A Robotic Execution Framework for Online Probabilistic (Re)Planning

Caroline P. Carvalho Chanel
caroline.chanel@isae.fr
Université de Toulouse - ISAE
10 av. Édouard-Belin, 31055 Toulouse, France

Charles Lesire, Florent Teichteil-Königsbuch
{*charles.lesire,florent.teichteil*}.@*onera.fr*
Onera – The French Aerospace Lab
2 av. Édouard-Belin, 31055 Toulouse, France

Abstract

Due to the high complexity of probabilistic planning algorithms, roboticists often opt for deterministic replanning paradigms, which can quickly adapt the current plan to the environment's changes. However, probabilistic planning suffers in practice from the common misconception that it is needed to generate complete or closed policies, which would not require to be adapted on-line. In this work, we propose an intermediate approach, which generates incomplete partial policies taking into account mid-term probabilistic uncertainties, continually improving them on a gliding horizon or regenerating them when they fail. Our algorithm is a configurable anytime meta-planner that drives any sub-(PO)MDP standard planner, dealing with all pending and time-bounded planning requests sent by the execution framework from many reachable possible future execution states, in anticipation of the probabilistic evolution of the system. We assess our approach on generic robotic problems and on combinatorial UAVs (PO)MDP missions, which we tested during real flights: emergency landing with discrete and continuous state variables, and target detection and recognition in unknown environments.

Introduction

In many robotic applications, autonomous agents are faced to complex task planning problems, which may not be known before the mission, and thus must be solved online. Classical approaches consist in making deterministic assumptions about the future evolution of the system, then plan a sequence of actions on the basis of these assumptions, and potentially replan when the current state of the world differs from the predicted one. Complex plan monitoring and repairing techniques are required to maximize the chance of mission success (Lemai and Ingrand 2004; Finzi, Ingrand, and Muscettola 2004; Fazil Ayan et al. 2007; Doherty, Kvarnström, and Heintz 2009). These techniques may rely on "real-time" (or "anytime") planning algorithms (Korf 1990) to boost plan generation and reduce the overall mission time, but such algorithms are not sufficient in themselves to execute the mission. Indeed, they tend to compute successive suboptimal plans, whose quality increases over time, which can be executed without waiting for the final optimal plan to be produced. However, there is no strong guarantees that each plan, even the first action of the plan, is computed in a known limited time, which is necessary for smooth plan execution and integration with other processes.

On the other hand, some planning algorithms take into account probabilities about the possible future evolutions of the system, by generating a plan conditioned on all future contingencies, named *policy*. Due to the high complexity of planning under probabilistic uncertainties, most state-of-the-art algorithms do not construct full policies, i.e. defined over the entire state space. Instead, they compute either partial closed policies (Meuleau et al. 2009; Hansen and Zilberstein 2001), i.e. defined over a subset of the states but which are guaranteed to succeed if the planning model is physically accurate, or incomplete policies that are continually completed and improved over time (Barto, Bradtke, and Singh 1995; Bonet and Geffner 2009; Pineau, Gordon, and Thrun 2006; Ross and Chaib-Draa 2007). However, none of these approaches are fully reliable in robotics. In the first case, building closed policies would be useless because the planning model rarely match the real physical model, meaning that the execution has actually a high chance to fail in practice. In the second case, the algorithms are certainly said to be anytime, but they cannot guarantee to provide an action on time when queried by the execution engine, simply because they all rely on uninterruptible so-called Bellman backups (Puterman 1994) whose completion time is difficult to bound.

In this paper, we propose an intermediate approach where we continually improve an incomplete partial policy over time, like in anytime probabilistic methods, but we allow ourselves to query the policy before completion of the optimization at any time and stop the current planning instance in order to replan from a new execution state, much like deterministic plan/replan approaches. Our method is formalized as an execution model that embeds the `AMPLE` probabilistic meta-planner, which guarantees to provide safe and optimized actions in the current state of the world, at precise execution time points. `AMPLE` is a complete rewriting of routines for solving (Partially Observable) Markov Decision Processes in a general algorithmic schema, which *optimizes several local policy chunks at future possible observations of the system*, while executing the current action.

On the contrary, existing (PO)MDP anytime algorithms optimize the policy only from the robot's current observation, not anticipating its possible interactions with its environment, so that no applicable action may be available in the current execution state when queried by the execution controller. We first introduce (PO)MDP modeling and optimization. Then, we present a model for anytime policy optimization and execution under time constraints, and the corresponding AMPLE algorithm. We finally test our algorithm on random on-line MDP problems, as well as real complex UAV missions modeled as on-line (PO)MDPs that were successfully tested in real conditions. To the best of our knowledge, these are among the first applications of (PO)MDPs on-board UAVs, whose problems cannot be known prior to the flight, and whose policies are optimized and successfully executed during the flight. We show that AMPLE achieves good quality while being always reactive to the execution engine's queries, which provides a practical paradigm for on-line (PO)MDP planning in large robotic problems.

