Robots Predictive Execution Monitoring in BDI Recipes

Mika Barkan[1] and Gal A. Kaminka[1]

Bar Ilan University, Ramat Gan 5290002, Israel
barkanm1@biu.ac.il

Abstract. Execution monitoring allows robots to assess the execution of plans, determine the need for re-planning, identify opportunities, and re-evaluate their commitments. There exists extensive literature on monitoring the execution of classical and HTN plans. However, execution monitoring of BDI plans is often left implicit in the BDI control loop. In practice, many BDI plan execution systems monitor the current plan steps only. They do not project ahead the current knowledge of the robot to determine implications on future steps. Thus a failure of a future plan-step, which may already be predictable given the current knowledge of the robot, is not detected until the last possible moment. This paper examines the task of predictive execution monitoring in BDI plans. It provides a base algorithm, and shows that its complexity is super-exponential in the general case, even under mild assumptions. It then discusses several methods for pruning the search space, and formally shows their completeness. It evaluates these methods in hundreds of experiments, utilizing approximately 4000 hours of modern CPU time.

Keywords: Execution Monitoring · Hierarchichal Control · BDI.

1 Introduction

Robots do not only generate and choose plans for execution, they also monitor the execution of plans and handle contingencies [**?**,**?**,**?**]. The capacity for *execution monitoring* allows robots to assess the execution of plans, determine the need for re-planning, identify opportunities, and re-evaluate goal selection.

While there exists extensive literature on execution monitoring of classical and HTN plans (see Section **??**), execution monitoring of BDI plans is often left implicit in the BDI control loop. Specifically, the BDI control loop contains a *perception* step which updates the robot's beliefs, commonly followed by a *reconsider* step, where the robot re-evaluates its intentions and plans based on its revised beliefs [**?**,**?**,**?**]. However, the process by which a future step is re-examined with respect to current beliefs is generally unspecified.

In practice, many BDI plan execution systems focus only on the current plan step. They do not project ahead the current knowledge of the agent to determine implications on future steps[1]. Thus a failure of a future plan-step, which may

[1] Indeed, planning is also in generally an open challenge in BDI sytems [**?**].

already be predictable given the current knowledge of the robot, is not detected until the last possible moment. For example, consider a future Mars Rover which is equipped with a drill, and goes out to visit three sites in order, using the drill in the last one only. The drill breaks early on the way to the first site. In classic and HTN plan execution and monitoring systems (e.g., IPEM [**?**], partial-order plan-based controllers [**?**,**?**], SIPE [**?**]), the unsatisfiable requirement for a drill in the last site would be detected as soon as the drill breaks. However, most if not all existing BDI systems—which emphasize reactivity [**?**]—would only detect it upon arriving at the last site.

This paper examines the task of *predictive* execution monitoring in BDI recipes. Such capability is similar in principle to BDI planning [**?**,**?**,**?**], in the sense that both tasks require prediction of future world or plan states, based on simulation of actions taken. However, execution monitoring of recipes does not require ordering decisions: the sequence of steps is constrained by the structure of the recipe, and would seem to therefore require lighter computation. Alas, this is not the case.

We provide a base algorithm for predictive execution monitoring, intended for flat and hierarchical plan recipes. We show that its complexity is super-exponential in the general case. We then discuss several methods for reducing the projected execution space, and formally prove their completeness. We evaluate these methods in various combinations in hundreds of experiments, utilizing approximately 4000 hours of modern CPU time.

2 Background and Related Work

Execution monitoring is an important capacity in robots. The ability to monitor the execution of plans seems an implicit requirement, but has long since been recognized as a challenge in itself [**?**,**?**]. One general approach is *model free* monitoring. It relies on superficial models of execution (e.g., by contrasting execution times with those previously observed, or by means of anomaly detection) [**?**,**?**,**?**,**?**,**?**,**?**]. A different general approach uses the plan as a model, to set validity conditions to be checked during execution[**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**]. Our work is closer to the latter.

Specifically, we focus on *predictive* execution monitoring of hierarchical BDI recipes, which requires the robot to project its current knowledge forward in time, to simulate future execution paths and decisions with respect to contingencies. However, unlike plan-based monitoring, BDI recipes have only partial information about the effects of plan steps, and thus the number of possible changes that can occur grows super-exponentially, as we show.

Walczak *et al.* [**?**] augmented a BDI system with a simple state based planning. The BDI controller invokes the planner whenever the planes in the plan library are not sufficient to satisfy the goals. Our intention is not to create plan but use the already existing plans to estimate the current actions influence in the future regarding the information the agent already has. Their work can handle situation where a new plan needs to be created, in this situation our system will fail to achieve the goal.

BDI recipes are very often hierarchical. Thus execution monitoring of hierarchical plans is relevant. de Silva *et al.* has shown the close similarities between BDI systems and HTN planning [**?**]. In their work they compare the runtime of an HTN planner and BDI system in both static and dynamic environments, using the blocks world environment. Their work shows that the BDI system has better results both in the static environment and the dynamic one. However, the problems are created in a way that there is no need for an HTN-style lookahead (prediction). This is done since BDI does not have the capabilities to do so. In contrast, such capability for prediction is exactly what we seek to investigate.

Sardina *et al.* [**?**] used HTN planner to add lookahead capabilities to BDI for planing purposes. As in [**?**] the HTN planner derives its knowledge from the plan library of the BDI agent and its beliefs. The HTN planner is invoked and does a full lookahead search. If a plan is found then the BDI agent will follow it until goal is reached or until a step in the plan is no longer possible. Detection of such a failure occurs late. To address this, the algorithms we present attempts to provide early detection of failures.

de Silva and Padgham [**?**] proposed a mechanism for on-demand planning in BDI system. In their work, the programmer can specify places during runtime, where an HTN planner should be run. The planner derives its knowledge from the BDI goals and plan library, as well as the beliefs of the agent at runtime. Our work intend to use lookahead automatically without the programmers needing to do anything. However, in this paper we do not examine the question of selective execution of the monitoring system—instead we focus on its operation once invoked.

