CENTRO PER LA RICERCA
SCIENTIFICA E TECNOLOGICA

38050 Povo (Trento), Italy
Tel.: +39 0461 314312
Fax: +39 0461 302040
e−mail: prdoc@itc.it − url: http://www.itc.it

Interleaving Execution and Planning for Nondeterministic, Partially Observable Domains

Bertoli P., Cimatti A., Traverso P.

May 2004

Technical Report # T04−05−02

Interleaving Execution and Planning for Nondeterministic, Partially Observable Domains

Piergiorgio Bertoli and **Alessandro Cimatti** and **Paolo Traverso**[1]

Abstract. Methods that interleave planning and execution are a practical solution to deal with complex planning problems in nondeterministic domains under partial observability. However, most of the existing approaches do not tackle in a principled way the important issue of termination of the planning-execution loop, or only do so considering specific assumptions over the domains.

In this paper, we tackle the problem of interleaving planning and execution relying on a general framework, which is able to deal with nondeterministic, partially observable planning domains. We propose a new, general planning algorithm that guarantees the termination of the interleaving of planning and execution: either the goal is achieved, or the system detects that there is no longer a guarantee to progress toward it.

Our experimental analysis shows that our algorithm can efficiently solve planning problems that cannot be tackled with a state of the art off-line planner for nondeterministic domains under partial observability, MBP. Moreover, we show that our algorithm can efficiently detect situations where progress toward the goal can be no longer guaranteed.

1 Introduction

Planning in nondeterministic domains under partial observability is one of the most significant and challenging planning problems. Several approaches have been proposed in the past [15, 18, 3, 2, 1]. However, the problem has been shown to be hard, both theoretically and experimentally, and building plans purely off-line still remains unfeasible in most realistic applications. Methods that interleave planning and execution, see, e.g., [13, 11, 17] are the practical alternative to the problem of planning off-line with large state spaces. In safely explorable domains [11], i.e., domains where execution cannot get trapped in situations where plans that lead to the goal no longer exist, it is possible to devise methods that are complete, i.e., that guarantee to reach the goal if there exists a solution, and that guarantee the termination of the planning/execution loop if no solution exists.

In this paper, we tackle the problem of interleaving planning and execution in the general case of nondeterministic domains and partial observability. We define an architecture for interleaving plan generation and plan execution, where a planner generates conditional plans that branch over observations, and a controller executes actions in the plan and monitors observations to decide which branch has to be executed. This extends an off-line approach to planning in nondeterministic, partially observable domains, based on symbolic model checking [2, 1]; the framework exploits the same data structures used in off-line planning to generate plans, to execute them and to monitor their execution. The plan generation component exploits a novel "embedded" algorithm that, at each plan generation step, generates a plan that, though possibly partial, makes a progress toward the goal. Clearly, this has the advantage that the algorithm does not necessarily need to consider all possible contingencies, and can thus scale up to large state spaces. In addition, the algorithm deals nicely with the case of non safely-explorable domains — i.e. where execution can get trapped in situations where no strong plan, guaranteed to reach the goal, exists anymore. Even though it is in general impossible to guarantee that a goal state will be reached, the embedded algorithm guarantees that the planning/execution loop always terminates: either the goal is reached, or it is recognized that a state has been reached from which there is no chance to find a strong plan [2].

We evaluate experimentally the proposed solution and compare the new planner with a state-of-the-art off-line planner based on symbolic model checking, MBP [2, 1]. The experimental results show that the new planner can scale up to much harder problems than the off-line techniques, and efficiently detect conditions where no progress can be guaranteed anymore toward the goal.

The paper is organized as follows. We first describe the problem of planning with nondeterminism and partial observability. We then formalize the framework for interleaving planning and execution, and discuss the embedded planning algorithm. Finally, we describe the experimental results and discuss some related work.

2 Planning with Nondeterminism and Partial Observability

A nondeterminstic, partially observable domain can be thought of as an automaton whose internal *state* evolves, starting from some initial value, on the basis of *actions* given as input to the domain. The domain provides *observations* whose values are related to the domain state, and which are the only mean for an external agent to acquire knowledge over the actual domain status.

Consider the simple robot navigation example presented in Fig. 1, upper left corner. A robot can move in a 2x2 room square, and is equipped with sensors that perceive the walls around it. The state of this domain consists of the robot position; the actions are the movement commands issued to the robot; the observations describe which is the current wall configuration around the robot. Notice that the observation may be not sufficient to achieve perfect knowledge on the domain status; for instance, since rooms NW and SW have the same wall configuration, it is impossible to distinguish whether the robot is in one or the other.