Policy optimization under probabilistic uncertainties

In this section, we formally define Markov Decision Processes (MDPs), a popular and convenient general model for sequential decision-making under uncertainties. We present both the complete observation case (simply noted as MDPs, see (Puterman 1994)) and the Partially Observable one (POMDPs, see (Kaelbling, Littman, and Cassandra 1998)), as our AMPLE planner matches both cases.

Complete and Partially Observable MDPs

A (Partially Observable) Markov Decision Process is a tuple $\mathcal{M} = \langle S, O, A, T, R, \Omega \rangle$ where: S is a set of *states* ; O is a set of *observations*, in the complete observable case, $O = S$; A is a set of *actions* ; T is the probabilistic *transition* function, such that, for all $(s, a, s') \in S \times A \times S$, $T(s, a, s') = Pr(s' \mid a, s)$; R is the *reward* function, such that, for all $(s, a, s') \in S \times A \times S$, $R(s, a, s')$ is the reward associated to the transition $T(s, a, s')$; Ω is the probabilistic *observation* function, such that, for all $(s', a, o) \in S \times A \times O$, $O(s', a, o) = Pr(o \mid a, s')$; in the complete observable case, $O(s', a, o) = \mathbf{1}_{\{s'=o\}}$. It is often convenient to 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 $succ : S \times A \to 2^S$ defines, for all $s \in S$ and $a \in A$, the set of states $succ(s, a)$ that are reachable in one step by applying action a in state s.

Algorithms for solving (PO)MDPs

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 π from the decision process' beginning up to the planning horizon which may be infinite. In the MDP case, states are completely observable so that $\pi : S \to A$ is directly applicable over the state space. In the POMDP case, states are not directly observable and the policy is not applicable by the controller 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 states of the system at each time step, named *belief state*, thanks to successive applications of Bayes' rule (Kaelbling, Littman, and Cassandra 1998). We note $B \subset [0; 1]^S$ the set of belief states ; for all $b \in B$, $\sum_{s \in S} b(s) = 1$. It often happens that the belief state is computed by the controller itself instead of the planner: in this case, the policy is defined over belief states, $\pi : B \to A$. Some algorithms optimize π over all the possible initial (belief) states of the world, like linear programming (Kaelbling, Littman, and Cassandra 1998), value iteration, policy iteration (Puterman 1994; Hoey et al. 1999). Some other recent planners use the knowledge of the system's initial (belief) state, and sometimes of goal states to reach via heuristic search, in order to compute a partial policy that is defined only over a subset of (belief) states that are reachable from the initial (belief) state by successively applying feasible actions from it (LAO* (Hansen and Zilberstein 2001), (L)RTDP (Bonet and Geffner 2003), RFF (Teichteil-Königsbuch, Kuter, and Infantes 2010), HAO* (Meuleau et al. 2009), HSVI (Smith and Simmons 2005), RTDP-Bel (Bonet and Geffner 2009), or AEMS (Ross and Chaib-Draa 2007)).

General algorithmic schema of (PO)MDP solving

As far as we know, most (PO)MDP algorithms, at least iterative ones, follow the algorithmic schema depicted in Alg. 1. They first initialize data structures and, for some algorithms, store the set of possible initial states (MDPs) or the initial belief state (POMDPs) (Line 1). Then, they improve the value function (Line 2), possibly using the discount factor γ. During this step, some algorithms also update the policy or the set of explored states (heuristic search MDP algorithms) or beliefs (point-based or heuristic search POMDP algorithms). They iterate until a convergence test becomes true (Line 2), for instance when the difference between two successive value functions is ϵ-bounded, 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 HSVI may eventually converge after an infinite number of iterations, so that the convergence test should be always false in theory. Finally, data structures are cleaned before returning the optimized value function and corresponding policy (Line 3); in some cases, the policy is computed for the first time during this final step, once the value function has converged.

Algorithm 1: General schema of (PO)MDP algorithms

```
  input : (PO)MDP $\mathcal{M}$, $0 < \gamma \leqslant 1$, $\epsilon > 0$, $N \in \mathbb{N}^*$, set of
          initial states or belief state $I$
  output: Value function $V$ and policy $\pi$
1 solve_initialize($I$);
2 repeat solve_progress($\gamma$);     // update $V$ or $\pi$
  until solve_converged($\epsilon, N$) ;
3 solve_end();   // compute $\pi$ if not done yet
4 return ($V, \pi$);
```

Anytime algorithms (Bonet and Geffner 2003; Pineau, Gordon, and Thrun 2006; Bonet and Geffner 2009) guarantee to *quickly* build an applicable policy at initialization (Line 1) or at each improvement (Line 2), but *no time threshold is given* that correspond to the current action's expected execution time in order to constrain the policy's building or improvement time in such a way that it is always applicable in the robot's next execution state. Therefore, when the execution controller queries an action in the next execution state, the policy may be broken and not applicable at all, which is not acceptable for time-constrained robotic missions. To overcome this issue which has been nearly never studied in the (PO)MDP literature, a finer control over the optimization process and its interaction with the possible future execution states of the robot is needed, as we propose in the next.

