces: in many cases the formulations appear identical, but the continuous case is generally more complicated, even if it usually maintains some of the properties from the discrete case. In the following pages, we will consider only discrete state spaces, for sake of simplicity; in the AMPLE framework, neither the applications, nor the use of the planner are dependent from the paradigm considered.

In the next section, we first describe general models of classical planning, and then extensions to models used for decision-making under uncertainty, namely conformant and contingent planning. Such models are then extended to include probabilistic models, where uncertainty about actions' outcomes are modeled as a probability distribution function; consequently the goals are represented as utility functions.

2.1 Classical Planning

The paradigm of classical planning can be translated as a directed graph whose nodes represent *states*, and whose edges represent *actions*. The change of state is then represented as a transition from a source node toward a target node representing the next state. The objective of automated planning is to find a *plan*, a path from

the node in the graph representing the initial state to a goal node representing a state recognized as a goal state of the problem. In classical planning, the plan is represented as a linearly ordered finite sequence of actions that drive the system from an initial state to a goal state. The formal model underlying the domain-independent planning problem can be described as follows:

Definition 2.1 (Classical Planning Model) *The classical planning model $\mathcal{M}$ is defined as the tuple $\mathcal{M} = \langle S, s_0, S_G, A, \varphi \rangle$ where:*

- *S is a finite set of states,*
- *$s_0 \in S$ is the initial state,*
- *$S_G \subseteq S$ is the set of goal states,*
- *A is the set of actions,*
- *$\varphi : \left| \begin{array}{lcl} S \times A & \rightarrow & S \\ (s,a) & \mapsto & \varphi(s,a) = s' \end{array} \right.$ is a state* transition *function where s' is the state resulting from applying the action a to a given state s.*

An action a is applicable *in a state s when at least one target state s' such that $\varphi(s,a) = s'$ exists. We define the applicability domain of an action a as the subset S_a such that, $\varphi|_{S_a}$ is injective. Executing a sequence of applicable actions $[a_0, \ldots, a_n]$ in a given state s_0, results in a chain of states such that $\varphi(s_0, [a_0, \ldots, a_n]) = [s_0, \ldots s_{n+1}]$, with $\varphi(s_i, a_i) = s_{i+1}$, for $0 \leq i \leq n$, and a_i an applicable action in s_i. We define the action a_0 as the* noop *action, such that $\varphi(s_0, a_0) = s_0$.*

2.1.1 General algorithmic schema for classical planning.

Several approaches to solve classical planning problems have been developed, all based on building a structure to express the problem, and then searching in that structure for solutions. SATPlan expresses the problem as a set of clauses derived from a GraphPlan-style planning graph up to a finite horizon (Blum and Furst, 1995), where each specific instance of an action or fluent at a point in time is a proposition; a general purpose SAT solver is then used to try to find a satisfying truth assignment for the formula, allowing parallel computations when certain conditions do hold on actions (Kautz and Selman, 1996; Rintanen et al., 2006).

However, the majority of actual planners are based on search in a graph (Hoffmann and Nebel, 2001; Richter and Westphal, 2010; Helmert et al., 2011; Lipovetzky and Geffner, 2012). The search can be either boosted by fast heuristics automatically extracted from the declarative representation of the problem, or not. Satisfiability planning has, in fact, considerably improved thanks to state-of-the-art heuristics that select, from a relaxed version of the problem, the more promising nodes of the graph to expand (Bonet and Geffner, 2001; Keyder et al., 2014; Domshlak et al., 2015), while blind searches use different criteria for node selection. Even if heuristics based on an estimation of the goal distance are the most historically successful and widespread, evaluation techniques using width-based algorithms (i.e. considering the *novelty* of the values encoutered in the newly expanded nodes in the search graph) that make no use of heuristic estimators, helpful actions (Hoffmann and Nebel, 2001), landmarks (Hoffmann et al., 2004; Richter et al., 2008), have recently been proved to obtain excellent performances (Lipovetzky and Geffner, 2017; Katz et al., 2017).

We represent in Alg. 1 the general search algorithm for a solution in a graph, using the heuristic evaluation function $f(n)$ to order the *open* list to select the most promising node to be expanded next. The search algorithm we use here to illustrate a search in a graph is a variant of the best-first search (BFS) algorithm.

Algorithm 1: General schema of BFS algorithms

```
   input  : M = ⟨S, s0, SG, A, φ⟩
   output: Plan π
 1 open = {s0};
 2 while open ≠ ∅ do
 3     n ← pop(open);
 4     if is_goal( n ) then
 5         π ← [ ];
 6         while n ≠ s0 do
 7             n ← parent(n);
 8             π ← π ∘ n;
 9         return π;
10     else
11         for each a ∈ A do
12             open ← next(n, a) ;
13 return fail;
```

The data structures are initialized by inserting the initial state s_0 in the open list, at line 1. The nodes in `open` are then expanded, meaning that their successors are added to the list (line 12), following the heuristic order, and while the goal is not reached (line 2). When a goal is encountered, the plan is built backward up to the initial state as shown in line 8.

This algorithm stops when the first valid plan is encountered, without searching for optimality. An optimal solution, even in the case of an almost perfect heuristic, would require an exponential number of node expansions with the algorithms of the family of A* that are used currently (Helmert et al., 2008).

The classical planning model is based on some strong assumptions, namely: a finite state space, deterministic transitions between states, caused by the agent's actions, and full information about the initial state. Thus, the only means of changing the environment are the actions triggered by the planning agent.

Such assumptions on the behavior of the environment limit the applicability scope of classical planning. However, even if limited, the good results of classical planning techniques have cleared the path to approaches that allow several relaxations for more general classes of models that extend both the expressive power and the solving capacity of automated planning systems.

2.2 Non-deterministic planning, incomplete information, and partial observability

A research agenda has emerged in the last 20 years focused on developing solvers to a range of intractable models. The planning models that address to more general classes of models relax several aspects of classical planning. For instance, relaxing the determinism in the action effects brings to non-deterministic models, that permit to describe tasks with exogenous events and unexpected action's outcomes.

Directly solving the planning model involving non-deterministic actions' effects

is a challenging and difficult problem, that is generally tackled by techniques that use classical planners to find a solution plan, and eventually, replan from a newly generated problem (Kuter et al., 2008). In fact, even if it is generally considered as a separate characteristic of the planning task, non-determinism is usually tackled by regarding it as uncertainty on the initial state. It can indeed be proven that those two features (non-deterministic effects, and incomplete information on the initial situation) are equivalent, as non-deterministic effects can be eliminated by using hidden artificial conditions on the initial state that must be introduced afresh each time a non-deterministic action is applied (Albore et al., 2010).

Considering uncertainty about the current state of the planning agent extends the classical planning paradigm toward two powerful models for decision-making under uncertainty: *conformant* and *contingent* planning. These two generalizations relax the conditions of classical planning on the initial situation, considering problems where the current state is uncertain. On top of this uncertainty, the contingent planning paradigm relaxes further the classical planning model by considering that the current state is only partially observable. It implies then to deal with *belief states*. Formally, in this context, a *belief state* can be defined by a set of states deemed possible in a given time step. The formal model of contingent planning can be described as follows.

Definition 2.2 (Contingent Planning) *The deterministic contingent planning model* $\mathcal{M}$ *is a tuple* $\langle S, S_0, S_G, A, O, \varphi, \omega \rangle$*, where:*

- S *is a finite set of states,*
- $S_0 \subseteq S$ *is the initial non empty belief state,*
- $S_G \subseteq S$ *is the set of goal states,*
- A *is the set of actions,*
- O *is the set of sensing actions (the observations), s.t.* $O \subseteq S$*,*
- $\varphi : S \times A \to S$ *is a state* transition *function that maps states to states s.t.* $(s, a) \mapsto \varphi(s, a) = s'$.
  The successor state $s' = \varphi(s, a)$ *results from applying the executable action* a *to a given state* s
- $\omega : S \times O \to S$ *is an observation function that associates each state to itself if the state is observable, the empty set otherwise: s.t.* $(s, o) \mapsto \omega(s, o) = \{\emptyset, s\}$.

We will indicate with $b' = \varphi(b, a)$ *the belief state* b' *resulting from applying the action* a *to the belief state* b*. Applying an observation* o *on a belief* b *results in the set of states compatible with observation coming from* o *and is denoted* $b^o = b \cap \{\omega(s, o) \,|\, s \in b\}$.

Conformant planning is a particular case of contingent planning, where no disambiguation can be done on the belief state as the model considers null observability. In a conformant planning problem the set of observations O is empty, and consequently the observation function $o : S \to O$ never associates an observation to a state. A conformant problem can be thus defined by the tuple $\langle S, S_0, S_G, A, \varphi \rangle$.

Conformant planning with deterministic actions is one of the simplest form of planning with uncertainty. A deterministic conformant problem is like a classical

problem but with many possible initial states instead of one, and a plan is conformant when it is a valid plan for each possible initial state. In spite of its simplicity, the conformant planning problem is harder than classical planning, as plan verification remains hard even under polynomial restrictions on plan length (Haslum and Jonsson, 1999; Baral et al., 2000; Rintanen, 2004; Turner, 2002). Few practical problems are purely conformant, but the ability to find conformant plans is needed in planning with sensing, of which conformant planning is a special case where no sensing is allowed. Indeed, relaxations of planning with sensing into conformant cases lies at the heart of recent methods for computing contingent plans (Hoffmann and Brafman, 2005; Albore et al., 2009) and deriving finite-state controllers (Bonet et al., 2009).

The plan can be interpreted as an automaton, whose execution controls the system by synchronously reading the output of the system (sensing) and providing step by step the input for the system (the planned actions).

The idea of performing sensing is strongly bound to solving contingent planning problems, as it provides information that consequently reduces the size of the belief state: after applying a sensing action, only the states compatible with what is observed are possible. Such sensing actions can be considered as part of the control system, when an *offline* plan is generated for contingent problems; the plan is tree-shaped[1], branching on every possible sensing result. But, in dynamic tasks, an *online* approach is often preferred, where observations are gathered at execution time and new actions are synthesized consequently by the planner.

2.2.1 General algorithmic schema for non-deterministic planning.

Planning under uncertainty as described in the previous sections, can be formulated as a path-finding problem in belief space. Computational challenges faced in this formulation are the derivation of heuristics to guide the search, and belief representation and update (Bonet and Geffner, 2000). This formulation is the basis of the most recent conformant planners such as Conformant-FF (Brafman and Hoffmann, 2004), MBP (Bertoli et al., 2006), POND (Bryce et al., 2006), CNF (To et al., 2010), DNF (To et al., 2011), and T1 (Albore et al., 2011). The exception is the planner T0 which is based on a translation of conformant problems P in classical problems $K(P)$ that are solved by off-the-shelf classical planners (Palacios and Geffner, 2009).

The approach to planning with incomplete information about the agent's current state is however solvable by heuristic search in the belief states space (Bonet and Geffner, 2000), thus considering sets of states instead of single states during the search. The general schema for the search algorithm follows Alg. 1 above, and is particularly true in the case of conformant planning, which is a deterministic planning model in the belief space. Thus, at line 1, the `open` list is initialized with the initial situation, that is a set of states in the case of conformant planning.

When sensors are part of the planning problem, the outcome of observing the truth value of some state variable affects the belief state and thus the search. Contingent planning can then be solved as a non-deterministic search in the belief space, considering a noiseless sensor model (noise in the observations are considered when discussing the POMDPs in Sect. 2.3). The difference with previous models is that a *strong solution* for a non-deterministic problem exists only if the goal is reachable

[1] A graph can be considered instead, when repeated states are taken into account.

regardless the possible observations that can occur during the execution path. This implies that branching is introduced during the search not only when a deterministic choice occurs (like in previous planning models), but when non-deterministic outcomes are dictated by the observations. We will thus differentiate the search between what are called the *And nodes* (the non-deterministic choices) and the *Or nodes* (the deterministic choices) in Alg. 2. This representation yields a solution shaped as a "decision tree" rather than a sequence of actions.

Algorithm 2: General schema of AND-OR search algorithm

```
   input  : M = ⟨S, S_0, S_G, A, O, φ, ω⟩
   output: Plan π
1  return OR_search(S_0, [ ])
1  Procedure OR_search(n,π)
2      if is_goal( n ) then
3          return [ ];
4      foreach a ∈ A do
5          n' ← next(n, a);
6          π ← AND_search(n', [ ]);
7          if π ≠ fail then
8              return π ∘ a;
9      return fail;
1  Procedure AND_search(n,π)
2      foreach o_i ∈ O do
3          n_i ← next(n, o_i);
4          π_i ← OR_search(n_i, [ ]);
5          if π = fail then
6              return fail;
7      return [if o_i then π_i]_i ;
```

In Alg. 2, the search is then carried on differently depending if an observation node or an action node are expanded, and starts from calling `OR_search` on the initial belief state (line 1), which represents all the states that are deemed possible initially. Procedure `OR_search` acts very closely to Alg. 1, expanding nodes following an order possibly determined by an heuristics. The main difference is in the `AND_search` procedure, that considers the solutions from *all the possible observation* o_i, concatenating them in a decision tree, where selecting the branch to execute depends on the sensing (line 7).

Note that observations can be passive, meaning that they are applied directly to the belief state during action execution; in such cases, the function `next` yields a belief state resulting from both (active) acting and sensing.

2.3 Planning under probabilistic uncertainties

The models of planning under uncertainty we saw in the former section can be generalized by considering probabilistic effects, where the problem changes from finding a solution that drives the environment/agent to a desired goal, into a minimization problem related to finding a path to the goal with the highest probability of success. Decision-theoretic planning (DTP) frameworks (Boutilier et al., 1999; Blythe, 1999) deal with the problems of planning with incomplete information about the environ-

ment by adopting, as underlying models, the stochastic planning models MDPs and POMDPs to consider the expected information or reward that could be gained by selecting an action rather than another. DTP is successfully applied in the field of robotics, providing, among the other advantages, a common framework that take advantage of utility functions to reason about localisation quality and navigation cost (Carrillo et al., 2015; Makarenko et al., 2002).