Belker *et al.* [**?**] used HTN planning to estimate the outcome of actions in navigation tasks. This in turn allows the agent to choose alternative actions (if available) that improve the projected outcome over the original chosen action, and results in a considerable performance improvement (42%). Encouraged by this, we seek to use predictions to improve the execution of BDI plans in general.

Tambe and Zhang [**?**] use predictive monitoring in the context of multi-agent teams, which work for long period of times and need to reason about future resource allocation, rather than just the resources needed for the immediate goal. They added lookahead capabilities to a BDI team execution system. In contrast, we use lookahead capabilities to reason about action choices (including their resources). However, our work focuses on single robots.

3 Predictive Recipe Monitoring

We start by clarifying our view of BDI plans and recipes. A recipe specifies multiple possible executions and orderings, and actions that are not fully instantiated (grounded). These are built to cover multiple possible contingencies, but are not a full pre-planned policy that covers every possible state. Indeed, BDI systems advocate runtime decision making and replanning if needed. Most BDI systems work by presenting the robot with a recipes that may be relevant to its task, and allowing it to choose how to ground the recipe and instantiate it so as to turn it into a concrete, grounded, *plan*. The process typically proceeds incrementally:

the robot instantiate, execute, and monitor only the current step in the recipe. Thus existing systems typically do not project ahead the current knowledge of the robot, and thus in practice monitor only the current executing subplan, and check that immediately following plan-steps are indeed selectable.

In contrast, we focus on *predictive* execution monitoring of hierarchical BDI recipes. The goal here is to detect branches in the recipe which can be predicted to fail under current conditions, as early as possible—well before the robot faces the opportunity to select them. This is difficult given that their grounding is not yet complete, and also given that BDI recipes have subtle, but critical, differences compared to hierarchical plans in HTN planing.

3.1 BDI Beliefs, Recipes, and Plans

A robot using BDI recipe has a knowledgebase of beliefs, which are revised and modified during the operation of the robot. For simplicity here, we think of beliefs as fluents, represented as tuples. Each belief is a tuple $\langle k, v \rangle$, where k is a unique key (the fluent name and parameters) and v is its value. The collection of all such tuples is the knowledgebase of the robot.

For us, a BDI recipe is an augmented connected directed graph, defined by a tuple $\langle B, H, N, b_0 \rangle$. B is a set of vertices representing *behaviors* (see below). $b_0 \in B$ is the behavior in which execution begins. H is a set of hierarchical *task-decomposition* edges, which allow a higher-level behavior to be broken down into lower level behaviors, until reaching a primitive behavior. N is a set of *sequential* edges, which constrain the execution order of behaviors: Given $b_1, b_2 \in B$, a sequential edge from b_1 to b_2 specifies that b_1 must be executed before executing b_2. Sequential edges may form circles, but hierarchical edges cannot.

Behaviors change the current beliefs of the robot (knowledgebase), and its state in the world (e.g., a command to move forward, changing its position in the world). Every behavior b is associated with the following: preconditions (preconds(b)), a set of beliefs that need to be true (in the knowledgebase) in order for this behavior to be selectable by the robot; termination conditions (termconds(b)), a set of beliefs that signal that execution of the behavior should terminate (typically, because of the achievement of the behavior goal, or its failure); and support keys (support(b)), a set of *keys* for beliefs whose value might be changed by the behavior.

For lack of space, we will not detail here the execution algorithm of a BDI recipe, but instead settle for a brief overview. The robot executes a recipe by matching its beliefs against the preconditions of behaviors, and selecting between matching behaviors for execution. A selected behavior logically allows one of its hierarchical children to be selected (if their preconditions hold), and so on until no child behavior is available whose preconditions match. Execution commences: the robot continually perceives the world, revising its beliefs, and matching them against the executing behaviors' termination conditions. If they match, execution of the behaviors stops, and the agent re-evaluates its goals (possibly choosing a new recipe), and the behavior selection begins anew, considering behaviors that can be reached via sequential edges.

We emphasize that while the BDI recipe structure above look similar to hierarchical plans (e.g., in HTN planning [**?**,**?**]), the definition of higher level behaviors is different. In HTN a *compound task* is not directly executed by the agent, but instead is decomposed into other tasks, such that actions are carried out only by primitive, non-decomposed tasks (the leaves of the HTN hierarchy). In contrast, here a higher level behavior is a program in and of itself, executed by the robot to affect change, in parallel to its task decomposition children behaviors. Thus after choosing a behavior and its decomposition, all the behaviors in the hierarchy work simultaneously. Indeed, it is entirely plausible that a higher level behavior has reached its termination conditions before its children. In this case, that behavior along with all its children (every behavior lower then it in the hierarchy) is stopped. This means that a behavior can be stopped before reaching its termination conditions. This type of layered-parallel execution is not as common in HTN planning systems, but quite common in robots.

3.2 Predicting Execution Possibilities

Predictive execution monitoring begins with (i) a recipe, (ii) the current execution state in the recipe (that is, which behaviors are currently selected), and (iii) the current knowledgebase of the robot. It then projects ahead, given the current beliefs of the robot, whether any future behaviors can be shown to be *un-selectable*, given potential changes to the beliefs of the robot, by behaviors preceding this future behavior, in the execution.

Algorithm Lookahead (Alg. **??**) searches *the space of possible recipe executions*. Each discrete point in this space is a combination of a valid path through the recipe graph (along hierarchical and sequential edges), coupled with the knowledgebase which holds at the end of the path. With each search iteration, the algorithm considers extending the path structurally. Each such expansion can involve multiple possible knowledgebase revisions. Thus each search iteration results in multiple discrete points in the search space, to be considered. We describe the process in detail below.