Anytime policy optimization and execution

In this section, we present our AMPLE (Anytime Meta PLannEr) planner, which drives any (PO)MDP planner that conforms to the general algorithmic schema depicted in Algorithm 1, and which is proved to be strictly anytime in the sense of policy execution under time constraints: it ensures to return an applicable *relevant* action in any possible given state at a precise time point, when required by the execution engine. AMPLE is designed as a configurable bi-threaded program: at first glance, an "execution" thread reactively interacts with the execution engine by managing multiple present and future planning requests, while an "optimization" thread deliberatively optimize planning problems in background by controlling the actual (PO)MDP optimizer. To this end, the (PO)MDP optimizer, which was normally conceived to be run as in Algorithm 1, is now broken up into several functions (solve_initialize, solve_progress, solve_converged, solve_end) that are directly managed by the optimization thread of the meta-planner according to the planning requests pending in the execution thread. In this sense, the optimization thread can be seen as an algorithmic rewriting of Algorithm 1. Planning requests management in the execution thread conforms to a formal model defined as a configurable (controllable) finite state machine (Figure 1), whose particular configuration in conjunction with the optimization thread algorithm actually yield to a well-defined planning algorithm. To be more concrete, Algorithm 1 is a particular instance of the execution thread's finite state machine and optimization thread's algorithm together, so that we can exactly reproduce algorithms like VI, LAO*, RTDP, HSVI, AEMS, etc. Most importantly, other configurations yield to anytime original algorithms, built upon the previous ones.

We first formalize planning requests, which AMPLE can deal with. Then, we define the configurable reactive finite state machine of the execution thread, as well as the deliberative algorithm of the optimization thread. Finally, we formalize two particular anytime instances of the AMPLE meta-planner, which are actually two original anytime probabilistic planning algorithms based on the solve_* functions of standard (PO)MDP algorithms, disassembled from Algorithm 1 and reorganized in a more general algorithm.

Planning requests

Planning requests are sent by the execution engine to the AMPLE planner. When received, the latter must update the current policy depending on information included in the planning request, which is a tuple $\mathcal{R} = \langle I, \Delta, \alpha, \alpha_p \rangle$ defined as follows. First, I is a set of states (MDPs) or *belief* states (POMDPs) from which the policy must be updated; belief states are computed using the history of actions, observations and Bayes' rule. Second, Δ is the (continuous) maximum duration of the policy update; if the planning request is received at t_0 by the AMPLE planner, then the execution engine may ask for an optimized action to execute in any states or observations $x \in I$ at any time point $t \geqslant t_0 + \Delta$. Third, α is the particular algorithm of the sub-(PO)MDP planner to use in order to update the policy; different algorithms may be used in different planning requests by the execution engine, in order for instance to adapt optimization to the particular (PO)MDP that is being solved; it may be also useful to unjam the AMPLE planner if it is trapped because of the current algorithm that would not be appropriate to the (PO)MDP problem solved. Fourth, α_p contains the parameters of algorithm α (e.g., ϵ and γ).

It is worth noting that, according to the model of the planning problem, the set of (belief) states I does not need to be physically reachable from the (belief) state of the system that is currently being executed. It has two pragmatic advantages: first, the execution engine can broaden the set of future states or observations from which the AMPLE planner is requested to plan, *in anticipation of errors in the model of the planning problem* that would shift the most expected future states or observations away from the true current state of the system. Second, in case this shift would not have been anticipated and the true current state is not included in the policy, the execution engine can still ask AMPLE to reconstruct a fresh local policy from the current state after a short bootstrap duration.

The AMPLE framework

Planning requests constitute the core information exchanged between the two threads of AMPLE, whose main routine is depicted in Procedure 1 and explained in the next paragraph. Each procedure or algorithm presented in the next relies on some parts of the following data structures and notations (***bold italic data*** are shared between the execution and optimization threads): $\mathcal{M}$, (PO)MDP model of the problem; psm, state machine that formalizes the interaction between the execution and optimization threads of AMPLE; pln, sub-(PO)MDP planner driven by AMPLE in the optimization thread; π_d, default policy generated before execution; ***pr***, list of planning requests managed by the execution thread and solved by the optimization thread, modelled as a first-in first-out queue (first received requests are first solved); $\boldsymbol{\pi_{sr}}$, backup policy defined on the set of (belief) states I of each solved planning request; ***stopCurrentRequest***, boolean indicating whether the current request being solved in the optimization thread should be interrupted; ***stopPlannerRequested***: boolean indicating whether AMPLE should be stopped (for instance when the mission is finished).