[1] ITC-Irst, Trento, Italy. E-mail: [bertoli,cimatti,traverso]@itc.it

[2] Notice that considering weak plans, that may not reach the goal, would remove the guarantee of termination. Nondeterministic behaviors of the domain might in fact cause endless planning/execution loops where weak plans always exist, and every time fail to reach the goal.

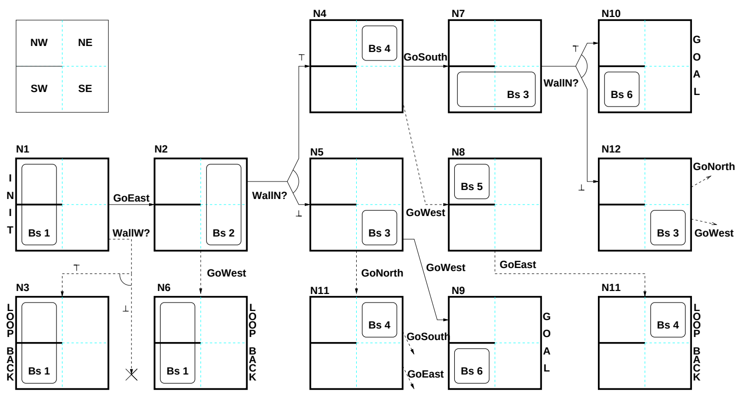


Figure 1. A simple robot navigation domain

The way actions evolve the domain status, and observations are related to the current status, can be described by means of nondeterministic functions; more formally:

Definition 1 *A* nondeterministic planning domain with partial observability *is a tuple $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$, where:*
• $\mathcal{S}$ *is the set of* states*.*
• $\mathcal{A}$ *is the set of* actions*.*
• $\mathcal{O}$ *is the set of* observations*.*
• $\mathcal{I} \subseteq \mathcal{S}$ *is the set of* initial states*; we require $\mathcal{I} \neq \emptyset$.*
• $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$ *is the* transition function*; it associates to each current state s and action a the set $\mathcal{T}(s, a) \subseteq \mathcal{S}$ of next states.*
• $\mathcal{X} : \mathcal{S} \to \mathcal{P}(\mathcal{O})$ *is the* observation function*; it associates to each state s the set of possible observations $\mathcal{X}(s) \subseteq \mathcal{O}$.*
Action a is executable *in state s if $\mathcal{T}(s, a) \neq \emptyset$; it is executable in a set of states b iff it is executable in every state $s \in b$. We require that in each state $s \in \mathcal{S}$ there is some executable action. Also, some observation must be associated to each state $s \in \mathcal{S}$, i.e., $\mathcal{X}(s) \neq \emptyset$.*

This model allows for uncertainty in the initial states and in the outcome of action execution. Also, since the observation associated to a given state is not unique, it is possible to model noisy sensing and lack of information.

In the domain of figure 1, the actions are `GoNorth`, `GoSouth`, `GoWest` and `GoEast`. We have four states, corresponding to the four positions of the robot in the room. It is in general possible to present the state space by means of state variables, where each state is presented by a truth assignment to the state variables. In the example, the state variables might be `E` and `S`. In state `NW` they would be both associated with a false value ($\perp$), while in `SE` they would be associated with $\top$. We have 16 observations, each corresponding to one of the possible configurations of walls around the robot. The space of observation can also be presented by means of observations variables, (e.g. `WallN`, `WallS`, `WallW` and `WallE`). Each observation associates a truth value to each observation variable. In the following, we will assume a variable-based presentation for $\mathcal{S}$ and $\mathcal{O}$. We define $\mathcal{X}_o[\top]$ to denote the set of states that are compatible with a $\top$ assignment to the observation variable o; $\mathcal{X}_o[\perp]$ is interpreted similarly. In partially observable domains, we consider plans that branch on the value of observation variables.

Definition 2 ((Conditional) Plan) *A plan for a domain $\mathcal{D}$ is either the empty plan ϵ, an action $a \in \mathcal{A}$, the concatenation $a; \pi$ of an action and a plan, or the conditional plan* **if** *o* **then** *π_1* **else** *π_2, with o an observation variable of the domain.*

For instance, **if** `WallN` **then** `GoSouth` **else** `GoWest` corresponds to the plan "if you see a wall north, then move south, otherwise move west".

