Bar Ilan University

Predictive Execution Monitoring for Layered Hierarchical Recipes

Mika Barkan

Submitted in partial fulfillment of the requirements for the
Master's Degree in the Department of Computer Science
Bar Ilan University

Ramat Gan, Israel 2019

This work was carried out under the supervision of Prof. Gal A. Kaminka. Department of Computer Science. Bar Ilan University.

This work was carried out under the supervision of Prof. Gal A. Kaminka. Department of Computer Science. Bar Ilan University.

ACKNOWLEDGMENTS

First of all, I would like to express my sincere gratitude to my advisor, Professor Gal A. Kaminka, for his support of my research, his patience, knowledge, enthusiasm, and encouragement. He was always accessible and willing to help me with my research and general life question as well - and for that, I am very grateful.

Secondly, I also would like to thank my lab colleagues, both from MAVERICK and SMART labs, who supported me and entertain me in the most difficult moments.

Last, but definitely not least, I would also like to thank my family. My grandfather for encouraging me to get the high education, my parents for always believing that I am capable of anything and my brothers for listening to all my problems with patience and love.

Contents

English Abstract

Execution monitoring allows agents to assess plan execution progress, determine the need for re-planning, identify opportunities, and re-evaluate their commitments. While there exists extensive literature on execution monitoring of classical and HTN plans, monitoring of *layered hierarchical recipes* is typically myopic, discovering failures late in the execution, even if a failure of a future step may already be determined given the current knowledge of the agent. This thesis examines the task of predictive execution monitoring in layered hierarchical recipes. 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 to determine what nodes where visited thus reducing the search space, and formally shows their completeness. Then we explore how using the results of previous calls to execution monitoring can help reduce the time to execute it again. It evaluates these methods in hundreds of experiments, and on a NAO robot.

List of Figures

List of Tables

List of Algorithms

1 Introduction

Agents do not only generate and choose plans for execution, they also monitor the execution of plans and handle contingencies and opportunities [41, 39, 15]. The capacity for *execution monitoring* allows agents to assess the execution of plans, determine the need for re-planning, identify opportunities, and re-evaluate goal selection.

There exist many techniques for execution monitoring (see Section 2 for details). *Model-free*, anomaly-detection methods utilize behavioral expectations—e.g., action duration—to monitor for failures [9, 4, 40, 33, 31, 13, 30]. *Model-based* monitoring methods use the *plan as a model*, identifying validity conditions to be checked during execution [1, 5, 49, 10, 47, 27, 28, 29, 26, 50, 3, 24, 36]. These model-based methods are currently limited to agents utilizing classical or HTN plans.

However, agents and robots in dynamic settings often use *recipes*, not plans, to guide their actions[41]. Recipes encode procedural knowledge, and succinctly represent multiple potential execution trajectories, with no commitment to complete grounding nor total ordering of actions until execution. Notably, while recipe actions have *preconditions* (which check applicability just as in planning), they do not have *effects* (which allow projection of state changes). Instead, recipe actions have *termination conditions* that convey partial information, allowing the agent to decide to terminate the action, but not the full state of the world upon termination.

This is a source for the succinctness of recipes, as each recipe succinctly represents many potential execution trajectories. It also makes it easy to dynamically compose them, e.g., by sequencing. The control loop contains a *perception* step which updates the agent's beliefs, commonly followed by a *decision* step, where the agent decides on the next action to take based on its revised beliefs (given the recipe). It may also revise its plans (re-plan or attempt plan repair), or reconsider its goals. The process by which a future step is re-examined with respect to current beliefs is generally unspecified.

In practice, this often leads to *myopic execution monitoring* and decision-making [17, 46]. To facilitate reactivity, practical recipe execution systems test current beliefs against preconditions of possible current actions allowed by the recipe. But they generally do not test future subsequent actions against this belief: First because the partial action models do not facilitate straightforward simulation of the combination number of future world states and second, because there are combination number of paths through

the recipe proceeding from a certain behavior. As a result, decisions are made *late* (when a current belief is tested in a current condition), instead of *early* (when a current belief is tested in a future step). This characterizes most forms of behavior-based control and procedural plan execution systems used in robots [23], and many *BDI* (belief, desire, intention) architecture implementations [43, 16, 21].