The uncertainty about the outcomes of the actions can be modeled as a probability distribution, adding to the non-deterministic model discussed above a probability to each possible effect of an action. Markov Decision Processes (MDPs, Puterman, 1994) –in the fully observable case– and Partially Observable MDPs (Kaelbling et al., 1998) when probabilistic uncertainty affects the sensing results (observations) as well, are then similar in spirit to non-deterministic planning in the sense that each action can have different outcomes, but they differ from contingent planning in several important aspects.

The probability attached to each action effect implies that the goals are usually represented as utility functions, i.e. numeric functions that can express preferences over the executed actions and the desired final states, instead of a set of goal states. Thus, to a state and an action are associated a probability of transition to a state, a possible reward, and, in the partially observable case, an observation probability. In such frameworks, a plan is expressed as a *policy* π that provides the action to execute in each state, s.t. $\pi : S \to A$, with the planning problem reduced to an optimization problem where a solution is a (optimal) policy maximizing the expected utility. This implies also that the planning algorithms for solving the probabilistic problems differ substantially from the search algorithms in a graph seen above.

Definition 2.3 ((Partially Observable) Markov Decision Process) *The (Partially Observable) Markov decision process model $\mathcal{M}$ is a tuple $\mathcal{M} = \langle S, A, O, \Phi, R, \Omega, \gamma, b_0 \rangle$ where:*

- *S is a finite set of states,*
- *A is the set actions,*
- *O is the set of observations*
- *$\Phi : S \times A \times S \to [0, 1]$ is the transition function where $(s, a, s') \mapsto \Phi(s, a) = Pr(s' \mid a, s)$ is a state* probabilistic transition *function that maps a transition from state to state in a transition probability, in which the successor state s' results from applying the executable action a to a given state s.*
- *$R : S \times A \times S \to \mathbb{R}$ is the* reward function *assigning a reward $R(s, a, s')$ to the outcome of the transition $\Phi(s, a, s')$;*
- *Ω is the* probabilistic observation function *$\Omega : S \times A \times O \to [0, 1]$ associating a probability $\Omega(s', a, o) = Pr(o \mid a, s')$ to an observation in a state; in the fully observable case, $\Omega(s', a, o) = \mathbf{1}_{\{s'=o\}}$,*
- *b_0 is the initial probability distribution over states,*
- *$\gamma \in [0, 1]$ is the discount factor.*

For convenience, we define the *application* function $app : A \to 2^S$, such that, for all $a \in A$, $app(a)$ is the set of states where action a is applicable. Besides, the

successor function $succ : S \times A \rightarrow 2^S$ defines, for all $s \in S$ and $a \in A$, the set of states $succ(s, a)$ that are directly reachable in one step by applying action a in state s. Please note that it holds only for the fully observable case; for the partial observable case an extension of the POMDP model has been proposed by Carvalho Chanel and Teichteil-Königsbuch (2013).

When planning for MDPs, the state of the system is always observable, but when partial observability is considered, in a time step, the state space is given in terms of *belief states*. In this case, belief states b are probability distributions over all possible states s such that $\sum_s b(s) = 1$, where $b(s)$ is the probability assigned to the particular state s in the belief b. In this context, at each time step t, after the application of an action $a \in A$ in a *belief state*, defined as an element $b_t \in \Delta$, and an observation $o \in O$ perceived, the agent updates its *belief state* using the Bayes' rule (Smallwood and Sondik, 1973). A policy is then a function that maps belief states to actions, s.t. $\pi : \Delta \rightarrow A$, where Δ is the belief state space.

2.3.1 General algorithmic schema of (PO)MDP solving.

Solving (PO)MDPs consists in computing an action *policy* that optimizes some numeric criterion V, named *value function*. In general, for a given policy π, this criterion is defined as the discounted expected sum of stochastic rewards gathered when successively applying $\pi(s)$ (or $\pi(b)$) from the beginning of the decision process over an infinite horizon.

In the MDP case, the execution of the policy corresponds to a Markov Chain, an infinite sequence of states where each state depends on the previous one, as states are completely observable and then $\pi : S \rightarrow A$ is directly applicable over the state space. As each state has a different probability to be visited during the execution, the expected reward for being in a particular state s, following some fixed policy π in the infinite horizon case, is given by the equation below:

$$V_\pi(s) = R\big(s, \pi(s)\big) + \gamma \sum_{s'} Pr\big(s'|s, \pi(s)\big) V_\pi(s') \tag{1}$$

where γ is the discount factor such that $\gamma \leq 1$. Equation (1) describes the expected reward $V_\pi(s)$ for starting in a state s and applying the actions in some policy π, considering the immediate reward $R(s, \pi(s)) = \sum_{s'} Pr(s'|s, \pi(s)) \cdot R(s, \pi(s), s')$ for applying action $\pi(s)$ in s.

Like contingent planning, POMDPs model planning problems when the assumption of full (or null) observability does not hold. This implies that an optimal policy will not depend on the current state only, but as well on the information available up to that point from the sequence of past observations. In the POMDP case, states are not directly observable, so that the policy is not applicable over the hidden states of the system. Instead, the policy is applied over the history of actions performed and of observations gathered from the beginning of the decision process, which allows the planner to compute a probability distribution over the possible current states of the system at each time step, thanks to successive applications of Bayes' rule. In other words, if an action is executed and an observation is gathered, the agent can use Bayes' rule to update its belief state. Note that the new belief state only depends on the action, on the observation, and on the previous belief state (the belief state is Markovian). Such modelling allows a POMDP to be formulated as a MDP where every belief is represented as a state, and state transitions probabilities are products

of actions and observations, resulting in a *belief MDP* whose value function is given in terms of belief states (Kaelbling et al., 1998).

Some algorithms optimize π over all the possible initial (belief) states of the world, like linear programming, value iteration, policy iteration (Puterman, 1994; Hoey et al., 1999), exact value iteration (Sondik, 1978) or witness (Kaelbling et al., 1998). Some others use the knowledge of the system's initial (belief) state, and sometimes of goal states to reach, in order to compute a partial policy that is only defined over a subset of (belief) states reachable from the initial (belief) state when successively applying all possible actions from the initial (belief) state. Examples of such algorithms are `LAO`* (Hansen and Zilberstein, 2001), `(L)RTDP` (Bonet and Geffner, 2003), `RFF` (Teichteil-Königsbuch et al., 2010), `HAO`* (Meuleau et al., 2009), `PBVI` (Pineau et al., 2006; Spaan and Vlassis, 2005), `HSVI` (Smith and Simmons, 2004), `SARSOP` (Kurniawati et al., 2008), offline and online `AEMS` (Ross and Chaib-Draa, 2007), `RTDP-Bel` (Bonet and Geffner, 2009).

To the best of our knowledge, most recent (PO)MDP algorithms, at least heuristic ones, follow the algorithmic schema depicted in Alg. 3.

Algorithm 3: General schema of (PO)MDP algorithms

input : (PO)MDP $\mathcal{M}$, $0 < \gamma \leqslant 1$, $\epsilon > 0$, $N \in \mathbb{N}^*$, initial (belief) state I
output: Value function V and policy π

```
1 initialize_bounds(I,V) ;
2 while stop criterion ε not reached do
3 |  explore(I,ε,0) ;
4 computing π from V;
5 return (V,π) ;
1 Procedure explore(I,ε,t)
2 |  if V has converged for the (belief) state I then
3 |  |  return;
4 |  else
5 |  |  choose an action a* (and an observation o*) according to the exploration
  |  |  heuristics ;
6 |  |  computing next (belief) state I';
7 |  |  call explore(I',ε,t+1) ;
8 |  |  update bounds, value function V and policy π for the (belief) state I);
```

They first initialize data structures and store the set of possible initial states (MDPs) or the initial belief state (POMDPs) and initialize bounds usually used to define the heuristic of (belief) state exploration as shown in line 1. Then, they improve the value function (and the policy) performing an exploration considering future steps (see line 3). This function implements trials choosing actions and observations, randomly or based on the heuristic function (bounds), and calling recursively this same function (lines 5 to 7) until the value of the considered (belief) state has converged possibly using the discount factor (line 2). In the end of a trial, these algorithms update the policy for the set of explored states (heuristic search MDP algorithms) or belief states (point-based and heuristic search POMDP algorithms) by backtracking the value of these explored states (line 8). The main iteration is performed until a convergence test becomes true (line 2), for instance when the difference between two successive value functions is ϵ-bounded, or when the difference of the value of the bounds of V is less than ϵ for the initial (belief) state, or when two successive policies are equal, or when the policy becomes closed, or when a

maximum number of iterations N is reached. Some sampling-based algorithms like `RTDP` or `RTDP-Bel` may eventually converge after an infinite number of iterations. Finally, the optimized value function is returned and the corresponding policy (see line 5) is extracted from this value function. Furthermore, in some cases, the policy is computed for the first time during this final step, once the value function has converged. An optimal policy can be of course obtained for POMDPs by pushing the convergence until a fix point; but, as in the case of classical planning, computation time could exponentially rise in that case.

2.4 General schema

Most recent planning algorithms can be rewritten into a more general algorithmic schema shown in Alg. 4. Please note that Alg. 1, 2 and 3 have the same main steps: *initialization* (line 1 in Alg. 1, line 1 in Alg. 2 and line 1 in Alg. 3), *policy improvement* until some criterion is reached (lines 2-12 in Alg. 1, lines 2-3 in Alg. 3, and procedures `OR_search` and `AND_search` in Alg. 2), and *returning* the result (line 8 in Alg. 1, line 5 in Alg. 3, and lines 3, 8 and 7 in Alg. 2). In this sense, the purpose of Alg. 4 is to generalize in a unique algorithm these common functions of planning algorithms: `solve_initialize` for the initialization, `solve_progress` for the policy improvement, and `solve_end` when returning the resulted policy. In the sequel of this paper, we will refer to Alg. 4 as the general algorithm schema for solving planning problems presented before. Without loss of generality, this algorithm returns the policy for a planning problem given an initial set of (belief) states.

Algorithm 4: General schema of planning algorithms

input : Planning problem $\mathcal{P}$, parameters α_p, stop criterion ϵ, states or belief state I
output: policy π

```
1 solve_initialize(P,I,α_p);
2 repeat solve_progress(P,I,α_p);        // graph exploration, update V or π
3 until solve_converged(ε) ;
4 solve_end();                                  // compute π if not done yet
5 return π;
```

2.5 Towards planning for robotic applications

As discussed in Sect. 1, robotic problems are challenging for decision making algorithms. They involve uncertainty in the environment changes, in the effects of the actions, in the observations outcomes. Some of these aspects can be dealt with by using some automated planning algorithms presented in the previous sections (e.g., conformant planning, POMDP, ...). Nevertheless, an important requirement for efficient decision making in robotics is the reactiveness to environment changes (from the point of view of the robot, including observing new parts of the environment). When tackling real critical applications, the autonomous system is required to provide some basic guarantees to ensure the safety of the mission. For instance, in case of UAVs, operators require that the executed policy never puts the robot in danger, which may happen in many situations like being out of battery or fuel. More generally, the absence of an action to execute in a given situation can cause system failures, or an unbearable waste of mission time.

To tackle the problem of reactive planning and execution, decision making al-

gorithms have been improved to provide anytime solutions. An anytime solution guarantees to provide an executable action for any time threshold specified. Such *anytime* algorithms generally improve their first solution using as much results from former computation efforts as possible (Likhachev et al., 2008), and they are effective when the environment does not change much between two plan computations (Stentz, 1995; Koenig and Likhachev, 2001). Anytime algorithms *quickly* build an applicable suboptimal policy, and then refine the solution, improving it if more planning time is available. A particular case of anytime planning algorithms consists in the online planning algorithms which aim at providing as fast as possible a feasible policy in the current execution state.

Different anytime algorithms were proposed for the different planning models presented before (Smith and Simmons, 2004; Likhachev et al., 2008; Bonet and Geffner, 2009; Richter et al., 2010; Korf, 1990; Cannon et al., 2012; Barto et al., 1995; Keller and Eyerich, 2012; Ross et al., 2008; Ross and Chaib-Draa, 2007). These so-called real-time or anytime planning approaches implement the same steps of the general schema of planning algorithms presented before (Alg. 4). They control, for example, how many policy improvements (trials) they will perform, like in `(L)RTDP` (Bonet and Geffner, 2003), in Point-Based Value Iteration (`PBVI`) algorithm for POMDP planning (Pineau et al., 2006; Spaan and Vlassis, 2005), or in `SARSOP` (Kurniawati et al., 2008).

In a more classical *plan-replan* approach, even using an anytime algorithm for replanning, there is no guarantee that the replanning episode will end before the given deadline is reached. Furthermore, the replanning requests are usually performed when a planned action is not applicable in the current state, this is done regardless of the time still available for planning, or of the information available to quickly update or optimize the current plan. Consequently, when the execution controller queries for an action in the current state at a *precise time point*, the previously calculated policy may be broken and not applicable, even when using online algorithms. Such a circumstance is not acceptable for time-constrained robotic missions, specially in UAV applications, as battery life is precious and execution has to be continuous.

We claim that these anytime algorithms are not *strictly anytime* in the sense that the execution of their continually improved policy is not guaranteed to succeed under strict time constraints. While the prediction of the environment evolution is properly dealt with by most planning approaches, very few of them take into account their own computational time as a cost to optimize along with standard action costs. These algorithms only aim at rapidly producing a first feasible plan from the current state that is then continuously improved, yet without controlling the time required to generate the first solution with regards to the time point where the current state will change and the plan will not be valid anymore. For instance, when improving a policy in the probabilistic case, *no time threshold is given*, except for `AEMS` (Ross and Chaib-Draa, 2007) which suggests to improve the value function until a time deadline is reached: $t_{elapsed} > t_{deadline}$, but there is no guarantee that the new launched optimization loop will terminate when the deadline is reached. Guaranteeing to update the plan as quickly as possible in order to minimize the unavailability time of the planning system, is clearly unsatisfactory for robotics missions where time is critical and actions have to be immediately available when queried by the execution engine. To overcome these issues, fine control of the planning process and its interaction with the possible future execution states of the robot are required.