A plan π is executable on a set of states b if b is empty, if $\pi = \epsilon$, or if one of the following holds:

- $\pi = a; \pi_1$, a is executable on b, and π_1 is executable on $exec(b, a) = \bigcup_{s \in b} \mathcal{T}(s, a)$
- $\pi =$ **if** o **then** π_1 **else** π_2, and the plans π_1 and π_2 are executable over $b \cap \mathcal{X}_o[\top]$ and $b \cap \mathcal{X}_o[\perp]$, respectively.

Intuitively, given a domain $\mathcal{D}$, a set of initial states I and a set of goal states G in $\mathcal{S}$, a plan π is a strong solution for the planning problem $\langle \mathcal{D}, I, G \rangle$ iff it is executable on I, and every execution on the states of I results in G [2].

During the execution of a plan, in general the executor has to consider a set of states which are equally plausible given the initial knowledge, and given the informations acquired through current and past observations. We call this set a *belief state*; it represents the current knowledge about the domain status. The search space then can be seen as an and-or graph whose nodes are belief states, considering that actions transform belief states into new belief states, and observations identify subsets of the current belief state. Search for a plan can be performed by visiting and-or graph representing the search space; given its size, a convenient "lazy" approach recursively constructs the graph from the initial belief state, expanding each encountered belief state by every possible combination of applicable actions and observations. The graph is possibly cyclic; in order to rule out cyclic plan behaviors, however, its exploration – at planning time – can be limited to the acyclic prefix of the graph.

The initial belief state in Figure 1 is $\{NW, SW\}$; our goal is to reach the condition $\{SW\}$. In the example, the actions are deterministic, with the exception of moving `GoSouth` from $\{NE\}$, which may cause the robot to slip in one of two states.

Figure 1 depicts a portion of the finite prefix of the search space for the described problem. The prefix is constructed by expanding each node in all possible ways, each represented by an outgoing arc. Single-outcome arcs correspond to simple actions (action execution is deterministic in belief space). For instance, N4 expands into N7 by the action `GoSouth`. Multiple outcome arcs correspond to observations. For instance, node N2 results in nodes N4 and N5, corresponding to the observation of `WallN`.

3 Interleaving Planning and Execution

Rather than searching the and-or graph of belief states off-line, taking into account all possible contingencies that can arise, we propose a framework where a *planner* searches the graph partially, and a *controller* executes the partial plan and monitors the current state of the domain. The process is iterated until the goal is (hopefully) reached. The top level algorithm for interleaving planning and execution, called PlanExecMonitor, is the following:

```
PlanExecMonitor($bs$, $G$)
1  if ($bs \subseteq G$)
2      return $Success$;
3  $\pi$ := ProgressivePlan($bs$, $G$);
4  if ($\pi$ = $Failure$)
5      return $Failure$;
6  else
7      $newbs$ := ExecuteMonitoring($bs$,$\pi$);
8      PlanExecMonitor($newbs$, $G$);
```

PlanExecMonitor is initially invoked by passing to it the set I of possible initial states, and the set of goal states G. We assume the domain representation to be globally available. PlanExecMonitor recursively implements a loop, that alternatively calls ProgressivePlan, which generates a plan, and the monitored executor ExecuteMonitoring, that executes the plan, and at the same time reports the new belief resulting from execution. PlanExecMonitor stops either when given a belief b such that the goal is known to be reached (that is, $b \subseteq G$), or when the planner returns failure.

```
ExecuteMonitoring($bs$, $\pi$)
1  MarkExecuted($bs$);
2  if ($\pi = \epsilon$)
3      return $bs$;
4  if ($\pi = \alpha; \pi'$)
5      Actuate($\alpha$);
6      $newbs$ := $exec(bs, \alpha)$;
7      ExecuteMonitoring($newbs$, $\pi'$);
8  if ($\pi =$ if $o$ then $\pi_1$ else $\pi_2$)
9      if CurrentValue($o$)
10         returnExecuteMonitoring ($bs \cap \mathcal{X}_o[\top]$,$\pi_1$);
11     else
12         returnExecuteMonitoring ($bs \cap \mathcal{X}_o[\bot]$,$\pi_2$);
```