For example, consider a robot sent to fix a loose screw in a room containing drawers. It is given a recipe for the task: (1) if you do not have a screwdriver, go pick one from the tool-shed; (2) if you are not next to the drawers, navigate to room A, then B, then C where the drawers are located; and then (3) if you have the screwdriver, use it to tighten the screw. Each of these steps includes many sub-steps(see Figure 1), which may or may not be executed depending on the position of the robot and contingencies (for instance, the robot may need to stop to cool off its motors[1]). With myopic monitoring, if the robot drops the screwdriver during navigation, it will not immediately go back to the tool-shed, as navigation only stops when room C is reached. Only then will the robot test whether it is holding a screwdriver. Had the robot been following a classical or HTN plan, it could have utilized predictive monitoring for detecting this as early as possible. However, no such methods exist for recipes.

This paper examines the task of *predictive* monitoring in recipe execution. Such capability is related in principle to BDI planning [44, 48, 8], in the sense that both tasks require prediction of future world states, based on simulation of actions taken. However, predictive monitoring does not require making ordering decisions, as the order of steps is constrained by the structure of the recipe. Monitoring would seem to therefore require lighter computation; alas, such is not the case.

We provide a base algorithm for predictive execution monitoring for flat and hierarchical plan recipes. We show that its complexity is super-exponential in the general case. We then discuss how to implement the *visited test*, an important part of graph search algorithms that usually curtails the complexity and make sure the same node will not be searched twice. First, we show two different methods to compare our search node and analyze this method in terms of completeness (returning ***all*** feasible execution paths) and whether the algorithm will halt with them.

We explore visited methods that will return **simple** execution paths, a

[1]This really happens with the NAO used in the experiments.

path where all vertices are distinct, instead of all feasible paths. We then explore how to use the information already collected in previous calls to monitoring, and show that repeated calls can use this information to reduce the time execution monitoring takes. We evaluate these methods in various combinations in hundreds of experiments, utilizing approximately 4000 hours of modern CPU time. We show the algorithm in action, with a recipe for an actual robot, and show the effect of the algorithm on the general run time in real life scenario.

The baseline lookahead algorithm proposed in this paper adds predictive monitoring capabilities to layered hierarchical recipes by searching the recipe . The lookahead algorithm traverses this recipe and finds all paths that are feasible according to the current knowledge. The agent will calls this lookahead algorithm before choosing the next behavior. The decision of which behavior to use next will be done with consideration of which paths were deemed feasible by the lookahead.

2 Background and Related Work

Plan execution monitoring is a broad and active area of research [14, 1, 10, 47, 50, 39, 15, 3, 37]. The capacity for *execution monitoring* allows agents to assess the execution of plans, determine the need for re-planning, identify opportunities, and re-evaluate goal selection.

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 other means of anomaly detection) [9, 4, 40, 33, 31, 13]. While broadly useful, model-free monitoring only provides general indications of failures, and does not guide the execution towards deeper diagnosis of the causes.

Other execution monitoring methods utilize a model of the agent and its interactions with its environment. For example, Khalastchi and Kalech [30] survey model-based fault detection and diagnosis (FDD) methods. First they survey different characteristic of robotics systems and analyze which FDD are appropriate for the varying degrees of each characteristic. They show that most execution monitoring of this crucial deliberation (decision-making) step is done using the plan as a model, showing different method using the plan as a model.

Bouguerra et al.[3] proposes execution monitoring using the models of the domain, rather than the plan, to detect plan execution failures. There work focus on ways to determine if an outcome of an action is as expected by using domain knowledge to derive implicit exception of the actions outcome from the explicit expectation. They first present an algorithm that gives a boolean outcome, that is whether the action failed or succeeded and then a probabilistic outcome giving probabilities to the possible outcomes. Once an action is determined to have failed then they re-plan using the current knowledge with consideration of the probabilities of the outcomes in the probabilistic outcomes. Their work still relies on the effects, or probabilities on multiple effects, of an action. Our work differs in a few ways: first we do not seek to determine if the outcome of our action is has expected, in fact rely only on a partial knowledge of the changes an action can have on the environment. Neither we have probabilities on those possible changes. In addition we are not only considering whether an action failed, but whether there were additional changes to the state of the world regardless of the success of the action that will never-the-less affect future steps in the recipe. For example if the action taken was the moving of a robot from one location

to another and then dig a hole in that location, then the robot can very well succeeded in the action and reach the location but if its shovel was broken on the way the plan can no longer go has planned.

Cohen, Amant and Hart[5] analyze the trade off between early detection of failure and detecting a failure when there is none (false positive). They analyze results from a slack time envelopes, decision rules that warn of plan failures, for path planning. They show a way to build such envelopes where the rate of false positive is lower, this envelopes are based on distance remaining to the goal. The trade off between early detection on false positive is determined by the slack, a period of no progress that they permit at the beginning.