Actually, the approach described below recalls goal reasoning and other techniques that manage goals ordering which have been used in planning in order to

increase agents efficiency and autonomy (Vattam et al., 2013). Sequencing the goals of a problem to make easy the search of an overall solution, sometimes reactively, has been used in past approaches: a form of goal reasoning is present in almost all the recent planning agents, from the FF's "goal agenda" (Hoffmann and Nebel, 2001) to other systems like goal managment (De Giacomo et al., 2016), goal motivation (Munoz-Avila et al., 2015), learning goal priorities (Young and Hawes, 2012), using heuristics for goal formulation (Wilson et al., 2013), and operational goal semantics (Harland et al., 2014). ActorSim (Roberts et al., 2016) is a development environment to simulate goal reasoning, and it provides the semantics to model both hierarchical task and goal planning in a single framework. ActorSim relates to our work with `AMPLE` because it addresses the same concern about providing a single framework for planning and executing tasks while reasoning about goal selection strategies, even if ActorSim does not develop the (actual) anytime planning aspect, even if it can theoretical rely on existing anytime planners; instead, it relies on a refinment of HTN (Hierarchical Task Network) planning that integrates goal planning. Like ActorSim, other platforms deal with the need of integrating in the same framework planning tasks, monitoring, and execution. The goals analysis and selection is so central in complex systems implementations that frameworks as SOTA (State Of Affairs) (Abeywickrama and Zambonelli, 2012) have been developped to permit an early and goal-oriented analysis of the systems in order to adapt the design to the identified requirements in an automated manner. SOTA uses model-checking to identify off-line, thus differently from `AMPLE` which is a reactive platform, any incompleteness of the goal-oriented requirements model, by checking for single goal or utility to detect deadlocks or any inconsistency.

In the more general framework of planning, recent theoretical advances have been made to intimately reason about both the evolution of the environment and the computational time of the planning algorithm (Burns et al., 2013; Lin et al., 2015). Such systems are required to perform some kind of meta-reasoning, meaning that the planners should reason about the performance of their planning algorithms, yet under certain limits since they still cannot guarantee to produce valid executable actions exactly when the execution engine requires them.

On the same line of these approaches, `AMPLE` aims at generalising the unified framework for execution-driven planning in robotics, steering and abstracting existing planning algorithms – if possible being anytime, for a better responsiveness of deliberative solutions – in such a way that valid actions are always available when queried at execution-time. In the next section, we present `AMPLE`, an anytime planning *and execution* framework which precisely manages planning requests and the planning process in such a way that the execution engine can query for feasible actions at any time.

3 `AMPLE`: Anytime planning, conditional planning, and parallel policy optimization and execution

`AMPLE` (Anytime Meta PLannEr), is a generic planning framework dedicated to robotic embedded architectures. The proposed paradigm can be seen as an *anytime generic planner*. `AMPLE` was designed to be *strictly anytime* in the sense of policy execution under time constraints. It is composed of an execution thread and a planning thread, which are a complete rewriting of routines for solving the planning models described in Sect. 2 that conform to the general algorithmic schema

depicted in Alg. 4. The generic planning thread, which drives any dedicated planner, is designed to fulfill high-level requirements hereafter discussed:

- strictly *anytime* in the sense of policy execution under time constraints: the planning thread ensures to return an applicable action in any possible execution state at a *precise* time point, exactly when required by the execution thread;
- *reactive* to environment changes dictated through other components' requests. It can be achieved by a parallel management of policy execution and planning requests;
- *conditional*, by prioritizing future execution states and computing a partial (incomplete) but applicable policy for each of them.

3.1 The AMPLE architecture

The AMPLE framework is a multi-thread architecture as depicted in Fig. 1 where the main *execution thread* requests optimization chunks from a *planning thread* by using some meta logics described thereafter. The *planning* thread can be compared to a *server thread*; it manages the plan optimization while answering to client requests. The *execution* thread is a *client thread* which sends requests to the planning thread in order to build and execute a policy according to the system's and environment's evolutions.

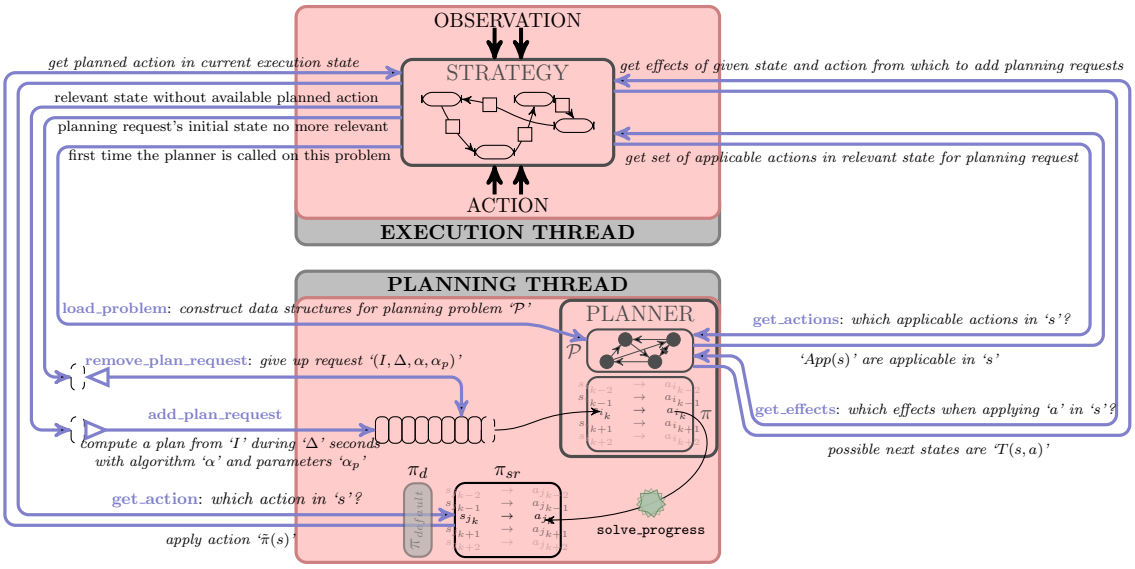


Figure 1: AMPLE architecture: connections between the execution thread and the planning thread

The execution thread's strategy[2] conforms to a state machine whose concrete instantiation can lead to various *"planning-while-executing"* logics as described in further subsections. The planning thread contains an instance of the planning algorithm, called *Planner*. Most of the requests that can be addressed to the planning

[2]in this paper, we call *strategy* the way the execution thread manages planning requests and actions execution; the optimized action plan given a planning request is called *policy*; note that these two terms are sometimes used as synonyms in the planning under uncertainty community.

thread from the execution thread concern *planning requests*, the others corresponding to getting information about the planning problem being solved.

3.2 Planning Requests

Planning requests are sent by the execution thread to the planning thread. When received, the latter must compute and update the current policy depending on the information included in the request. We formalize planning requests as follows:

Definition 3.1 (Planning Request) *A planning request $\mathcal{R}$ is a tuple $\langle I, \Delta, \alpha, \alpha_p \rangle$ with:*

- *I is a (belief) state from which the policy must be updated ;*
- *Δ is the (continuous) maximum duration of the policy update ;*
- *α is an algorithmic variant of the planner; note that some planners can only provide one variant;*
- *α_p are the parameters of α; note that α_p is general enough to take into account any parameter required to handle the request.*

State I is not necessarily reachable from the current execution state of the system; this opens the execution engine to the possibility of requesting plans from different initial states, anticipating future requests or dynamically reacting to unexpected situations.

The next paragraphs dig deeper in the planning and execution threads to understand how the items described above are designed and implemented.

3.3 `AMPLE` initialization

`AMPLE`'s main routine is depicted in Proc. 1 and explained in the next, where each procedure or algorithm presented relies on parts of the following data structures and notations. Please note that ***bold italic data*** are shared between the execution and planning threads. We define:

- $\mathcal{P}$, the planning problem;
- *psm*, the state machine that formalizes the interaction between the execution and planning threads of `AMPLE` (detailed in Sect. 3.5);
- *pln*, the planner driven by `AMPLE` in the planning thread;
- π_d, the default policy generated before the execution;
- ***pr***, list of planning requests managed by the execution thread and solved by the planning thread.
- $\boldsymbol{\pi_{sr}}$, the backup policy for solved requests;
- ***stopCurrentRequest***, a boolean indicating whether the current request being solved in the planning thread must be interrupted;
- ***stopPlannerRequested***, a boolean indicating whether `AMPLE` must be stopped (for instance when the mission is finished).

Procedure 1: AMPLE_main

input: $\mathcal{P}$, pln

1 Create empty list of planning requests: $\boldsymbol{pr}$;
2 Create empty backup policy for solved requests: $\boldsymbol{\pi_{sr}}$;
3 $\boldsymbol{stopCurrentRequest} \leftarrow$ false;
4 $\boldsymbol{stopPlannerRequested} \leftarrow$ false;
5 $psm \leftarrow$ `load_AMPLE_state_machine`();
6 $\pi_d \leftarrow$ `load_default_policy`($\mathcal{M}$);
7 `launch_thread`(`AMPLE_planning`($\mathcal{P}$, pln, psm, $\boldsymbol{pr}$, $\boldsymbol{\pi_{sr}}$, $\boldsymbol{stopCurrentRequest}$, $\boldsymbol{stopPlannerRequested}$));
8 `launch_thread`(`AMPLE_execute`($\mathcal{P}$, psm, π_d, $\boldsymbol{pr}$, $\boldsymbol{\pi_{sr}}$, $\boldsymbol{stopCurrentRequest}$, $\boldsymbol{stopPlannerRequested}$));

After creating an empty list of planning requests, an empty backup policy for solved requests and initializing stopping condition booleans (lines 1 to 4) , `AMPLE` loads the state machine (line 5) that defines the states and interactions of the planning thread with the system's execution engine. Next, `AMPLE` loads the default policy, required to guarantee reactivity (line 6).

It then launches two concurrent threads: the planning thread, where queued planning requests are solved, and the execution thread, that interacts with the system's execution engine and queues planning requests (lines 7 and 8).

3.4 `AMPLE` execution thread

The execution thread adds and removes planning requests according to the current execution state and to the future evolutions of the system, and gets the action to execute in the current state from the current updated policy, or from the default one. When interacting with the planning thread, the execution thread can use several methods:

- `load_problem` loads an initial problem in the planner;
- `add_plan_request` adds a request $\mathcal{R}$ to the queue of pending planning requests $\boldsymbol{pr}$;
- `remove_plan_request` directly removes a request $\mathcal{R}$ if it is not being solved by the planning thread, i.e. only if it is not in the front of the list of pending planning requests $\boldsymbol{pr}$; otherwise, it sets the $\boldsymbol{stopCurrentRequest}$ variable to true to inform the planning thread to stop solving and to remove this request;
- `get_action` reads the optimized action to be executed in the current state if it is included in the (backup) policy $\boldsymbol{\pi_{sr}}$, otherwise an action defined by the default policy π_d is read.

Other methods allow to make queries on the planning model itself:

- `get_effects` returns the set of states that can be reached by applying action a in state s;
- `get_actions` returns the set of actions that are applicable in state s.

Managing planning requests can follow different strategies. Some of them are described later (see Sect. 3.6).

3.5 AMPLE planning thread

The planning thread is a complete reorganization of standard planning main steps depicted in Alg. 4, in order to automatically manage the queue of planning requests and to locally update the policy in bounded time for each planning request. The revisited planning algorithm is presented in Alg. 5.

Algorithm 5: AMPLE_planning

```
1  solvingRequest ← false;
2  while true do
3      if psm.state = LOADING_PROBLEM then
4          pln.load_problem(P);
5          psm.state ← PROBLEM_LOADED;
6      else if psm.state = PLANNING then
7          if solvingRequest = false then
8              launch_front_request();
9          else
10             t ← get current CPU time ;
11             if pln.solve_converged(pr.front.α_p)
                   or t − requestStartTime > pr.front.Δ
                   or stopCurrentRequest = true
                   or stopPlannerRequested = true then
12                 pln.solve_end();
13                 π_sr ← π_sr ∪ subpln.policy(pr.front.I);
14                 pr.pop_front();
15                 stopCurrentRequest ← false;
16                 if pr is not empty then
17                     launch_front_request();
18                 else
19                     psm.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 CPU time ;
28 pln.solve_initialize(pr.front.I);
```

The algorithm is conceived as an endless loop that looks at, and accordingly reacts to, the current mode of the state machine defined in Fig. 2.

If the mode is LOADING_PROBLEM (further to an activation of the load_problem command in the execution thread), it loads the planning problem $\mathcal{P}$ and changes the mode to PROBLEM_LOADED (lines 3 to 5). If the mode is PLANNING (further to an add_plan_request command in the execution thread), it first tests if the procedure solving the current planning request must end (lines 7 then 11), which can happen if the sub-planner procedure has converged or finished, or if the time allocated to solving the current request has been consumed, or if the solving of the current request must end (further to an activation of the remove_plan_request in the execution thread), or if the AMPLE planner has to be stopped.