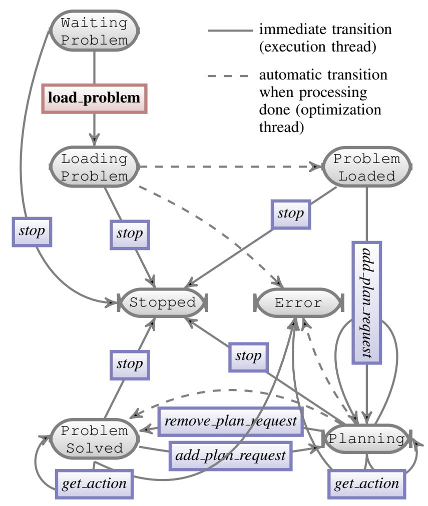


Figure 1: AMPLE's Controllable Finite State Machine

Procedure 1 describes the main routine of AMPLE. 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 that defines the interactions with the system's execution engine (Line 5), and the default policy required to guarantee reactivity in case the optimized policy would not be available in the current execution state (Line 6). Note that the default policy can be a parametric or heuristic policy specifically designed for a given mission. Then, it launches two concurrent threads: the optimization thread (Line 7), where queued planning requests are solved, and the execution thread (Line 8), which interacts with the system's execution engine and queues planning requests.

Procedure 1: AMPLE_main()

```
  input: $\mathcal{M}$, $pln$
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_optimize($\mathcal{M}$, $pln$, $psm$, $\boldsymbol{pr}$,
  $\boldsymbol{\pi_{sr}}$, $\boldsymbol{stopCurrentRequest}$, $\boldsymbol{stopPlannerRequested}$));
8 launch_thread(AMPLE_execute($\mathcal{M}$, $psm$, $\pi_d$ , $\boldsymbol{pr}$, $\boldsymbol{\pi_{sr}}$,
  $\boldsymbol{stopCurrentRequest}$, $\boldsymbol{stopPlannerRequested}$));
```

AMPLE's execution thread. The execution thread adds and removes planning requests according to the current execution state and to the future probable evolutions of the system, and reads the action to execute in the current execution state in the backup policy or in the default one. Different strategies are possible, leading to different actual anytime algorithms described later. Sequences of additions and removals of planning requests and of action retrievals are formalized in the state machine defined in Figure 1, which also properly defines synchronizations between the execution and the optimization threads.

The add_plan_request command of this state machine simply puts the state machine in the PLANNING mode and adds a request r to the queue of pending planning requests ***pr***. The remove_plan_request command removes a request r only if it is not being solved by the optimization thread, i.e. only if it is not at the front of the list of pending planning requests ***pr***. Otherwise, it sets the ***stopCurrentRequest*** variable to true in order to request the optimization thread to stop solving this request. Finally, the get_action command reads the optimized action to execute in the current execution state if it is included in the backup policy $\boldsymbol{\pi_{sr}}$. If not, the action from the default policy $\boldsymbol{\pi_d}$ is used, so that the planning process is always reactive to the robot's global execution engine.

AMPLE's optimization thread. The optimization thread (Algorithm 2) is a complete reorganization of standard (PO)MDP algorithmic schema depicted in Algorithm 1, in order to automatically manage the queue of planning requests and to locally update the policy in bounded time for each planning request. It is conceived as an endless loop that looks at, and accordingly reacts to, the current mode of the state machine defined in Figure 1. If the mode is LOADING_PROBLEM (further to an activation of the load_problem command in the execution thread),

Algorithm 2: AMPLE_optimize

```
1  $solvingRequest \leftarrow$ false;
2  while true do
3      if $psm.state =$ LOADING_PROBLEM then
4          $subpln$.load_problem($\mathcal{M}$);
5          $psm.state \leftarrow$ PROBLEM_LOADED;
6      else if $psm.state =$ PLANNING then
7          if $solvingRequest =$ false then
8              launch_front_request();
9          else
10             $t \leftarrow$ get current CPU time ;
11             if $subpln$.solve_converged($\boldsymbol{pr}.front.\alpha_p$)
               or $t - requestStartTime > \boldsymbol{pr}.front.\Delta$
               or $\boldsymbol{stopCurrentRequest} =$ true
               or $\boldsymbol{stopPlannerRequested} =$ true then
12                 $subpln$.solve_end();
13                 $\boldsymbol{\pi_{sr}} \leftarrow \boldsymbol{\pi_{sr}} \cup subpln$.policy($\boldsymbol{pr}.front.I$);
14                 $\boldsymbol{pr}.pop\_front()$;
15                 $\boldsymbol{stopCurrentRequest} \leftarrow$ false;
16                 if $\boldsymbol{pr}$ is not empty then
17                     launch_front_request();
18                 else
19                     $psm.state \leftarrow$ PROBLEM_SOLVED;
20                     $solvingRequest \leftarrow$ false;
21             else
22                 $subpln$.solve_progress($\boldsymbol{pr}.front.\alpha_p$);
23                 $\boldsymbol{\pi_{sr}} \leftarrow \boldsymbol{\pi_{sr}} \cup subpln$.policy($\boldsymbol{pr}.front.I$);

24 Procedure launch_front_request
25 $solvingRequest \leftarrow$ true;
26 $subpln$.set_algorithm($\boldsymbol{pr}.front.\alpha, \boldsymbol{pr}.front.\alpha_p$);
27 $requestStartTime \leftarrow$ get current CPU time ;
28 $subpln$.solve_initialize($\boldsymbol{pr}.front.I$);
```