Searching Possible Future Executions. The algorithm proceeds by iterating over a queue of execution traces to be considered. Each element in the queue is a *search node* $\langle n, w, p, c \rangle$, where n is the current vertex in the graph, w is the current knowledgebase, p is an execution path (see below), and c is the expansion type to be considered. In each iteration, a new search node q is taken from the queue (line 6). If the vertex n associated with it is a leaf (structurally, has no outgoing edges and none of its parents has outgoing edges) then it is a possible termination of the execution, and the path leading to it ($q.path$) is added to the set of successful executions (lines 7–9). Otherwise, if all edges of the recipe graph are accounted for in the set of successful paths, then this means that no future behavior can be proven to be unselectable at this point, and thus the search can terminate (lines 10–12).

Algorithm 1 Lookahead

Require: The Recipe $P = \langle B, H, N, b_0 \rangle$
Require: Current behavior b_c
Require: Knowledgebase W
Require: A function to create the next states EXPAND
Require: A function to create the next states PRUNE

1: $Q \leftarrow$ EMPTYQUEUE()
2: $successful\_paths \leftarrow \emptyset$
3: $visited \leftarrow \emptyset$
4: ADD($\langle b_c, W, [\langle b_c, w \rangle], pre\_check \rangle, Q$)
5: **while** $Q \neq \emptyset$ **do**
6: $q \leftarrow$ POP(Q)
7: **if** CHECKIFLEAF($plan, q.n$) **then**
8: ADD($q.path, successful\_paths$)
9: **Goto ??**
10: **if** ALLEDGESCOVERED($plan, successful\_paths$) **then**
11: ADD($q.path, successful\_paths$)
12: **Goto ??**
13: $E \leftarrow$ EXPAND($q, P, Revise, Test$)
14: $E' \leftarrow$ PRUNE($E, visited, successful\_paths$)
15: **for all** $nq \in E'$ **do**
16: **if** $E' \notin Visited$ **then**
17: ADD(nq, Q)
18: ADD($nq, visited$)
19: $successful\_edges \leftarrow$ GET_EDEGES($successful\_paths$)
20: $failed\_edges \leftarrow (H \cup N) \backslash successful\_edges$
21: $newP =$ REMOVE($P, failed\_edges$)
22: **return** $newP$

The expansion of the search occurs in lines 13–18. First (line 13), the algorithm asks for the set E, all possible expansions of the current search node q, by structural and belief revisions. This set is then pruned (line 14) if possible, to reduce the number of such expansions (this key step is the subject of Section **??**). Then, the new nodes are put on the queue and marked as visited, so they do not get expanded again.

The process continues until the queue is empty (line 5), or all edges of the recipe graph are accounted for by successful paths (lines 10–12). It bears some similarity to a BFS search through a graph, however we note that unlike BFS, the search does not stop when we found a single path to a target behavior, but continues examining other paths, to other behaviors. Moreover, as we discuss in detail below, the presence of both hierarchical and sequential links, which carry different execution semantics (parallel and sequential, resp.) is also a significant challenge.

Execution Paths. Each search node q contains a valid *possible execution path*. This path records a potential execution trace (behaviors and beliefs), beginning

with the robot's beliefs and behaviors when Alg. **??** was called. The execution path contains a sequence of behaviors selected for execution by the BDI executive, in response to possible revisions to the knowledgebase, made by behaviors.

An execution path p is an ordered sequence of *execution elements*. This element is itself an ordered sequence of tuples $\langle b, w\rangle$ where b is a behavior and w is the knowledgebase in effect when b was selected. An execution elements represent one hierarchical decomposition of a behavior. Thus each b in a tuple is the child of the behavior directly preceding it. That child does not have to be a direct child (by hierarchical edge), but can be a sequential follower of a child. In this case w is the knowledgebase created after the termination conditions of the previous child. Thus the execution element does not just give us the structural decomposition, but also the changes of the knowledgebase during the parallel execution of lower level behaviors. For example if we have the recipe $\langle B, H, N, b_0\rangle$ where $B = \{b_0, b_1, b_2, b_3\}, N = \{(b_0, b_1), (b_3, b_4)\}, H = \{(b_1, b_3)\}$. A path in the recipe can be: $p = \langle b_0, w_0\rangle \rightarrow \langle b_1, w_1\rangle{\downarrow}\langle b_3, w_1\rangle \rightarrow \langle b_1, w_1\rangle{\downarrow}\langle b_4, w_4\rangle$. Thus path p above has 3 execution elements: $\{\langle b_0, w_0\rangle\}, \{\langle b_1, w_1\rangle{\downarrow}\langle b_3, w_1\rangle\}\{\langle b_1, w_1\rangle{\downarrow}\langle b_4, w_4\rangle\}$. We denote $last(path)$ to be the last execution element in this path (i.g $last(p) = \langle b_1, w_1\rangle{\downarrow}\langle b_4, w_4\rangle$). We also define subtraction between execution element of a path and a tuple in the path. The difference is the element until the last place the node appeared. For example if we take a path element of the form $e = \langle b_0, w_0\rangle{\downarrow}\langle b_1, w_0\rangle{\downarrow}\langle b_2, w_1\rangle$ and subtruct b_1 we get: $e\backslash\langle b_1, w_1\rangle = \langle b_0, w_0\rangle{\downarrow}\langle b_1, w_0\rangle$.

Simulating a Future Decision: Expanding an Execution Path. The role of the EXPAND procedure is to simulate the effects of all possible executions of a behavior. Given a search node q to expand, the procedure checks the expansion type specified in q, and generates new search nodes to be put on the queue (possibly after pruning). Each of these revises q in some fashion, in accordance with the execution logic described above, but without having access to a full model of the behavior. There are three possible expansion type (*PreCheck*, *TermCheck*, *InCheck*), described in detail below. We remind the reader that q contains the execution path p, the behavior n to be expanded, and the knowledgebase w assumed to hold currently.