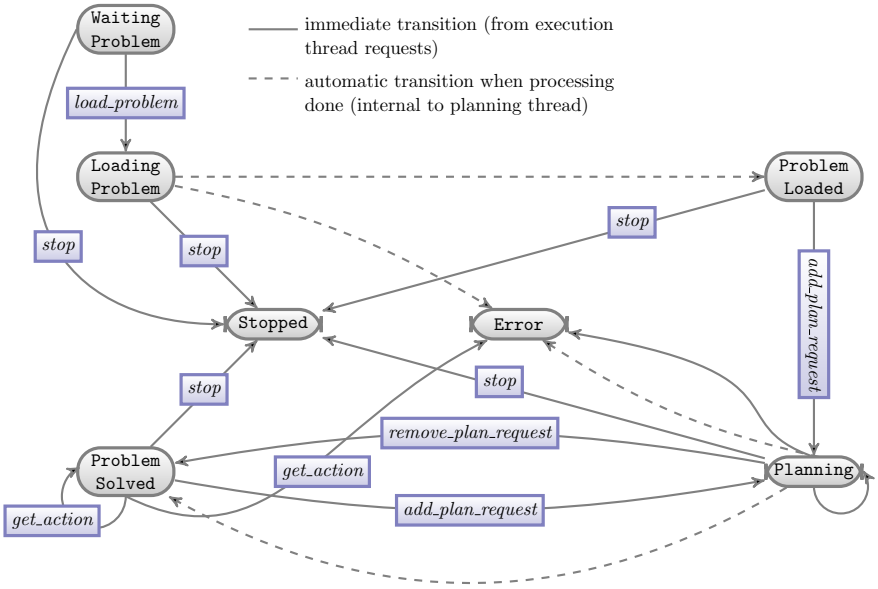


Figure 2: State machine of the `AMPLE` planning thread

Otherwise, it launches the optimization of the front request in the queue (lines 25 to 28), which mainly consists in recording the current CPU time and calling the `solve_initialize` procedure of the sub-planner. Lines 22 to 23 consist in optimizing the current planning request by calling the `solve_progress` procedure of the sub-planner and then updating the backup policy $\boldsymbol{\pi_{sr}}$ for the (belief) state queried by the current planning request.

Note that, the backup policy $\boldsymbol{\pi_{sr}}$, which is, roughly speaking, a copy of the policy π being optimized, needs to be updated during the optimization process (after `solve_progess` procedure, or for solved requests – see Alg. 5 and Fig. 1). This backup policy is the one used to answer the `get_action` requests of the execution thread. Note also that the default policy π_d is different from the backup policy. The default policy is the "rescue" policy, which can be: either a parametric off-line expert policy whose parameters are on-line adapted to the actual problem; or a heuristic policy quickly computed on-line before computing the optimal policy, or even written by hand. The default policy, contrary to the backup one, is not updated during the planning process.

3.6 Some helpful instantiations of `AMPLE`

Now, we have defined all necessary elements to design anytime planning algorithms, depending on how the `AMPLE` state machine (Fig. 2) is used. Since `AMPLE` planning thread automatically manages the queue of the planning requests to solve (see Alg. 5), we just have to take care of adding and removing planning requests in the execution thread according to the current execution state and to the future probable evolutions of the system.

First of all, we have to bootstrap the planner so that it computes a first optimized action in the initial (belief) state of the system. Many strategies seem possible, but we present a simple one in Proc. 2.

We first load the planning problem model $\mathcal{M}$ and wait for its completion by looking at the mode of the state machine (lines 1 to 2). Note that the bootstrap is

Procedure 2: bootstrap_execution

input : $\mathcal{M}$, psm, $\boldsymbol{pr}$
output: I: initial (belief) state

1 `load_problem`($\mathcal{M}$);
2 Wait until $psm.state =$ `PROBLEM_LOADED`;
3 Create initial planning request r;
4 $r.I \leftarrow$ initial (belief) state;
5 $r.\Delta \leftarrow$ choose bootstrap planning time;
6 $r.\alpha, r.\alpha_p \leftarrow$ choose solving algorithm;
7 `add_plan_request`($psm, \boldsymbol{pr}, r$);
8 Wait $r.\Delta$ amount of time;
9 **return** I;

a blocking procedure since any planning algorithm needs to be initialized. We then create a first planning request, filled with the initial (belief) state of the system, a chosen bootstrap planning time Δ, and a planning algorithm with its parameters (lines 3 to 6). Finally, we add a plan request to the queue of planning requests (lines 7 to 9), and wait an amount Δ of time (meanwhile the plan request is optimized in the planning thread).

Once the planner is bootstrapped, we can go into the "planning-while-executing" loop, for which we propose three different strategies instantiated within the execution thread (see Fig. 1): `AMPLE-NEXT`, `AMPLE-PATH` and `AMPLE-PORTFOLIO`. The first two are based on a probabilistic model of the system, allowing the decision process to either anticipate all the next execution states, which is suitable to problems with high variance on action effects, or to prioritize the most probable path, which is effective if the probability to deviate from this path is low. The last strategy, which inspires from portfolio-like planner selection (Helmert et al., 2011), has been developed for classical planning in dynamic environments where it is profitable to use various planners reasoning in concert about different symbolic levels or temporal horizons.

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

In this setting, each time the system begins to execute an action a, all the next possible (belief) states coming from this action are computed, based on the problem model, and plan requests are added for each of these (belief) states. With this strategy, the short-term evolution of the system is anticipated independently of the heuristic used in the underlying planning algorithm. Thus, the planning thread has a chance to provide an optimized action on time as soon as the current action terminates in the execution thread. The `AMPLE-NEXT` strategy is described in Alg. 6.

Once an action has completed, the next to be executed action a is gathered by calling the `get_action` command for the current execution state (line 3); it is thus executed and its expected duration Δ_a is computed[3] (line 5). 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 effect of executing a (lines 6 to 11). The system waits for action a to complete (meanwhile added planning requests are solved in the planning thread) and removes the previous planning requests in case they have not yet been solved by the planning thread (line 12). Finally, the current execution state is observed and the algorithm goes

[3]This expected duration can be computed based on return on experience from previous robotic missions, or, for instance, based on the expectation of minimum travel time.

back to the beginning of the execution loop.

Algorithm 6: `AMPLE-NEXT`

```
1  I ← bootstrap_execution(M, psm, pr);
2  while stopPlannerRequested = 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.push_back(r);
12     Wait until action a has completed;
13     for r ∈ prNext do
14         remove_plan_request(psm, pr, r,
           stopCurrentRequest);
15     I ← observe and update current (belief) state;
```

3.6.2 `AMPLE-PATH`: reasoning about the most probable evolution of the system

The previous strategy may lack of an execution-based long-term reasoning, even if the sub-planner would reasons about the long-term evolution of the system while optimizing planning requests. The strategy presented in this paragraph rather analyzes the most probable execution path of the system, which can be 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. This strategy is formalized in Alg. 7.

The depth for analyzing the (belief) state trajectory of the most probable execution path is set (line 1), noted *pathDepth*. As in the `AMPLE-NEXT` strategy, we then bootstrap the execution (line 2) and enter the execution loop, where the action a to be applied in the current execution state is obtained, starts its execution and its expected duration Δ_a is computed (lines 4 to 6). Then (see lines 7 to 14), the `get_action` procedure is applied and the most probable (belief) state at each step is obtained, starting from the current (belief) state b up to *pathDepth*. Note that the `get_action` request will return an action with respect to the current policy. For each visited (belief) state of the explored trajectory, a plan request is added starting from this (belief) state with a maximum computation time proportional to Δ_a and to the inverse of *pathDepth*. The end of the loop (lines 15 to 18) is identical to `AMPLE-NEXT`.

3.6.3 `AMPLE-PORTFOLIO`: using several planners for solving a deterministic problem

This strategy uses the approach of solvers portfolio from (Helmert et al., 2011): we have a set of planners for solving the current problem, and we give each of them some time to solve the current problem, in order to have more chance to find a solution. Contrary to (Helmert et al., 2011), we do not train our solvers based on

Algorithm 7: `AMPLE-PATH`

1 $pathDepth \leftarrow$ choose path lookahead depth;
2 $I \leftarrow$ `bootstrap_execution`$(\mathcal{M}, psm, \boldsymbol{pr})$;
3 **while** ***stopPlannerRequested*** = false **do**
4 $\quad a \leftarrow$ `get_action`$(psm, \pi_d, \mathcal{P}, \boldsymbol{\pi_{sr}}, b)$;
5 $\quad$ Execute action a;
6 $\quad \Delta_a \leftarrow$ expected duration of action a;
7 $\quad prPath \leftarrow$ empty list of planning request pointers;
8 $\quad I' \leftarrow I$;
9 $\quad$ **for** $0 < k < pathDepth$ **do**
10 $\quad\quad I' \leftarrow \underset{I'' \in succ(I', a)}{\text{argmax}} Pr(I'' \mid \texttt{get\_action}(I'), I')$;
11 $\quad\quad r.I \leftarrow I'$;
12 $\quad\quad r.\Delta \leftarrow \frac{\Delta_a}{pathDepth}$;
13 $\quad\quad$ `add_plan_request`$(psm, \boldsymbol{pr}, r)$;
14 $\quad\quad prPath.push\_back(r)$;
15 $\quad$ Wait until action a has completed;
16 $\quad$ **for** $r \in prPath$ **do**
17 $\quad\quad$ `remove_plan_request`$(psm, \boldsymbol{pr}, r,$ ***stopCurrentRequest***$)$;
18 $\quad I \leftarrow$ observe and update current (belief) state;

prior problems. The strategy is presented in Alg. 8.

The algorithm first defines a set of planning algorithms to use (line 1), then bootstraps the optimization and enters the execution loop. Like other strategies, the action to perform is obtained according to the current state, and then executed (lines 4 to 6). As we are in a deterministic model, performing the given action can lead to only one successor state (line 8). For each available algorithm, a planning request is added, and the same computation time is allowed to all requests (lines 9 to 14). The following lines are identical to the other strategies. Some slight changes can be made to this strategy depending on the specific problem (see 4.4 for a concrete

Algorithm 8: `AMPLE-PORTFOLIO`

1 $\Gamma \leftarrow \{\alpha_k\}$ a set of algorithms;
2 $I \leftarrow$ `bootstrap_execution`$(\mathcal{M}, psm, \boldsymbol{pr})$;
3 **while** ***stopPlannerRequested*** = false **do**
4 $\quad a \leftarrow$ `get_action`$(psm, \pi_d, \mathcal{P}, \boldsymbol{\pi_{sr}}, b)$;
5 $\quad$ Execute action a;
6 $\quad \Delta_a \leftarrow$ expected duration of action a;
7 $\quad prPointers \leftarrow$ empty list of planning request pointers;
8 $\quad I' \leftarrow$ `get_effects`(I, a);
9 $\quad$ **for** $\alpha_k \in \Gamma$ **do**
10 $\quad\quad r.I \leftarrow I'$;
11 $\quad\quad r.\Delta \leftarrow \frac{\Delta_a}{\#\Gamma}$;
12 $\quad\quad r.\alpha = \alpha_k$;
13 $\quad\quad$ `add_plan_request`$(psm, \boldsymbol{pr}, r)$;
14 $\quad\quad prPointers.push\_back(r)$;
15 $\quad$ Wait until action a has completed;
16 $\quad$ **for** $r \in prPointers$ **do**
17 $\quad\quad$ `remove_plan_request`$(psm, \boldsymbol{pr}, r,$ ***stopCurrentRequest***$)$;
18 $\quad I \leftarrow$ observe and update current (belief) state;

application). In case several planners have computed an action executable in the current state, the `get_action` method (line 4) uses a fixed priority list to return the action of the most priority planner.

4 Experimental evaluation

The AMPLE architecture, with implementations of the several strategies, has been applied to real field experiments with complex missions involving autonomous planning under high environment uncertainty. The objective of the experiments reported in this section is threefold:

1. demonstrating that for the first time many different planning frameworks (deterministic/probabilistic,
   partially-observable/fully-observable) can be efficiently managed within a robotic architecture under a unique versatile planning component, the AMPLE umbrella;

2. showing that complex (PO)MDP policies can be optimized in real time onboard robots whereas most previous approaches opted for breaking the probabilistic problem down to several deterministic plan-replan problems due to the too high complexity of solving (PO)MDPs for embedded robotic computers;

3. highlighting the original ability of AMPLE to switch between several different planners on the fly when solving on-line a given robotic problem to reason about various semantic levels at the same time.

In the following subsection, we first present a benchmarking of the AMPLE-NEXT and AMPLE-PATH strategies on random MDP problems. This benchmarking shows that AMPLE-PATH is not very well suited to problems with high variance in action effects since the chance of leaving the most probable path is most often high, leading to an invalid strategy at almost every action, then using only the default policy. Then, each field experiment is described in the successive subsections, with a mission description, the way AMPLE has been instantiated, and some results obtained from these experiments. In these applications, we have only used the AMPLE-NEXT and AMPLE-PORTFOLIO execution strategies due to high uncertainty in the environment making the AMPLE-PATH strategy not adequate.

4.1 Random MDP problems

We have first evaluated AMPLE on random MDP problems using both the NEXT and PATH execution strategies. Obviously, randomness in the problems is not suitable to the PORTFOLIO strategy because the various portfolio planners should be logically chosen to be efficient at solving a specific problem. The solved problems are random probabilistic graphs composed of 10000 states solved using the LAO* optimal heuristic algorithm (Hansen and Zilberstein, 2001) via AMPLE planning requests. AMPLE results are illustrated through execution and planning timelines until success (Fig. 3). Each time slice of the execution (resp. planning) thread corresponds to the execution of one action (resp. solving of one planning request).

Figure 3(a) shows the timelines for the AMPLE-NEXT execution strategy. After a first bootstrap (where only the planning thread is active), we can notice that planning continues for a few time. Then, small planning pieces are still processed when new planning requests are sent to the planner, as it still requires the value

function to converge on next possible states (requests' initial states) that have not been totally explored. Finally, the value function quickly converges for the whole state space as shown by the evolution of the Bellman error, and we can notice that only the execution thread still goes on.