it loads the problem from the (PO)MDP model $\mathcal{M}$ and changes the mode to PROBLEM_LOADED (Lines 3 to 5). If the mode is PLANNING (further to an activation of some add_plan_request commands in the execution thread), it tests if the front planning request in the queue is already being solved (Line 7). If not, it launches its optimization (Lines 25 to 28), which mainly consists in recording the current CPU time and calling the solve_initialize procedure of the sub-(PO)MDP planner. Otherwise, it means that the optimization thread was already solving this request from a previous iteration of its endless loops. In this case, it tests if the optimization of this request must end now (Line 11), which can happen if the sub-(PO)MDP planner has converged, or if the time allocated to solve the request has been consumed, or if requested by the execution thread via the remove_plan_request command, or if the AMPLE planner has to be stopped. If all these conditions are false, we continue to optimize the request by calling the solve_progress procedure of the sub-(PO)MDP planner and we update the backup policy $\boldsymbol{\pi_{sr}}$ at the initial (belief) state of the request (Lines 22 to 23). Otherwise, we call the solve_end procedure of the sub-(PO)MDP planner, update the backup policy and remove the current planning request from the queue (Lines 12 to 15). Then, we launch the next planning request in the queue if any, or change the mode of the state machine to PROBLEM_SOLVED.

At least two helpful instantiations of AMPLE

All ingredients are there to design anytime probabilistic planning algorithms, depending on how AMPLE's execution thread's state machine (see Figure 1) is used. Since AMPLE's optimization thread automatically manages the queue of planning requests to solve (see Algorithm 2), 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 Procedure 2.

Procedure 2: bootstrap_execution()

```
input  : $\mathcal{M}, psm, \boldsymbol{pr}$
output: $b$: 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 (PO)MDP solving algorithm;
7  add_plan_request($psm, \boldsymbol{pr}, r$);
8  Wait $r.\Delta$ amount of time;
9  $b \leftarrow$ observe and update current (belief) state;
10 return $b$;
```

We first load the problem from the (PO)MDP model $\mathcal{M}$ and wait for its completion by looking at the mode of the state machine (Lines 1-2). We then create a first planning request, filled with the initial (belief) state of the system, a chosen bootstrap planning time Δ, and a (PO)MDP algorithm with its parameters (Lines 3-6). Finally, we add a planning request to the queue of planning requests, wait an amount Δ of time (meanwhile the request is optimized in the optimization thread), and return the (belief) state corresponding to the observation of the system (Lines 7-10).

Now that the planner is bootstrapped, we can go into the "act-sense-plan" loop, for which we propose two different planning strategies presented in the next paragraphs.

AMPLE-NEXT: predicting the evolution of the system one step ahead. In this setting, each time the system begins to execute an action a, we compute the next possible (belief) states of this action and add planning requests for each of these (belief) states. With this strategy, we can anticipate the short-term evolution of the system so that we give a chance to the optimization thread to provide an optimized action on time when the current action has completed in the execution thread. Moreover, as the internal policy of the sub-(PO)MDP planner is not cleared between successive planning requests, the chance to provide an optimized action in the current execution state increases with time. The AMPLE-NEXT strategy is described in Algorithm 3. Once an action has completed, we look at the next action a to execute by calling the get_action command for the current execution state; we start executing it and compute its expected duration Δ_a (Lines 3 to 5). Then, we add planning requests for each possible next observation of the system (translated into a belief state b' in the partially observable case), whose maximum computation time is proportional to Δ_a and to the probability of getting b' as effect of executing a (Lines 6 to 11). We wait for action a to complete (meanwhile added planning requests are solved in the optimization thread) and remove the previous planning requests in case they are not yet solved by the optimization thread (Lines 12 to 14). Finally, we observe the current execution state (Line 15) and go back to the beginning of the execution loop (Line 3).