(i) *PreCheck*: Select hierarchical child Given that n was selected for execution, one or more of its hierarchical children's may be selected for execution. The precoditions of all children behaviors (reached by following a single hierarchical edge from n) are tested against w. In actual execution, only one would get selected. But as we are simulating all possible executions, each possible matching child b_i would be a possible expansion of the current execution path. This is done by generating a new search node for each match: a node in which w is the same, but the execution path was amended to include $n{\downarrow}b_i$ at the end of the last element. Finally, the behavior n must also be expanded as it modified its beliefs during its own execution (remember, n runs in parallel to any child b_i). Thus a final new expansion duplicates the original node, but with the type of expansion set to *InCheck* (see below).

(ii) *InCheck*: Simulate revisions by the behavior. When n begins execution, it may directly revise the beliefs in w. A simulation of its execution requires us to predict such revisions. The behavior's *support keys* indicate the specific beliefs (fluents) whose values may change, though we do not know how (as we do not have *effects*, as in classical planning). We therefore expand the original search node by creating a duplicate, but with a revised knowledgebase w', where the value of the keys specified in support(n) is set to *unknown*. In addition, the behavior n may also terminate, and so we also set the expansion type set to *TermCheck*.

(iii) TermCheck: Simulate behavior termination. A final set of expansions of n simulates the effects of its termination. Algorithm **??** describes the process. It relies on two procedures: REVISE which generates a new knowledgebase w' from the existing w and a set of new beliefs, and TEST which carries out the matching of the preconditions of behaviors f against the revised w'. These same two procedures are used in the previous expansion types, but for lack of space we did not provide algorithms for the other expansion types, and thus did not explain them earlier.

When n terminates, then the termination conditions termconds(n) are true. Thus in any *TermCheck* expansion of q, new nodes must have a revised knowledgebase w' where the termination conditions are represented. In the common case where termconds(n) are arranged as a disjunction (i.e., any one condition may indicate termination), this means that each combinations of the beliefs in termconds(n) (loop, line 2) generates a new w' (line 3). In addition, there are two ways in which execution continues after n terminates. First, its parent may terminate given the new knowledgebase w' (line 4). Second, any behavior f that *follows* n (i.e., edge $(n, f) \in N$, loop in line 5) may be selected, should its preconditions hold in w' (line 6). Each f must replace n as the last executing behavior in the path, with knowledgebase w' (lines 7–9).

Note that as there are often multiple f, and given the combinatorial number of possible w', this expansion is where most search nodes are created and put on the queue.

The *TermCheck* expansion is where cycles are encountered, as cycles occur when a follower of n is either n or a behavior that precedes it in execution. We note that this type of expansion necessarily revises the knowledgebase; when n terminates, it is always with a revised w'. Thus re-expanding a behavior that has been expanded before is essentially valid, as it needs to be expanded with w'. As there is a combinatorial number of w', even a cycle from n to itself in the recipe graph can result in a combinatorial number of expansions to the earlier behavior.

Testing Unknown Values. The TEST procedure is in use in all the expansion types. Its task is to match (or test) a belief or a set of beliefs against a given knowledgebase W, returning true if the beliefs are in the knowledgebase. However, a complication arises. The *InCheck* expansion sets some beliefs in W to value *unknown*. How should a belief $\langle k, v \rangle$ with a known value v in a precondition

Algorithm 2 Expand TermCheck.

Require: Current search node $q = \langle n, p, w, c\rangle$
Require: The recipe $P = \langle B, H, N, b_0\rangle$
Require: Belief Revision Procedure REVISE
Require: Condition Testing Procedure TEST
1: $E \leftarrow \emptyset$
2: **for all** $t \in 2^{\text{termconds}(n)}$ **do** ▷ Disjunction? all belief combinations
3: $w' \leftarrow \text{REVISE}(w, t)$
4: $E \leftarrow E \cup \{\langle parent(n), w', p, TermCheck\rangle\}$
5: **for all** $\{f|(n, f) \in N\}$ **do**
6: **if** $\text{TEST}(\text{preconds}(f), w')$ **then**
7: $p' \leftarrow last(p)\backslash\langle n, w\rangle$ ▷ Remove n from end of execution path
8: $p' \leftarrow p + p'\downarrow\langle f, w'\rangle$ ▷ Add f sequential follower of n
9: $E \leftarrow E \cup \{\langle f, w', p', PreCheck\rangle\}$
10: **return** E

or termination condition be matched against a belief $\langle k, unknown\rangle \in W$ with the same key but value *unknown*. We propose two possibilities:

Optimistic Testing. Here, explicitly unknown values *pass* the test: $\forall v, \langle k, v\rangle = \langle k, unknown\rangle$. Thus, if there is a precondition that demands that some key k will have a value v, but instead $< k, unknown >\in W$ then the precondition holds. Trivially, we can see that optimistic testing gives us complete but not necessarily sound matchings: It never rules out a possibility unless there is no way for it to exist. Thus it never rejects possible matches, but may allow solutions that turn out to be false.

Pessimistic Testing. The inverse of optimistic testing—unknown values *do not pass* the test. By definition, $\forall v, \langle k, v\rangle \neq \langle k, unknown\rangle$. Trivially, it gives sound solutions, but is potentially incomplete. We found pessimistic testing to be ineffective in practice, since it almost invariably predicts complete plan failure within a few iterations of Algorithm **??**. In the experiments, we therefor use optimistic testing.

3.3 Complexity

We analyze the run-time complexity of Algorithm **??**. Let us denote $deg^-_S(b)$, $deg^+_S(b)$, $deg^-_H(b)$, $deg^+_H(b)$ the sequential in-degree of b, the sequential out-degree of b, the hierarchical in-degree of b and the hierarchical out-degree of b, respectively. We start by examining the number of execution paths for a *simple recipe*, which is really just a set of behaviors arranged linearly in a linked-list type of structure. No cycles, no hierarchical children, no choices about order of execution.