The executor ExecuteMonitoring recursively applies the plan actions to the domain, via Actuate. The plan execution is driven by the observations in the plan: it branches over the actual observation values, retrieved from the domain via CurrentValue. Parallel to this, ExecuteMonitoring uses a domain model, namely $\mathcal{X}$ and $exec$, to propagate the initial belief consistently with the execution. Each belief state traversed during the monitored execution is marked as traversed via MarkExecuted. We call a sequence of beliefs traversed by the plan during its monitored execution a *run* of a plan; several runs are possible from a starting belief state b_0, depending on the behavior of the domain:

Definition 3 (Runs of a plan) *Let π be a plan for a domain $\mathcal{D}$. The set of runs of π from an initial belief state $b_0 \subseteq \mathcal{S}$ is inductively defined as follows.*
- *If π is ϵ, then b_0 is a run of π from b_0.*
- *If π is $a; \pi_1$, then the sequence b_0, r is a run of π from b_0, where r is a run of π_1 from $exec(b_0, a)$.*
- *If π is* **if** o $\mathbf{then}$ π_1 $\mathbf{else}$ π_2*, then the sequences b_0, r_1 and b_0, r_2 are runs of π from b_0, where r_1 is a run of π_1 from $b_0 \cap \mathcal{X}_o[\top]$, and r_2 is a run of π_2 from $b_0 \cap \mathcal{X}_o[\bot]$.*

```
ProgressivePlan($I$, $G$)
1  $graph$ := MkInitialGraph($I$, $G$);
2  while ( ¬isSuccess(getRoot($graph$)) ∧
3          ¬isEmptyFrontier($graph$) ∧
4          ¬(isProgress(getRoot($graph$)) ∧
5            TerminationCriterion($graph$)))
6      $node$ := ExtractNodeFromFrontier($graph$);
7      if (SuccessPoolYieldsSuccess($node$, $graph$))
8          MarkSuccess($node$);
9          NodeSetPlan($node$,RetrievePlan($node$, $graph$));
10         PropagateSuccess($node$,$graph$);
11     else
12         $exp$ := ExpandNode($node$);
13         ExtendGraph($exp$, $node$, $graph$);
14     if (¬isExecuted($node$))
15         MarkProgress($node$, $graph$);
16         PropagateProgress($node$, $graph$);
17 end while
18 if (isSuccess(getRoot($graph$)))
19     return ExtractSuccessPlan($graph$);
20 if (reachedTermination)
21     return ExtractPartialPlan($graph$);
20 if (isProgress(getRoot($graph$)))
21     return ExtractProgressingPlan($graph$);
22 return $Failure$;
```

Figure 2. The planning algorithm

4 Planning for Interleaving

Consider the planning algorithm depicted in Figure 2, disregarding the lines with boldfaced labels. This is a slight modification for strong planning under partial observability described in [1]. The algorithm takes as input the initial belief state and the goal belief state, and proceeds by incrementally constructing a finite acyclic prefix of the search space, implemented as a $graph$. In the graph, each node n is associated with a belief state $b(n)$; a directed connection between a node n_1 and a node n_2 results either from an action α such that $Exec(\alpha, b(n_1)) = b(n_2)$, or from an observation o such that $b(n_1) \cap \mathcal{X}_o[v] = b(n_2)$, with $v = \top$ or $v = \bot$. We call n_1 the father of n_2 and n_2 the son of n_1; we call "brothers" all the nodes that result from the same observation expansion of the same node. The graph is annotated with a frontier of the nodes that have not yet been expanded, and with a success pool, containing the nodes for which a strong plan has been found.

The algorithm has its core in a search loop (lines 2-21), iteratively selecting and expanding a node in the graph. Namely, at each iteration, a node is extracted from the frontier, and evaluated for success against the success pool (lines 6-7). If the node successful, a strong plan is extracted and associated to it, the success pool is expanded, and success is propagated backward on the graph (lines 8-10). Otherwise, the node is expanded by applying every executable action, and non-trivial observation to it, resulting into a graph expansion (lines 12-13). The expansion routine avoids generating ancestors of the expanded node, inhibiting the presence of loops in the graph. The search loop terminates either when (a) the root of the graph is signaled as a success node, (b) the graph frontier is empty, or (c) a termination criterion is met. Condition (a) signals that a strong plan has been found; condition (b) indicates that no strong plan exists; condition (c) is responsible for the search being stopped while only partial plans have been expanded.