Algorithm 3: AMPLE-NEXT_execute

```
1  $b \leftarrow$ bootstrap_execution($\mathcal{M}, psm, \boldsymbol{pr}$);
2  while stopPlannerRequested = false do
3      $a \leftarrow$ get_action($psm, \pi_d, \boldsymbol{\pi_{sr}}, b$);
4      Start execution of action $a$;
5      $\Delta_a \leftarrow$ expected duration of action $a$;
6      $prNext \leftarrow$ empty list of planning request pointers;
7      for $b' \in succ(b, a)$ do
8          $r.I \leftarrow b'$;
9          $r.\Delta \leftarrow Pr(b'|a,b) \times \Delta_a$;
10         add_plan_request($psm, \boldsymbol{pr}, r$);
11         $prNext.push\_back(r)$;
12     Wait until action $a$ has completed;
13     for $r \in prNext$ do
14         remove_plan_request($psm, \boldsymbol{pr}, r$,
               stopCurrentRequest);
15     $b \leftarrow$ observe and update current (belief) state;
```

AMPLE-PATH: reasoning about the most probable evolution of the system. The previous strategy lacks from execution-based long-term reasoning, even if the sub-(PO)MDP planner actually reasons about the long-term evolution of the system when 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 Algorithm 4. We first choose the depth for analyzing the (belief) state trajectory of the most probable path (Line 1), noted $pathDepth$. As in the `AMPLE-NEXT` strategy, we then bootstrap the execution (Line 2) and enter the "act-sense-plan" loop where we first get the action a to apply in the current execution state, begin to execute it and compute its expected duration Δ_a (Lines 4-6). Then (Lines 7-14), we successively apply the `get_action` procedure and get the most probable (belief) state at each iteration, starting from the current (belief) state b up to $pathDepth$. For each visited (belief) state of the explored trajectory, we add a planning request starting at this (belief) state with a maximum computation time proportional to Δ_a and to the inverse of $pathDepth$. The end of the loop (Lines 15-18) is identical to `AMPLE-NEXT`.

Algorithm 4: AMPLE-PATH_execute

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

Experimental evaluation

Random MDP problems

We first evaluated the `AMPLE` meta-planner on random MDP problems using both the `NEXT` and `PATH` execution processes. 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 optimization timelines until success (Figure 2). Each time slice of the execution (resp. optimization) thread corresponds to the execution of one action (resp. the optimization of one planning request).

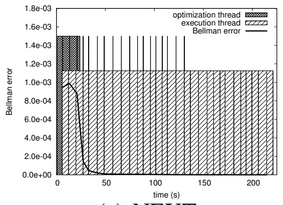
Figure 2(a) shows the timelines for the `AMPLE-NEXT` execution process. After a first bootstrap (where only the optimization thread is active), we can notice that the optimization continues for a few time. Then, small optimization pieces are still processed when new planning requests are sent to the 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.

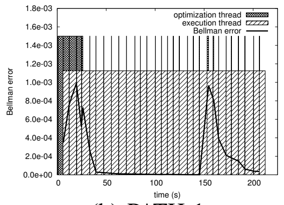
Figure 2(b) shows the timelines for the `AMPLE-PATH` execution process with a path depth of 1 (only the next most probable state is *requested*). Two behaviors are noticeable: first, the optimization process continues much longer than in the `NEXT` strategy; the later indeed explores a larger state space 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 optimization time. This phenomenon is emphasized for a path depth of 3 (Figure 2(c)) and 5 (Figure 2(d)), where replanning requests require a longer optimization time. However, we can notice that when the execution follows the most probable path, the optimization converges quite quickly (e.g., no more optimization pieces after time 100 on Figure 2(c)).

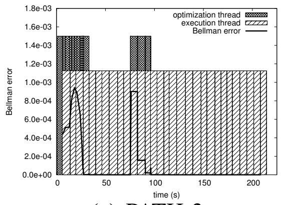
To conclude with, the `AMPLE-NEXT` process seems to provide a more convenient behavior with respect to problem optimization and mission execution. However, when the problem has a prevailing most probable path, the `AMPLE-PATH` execution process may be an efficient execution framework, with a fast optimization process, and online reactive repair phases when the system state leaves the most probable path. In the two next experiments, we then decided to only implement and test the `AMPLE-NEXT` approach.

Real-world probabilistic planning for autonomous rotorcrafts in unknown and uncertain environment

Emergency landing. The first UAV mission embedding the `AMPLE` framework consists in an autonomous emergency landing: the 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. Then, the `AMPLE` process is used to solve the landing problem: the UAV can go from a sub-zone to another, perform a scan of a sub-zone at a lower altitude to determine its probabilistic landability, and try to land on a given sub-zone. As both landability and action effects are uncertain, this problem is modeled as an MDP. To solve it, we use an `RTDP`-like algorithm (Barto, Bradtke, and Singh 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


(a) NEXT


(b) PATH-1


(c) PATH-3

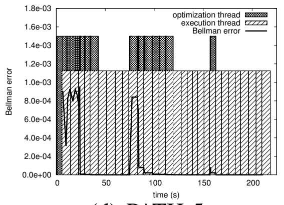

(d) PATH-5

Figure 2: Execution and optimization timelines for different execution strategies.