Definition 1. *A plan* $G = (B, H, N, b_0)$ *is a simple recipe when* $\forall b \in B, deg^-_S(b) = deg^+_S(b) = 1, deg^-_H(b) = deg^+_H(b) = 0$ *and each behavior has no support keys.*

Theorem 1. *Let P be a simple recipe with $|B| = n$ where $n \geq 2$ and each $b \in B$ has t termination conditions. P has at most t^{n-1} search paths.*

Proof. Let us prove by induction: Base case, $n = 2$: Behavior b_0 TermCheck expansion function will produce the search nodes $\forall t \in 2^{\text{termconds}(b_0)}, \langle b_0, \textsc{Revise}(w_0, t), \langle b_0, w_0 \rangle, PreCheck \rangle$. Notice that each Revise call produces a different knowledgebase, due to the different termination conditions. Expand will return at the most all t nodes, that is if no termination condition contradicted a precondition. Thus we have $t^{n-1} = t^{2-1} = t^1$ expanded nodes on the queue at the most.

Induction step: Assume $n = k - 1$, and show true for k: Let us have a simple recipe p with k nodes, let us denote the last behavior in the recipe b_k and the only directed edge to it be (b_{k-1}, b_k). We make a new recipe $p_{k-1} = (B_p \backslash b_n, H_p = \emptyset, N_p \backslash (b_{k-1}, b_k), b_0)$. We know be the induction that p_{k-1} has at the most $t^{k-1-1} = t^{k-2}$ search paths already in the queue. This means that if we add the behavior b_k to the end of p_{k-1} then the expand function will be could for all t^{k-2} search nodes again because now b_{k-1} has a sequential follower. For each such call the expand will produce t expanded nodes with the path $\langle b_0, w_0 \rangle \rightarrow ... \rightarrow \langle b_{k-1}, w_{t_k - 1} \rangle \rightarrow \langle b_k, w_t \rangle$ thus at the most, if termination condition do not contradict, we have $t \cdot t^{k-2} = t^{k-1}$ new nodes in the queue.

Indeed, the simplest recipe graph with one path of sequential links has an exponential number of possible execution paths, due to the combinatorial explosion in the combination of termination conditions. If we expand this graph to have more then one sequential in-degrees and more sequential out-degrees then we will have this for each path in the graph. A directed acyclic graph (DAG) has combinatorial number of paths. For each path of length m we will have this complexity, this mean that even when we look at a graph plan that only has sequential edges and no cycles, we get an super-exponential worst case run time: a combinatorial number of paths, each generating a combinatorial number of execution paths to be considered in Algorithm **??**. Of course, when we add hierarchical edges (which allow more complex paths), and when we allow cycles in our sequential edges, the runtime is exacerbated even further.

4 Pruning Possible Executions

We explore three different pruning methods, which cut the search space of possible executions.

4.1 Successful Visited

When a path p from successful paths already contains a tuple where the current vertex (from the expanded) with the same knowledge-base has been shown to be successful, then there is no use checking from the current node forward (the tuple from the successful path shows us that from this point on, the issue is resolved). Thus the only new information in the expanded tuple is in the prefix path, leading to the current node, which may be new. If so, we save it.

Successful visited derives from the successful paths list a set of successful visited $Successful\_visited = \{\langle n, w\rangle | \forall p \in Successful, \langle n, w\rangle \in p\}$. For each new search node $s = \langle n, w, p, c\rangle$ the method checks if $\langle n, w\rangle \in Successful\_visited$. If this is true then s is pruned and p is add to successful paths.

Algorithm 3 Successful visited.

Require: Successful paths list S
Require: List of expanded nodes E
1: $E' \leftarrow \emptyset$
2: $S_{nkb} \leftarrow \{\langle n, w\rangle | \forall p \in S, \langle n, w\rangle \in p\}$
3: **for all** $e \in E$ **do**
4: **if** $\langle e.n, e.w\rangle \in S_{nkb}$ **then**
5: ADD$(e.p, successful_paths)$
6: **else**
7: ADD(e, E')
8: **return** E'

Theorem 2. *Alg* **??** *with successful visited pruning and optimistic testing is complete.*

Proof. Let us assume for contradiction there is a path p that is feasible but was not returned by the algorithm. *We know from the completeness of optimistic testing* that without pruning we will explore all feasible paths. Thus we know that it was not returned by pruning. This means there is a vertex n and a knowledgebase w that where pruned. Notice that since a path p' in successful paths contains a node with the knowledgebase we started with, then if $\langle n, w\rangle$ was part of a search node that has PreCheck type of Expand then we already covered its expansion, with a different path. However we add this prefix path to successful paths, thus we have the path in successful paths. We then have to account for InCheck and TermCheck type of Expand. Notice that a node with *InCheck* type still has the same w so we have the same logic as before. We then need to look on search nodes with type check TermCheck. Notice that when $revise(w, \text{support}(n)) = w'$ then the only difference between w' and w are unknown values, this can only increase the number of nodes expanded, since unknown always satisfies the precondition. This means that if we started with w then we get more possibilities then ending with w. Thus TermCheck with w will at the most produce the same expansion has with w'.

4.2 Cycle Detection

A cycle in a search graph is when we reach the same vertex again. In our case, since search node also includes the path, and there are cycles in the recipe graph simply comparing the search node is not enough. Thus, to make sure the algorithm only goes in cycles through the graph until there is no new information

to gain from the cycle, we use cycle detection. Cycle detection prunes search node $s = \langle n, w, p, c\rangle$ if $\langle n, w\rangle \in p \backslash last(p)$.

This is possible since the search node $\langle n, w, p', c\rangle$, where p' is the part of p until the first occurrence of $\langle n, w\rangle$, was already explored and led to this search node. Thus $\langle n, w\rangle$ is already expanded.

Theorem 3. *Cycle detection with optimistic testing is complete.*

Proof. We saw in the proof for successful visited that if we have a path the includes the tuple $\langle n, w\rangle$ means we checked all the possible paths from then on starting with this knowledgebase. This is the case here as well, the only difference is that we are checking if the tuple is in the same path of the search node. For that reason we get that indeed if $\langle n, w\rangle \in (p - last(p))$ then one of this possibilities led us to the tuple again, but it also explored all the other paths, thus we already have the feasible suffixes explored.