Notice that the criterion defining condition (c) is the only distinction with the original off-line approach, for the purpose of integrating the planner within the interleaving framework, added to generate possibly partial plans rather than searching the whole search space. (Different termination criteria could be envisaged, e.g. partial success, number of nodes, run times. The specific details are not relevant here.) Unfortunately, this simple minded approach does not guarantee termination of the overall interleaving loop, even if a solution exists. The problem is that the planner should guarantee that, for every possible run, at least a new belief state is reached during execution. If this is not so, plan-execution loops are possible that keep visiting the same beliefs over and over, never terminating. If instead the guarantee is achieved, termination of plan-execution loops follows from the fact that belief states are finitely many. (Notice that weakening the condition, and accepting plans which only *might* result in beliefs never been visited before, does not guarantee termination.) The notion that guarantees the termination of the top level is what we call *progressiveness* of the planner: each plan must guarantee that at least one *belief state* (not just a state) is traversed that has not been previously encountered during execution.

Definition 4 (Progressive Plan) *Let r be a run $b_1, \dots, b_n$. Let π be a plan for $\mathcal{D}$. The plan π is progressive for the run r iff, for any run r_π of π from b_n, there is at least one belief state in r_π that is not a belief state of r.*

Let us consider again the statements in Figure 2 (with lines 20 and 21 bold, replacing the non-bold counterparts). In order to obtain a progressive planning algorithm, we consider all the plans originating from a node, to make sure that at least for one of them, execution will visit a belief state, which has never been visited during previous executions. The graph is therefore extended in order to maintain up-to-date information on progress of nodes. In order to guarantee progressiveness, at line 14, we check if $b(node)$ has already been visited at execution time. If not, we mark *node* as a "progress" node (line 15) (we remark that a successful node has surely not been visited by a previous run, and as such it is marked as progress). In that case, the progress information is recursively propagated bottom-up on the tree (PropagateProgressOnTree, line 16): if the node is the result of the application of an action, then its father is marked as progress. If the node is the result of an observation, in order to propagate its progress backward it is necessary to check that all of its brothers are also marked as progress nodes.

Finally, when the loop is exited, either a strong plan has been found, and is returned by ExtractSuccessPlan; or, a progressing plan exists in the graph, and is extracted by ExtractProgressingPlan; or, failure is returned. While extracting the success plan is simple (it is associated with I by the bottom-up propagation), the progressing plan might not be unique: several such plans may exist. The selection operated by ExtractProgressingPlan may affect the overall performance. Our implementation privileges, amongst progressing plans, the ones performing more observations.

5 Experiments

We implemented a planner called MBPp (Model Based Planner with Progressiveness) which extends the offline MBP [2] planner with the algorithm shown in Section 3, and is equipped with a simulator to trace executions. MBPp, just as MBP, relies on symbolic data structures to represent the search; these are based on Binary Decision Diagrams (BDDs), also exploited within planners such as UMOP [9].

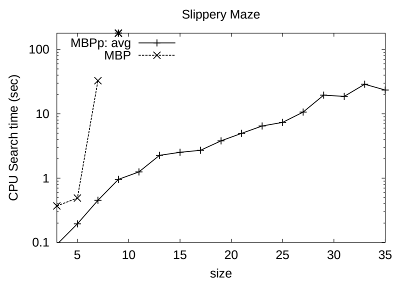
We compare the interleaved approach of MBPp with the state of the art (offline) MBP; comparison with UMOP is not possible since it does not handle partially observable domains. The tests were run on a Pentium III, 700 MHz with 6GB RAM, running Linux. The memory limit was set to 512MB, and CPU timeout was set at 180 sec. For each problem instance, we collect the planning time for MBP. For MBPp, the information is statistical, since its performance depends on the actual domain behavior chosen by the simulator: therefore, for each problem instance, 100 runs were generated, with initial states and nondeterministic outcomes selected randomly. We report average times for the total of MBPp planning and randomized plan execution.

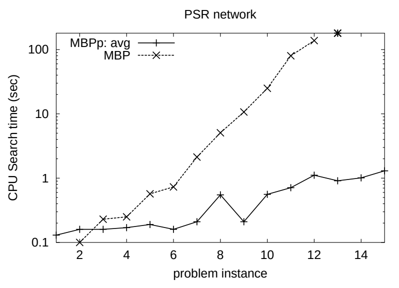
For our experiments, we considered three classes of experiments. The first is a robot navigation problem in a maze, with nondeterministic action effects. The robot may start at any position in the maze, and has to reach the top left corner. The robot may move in the four directions, and is equipped with reliable wall-presence sensors in the four directions. The robot may nondeterministically slip on the floor while trying to move, in which case it stays in the same position; this can occur at most 20% of the times. We consider a set of randomly generated mazes of increasing sizes.