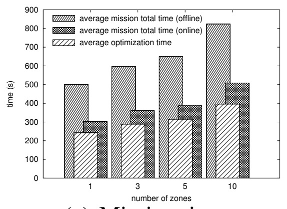
needed to optimize the policy with state-of-the-art MDP algorithms on a 1Ghz processor as the one embedded in our UAV is about 1.5 hours with 5 sub-zones, 4 months with 10 sub-zones, more than 700 millenniums with 20 zones; whereas the mission's duration is at most 10 minutes. Thus, time-constrained policy optimization like AMPLE is required to maximize the chance of achieving the mission.

Autonomous outdoor real-flight experiments have been conducted using the AMPLE-NEXT execution process. Data have been collected during the flights and incorporated in additional real-time simulations in order to statistically evaluate the performance of our planner (Figure 3).

Figure 3(a) shows the total mission time in two cases: the online AMPLE usecase, where the policy is optimized during execution, and an offline case that corresponds to first computing the policy, and then executing 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 get better than worst-case performances. The average optimization time is the time taken by the optimization 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 in 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 still succeeds. Note that mission times increase with the number of zones, but this is independent from the optimization framework: there are more zones to potentially explore, so that the overall actions take physically more times to be executed. Figure 3(b) 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 had not yet computed any policy in the corresponding state. Then, even if the default action is not optimal, it is the only way to guarantee reactivity, i.e. not wait for the optimization process to complete (which would lead to a "plan-then-execute" frame).

Target detection and recognition. The second UAV mission consists in a real target detection and identification mission, where the goal is to identify a particular car model among several cars, and land next to this target. Due to the nature of the problem, it is modeled as a POMDP, where the partial observability concerns the actual identity of cars. The UAV can act in the environment by moving between zones, changing height level or view angle in order to observe the cars, and land. The cars can be or not in any of the zones considered in the model (no more than one car per zone). The total number of states depends on the number of zones, the height levels and the car models. In this test case, we consider 5 possible observations in each state: {*car not detected, car detected but not identified, car identified as target A, car identified as target B, car identified as target C*}. The probabilistic observation model, which represents the uncertain outputs of the image processing based on the algorithm described in (Saux and Sanfourche 2011), was learnt during several outdoor test campaigns. Note that the POMDP problem is unknown before flight and thus must be solved online, because the actual number of zones is discovered at the beginning of the flight by using a dedicated image processing algorithm. We use a QMDP approximation as default policy (Littman, Cassandra, and Kaelbling 1995): although not optimal, it can be quickly computed at the beginning of the mission, once the zones are extracted from the map. As the formal model is a POMDP, the AMPLE's optimization thread handles the AEMS online POMDP algorithm (Ross and Chaib-Draa 2007). AMPLE was successfully tested during real flights: Figures 4 and 5 respectively show the UAV's global flight trajectory and the actual observation-action pairs obtained during the flight.

In order to analyze AMPLE'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 6 highlights the benefits of AMPLE compared with the classical approach of AEMS for different planning


(a) Mission times

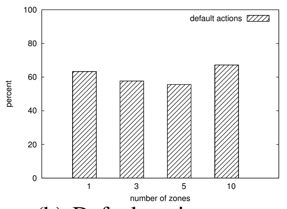

(b) Default actions rate

Figure 3: Autonomous emergency landing results.

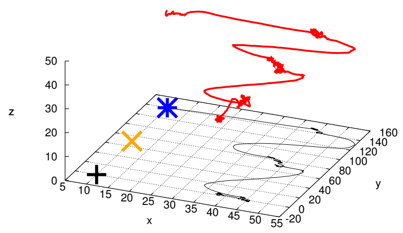


Figure 4: UAV's trajectory performed during the real flight.

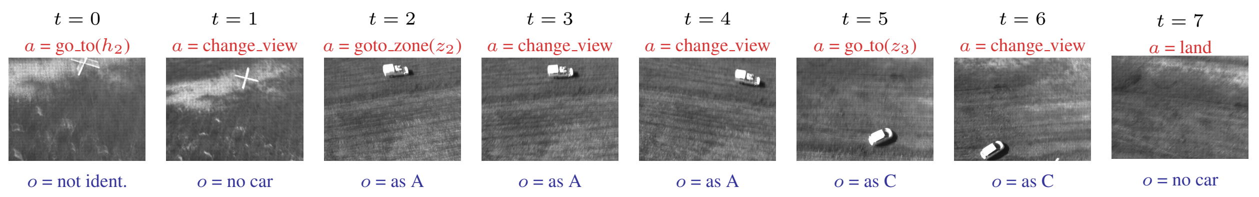
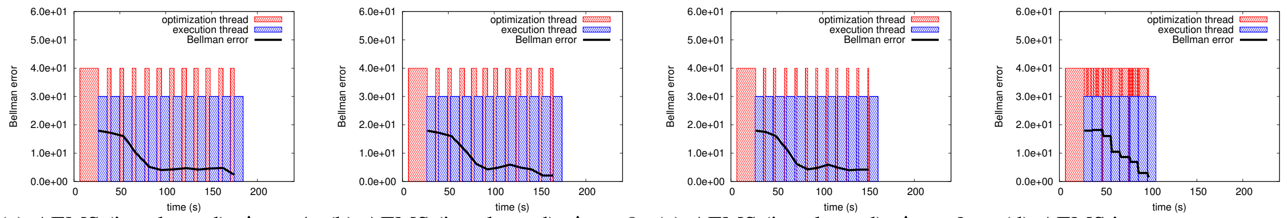


Figure 5: Sequence of decisions for time step t. Each image represents the input of the image processing algorithm after the current action a is executed. Observations o represent the successive outputs of the image processing algorithm's classifier.