Algorithm 4 Cycle Detection.

Require: List of expanded nodes E
1: $E' \leftarrow \emptyset$
2: **for all** $e \in E$ **do**
3: $p_{nodes} \leftarrow$ GET_NODES$(e.p \backslash LAST(e.p))$
4: **if** $\langle e.n, e.w\rangle \notin p_{nodes}$ **then**
5: ADD$(e, E's)$
6: **return** E'

4.3 Merging paths

In successful visited we tried to prevent making checks if there is already a proof of success. The problem was that we needed to succeed first. Until we succeeded for the first time we continued to expand search nodes that produced the same results. We need to prevent this.

We observe that the role of the path p in each search node is to maintain information about which edges we can keep in our new plan. However, if a path leads to the same behavior with the same knowledgebase and same type of expand, then the checks from there on will be the same. So a new search node duplicating this check need not be added to the queue.

To prevent duplication, we add a map allowing us to record search nodes which are already on the queue. The *keys* of the map are tuples $\langle n, kb, c\rangle$ where n is a behavior vertex in the recipe graph, kb is a knowledgebase when we reached n, and c is the expand type. We need the expand type so that different expanded nodes will not eliminate the next expand of different type. The *value* of each key is a set that holds all the paths that leads to the key tuple. For each search node q we check if its tuple $\langle n, w, c\rangle$ exist in the keys. If not then we add that tuple

to the map and also the path to that keys corresponding set of paths. If it exist then we add the path to the set of paths corresponding to this key and do not add the search node to the queue. In the end of Algorithm **??**, if the keys pair $\langle n, w\rangle$ is in any of the successful paths then we add the edges of all the paths in the set corresponding to the key, to successful paths.

We note that there is one kind of search node that is different then the others. That is the search node created in line 4 of algorithm **??**. This search node is different since it is the only search node in which its knowledgebase is the one we are *ending* with, while in all the other search nodes the knowledgebase is the one we had before they started. We cannot treat this node like the others. Specifically it does not correspond to tuples in our path elements, which are tuples of vertex and the knowledge we started with. For this reason we marked this search nodes and did not add them to the map, and they where not pruned. Rather we rely on the following vertex selection to account for a knowledge they already encountered.

This map saves us doing the same checks again for different prefix of paths and eliminates the multiplication by number of paths in the complexity of the problem, since we are merging paths. This saves not only doing successful checks again, has with the successful visited, but we also only go through a suffix of a path that fails or succeed only once.

Algorithm 5 Merging Paths.

Require: Map of expanded nodes M
Require: List of expanded nodes E
1: $E' \leftarrow \emptyset$
2: **for all** $e \in E$ **do**
3: **if** $\langle e.n, e.w, e.c\rangle \in M$ **then**
4: ADD$(e.path, M[\langle e.n, e.w, e.c\rangle])$
5: **else**
6: ADD$(\langle e.n, e.w, e.c\rangle, e.p, M)$
7: ADD(e, E')
8: **return** E'

Theorem 4. *Every path that is feasible will be added to successful paths.*

Proof. Let us assume there is a path p that is feasible and not in successful paths. This means there is a recipe graph vertex n in the path that was the last that we reached with w and expand check c, and was pruned. Let us denote the prefix of p until n as px. Notice that the path of a search node does not effect the graph nodes we choose to expand, only saves us the path that got us here. For that reason for two search nodes $s = \langle n, w, px, c\rangle, s' = \langle n, w, px', c\rangle$, there expansion will be the same with the only different being the path in the expansion node. We saw that optimistic testing is complete thus we know that from n, w onward if there is a path to a leaf we will traverse it and declare it a

successful path. Thus if only s' was put on the queue, then the path that will end in successful paths will be $p + s$ where s is a feasible path from n to a leaf when we started with w. We then have a path in successful path that contains $\langle n, w \rangle$ thus we have the suffix of p from n onward in successful paths. In addition because we have a successful path with the tuple $\langle n, w \rangle$ in successful paths, all the prefix paths that led to this tuple, who are kept in the map, are added to successful paths. One of this prefixes is px. Thus px is in successful paths. We get that all the edges of p are in successful paths, so p is in successful paths. In contradiction to our assumption.

5 Experiments

We seek to empirically evaluate two independent issues. First, the influence of the graph structure and the influence of the knowledge state space size (as reflected by the number of termination conditions used in behaviors), on the actual complexity of the execution algorithm. Second, we seek to evaluate the efficacy of the different pruning methods we introduced.

5.1 Experiment Environment

We ran our experiments on randomly generated recipes. The recipes where generated with 3 parameters. The first is the depth of the recipe graph, that is the height of the tree from the initial node to the lowest leaf. We will denote depth with d. The depth we choose are $d = 1, 3, 5$. The second parameter is the breadth of the tree, that is how many children are in each level of the tree. We will denote breadth with b. The breadth we choose are $b = 1, 3, 5$. For example a recipe graph with depth 1 and bread 2 is a recipe graph that has the initial node and this node either have 2 hierarchical children or 1 hierarchical child, and the child has one sequential follower that is not itself. Table **??** shows the number of behaviors that each such recipe graph has. Note that BDI recipes from significant research efforts appearing in the literature report on having a behavior count somewhere in the range of a few dozen [**?**] to well over a hundred [**?**,**?**], i.e., similar numbers to $d = 3, b = 3, 5$ in the experiments.

	b=1	b=3	b=5
d=1	2	4	6
d=3	4	40	156
d=5	6	364	3906

Table 1: Number behaviors in a recipe graph

The last parameter is the max number of termination conditions each node can have, we will denote it with t. The max termination conditions we choose are $1, 3, 9$. For each combination of d, b, t we generated 5 different recipe graphs. The knowledgebase we decided to go with has 10 keys with boolean values (True or False). Thus we have 2^{10} possible knowledgebases to start with. We choose randomly 5 of this knowledgebases to start five different runs on the same recipe

graph. This means that for every combination of d, b, t we have 25 runs. In total we ran the algorithm $25 * 3 * 3 * 3 = 675$ times *for each pruning method*. The runs were carried out in parallel, on a 24-core XEON server with $76G$ RAM. Each run was a single process, utilizing a single core. Overall, we used more than 4000 hours of CPU time for the experiments.