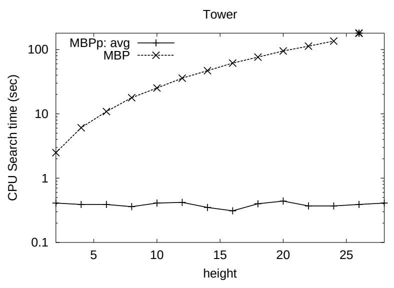
The second set of test cases is taken from the Power Supply Restoration (PSR) domain, recently proposed as a significant benchmark for planning under partial observability [16]. In PSR, a network of electricity supply lines, which can be reconfigured by turning (possibly unreliable) switches, are fed by a set of generators. Possible faults on the lines cause reopening of switches connected to generators. Direct observation of line faults is not available; it is only possible to detect, for each line, whether it has been de-connected due to a fault to lines below in the electricity flow. The PSR problem consists in feeding all possible lines within a given set, for a set of possible fault configurations, in spite of the limited sensing available. We consider a set of 16 PSR problems over a network featuring 3 generators, 5 switches and 6 lines. The problems have 0 to 3 unreliable switches, and four different set of possible faults configurations. We present the problems sorted by growing complexity.

Finally, we consider a robot navigation problem where the robot is in a cilindric tower of N floors, and can only move around the floor he's in. Inside any floor, every tenth room has a writing on the wall which indicates the floor number, and can be read by the robot. The robot has to reach a given room in floor 1; it starts not knowing the floor he's in, and is uncertain on the room also. We consider towers of increasing height, and 100 rooms per floor.

The average MBPp times and the MBP times for the three test cases are reported in Fig. 4. For the maze and PSR problems considered, a strong solution exist. However, due to the enormous number of contingencies introduced by nondeterminism (the robot slipping in the maze, faults and unreliable switches in the PSR), offline search becomes practically unfeasible for large/complex problem instances, while the on-line progressive search performed by MBPp scales up smoothly, dealing with contingencies as they arise. For the tower problem, the goal can never be reached: the robot might be in the wrong floor, and in any case it cannot detect precisely in which room it is situated. Once more, MBPp is able to effectively exploit the knowledge gathered by observations performed during executions, restricting the search and promptly discovering after a few runs that no progressive plan exists. The time spent to achieve this is basically independent of the domain size, since anyway after a few short runs, the robot gets to know the floor it is in, achieving the better knowledge possible given the situation. On the opposite, the larger the domain, the more MBP finds it complex to discover the absence of a solution. We experimentally verified that when the progressiveness check is disabled, MBPp fails discovering that no solution exists, and







does not terminate. In this setting, termination can only be achieved by adding a termination criteria that constraints the minimum length of the partial plans searched at each run to (at least) the number of rooms in a floor. On top of being problem specific, this strongly degrades the search, since it forces visiting off-line a vast portion of the search space at each planning run.

The experimental evaluation shows that the approach is very promising: despite being fully general, the interleaved approach by MBPP scales up much better, and is capable of efficiently dealing with very complex problem instances, that cannot be dealt by the state-of-the art planner MBP. The last experiment confirms the capability of MBPP to terminate when there is no more chance to find a strong solution to the planning problem. Regarding plan quality, we remark that neither MBP nor MBPP grant optimal (average) length of execution. Since both planners rely on the same node extraction heuristics, and progressiveness check only prevents looping behaviors, we do not expect relevant differences; however, a more accurate evaluation of this aspect is in our agenda.

6 Conclusions and Related work

The idea of interleaving planning and execution is certainly not new and has been around for a long time, see, e.g., [6]. In particular, some approaches exist that address the problem of interleaving planning considering nondeterministic domains and partial observability, defined similarly to here. Amongst those works, the most notable are [12, 13, 11], which propose different techniques based on real-time heuristic search. These algorithms rely on the existence of distance heuristics for the search space; in some case [11] they have the nice property that they can amortize learning over several planning episodes. These algorithms only guarantee termination (and reaching the goal) under the assumption that the domain is safely explorable, i.e. that the plan executor can never find himself in a position where it may be impossible to reach the goal due to unlucky nondeterministic action outcomes. By providing the notion of progressiveness, we are able to guarantee termination in a more unconstrained setting, where assumptions on safe explorability and admissible heuristic evaluations are not involved.