Howe and Cohen[22] uses analysis of execution traces of a planner to better improve future iterations of the planner. They show that statistic analysis of the multiple execution trace can give us interesting patterns and determine if they are significant. Then, using domain knowledge, the patterns are interpreted to produce better planners (or plans).

Plan-based execution monitoring methods use the plan as a model. It extracts validity conditions to be checked during execution. These conditions must be met for the plan to remain valid. If they fail, the agent may revise its plan, reconsider its goal, or re-plan [14, 1, 49, 47, 10, 35, 50, 37, 36, 30]. Some previous work is focused on the integration of execution and monitoring [14, 1, 10, 49].

Veloso, Pollack, and Cox [47] and later Pollack and McCarthy[35] discuss monitoring of plans in dynamic environments when the planing stage is still separate from the execution stage. They address the changes to the environment occurring in the planning stage, this stage can be over hours or days and thus the environment can indeed change. They propose mechanism that monitors the features of the state of the world that were instrumental in the plan construction and effected the decision of choosing one action over anther, i.e preconditions or usability condition (conditions that the robot cannot change but can still effect the plan construction). The planner monitors such features and if they change then it can trigger a re-planning. Our work focuses on preforming monitoring during execution stage.

McIlraith and Beck[37] proposed algorithms for execution monitoring of partial ordered plans. In there work they define conditions for partial ordered plan viability in a given state (in STRIPS language). They then describe how to check the viability of the partial ordered plan using regression from the goal to the current plan. They seek but one linearization of the plan, an ordering

of the action in a specific sequence, if such exist then the partial ordered plan is viable. They then analyze there methods by running it on different domains from IPC. In there analysis they too encounter the exponential explosion of the complexity once parallel execution of action is allowed.

Predictive execution monitoring of recipes requires the agent to project its current knowledge forward in time, to simulate future execution paths and decisions with respect to contingencies. [48] used a state-based planner to generate totally-ordered execution paths, one at a time, as the basis for BDI (belief, desire intention) recipes. In contrast, our work is concerned with eliminating paths which can be deemed a failure, but still leave the decision making to be reactive at the moment of choosing without committing to the full execution path.

We focus on hierarchical recipes, somewhat similar to HTN (hierarchical task network) plans [18]. We therefore survey related work in HTN planning.

Earl, Handler and Nau [12] formalize Hierarchical Task Network (HTN) planning. HTN has three types of tasks. *Goal tasks* represent what needs to be true in order for the goal to be achieved. *Primitive tasks* correspond to actions and their effect on the world. *Compound tasks* represent higher level tasks that can be achieved with a sequence of other lower level tasks (i.g having a house can be done either by building a house or buying a house). The tasks are connected with a *task network* that give the tasks, their order and conditions to accomplishing this task network. HTN planning builds a task network that achieves the given goal task. Erol et al. gives a procedure for HTN planning that is both sound and complete.

HTN total-order planning with variables (closest to our setting) is in 2-EXPTIME [11]. We show below that the number of execution paths grows exponentially, even in flat recipes, with no cycles and no alternatives to be considered. This is due to the partial description of actions in recipes (see below).

Belker, Hammel and Hertzberg [2] 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 layered hierarchical recipes in general, not only in navigation.

HATP (Hierarchical Agent-based Task Planner) is an HTN planner built to suit robotics problems [32]. It includes a domain representation language

for that purpose. This domain representation language is built in a way that allows to represent different agents in the environment and distinguish between them and objects. This allows for planning for several agents, though synchronization during execution is still needed. As HATP is an HTN planner it includes HTN style lookahead capabilities. However, HATP does not include the execution of the plan or the synchronization between the agents. This work intends to use lookahead during execution and not create a plan in advance. We intend to use the knowledge gained in the lookahead for decision making in real-time.

BDI is a well-known architecture used for incorporating deliberation with execution in agents . BDI has also been used in robot agents . However, BDI does not have a built in mechanism for planning. Lookahead and planning are desired capabilities in decision making. It can give additional and sometimes even critical information when choosing the next action.

BDI recipes are also very often hierarchical. Thus execution monitoring of hierarchical plans is relevant to BDI systems. [6] have shown the close similarities between BDI systems and HTN planning . In their work they compare the run-time 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.

Tambe and Zhang [46] 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. Each member of the team calculates the expected utility of an action suggested to be selected. This action is specifically a coordination action between the different team members. In contrast, we use lookahead capabilities to reason about action choices (including their resources) and see future implication not just for a certain action but all actions.