Since we know the time to run the algorithm for each problem can be very long we decided to restrict the time for each run with different knowledgebases to one hour of CPU time. We first ran the base algorithm without pruning on this problems but even smaller recipes timed out, even with 3 hours of CPU time given to them to run. We thus focused on the pruning method. We ran the 5 chosen knowledgebases on each of the examples with the pruning methods and their combinations, that is: merge paths (M), cycle detection (C), cycle detection and successful visited (C+S), merge path and successful visited (M+S), merge path and cycle detection (M+C) and all 3 pruning methods together (ALL). Successful visited without some sort of cycle detection or merge path proved to be as bad as the base algorithm and thus we decided to run it as part of a combination of pruning methods.

5.2 Recipe Graph Structure

Let us first discuss the recipe graph's structure influence on the run time complexity. For this purpose we set t to be 1 and looked at the changes in run time when changing the depth and breadth. The result are in Figure **??** and Figure **??**. Both figures has 9 graphs. Each graph corresponds to a different d and b combination. In the first figure, each bar represent the total runtime of all 25 runs of the algorithm with a given pruning method. In the second we have the number of recipe graphs (out of the 25), for which the algorithm finished within the 1 hour cutoff time. Note that the Y axis in the run-time figures changes scale between subfigures, sometimes dramatically.

We can see in these figures that if the breadth and depth are small, the run time is fast and all the runs reach the end, since we have less behaviors and less paths to go through. On the other end if we have a lot of behaviors (in this case 3906) then the time complexity is very high and very few recipe graphs actually finish before the 1 hour cutoff. One important conclusion from these figures is that breadth has more influence on the time complexity then depth. The jump in time from $d = 1, b = 1$ to $d = 5, b = 1$ is small, on the other hand the jump from $d = 1, b = 1$ to $d = 1, b = 5$ is tenfold. In addition we can see in Figure **??** that even though the jump in the number of behaviors from $d = 3, b = 3$ to $d = 5, b = 3$ is significantly bigger then the jump to $d = 3, b = 5$, the number of recipes that finished is not. Thus we can say that the breadth of the recipe graph is more influential than the depth of the recipe graph.

In addition we can see in Figure **??** that cycle detection does worse then the rest. It is the only one that did not menage to finish runs on all the recipe graphs in $d = 3, b = 3$, and when others start to fail, it fails more times. Thus, cycle detection total runtime jumps significantly more then the others in the corresponding graph in Figure **??**. Successful visited with cycle detection does

slightly better then cycle detection alone, but not by much. On the other end Merge Paths and all its combinations, finish faster and thus also finish more recipe graphs in the hour given. We can see that Merge Path by itself never has longer runtime then any combination with it. This is because it already incorporates the two methods in it. We can see that any combination of merge paths and a different method finish exactly the same number of recipe graphs. The longer running time of its combination can thus only be explained by the fact that we do more operation per iteration, since we are doing one or two more methods.

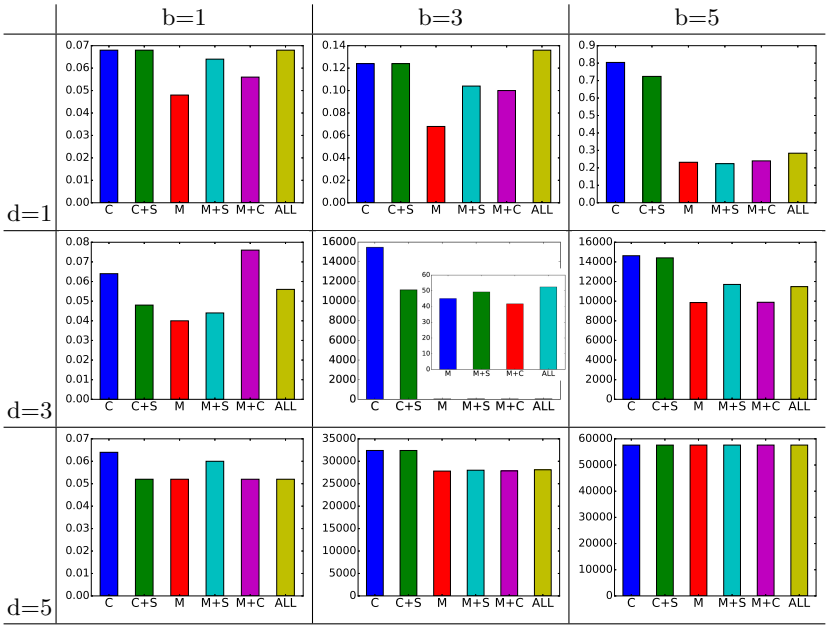


Fig. 1: Total runtime for (t=1) (Lower is better)

5.3 Knowledge State Space Size

To understand the influence of the knowledge state space size we look at the total running time when b and d are fixed and instead vary the number of termination condition per behavior. This result can be seen in Figures **??**, **??** and **??**. In each, we see the results of all tested combinations of pruning methods, for a given breadth (b) and depth (d) but varying t $(1, 3, 9)$. Figure **??** is the total run time on plans with $d = 1$ and $b = 5$. Figure **??** is the total run time on plans with $d = 3$ and $b = 3$. Figure **??** is the total run time on plans with $d = 3$ and $b = 5$.