Other approaches, still based on real-time heuristic search, address the problem of planning in stochastic domains with probability distributions on action outcomes, like in POMDP (see, e.g., [3, 10, 4, 5]). Our technique is very different, and relies on symbolic model checking techniques and on an efficient, BDD-based representation.

The maze domain presented in the experiments is inspired by the work by Koenig ([11]), where it has been tested extensively in the problem of robot navigation and localization. However, the experimental domain of Section 5 is much harder than the one used in [11], which assumes that there is no uncertainty in actuation and sensing. It would be interesting an in depth experimental comparison in different domains of the two different approaches.

Somehow related to our work, even if very different in scope and objective, are the works that propose architectures for interleaving planning and execution, reactive planning and continuous planning, see, e.g., [14]. Among them, CIRCA [8, 7] is an architecture for real-time planning and execution where model checking with timed automata is used to verify that generated plans meet timing constraints.

REFERENCES

[1] P. Bertoli, A. Cimatti, and M. Roveri, ‘Conditional planning under partial observability as heuristic-symbolic search in belief space’, in *Proc. of ECP’01*, (2001).

[2] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso, ‘Planning in nondeterministic domains under partial observability via symbolic model checking’, in *Proc. of IJCAI 2001*, (2001).

[3] B. Bonet and H. Geffner, ‘Planning with incomplete information as heuristic search in belief space’, in *Proc. of AIPS’00*, (2000).

[4] A. Cassandra, L. Kaelbling, and M. Littman, ‘Acting optimally in partially observable stochastic domains’, in *Proc. of AAAI94*, (1994).

[5] T. Dean, L. Kaelbling, J. Kirman, and A. Nicholson, ‘Planning Under Time Constraints in Stochastic Domains’, *Artificial Intelligence*, **76**(1-2), 35–74, (1995).

[6] M. Genereseth and I. Nourbakhsh, ‘Time-saving tips for problem solving with incomplete information’, in *Proc. of AAAI93*, (1993).

[7] R. P. Goldman, D. J. Musliner, and M. J. Pelican, ‘Using model checking to plan hard real-time controllers’, in *Proc. of the AIPS2k Workshop on Model-Theoretic Approaches to Planning*, (2000).

[8] R.P. Goldman, M. Pelican, and D.J. Musliner. Hard Real-time Mode Logic Synthesis for Hybrid Control: A CIRCA-based approach. Working notes of the 1999 AAAI Spring Symposium on Hybrid Control.

[9] R. M. Jensen, M. M. Veloso, and M. H. Bowling, ‘OBDD-based optimistic and strong cyclic adversarial planning’, in *Proc. of ECP’01*, (2001).

[10] L. Kaelbling, M. Littman, and A. Cassandra, ‘Planning and acting in partially observale domains’, *Artificial Intelligence*, **1-2**(101), 99–134, (1998).

[11] S. Koenig, ‘Minimax real-time heuristic search’, *Artificial Intelligence*, **129**(1), 165–197, (2001).

[12] S. Koenig and R. Simmons, ‘Real-time search in non-deterministic domains’, in *Proceedings of IJCAI-95*, (1995).

[13] S. Koenig and R. Simmons, ‘Solving robot navigation problems with initial pose uncertainty using real-time heuristic search’, in *Proc of AIPS-98*, (1998).

[14] K. L. Myers, ‘Towards a framework for continuous planning and execution’, in *Proc. of the AAAI Fall Symposium on Distributed Continual Planning*, (1998).

[15] L. Pryor and G. Collins, ‘Planning for Contingency: a Decision Based Approach’, *J. of Artificial Intelligence Research*, **4**, 81–120, (1996).

[16] S. Thiebaux and M. O. Cordier, ‘Supply Restoration in Power Distribution Systems: a Benchmark for Planning Under Uncertainty’, in *Proc. of ECP-01*, (2001).

[17] S. Thiebaux and J. Hertzberg, ‘A semi-reactive planner based on a possible models action formalization’, in Proc. of *AIPS-92*, (1992).

[18] D. S. Weld, C. R. Anderson, and D. E. Smith, ‘Extending graphplan to handle uncertainty and sensing actions’, in *Proc. of AAAI-98)*, (1998).