Figure 3(b) shows the timelines for the `AMPLE-PATH` execution strategy with a path depth of 1 (only the next most probable state is *requested*). Two behaviors are noticeable: first, the planning thread continues much longer than in the `NEXT` strategy; the later indeed explores a larger state space at each request, thus converges faster. Second, by only considering the most probable next state, the execution is more exposed to disturbances, i.e. to arriving in a state that has not been explored; this is observable around time 150, where the Bellman error suddenly increases, and replanning is needed, which leads to a slightly longer planning time. This phenomenon is emphasized for a path depth of 3 (Fig. 3(c)) and 5 (Fig. 3(d)), where replanning requests require a longer planning time. However, we can notice that when the execution follows the most probable path, planning converges quite quickly (e.g., no more planning piece after time 100 on Figure 3(c)).

To conclude with, the `AMPLE-NEXT` process seems to provide a more convenient behavior with respect to problem solving and mission execution in probabilistic environments. However, when the problem has a prevailing most probable path, i.e. deterministic environments or probabilistic ones but with small action effect variances, the `AMPLE-PATH` execution process may be an efficient execution framework, with a fast planning process, and online reactive repair phases when the system state leaves the most probable path. In the next experiments, which are conducted on highly uncertain environments, we then decided to not implement nor test the `AMPLE-PATH` strategy.

4.2 Autonomous emergency landing with uncertain observations

4.2.1 The mission:

The first UAV mission embedding the `AMPLE` framework consists in an autonomous emergency landing: a UAV is performing a mission (e.g., a search and rescue mission, an observation mission, or a cargo mission) when a critical disturbance occurs (e.g., one of the two engines is damaged). The UAV must then perform an autonomous emergency landing: first, the UAV scans the zone over which it is flying, builds a map of the zone, and deduces some flat landable sub-zones. Finding a zone to land is urgent, as fuel consumption may be increased by the potential engine damages, leading to an approximate landing time limited to 10 minutes. Moreover, the capacity to safely land on a zone is uncertain, as it is obtained from sensor data processing onboard (e.g., laser-based or image-based mapping). The available UAV actions are: (1) go from a sub-zone to another, (2) perform a scan of a sub-zone at a lower altitude to determine precisely its landability, (3) try to land on a given sub-zone. As both landability and action effects are uncertain, this problem is modeled as an MDP. See (Teichteil-Konigsbuch et al., 2011) for the complete MDP model.

4.2.2 `AMPLE` instantiation:

`AMPLE` is used to solve the landing problem (once the set of landable zones has been computed). To solve it, we use an `RTDP`-like algorithm (Barto et al., 1995) on an MDP with continuous variables (Meuleau et al., 2009). The MDP has two continuous state

variables, and more than twice the number of sub-zones as discrete state variables, so that the number of discrete states is exponential in the number of sub-zones. The theoretical worst-case time needed to optimize the policy with state-of-the-art MDP algorithms on a 1 Ghz processor as the one embedded in our UAV (a Yamaha RMax helicopter) is about 1.5 hours with 5 sub-zones, 4 months with 10 sub-zones, more than 700 millenniums with 20 zones; whereas the mission's duration is at most 10 minutes. Thus, time-constrained policy optimization frameworks like `AMPLE` are needed to maximize the chance of achieving the mission.

While defining and loading the problem, a default policy for `AMPLE` is computed by always selecting to explore the closest zone, whatever the probability distribution is. This default policy is used when the planning process has not yet computed a feasible action for the current state when the executor asks for one.

4.2.3 Experimental results:

Autonomous outdoor real-flight experiments have been conducted using the `AMPLE-NEXT` execution strategy on a Yamaha RMax helicopter. Real flights being quite expensive, we also extensively tested our `AMPLE` strategy with realistic simulations embedding computer vision algorithms (Echeverria et al., 2012). Figure 4 shows the map built during a real flight experiment over a small village.

Data have been collected during the flights and incorporated in additional real-time simulations, in order to compare the performance of the `AMPLE-NEXT` strategy with solving the problem offline without using `AMPLE`. Figure 5 shows the total mission time in two cases: 1) the online `AMPLE` case, where the policy is optimized during execution, and 2) an offline case that corresponds to a strategy that first computes the policy, and then executes it. Since we use `RTDP`, a heuristic algorithm, offline computation times are actually better than the previously mentioned worst-case times; yet we have a priori absolutely no guarantees to perform better than the worst-case performances. The average optimization time is the time taken by the planning thread which is the same in both cases (interleaved with execution in the online case, preceding execution in the offline case). Actually, this offline case is not used during flight, but shows the interest of the `AMPLE` execution framework: the mission time is shorter in the online case, and moreover, with a total mission time limited to 10 min, the offline process would have made the mission fail, while the online process succeeds.

Figure 6 shows the rate of default actions used in the mission. We can notice that the number of default actions is quite high in this mission ($\sim$ 60%). However, when a default action is used, it means that the planner has not yet computed any policy in the corresponding state. Then, even if the default action is sub-optimal, it is the only way to guarantee reactivity, i.e. the system does not have to wait for the planning process to complete (which would lead to a classical "plan-then-execute" framework). Nevertheless, the time spent to optimize the policy before the default one is applied is not lost: subsequent actions chosen in future states may indeed come from the optimized policy.

All the previous experiments were conducted on problems with perfect state sensing, i.e. the planner reasoned directly about the true state of the system. However, in many robotic applications, states are only partially observable which means that some internal state features required by the planner are not directly observable. The next section shows how the `AMPLE` framework performs for aerial robotics planning in partially observable environments, which are typically harder to solve than fully

observable problems.

4.3 Target detection and recognition with partially observable states

4.3.1 The mission:

In this section, we consider a robotic mission where an autonomous helicopter must detect and identify a car whose model is specific among several cars scattered in an unknown environment under real-world constraints, and has to land close to this car (Carvalho Chanel et al., 2014). Due to the partial observability of car targets in the scene, we model the mission as a POMDP with probabilistic beliefs on cars' models. The UAV can perform both high-level mission tasks and perception actions: the $go\_to(\hat{z})$ (resp. $go\_to(\hat{h})$) action, where $\hat{z}$ (resp. $\hat{h}$) represents the zone (resp. height level) to go to, the *change_view* action changes the view angle when observing a given zone, and the *land* action. In this mission we do not assume any prior number of cars in the scene, which can be in any of many zones in the environment (but no more than one car per zone). Zones are extracted once in flight by image processing, allowing for creating online the POMDP model before optimizing the policy. For this mission we consider that moving and landing actions are sufficiently precise to be considered as deterministic: the effect of going to another zone, or changing flight altitude, or landing, is always deterministic. However, the problem is still a POMDP, because observations of cars' models are probabilistic (Sabbadin et al., 2007). These observations rely on the result of image processing and on the number of car models in the database. In this mission, we consider 5 possible observations ($|O| = 5$): {*no car detected, car detected but not identified, identified as model A, identified as model B, identified as model C*}. We automatically learned from real data the observation function, whose symbols are produced by an image processing algorithm (Le Saux and Sanfourche, 2011). This learning method, which assign a prior probability to each symbol, i.e. $\hat{p}(o_i|s)$, can be performed off-line because it is independent from the zones that will be extracted during the mission. So, in order to be compliant with the POMDP model, which assumes that observations are available after each executed action, all actions of our model provide an observation of the zone in front of the helicopter. The cost associated with actions models the fuel consumption depending on the distance between zones (resp. difference between height levels) and the cost represents the time spent by the image processing algorithm. See (Carvalho Chanel et al., 2013) for the complete POMDP model.

4.3.2 `AMPLE` instantiation:

Due to the very high complexity of solving POMDPs, which prevents to optimize a full policy in flight with standard algorithms, `AMPLE` is used to solve on line the target detection and recognition problem using the `AMPLE-NEXT` instantiation (Alg. 6). Future probabilistic effects are the next possible observations which allow us to construct the future next belief states. The time allocated for each planning request is proportional to the probability of each observation considered to construct the possible next belief states ($r.\Delta \sim \hat{p}(o_i|b,a)$). `AMPLE`'s planning thread handles an `AEMS`-like online algorithm (Ross and Chaib-Draa, 2007). Alike the previous MDP-based mission where the default policy was defined as a parametrized rule, the default policy considered here is optimized using the QMDP approximation (Littman et al., 1995): although not optimal, it can be quickly computed at the beginning of the mission,

once the zones are extracted from the map and the POMDP model constructed.

4.3.3 Experimental results:

In order to analyze AMPLE-NEXT's behavior on this domain, we performed several realistic simulations on different instances of the problem, with 3 searching zones, 2 height levels and 3 target models. The mission's time limit is 3 minutes. Figure 7 highlights the benefits of AMPLE compared with the classical online approach of AEMS for different planning times (4, 3 and 2 seconds) for a given mission. The classical online approach of AEMS consists in interleaving planning and execution, i.e. it plans for the current belief state during a certain amount of time at every decision epoch, but not in advance for the future ones as in our AMPLE-NEXT approach. With the classical use of AEMS (Fig. 7(a), 7(b) and 7(c)), we can easily notice that the mission's total time increases with the time allocated to plan from the current execution state. In Fig. 7 successive red bars show that the POMDP needs to be (re-)optimized in each new execution state. On the contrary, our approach (Fig. 7(d)) continually optimizes for future possible execution states (i.e. future observations) while executing the action in the current execution state. Thus, the mission's duration is lower with our approach than with the interleaved approach (at least 30% less). In other words, in our approach the amount of saved time relies on the sum of time slices of the classical approach when the planning thread is idle. The more actions get time to be executed, the more time will be saved.

We performed some additional comparisons by running 50 software-architecture-in-the-loop (SAIL) simulations of the mission using images taken during real flights. Our SAIL simulations use the exact functional architecture and algorithms used on-board our UAV, as well as real outdoor images. We averaged the results and analyzed the total mission time and planning time, the percentage of timeouts and successes in terms of landing near the searched car. Action durations are uniformly drawn from $[T^a_{min}, T^a_{max}]$, with $T^a_{min} = 8s$ and $T^a_{max} = 10s$, which is representative of durations observed during real test flights. As expected (see Fig. 8), AMPLE continually optimizes the policy in background, contrary to the interleaved approach. As a result, it is more reactive: it has the minimum mission time, while providing the best percentage of success and the minimum number of timeouts. Note that, in Fig. 8(a), AEMS-2s performs better in averaged mission time (avg. over successful missions), but the percentage of successful missions is lower than in our approach (Fig. 8(b)). Furthermore, less than 20% of default actions were used, which shows the relevance of optimizing actions in advance for the future possible belief states that come from future possible observations.

AMPLE-NEXT was successfully tested during real flights: Fig. 9 and 10 respectively show the actual observation-action pairs obtained during the flight and the UAV's global flight trajectory.

Until now we have tested our AMPLE framework in uncertain environments whose probabilistic models of actions dynamics and observations are known. In the next set of experiments we rather assume that the underlying uncertainty model is unknown so that we use AMPLE to drive a portfolio of deterministic planners at various reasoning levels which are called to update the current plan each time the deviation of the observations with the model is too large or when new information is available and relevant to reoptimize the current plan.

4.4 Autonomous navigation in dynamic environments

4.4.1 The mission:

Navigating in dynamic environments is a complex problem for robotics. It needs path planning algorithms, able to find feasible paths while avoiding obstacles; guidance algorithms, able to follow as close as possible the computed paths; and localization and mapping algorithms able to produce an accurate model of the environment. However, highly dynamic environments, like office environments, require to regularly update these models, which may result in invalid plans or unreachable goals. The mission considered in this experiment is to reach a sequence of waypoints, given an imperfect model of the environment, which is updated on the fly as the robot moves.

4.4.2 `AMPLE` instanciation:

For our navigation problem we designed a simple path planner based on A* (Hart et al., 1968). The objective was not to define the best algorithm for this problem, but to have a representative deterministic planner to integrate into `AMPLE`. The planner reasons on an occupancy grid of the environment. A state is a 2D position of the robot, and an action is a 2D displacement along a straight line segment. The search graph is built on-the-fly, by using ray-casting to compute the actions that are feasible from a state regarding straight ahead obstacles. Planning requests then contain a starting state, the current occupancy grid of the environment and a goal state.

Our path planning algorithm has a specific *field-of-view* parameter that indicates which part of the map is taken into account while planning. We then use two variants of our path planner as two separate algorithms: one with an unlimited field-of-view (the planner uses the whole known map), and another one with a limited field-of-view. The rationale for this strategy is that limiting the field-of-view will fasten the search (more chance to find a local path before the execution thread asks for an action to execute), while the unlimited field-of-view will compute an optimal global path but may not have time to converge during the allocated time, or a global path may even not exist, e.g. because an opening is momentarily closed by a door or a person.

Figures 11(a) and 11(b) represent an environment with a 3D octomap on top of the associated projected 2D costmap. The field of view is depicted in black inside the enclosure. In Fig. 11(a) the field of view is unlimited so the costmap takes into account all observed obstacles. In Fig. 11(b) the field of view is limited to $1m$ so the costmap takes into account only the obstacles in the local neighborhood.

We have instantiated the `AMPLE-PORTFOLIO` strategy composed of a first algorithm with unlimited field-of-view, and of a second planner with a limited field-of-view set to 1.5m. The `AMPLE` default policy simply makes the robot turn in place, so that it forces the map to be updated, at least in the limited field-of-view variant, in order to maximize the chance to find a valid path to the next goal.

4.4.3 Experimental results:

We participated in the *Kinect Autonomous Navigation Contest*[4], held at the International Conference on Intelligent Robots and Systems (IROS) in Chicago, Illinois,

[4] `http://www.iros2014.org/program/kinect-robot-navigation-contest`

USA on September 18^{th}, 2014.

The environment was populated with realistic items that are known to challenge typical navigation sensors and systems, such as narrow chair and table legs and uneven surfaces. It was also featuring moving people and furniture. To perform localization and mapping, we have used an architecture made of the EVO algorithm (Sanfourche et al., 2013) and OctoMap (Hornung et al., 2013). Figure 12 represents part of the contest environment built during the contest. On the bottom left, one can guess a sofa and a table with a plant. Figure 13 represents the corresponding 2D costmap along with the robot trajectory. This costmap is the robot's representation of the environment when it arrived at the last validated waypoint (the farest bottom point of the robot's trajectory). After reaching this waypoint, our mapping and localization system failed, making the environment representation too inaccurate, so no global path could be found anymore. Even if the robot did not reach the next waypoint, thanks to the `PORTFOLIO` strategy, the robot still progressed a lot towards it by using local planning.