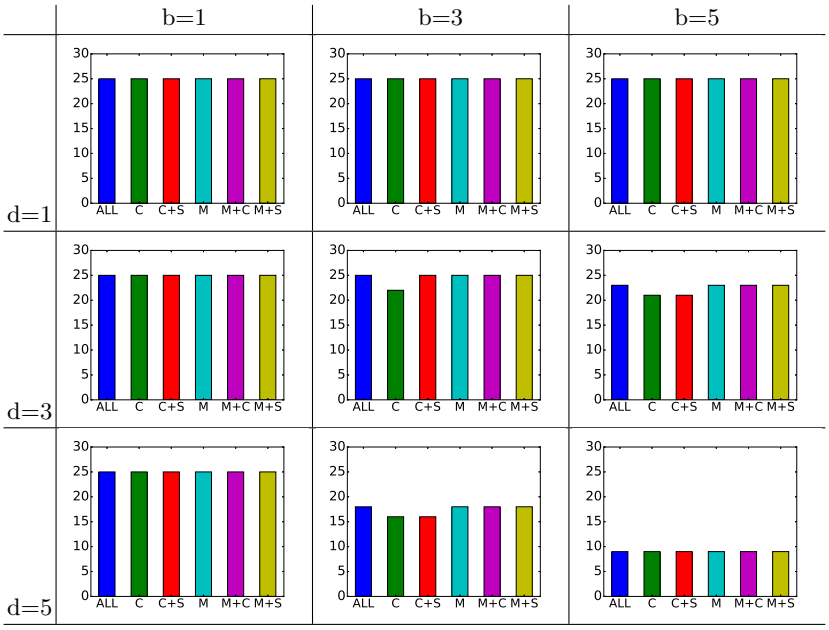
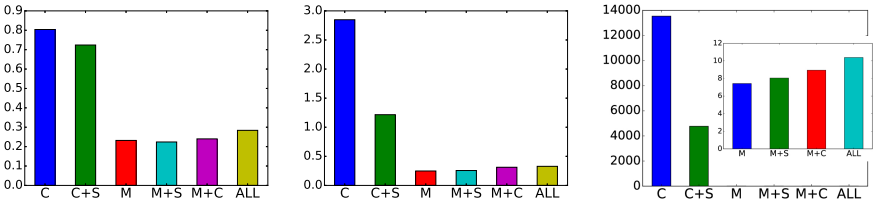


Fig. 2: Number of finished runs when (t=1) (higher is better). Note the scale on the Y axis changes dramatically between subfigures.

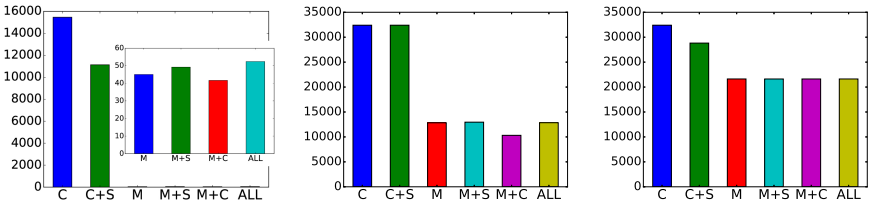
Notice that the scale of the graphs increases when t increases. This is in agreement with the complexity we found in Section **??**, where we saw that the number of termination conditions for a behavior, increases the number of possible exploration options. Thus we can conclude that the more active keys we have, the time complexity increases. This means that the complexity of the problem is not only dependent on the number of behaviors in the recipe. Even small recipes with large number of termination conditions can take a very long time to solve.

Another conclusion we can see from this graphs is that Merge Paths is better then Cycle Detection and Successful Visited. Even more surprising, the combination of all pruning methods together does not improve the running time, and sometimes even increases the runtime. The same can be said for Merge Paths with successful visited. Notice that Merge Paths is an improvement on Successful Visited, since it does not wait for a path to reach the end, rather prevent the double checks from happening even before that. Thus the runtime of Merge Paths with Successful Visited can only increase because of the overhead of running the pruning method itself. Merge Paths with Cycle Detection does slightly better then Merge Paths alone, but not by much. This means that the best method to use is Merge Paths with Cycle Detection.



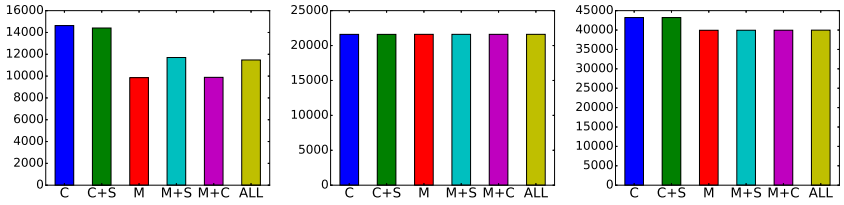
(a) Total runtime for (t=1) (b) Total runtime for (t=3) (c) Total runtime for (t=1)

Fig. 3: Total runtime for (d=1,b=5) (Lower is better)



(a) Total runtime for (t=1) (b) Total runtime for (t=3) (c) Total runtime for (t=9)

Fig. 4: Total runtime for (d=3,b=3) (Lower is better)



(a) Total runtime for (t=1) (b) Total runtime for (t=3) (c) Total runtime for (t=9)

Fig. 5: Total runtime for (d=3,b=5) (Lower is better)

6 Conclusion

In this paper we presented the problem of predictive execution monitoring of BDI recipes for robots; the ability to project forward the current knowledge of the agent to prevent future failures that can already be predicted. We then analyzed the complexity of the problem and showed that even on simple acyclic flat recipe graphs it is a super-exponential problem. We presented an algorithm that goes over the branches of the BDI recipe graph to try and find the branches that are predicted to fail. We then presented pruning methods to make the algorithm more efficient and reduce the running time. We proved that this pruning method are complete.

We then showed experimental results, run over multiple BDI recipes for more then 4000 CPU hours. These experiments showed that the runtime complexity is not directly effected by the number of behaviors in the recipe graph, rather by the structure of the behaviors in the recipe graph and the edges that connects them. In addition, the problem is effected by the size of the knowledgebase and the number of beliefs the recipe changes from this knowledgebase. We also saw that all the pruning method improve the running time, but that there is one better then the others, that is Merge Paths. We saw that combining Merge Paths with the other two pruning method is not always beneficial, and in most cases using all the pruning methods together is even detrimental.