(a) AEMS (interleaved), $\Delta = 4$s (b) AEMS (interleaved), $\Delta = 3$s (c) AEMS (interleaved), $\Delta = 2$s (d) AEMS in AMPLE-NEXT.

Figure 6: Timelines for classical AEMS (interleaved approach) with 4s, 3s, 2s for planning *versus* AEMS in AMPLE-NEXT.

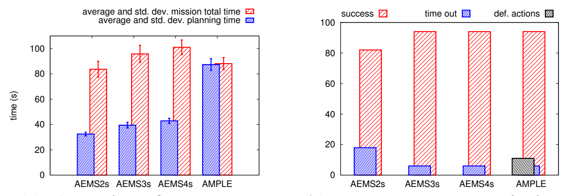
times (4, 3 and 2 seconds), which consists in interleaving planning and execution, i.e. it plans for the current belief state (for a long-term horizon) at every decision epoch, but not in advance for the future ones as in our *optimize-while-execute* approach. With the classical use of AEMS (Figures 6(a), 6(b) and 6(c)), we can easily notice that the mission's total time increases with the time allocated to plan from the current execution state. Successive red bars show that the POMDP needs to be (re-)optimized in each new execution state. On the contrary, our approach (Figure 6(d)) continually optimizes for future possible execution states while executing the action in the current execution state, so that optimized actions are always *immediately* available in each new 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 time saved relies on the sum of time slices of the classical approach when the optimization 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, a Yamaha Rmax adapted to autonomous flights, 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 Figure 7), AMPLE continually optimizes the policy in background, contrary to the interleaved approach. As a result, it is more reactive: it has the minimum mission's time, while providing the best percentage of success and the minimum number of timeouts. Note that, in Figure 7(a), AEMS2s performs better in averaged mission time (avg. over successful missions), but the percentage of successful missions is lower than in our approach (Figure 7(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.

Conclusion

In this paper, we have proposed AMPLE, a framework for anytime optimization of (PO)MDP problems, and anytime execution of the resulting policy. Contrary to existing (PO)MDP anytime algorithms, AMPLE's optimization thread locally updates the policy at *different future execution states*, by calling procedures of the underlying planning algorithm. The execution thread is in charge of managing planning requests and launching optimized actions according to the current system state. Based on this framework, we have proposed two possible instantiations: AMPLE-NEXT that anticipates all the next possible execution states, and AMPLE-PATH that anticipates the most probable path with a given depth. We have evaluated and shown the relevance of our approach on random problems and on two real and challenging UAV missions: AMPLE allows us to always complete the missions while a classical "optimize-then-execute" process would not succeed in the granted time.

The AMPLE framework can still be extended with new execution process instantiations. Possible instantiations could consider safety considerations, for instance by adding requests for states where the safety properties may be violated, or for very unprobable states that correspond to fault occurrences. Another interesting extension concerns the execution of policies in a reinforcement learning framework in such a way that learned actions are always safe and informative, which is often antagonist in practice.



(a) Avg. time for success. (b) Percent. among missions.

Figure 7: Target detection and recognition mission.

References

Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence Journal* 72:81–138.

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

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

Doherty, P.; Kvarnström, J.; and Heintz, F. 2009. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Journal on Autonomous Agents and Multi-Agent Systems* 19(3):332–377.

Fazil Ayan, N.; Kuter, U.; Yaman, F.; and Goldman, R. 2007. HOTRiDE: hierarchical ordered task replanning in dynamic environments. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

Finzi, A.; Ingrand, F.; and Muscettola, N. 2004. Model-based executive control through reactive planning for autonomous rovers. In *International Conference on Intelligent Robots and Systems (IROS)*.

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

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

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

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

Lemai, S., and Ingrand, F. 2004. Interleaving temporal planning and execution in robotics domains. In *AAAI Conference on Artificial Intelligence (AAAI)*.

Littman, M. L.; Cassandra, A. R.; and Kaelbling, L. P. 1995. Learning policies for partially observable environments: Scaling up. In *International Conference on Machine Learning*.

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

Pineau, J.; Gordon, G.; and 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*. New York, NY, USA: John Wiley & Sons, Inc., 1st edition.

Ross, S., and 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)*.

Saux, B., and 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)*.

Smith, T., and Simmons, R. 2005. Point-based POMDP algorithms: Improved analysis and implementation. In *International Conference on Uncertainty in Artificial Intelligence (UAI)*.

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