Thanks to `AMPLE`, we came in second place.

In order to analyse further the performance of `AMPLE` in such a mission, we set up an experiment inspired from the Robocup@Home league[5]. Figure 14 gives a schematic view of the environment with the waypoints of the mission going from 0 to 11. It represents a small apartment composed of a hall, a kitchen, a bedroom, and a living room. The dashed lines represent doors, that are alternately closed and opened during the experiment to integrate dynamic changes in the environment.

All doors (A, B, C, D) are initially open. The robot is moved from 0 to 6 in order to build a first knowledge of its environment. Then the mission is made of 5 steps, asking the robot to go:

1. from 6 to 7, meanwhile we close door D;
2. from 7 to 8, meanwhile we reopen D;
3. from 8 to 9, meanwhile we close C, then B when the robot has passed door D;
4. from 9 to 10, meanwhile we open B;
5. from 10 to 11, meanwhile we close A.

We have compared our `AMPLE-PORTFOLIO` implementation with a classical plan-replan (`PR`) architecture using the same A* implementation with the global map. In this `PR` architecture, a replanning is triggered when the current action is not feasible. The mission has been performed 5 times with each architecture.

In these experiments, the mission duration using the `AMPLE-PORTFOLIO` architecture is 5% lower (i.e. better) than the `PR` architecture.

Table 1 presents the number of replannings according to the several steps of the mission. Replannings for the `PR` architecture are triggered when the next action is no more feasible. Replannings for `AMPLE` correspond to situation where the returned action is a default action. The number of replannings is almost similar, except for step 3 (i.e. when going from 8 to 9). In this step, the two architectures have computed an initial path going trough door C, which is now closed. `PR` needs to replan once it realizes that its plan is not feasible anymore. `AMPLE` is able to find the new plan while executing the previous action, and then just follows its execution. The same behavior occurs when closing door B. It results in a clearly lower number of necessary replannings when using `AMPLE`, leading to a shorter mission time.

[5]information available on `http://www.robocupathome.org/`

Table 1: Number of replannings per mission step for each architecture.

mission step	1	2	3	4	5	Total
PR	8	5	16	0	0	29
AMPLE	5	5	1	0	0	11

5 Conclusion

We proposed a flexible algorithmic framework to allow for continuous real-time planning of complex tasks in parallel of their executions. Our framework, named AMPLE, is oriented towards robotic modular architectures in the sense that it turns planning algorithms into services that must be generic, reactive, and valuable. Services are optimized actions that are delivered at precise time points following requests from other modules that include states and dates at which actions are needed. To this end, our framework is divided in two concurrent processes: an execution thread which receives planning requests from other modules and accordingly orchestrates internal planning requests, and a planning thread which delegates deliberative action selections to embedded planning softwares in compliance with the queue of internal planning requests. Importantly, default actions specific to the given application must be provided to guarantee the reactivity of the service in cases where no optimized action could be found within a request's deadline. The behavior of the execution thread can be parametrized to achieve various optimization strategies: we especially proposed three variants, namely AMPLE-NEXT, AMPLE-PATH and AMPLE-PORTFOLIO, which differ depending on the distribution of internal planning requests over possible future execution states in anticipation of the uncertain evolution of the system, or over different underlying planners to account for several planning levels. This ability to customize the execution thread makes our framework a meta-planning service which can drive in background any existing specialized planner through internal planning requests.

We note two important features for which our framework particularly stands out from existing planning paradigms. First, it is truly real-time in the sense that it always reactively returns an action (possibly optimized, otherwise a default one) when requested by other modules. This is different from so-called "anytime" or "real-time" planners from the Artificial Intelligence community, which try to maximize the chance of providing optimized actions on request by optimizing the plan while executing it but without any guaranty to actually deliver actions at precise time points. Our framework can in fact use some planners in the planning thread, which help it deliver optimized actions instead of default ones, as highlighted by our experiments with the AEMS underlying anytime planner. But, perhaps more interestingly, planners like A* that are not natively real-time can be made so via embedding in AMPLE and its planning request proxy. Second, our framework is flexible and generic enough to tackle different natures of planning problems, from fully deterministic ones to partially observable ones with probabilistic action effects. We demonstrated its capabilities on three kinds of planning problems: path planning with unknown and moving obstacles, task planning with probabilistic action effects, information planning with partially observable states. It is worth mentioning that uncertain problems like the aforementioned class of problems can be tackled by AMPLE even with deterministic planners as we demonstrated in our experiments: the uncertainty about the environment is dealt with by the strategy implemented in the execution

thread which generates many deterministic planning requests for various possible future execution states.

There are still open questions that are not answered by our current framework. While we think that the need for default actions is mandatory whatever the implemented framework is, for any kind of critical applications where reactivity is at stake, we should provide means to check their validity in terms of quality and safety. This could be partially achieved by model checking the properties of the default action policy along with the available model of the environment. Further checks would be necessary on-line to guarantee that a default action will bring the system to only desired situations given the current assessed context. Optimized actions could be also compared with default ones since the latter do not need to be optimized up to provide the allowed optimization times in operation. Moreover, the class of planning models compatible with `AMPLE` is currently restricted to simple action execution schemes: more complex models involving concurrent (parallel) actions – scheduling problems for instance – or action hierarchies – like hierarchical task networks – arise in many robotic applications and thus should be considered in future extensions of `AMPLE`.

References

Abeywickrama D, Zambonelli F (2012) Model checking goal-oriented requirements for self-adaptive systems. In: International Conference and Workshops on Engineering of Computer Based Systems (ECBS), Novi Sad, Serbia

Albore A, Palacios H, Geffner H (2009) A translation-based approach to contingent planning. In: International Joint Conference on Artificial Intelligence (IJCAI), Providence, RI, USA

Albore A, Palacios H, Geffner H (2010) Compiling uncertainty away in non-deterministic conformant planning. In: European Conference on Artificial Intelligence (ECAI), Lisbon, Portugal

Albore A, Ramirez M, Geffner H (2011) Effective heuristics and belief tracking for planning with incomplete information. In: International Conference on Automated Planning and Scheduling (ICAPS), Freiburg, Germany

Baral C, Kreinovich V, Trejo R (2000) Computational complexity of planning and approximate planning in the presence of incompleteness. Artificial Intelligence 122(1-2):241–267

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

Barto A, Bradtke S, Singh S (1995) Learning to act using real-time dynamic programming. Artificial Intelligence Journal (AIJ) 72(1-2):81–138

Bertoli P, Cimatti A, Roveri M, Traverso P (2006) Strong planning under partial observability. Artificial Intelligence 170(4-5):337–384

Blum A, Furst M (1995) Fast planning through planning graph analysis. In: International Joint Conference on Artificial Intelligence (IJCAI), Quebec, Canada

Blythe J (1999) Decision-theoretic planning. AI magazine 20(2):37

Bonet B, Geffner H (2000) Planning with incomplete information as heuristic search in belief space. In: International Conference on Artificial Intelligence Planning and Scheduling Systems (AIPS), Breckenridge, CO, USA

Bonet B, Geffner H (2001) Planning as heuristic search. Artificial Intelligence 129(1):5–33

Bonet B, Geffner H (2003) Labeled RTDP: Improving the convergence of real-time dynamic programming. In: International Conference on Automated Planning and Scheduling (ICAPS), Trento, Italy

Bonet B, Geffner H (2009) Solving POMDPs: RTDP-bel vs. point-based algorithms. In: International Joint Conference on Artificial Intelligence (IJCAI), San Francisco, CA, USA

Bonet B, Palacios H, Geffner H (2009) Automatic derivation of memoryless policies and finite-state controllers using classical planners. In: International Conference on Automated Planning and Scheduling (ICAPS), Thessaloniki, Greece

Boutilier C, Dean T, Hanks S (1999) Decision-theoretic planning: Structural assumptions and computational leverage. Journal of Artificial Intelligence Research (JAIR) 11(1):94

Brafman R, Hoffmann J (2004) Conformant planning via heuristic forward search: A new approach. In: International Conference on Automated Planning and Scheduling (ICAPS), Whistler, Canada

Bryce D, Kambhampati S, Smith DE (2006) Planning graph heuristics for belief space search. Journal of Artificial Intelligence Research (JAIR) 26:35–99

Burns E, Ruml W, Do MB (2013) Heuristic search when time matters. Journal of Artificial Intelligence Research (JAIR) 47:697–740

Cannon J, Rose K, Ruml W (2012) Real-time motion planning with dynamic obstacles. In: Symposium on Combinatorial Search (SOCS), Niagara Falls, Canada

Carrillo H, Dames P, Kumar V, Castellanos JA (2015) Autonomous robotic exploration using occupancy grid maps and graph slam based on shannon and rényi entropy. In: International Conference on Robotics and Automation (ICRA), Seattle, WA, USA

Carvalho Chanel CP, Teichteil-Königsbuch F (2013) Properly acting under partial observability with action feasibility constraints. In: European Conference on Machine Learning and Knowledge Discovery in Databases (ECML), Prague, Czech Republic

Carvalho Chanel CP, Teichteil-Königsbuch F, Lesire C (2013) Multi-target detection and recognition by UAVs using online POMDPs. In: AAAI Conference on Artificial Intelligence (AAAI), Bellevue, WA, USA

Carvalho Chanel CP, Teichteil-Königsbuch F, Lesire C (2014) A robotic execution framework for online probabilistic (re) planning. In: International Conference on Automated Planning and Scheduling (ICAPS), Portsmouth, NH, USA

De Giacomo G, Gerevini AE, Patrizio F, Saetti A, Sardina S (2016) Agent planning programs. Artificial Intelligence 231:64–106

Domshlak C, Hoffmann J, Katz M (2015) Red–black planning: A new systematic approach to partial delete relaxation. Artificial Intelligence 221:73–114

Echeverria G, Lemaignan S, Degroote A, Lacroix S, Karg M, Koch P, Lesire C, Stinckwich S (2012) Simulating complex robotic scenarios with morse. In: International Conference on Simulation, Modeling, and Programming Autonomous Robots (SIMPAR), Tsukuba, Japan

Ghallab M, Nau DS, Traverso P (2014) The actor's view of automated planning and acting: A position paper. Artificial Intelligence 208:1–17

Hansen E, Zilberstein S (2001) LAO*: A heuristic search algorithm that finds solutions with loops. Artificial Intelligence Journal (AIJ) 129(1-2):35–62

Harland J, Morley DN, Thangarajah J, Yorke-Smith N (2014) An operational semantics for the goal life-cycle in BDI agents. Autonomous Agents and Multi-Agent Systems (JAAMAS) 28(4):682–719

Hart PE, Nilsson NJ, Raphael B (1968) A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems, Science and Cybernetics 4(2):100–107

Haslum P, Jonsson P (1999) Some results on the complexity of planning with incomplete information. In: European Conference on Planning (ECP), Durham, UK

Helmert M, Röger G, et al. (2008) How good is almost perfect? In: AAAI Conference on Artificial Intelligence (AAAI), Chicago, IL, USA

Helmert M, Röger G, Karpas E (2011) Fast downward stone soup: A baseline for building planner portfolios. In: Inernational Conference on Automated Planning and Scheduling (ICAPS) Workshop on Planning and Learning, Freiburg, Germany

Hoey J, St-Aubin R, Hu A, Boutilier C (1999) SPUDD: Stochastic planning using decision diagrams. In: International Conference on Unvertainty in Artificial Intelligence (UAI), Stockholm, Sweden

Hoffmann J, Brafman R (2005) Contingent planning via heuristic forward search with implicit belief states. In: International Conference on Automated Planning and Scheduling (ICAPS), Monterey, CA, USA

Hoffmann J, Nebel B (2001) The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research (JAIR) 14:253–302

Hoffmann J, Porteous J, Sebastia L (2004) Ordered landmarks in planning. Journal of Artificial Intelligence Research (JAIR) 22:215–278

Hornung A, Wurm KM, Bennewitz M, Stachniss C, Burgard W (2013) OctoMap: An efficient probabilistic 3D mapping framework based on octrees. Autonomous Robots 34(3)

Ingrand F, Ghallab M (2014) Robotics and artificial intelligence: A perspective on deliberation functions. AI Communications 27(1):63–80

Kaelbling L, Littman M, Cassandra A (1998) Planning and acting in partially observable stochastic domains. Aritificial Intelligence 101(1–2):99–134

Katz M, Lipovetzky N, Moshkovich D, Tuisov A (2017) Adapting novelty to classical planning as heuristic search. In: International Conference on Automated Planning and Scheduling (ICAPS), Pittsburgh, PA, USA

Kautz H, Selman B (1996) Pushing the envelope: Planning, propositional logic, and stochastic search. In: National Conference on Artificial Intelligence (AAAI), Portland, OR, USA

Keller T, Eyerich P (2012) PROST: Probabilistic Planning Based on UCT. In: International Conference on Automated Planning and Scheduling (ICAPS), Sao Paulo, Brazil

Keyder E, Hoffmann J, Haslum P (2014) Improving delete relaxation heuristics through explicitly represented conjunctions. Journal of Artificial Intelligence Research (JAIR) 50:487–533

Knight S, Rabideau G, Chien S, Engelhardt B, Sherwood R (2001) Casper: Space exploration through continuous planning. IEEE Intelligent Systems 16(5):70–75

Koenig S, Likhachev M (2001) Incremental A*. In: Conference on Neural Information Processing Systems (NIPS), Vancouver, Canada

Korf R (1990) Real-time heuristic search. Artificial Intelligence 42(2–3):189–211

Kurniawati H, Hsu D, Lee WS (2008) SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces. In: Robotics: Science and Systems (RSS), Zurich, Switzerland

Kuter U, Nau D, Reisner E, Goldman R (2008) Using classical planners to solve nondeterministic planning problems. In: International Conference on Automated Planning and Scheduling (ICAPS), Sydney, Australia

Le Saux B, Sanfourche M (2011) Robust vehicle categorization from aerial images by 3D-template matching and multiple classifier system. In: International Symposium on Image and Signal Processing and Analysis (ISPA), Dubrovnik, Croatia

Lemai S, Ingrand F (2004) Interleaving temporal planning and execution in robotics domains. In: AAAI Conference on Artificial Intelligence (AAAI), San Jose, CA, USA

Likhachev M, Ferguson D, Gordon G, Stentz A, Thrun S (2008) Anytime search in dynamic graphs. Artificial Intelligence 172(14):1613–1643

Lin C, Kolobov A, Kamar E, Horvitz E (2015) Metareasoning for planning under uncertainty. In: International Joint Conference on Artificial Intelligence (IJCAI), Buenos Aires, Argentina

Lipovetzky N, Geffner H (2012) Width and serialization of classical planning problems. In: European Conference on Artificial Intelligence (ECAI), Montpellier, France

Lipovetzky N, Geffner H (2017) A polynomial planning algorithm that beats LAMA and FF. In: International Conference on Automated Planning and Scheduling (ICAPS), Pittsburg, PA, USA

Littman ML, Cassandra AR, Kaelbling LP (1995) Learning policies for partially observable environments: Scaling up. In: International Conference on Machine Learning (ICML), Tahoe City, CA, USA

Makarenko AA, Williams SB, Bourgault F, Durrant-Whyte HF (2002) An experiment in integrated exploration. In: International Conference on Intelligent Robots and Systems (IROS), Lausanne, Switzerland

McGann C, Py F, Rajan K, Thomas H, Henthorn R, McEwen R (2008) A deliberative architecture for AUV control. In: International Conference on Robotics and Automation (ICRA), Pasadena, CA, USA

Meuleau N, Benazera E, Brafman R, Hansen E, Mausam (2009) A heuristic search approach to planning with continuous resources in stochastic domains. Jounal of Artificial Intelligence Research (JAIR) 34(1):27–59

Munoz-Avila H, Wilson MA, Aha DW (2015) Guiding the ass with goal motivation weights. In: Annual Conference on Advances in Cognitive Systems: Workshop on Goal Reasoning, Atlanta, GA, USA

Nau D, Ghallab M, Traverso P (2004) Automated Planning: Theory and Practice. Elsevier

Nau DS, Ghallab M, Traverso P (2015) Blended planning and acting: Preliminary approach, research challenges. In: AAAI Conference on Artificial Intelligence (AAAI), Austin, TX, USA

Palacios H, Geffner H (2009) Compiling Uncertainty Away in Conformant Planning Problems with Bounded Width. Journal of Artificial Intelligence Research (JAIR) 35:623–675

Pineau J, Gordon G, Thrun S (2006) Anytime Point-Based Approximations for Large POMDPs. Jounal of Artificial Intelligence Research (JAIR) 27:335–380

Puterman M (1994) Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., New York, NY, USA

Richter S, Westphal M (2010) The LAMA planner: Guiding cost-based anytime planning with landmarks. Journal of Artificial Intelligence Research (JAIR) 39(1):127–177

Richter S, Helmert M, Westphal M (2008) Landmarks revisited. In: AAAI Conference on Artificial Intelligence (AAAI), Chicago, IL, USA

Richter S, Thayer JT, Ruml W (2010) The joy of forgetting: Faster anytime search via restarting. In: International Conference on Automated Planning and Scheduling (ICAPS), Toronto, Canada

Rintanen J (2004) Complexity of planning with partial observability. In: International Conference on Automated Planning and Scheduling (ICAPS), Whistler, Canada

Rintanen J, Heljanko K, Niemelä I (2006) Planning as satisfiability: parallel plans and algorithms for plan search. Artificial Intelligence 170(12–13):1031–1080

Roberts M, Shivashankar V, Alford R, Leece M, Gupta S, Aha D (2016) Goal reasoning, planning, and acting with ActorSim, the Actor Simulator. In: Annual Conference on Advances in Cognitive Systems (CogSys), Evanston, IL, USA

Ross S, Chaib-Draa B (2007) AEMS: An anytime online search algorithm for approximate policy refinement in large POMDPs. In: International Joint Conference on Artificial Intelligence (IJCAI), Hyderabad, India

Ross S, Pineau J, Paquet S, Chaib-Draa B (2008) Online planning algorithms for POMDPs. Journal of Artificial Intelligence Research (JAIR) 32(1):663–704

Sabbadin R, Lang J, Ravoanjanahary N (2007) Purely epistemic Markov Decision Processes. In: AAAI Conference on Artificial Intelligence (AAAI), Vancouver, Canada

Sanfourche M, Vittori V, Besnerais GL (2013) eVO: A realtime embedded stereo odometry for MAV applications. In: International Conference on Intelligent Robots and Systems (IROS), Tokyo, Japan

Smallwood R, Sondik E (1973) The optimal control of partially observable Markov processes over a finite horizon. Operations Research 21(5):1071–1088

Smith T, Simmons R (2004) Heuristic search value iteration for POMDPs. In: International Conference on Uncertainty in Artificial Intelligence (UAI), Banff, Canada

Sondik EJ (1978) The optimal control of partially observable Markov processes over the Infinite Horizon: Discounted Costs. Operations Research 26(2):282–304

Spaan M, Vlassis N (2005) Perseus: Randomized point-based value iteration for pomdps. Journal of Artificial Intelligence Research (JAIR) 24(1):195–220

Stentz A (1995) The focussed D* algorithm for real-time replanning. In: International Joint Conference on Artificial Intelligence (IJCAI), Quebec, Canada

Teichteil-Königsbuch F, Kuter U, Infantes G (2010) Incremental plan aggregation for generating policies in MDPs. In: International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), Toronto, Canada

Teichteil-Konigsbuch F, Lesire C, Infantes G (2011) A generic framework for anytime execution-driven planning in robotics. In: International Conference on Robotics and Automation (ICRA), Shangai, China

To S, Son T, Pontelli E (2010) A New Approach to Conformant Planning using CNF. In: International Conference on Automated Planning and Scheduling (ICAPS), Toronto, Canada

To S, Son T, Pontelli E (2011) Contingent Planning as AND/OR Forward Search with Disjunctive Representation. In: International Conference on Automated Planning and Scheduling (ICAPS), Freiburg, Germany

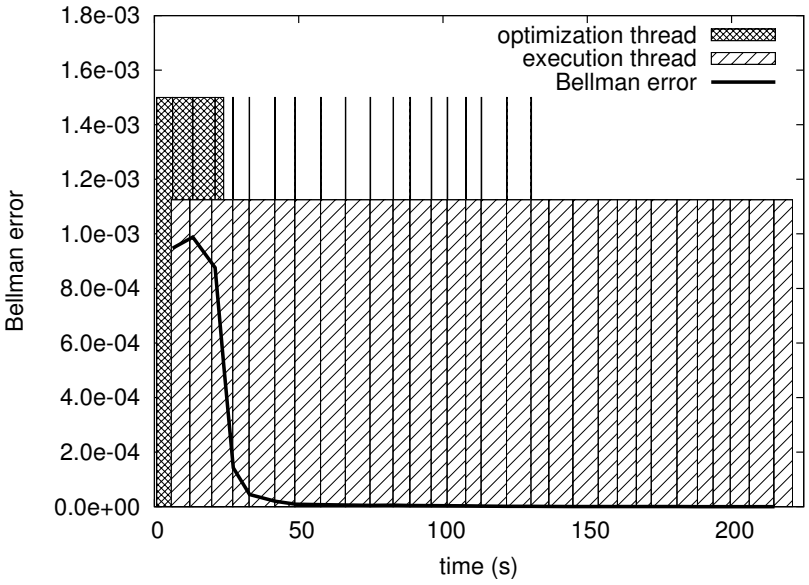
Turner H (2002) Polynomial-length planning spans the polynomial hierarchy. In: European Conference on Logics in AI (JELIA), Cosenza, Italy

Vattam S, Klenk M, Molineaux M, Aha D (2013) Breadth of approaches to goal reasoning: A research survey. Tech. Rep. CS-TR-5029), College Park, University of Maryland, Department of Computer Science

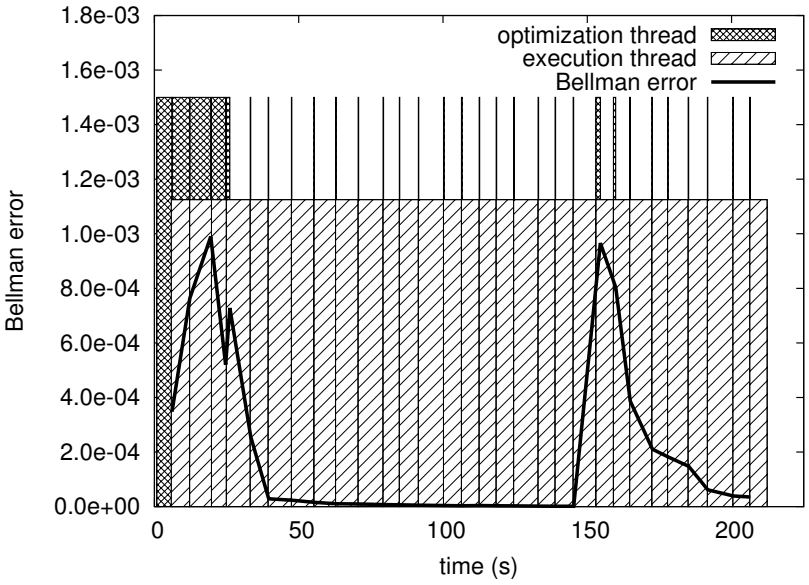
Verma V, Jónsson A, Pasareanu C, Iatauro M (2006) Universal executive and PLEXIL: Engine and language for robust spacecraft control and operations. In: AIAA Space Conference, San Jose, CA, USA

Wilson MA, Molineaux M, Aha DW (2013) Domain-independent heuristics for goal formulation. In: Florida Artificial Intelligence Research Society Conference (FLAIRS), St. Pete Beach, FL, USA

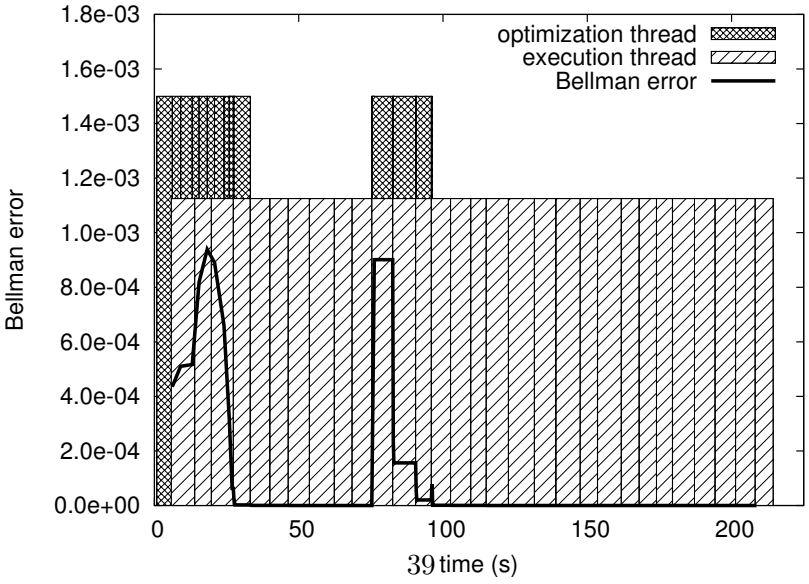
Young J, Hawes N (2012) Evolutionary learning of goal priorities in a real-time strategy game. In: AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), Palo Alto, CA, USA



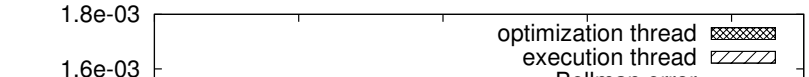
(a) NEXT



(b) PATH-1



(c) PATH-3



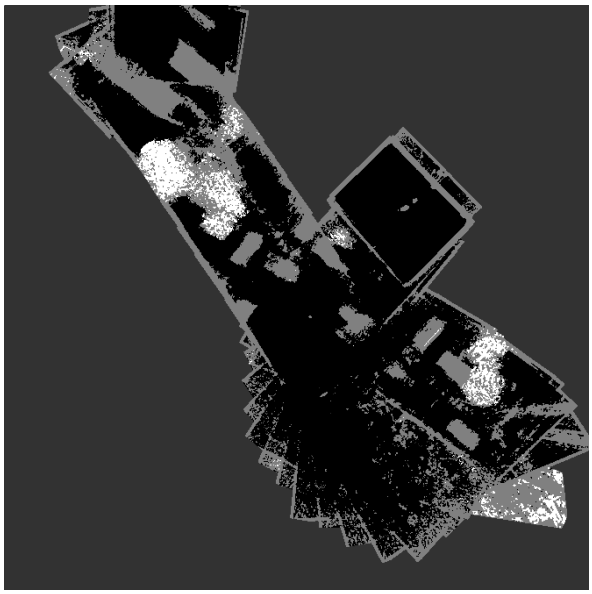


Figure 4: Map built during experiment. Map size is approximately 200 meters wide.

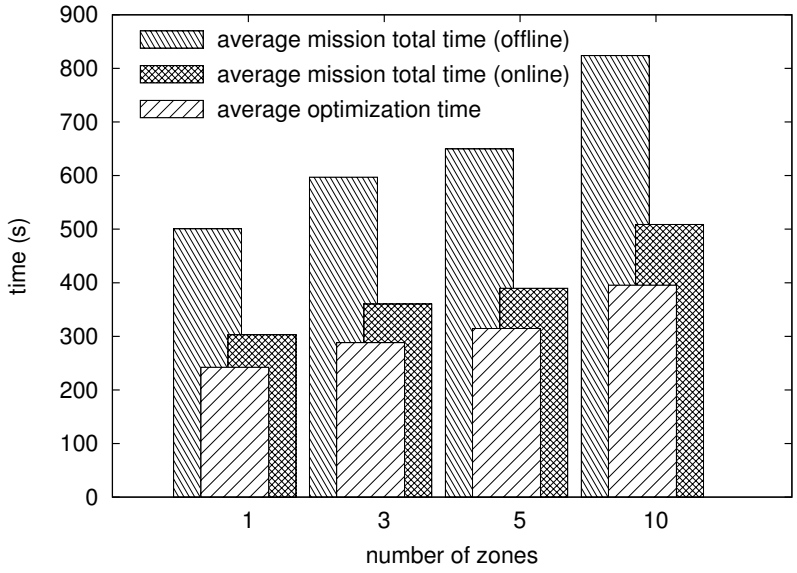


Figure 5: Mission time results for the autonomous emergency landing experiment.

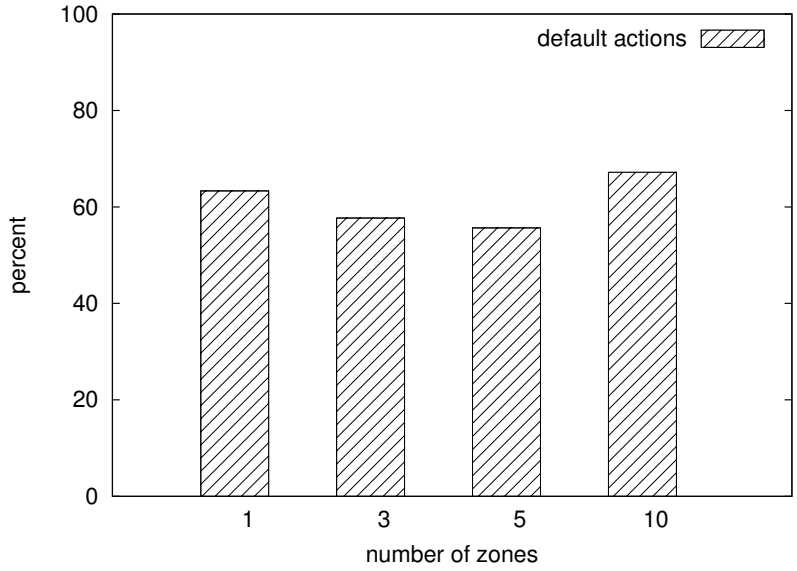
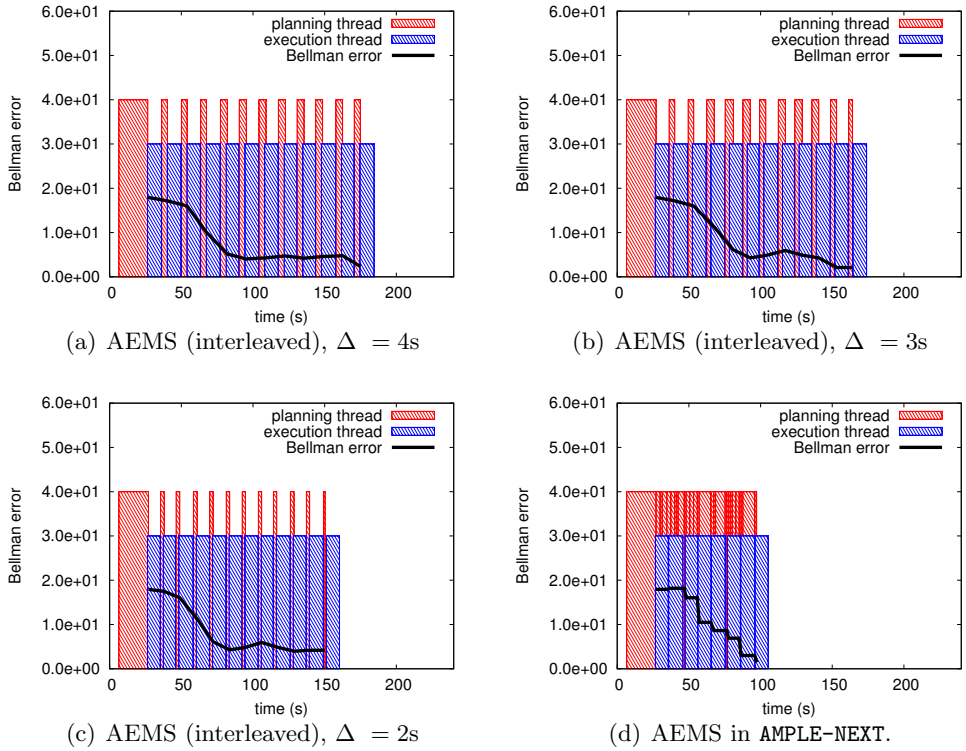


Figure 6: Default actions for the autonomous emergency landing experiment.



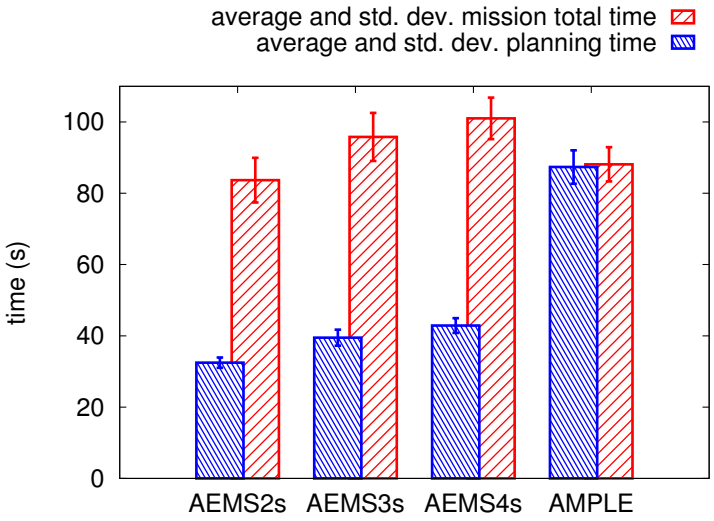
(a) AEMS (interleaved), $\Delta = 4$s

(b) AEMS (interleaved), $\Delta = 3$s

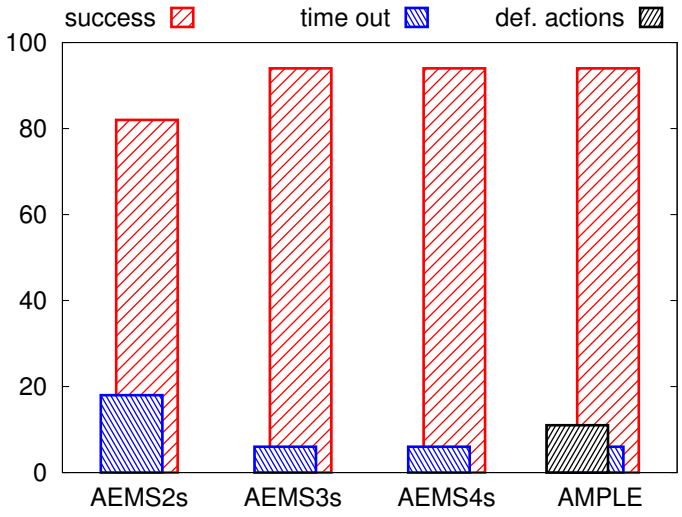
(c) AEMS (interleaved), $\Delta = 2$s

(d) AEMS in `AMPLE-NEXT`.

Figure 7: Timelines for classical AEMS (interleaved approach) with 4s, 3s, 2s for planning *versus* AEMS in `AMPLE-NEXT`.



(a) Avg. time for success.



(b) Percent. among missions.

Figure 8: Target detection and recognition mission.

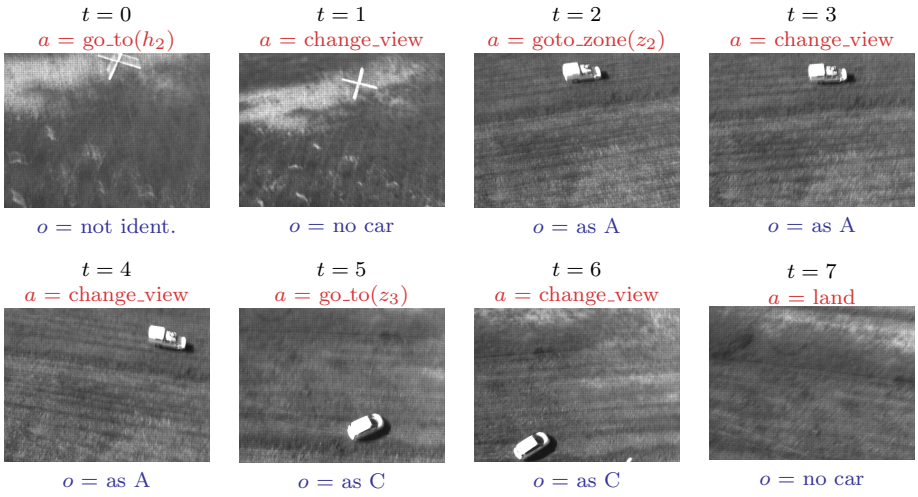


Figure 9: Sequence of decisions for successive time steps t. Each image represents the input of the image processing algorithm after the current action a is executed during a real flight. Observations o represent the successive outputs of the image processing algorithm's classifier. In this mission, the searched car model was C.

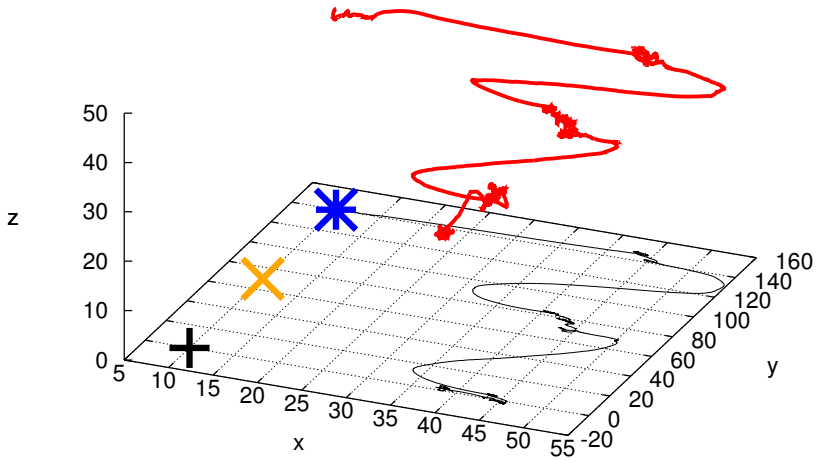
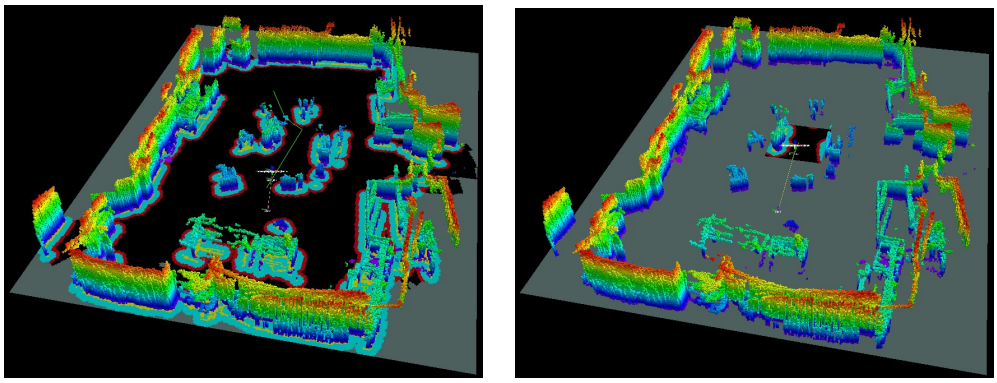


Figure 10: UAV's trajectory performed during the real flight.



(a) with unlimited field-of-view (b) with limited field-of-view of 1m

Figure 11: Octomap and costmap on a 3D environment

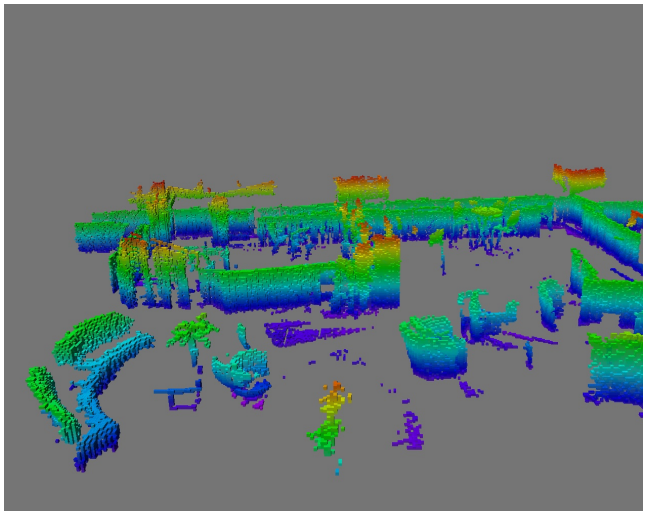


Figure 12: 3D octomap built during the contest.

Figure 13: Costmap and robot trajectory during the contest.

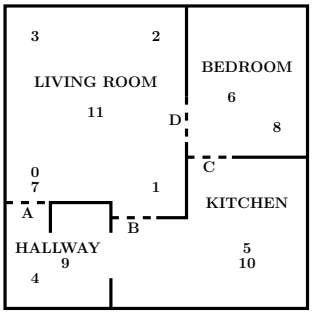


Figure 14: Navigation environment setup