De Silva and Padgham [7] proposed a mechanism for on-demand planning in BDI system. In their work, the programmer can specify places during run-time, 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 run time, relying on the similarities between BDI systems and HTN planning. Our work uses 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.

Walczak *et al.* [48] 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 plans 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.

Sardina, de Silva and Padgham [44] used HTN planner to add lookahead capabilities to BDI for planing purposes. As in [6] 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. They make the assumption "that agents are coherent—only the environment or other concurrent intentions may make the failure condition of a goal-program true"[44, p.5]. When dealing with robots we cannot make this assumption, the outcome of an action is not always guaranteed. To address this, the algorithms we present attempts to provide early detection of failures.

Sardina, de Silva and Padgham [8] worked on integrating first principle planning in BDI. They created an algorithm for creating hybrid planes. These plans included abstract operators, which can be mapped to BDI goals. This allowed the BDI system to choose a plan from the plan library, whenever such operators existed in the created plan. In that manner they are using BDI plan selection and failure recovery. Our system still cannot create such new plan. We are focusing first on using knowledge inherited in the existing plans, to drive the best course of action from the possibilities given.

De Giacomo, Patrizi and Sardina [20] devised a technique to create a controller for a goal behavior from available behaviors. In their work behavior stands for any artifact that can operate in the environment, this means that different robots can be represented as behaviors. Their technique creates something they call "controller generator", that represents all possible compositions. This is similar to a recipe that represents the different ways of achieving a goal. De Giacomo *et al.* indeed show that composition problem are more similar to a generalized type of planning for maintaining a goal. Thus their technique efficiently deals in a non-deterministic environment,

and maintaining goals. Their work creates the controller from the set of available behaviors, unlike ours that receives the recipe form the programmer. However, they assume fully observable world, something that is rarely true in the robotics field.

Ramirez, Yadav and Sardina [42] showed that solving behavior composition problem, is akin to finding "a strong-cyclic plan for a special fully-observable non-deterministic planning problem"[42, p.180]. They solve this using a technique that dynamically define a goal to reach, that then allows the plan to be executed infinite times to maintain the goal. While as with [20], they create a plan, rather than get it from outside source, this technique is still dealing with a fully observable world, as with maintaining a goal. Our algorithm deals with a partially observable goal, and is intended to work without knowing whether the robots goal is maintain a certain goal or simply achieve a goal.

3 Recipes

We start by clarifying our view of recipes. A recipe encodes procedural knowledge. It 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, systems utilizing recipes are built to detect recipe failures, and are built to replan if needed. Recipe execution systems work by presenting the agent 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 grounded plan. The process typically proceeds incrementally. The agent instantiates, executes, and monitors only the current step in the recipe. It does not project ahead the current knowledge of the agent, and thus only checks the preconditions of immediately-following plan-steps.

In contrast, we focus on *predictive* execution monitoring of hierarchical 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 agent faces the opportunity to select them. This is difficult given that their grounding is not yet complete, and also given that recipes have subtle, but critical, differences compared to classical or hierarchical plans.

3.1 Beliefs, Recipes, and Plans

An agent utilizing recipes maintains a knowledge-base of *beliefs*, which are revised and modified during the operation of the agent. Beliefs are 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 agent. For simplicity, we consider values of $\top$, $\bot$, and ? (for *true*, *false*, and *unknown*).

A recipe is an augmented connected directed graph, defined by a tuple $\langle F, B, H, N, b_0 \rangle$. F is a set of keys and their possible values (i.e., the space of fluents that may appear in the knowledgebase). 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 values of beliefs in the knowledge-base, and its state in the world (e.g., a command to move forward, changing its position in the world). For every behavior b, we have preconditions (preconds(b)), a set of beliefs that need to be true in order for this behavior to be selectable by the agent; termination conditions (termconds(b)), a disjunction of beliefs that signals 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.

An example for a recipe, for a robot trying to fix the drawer from section 1 can be found in Figure 1. The recipe works as follows, we start by assessing the location of the robot, if the robot is in the start point then it will move forward using the behavior from_init, otherwise it will choose the behavior of face_west to turn until the robot is facing west, that is because the tool shed is the farther west point in the robots environment. After going to the tool shed the robot will pick a screwdriver if it is not holding one already, otherwise it will move towards the resting point (if the robot does not rest then its motors warm up and it collapses). Notice that reaching the resting point can only be achieved by first visiting the tool shed, thus reaching the resting always follows the same steps, turning east then moving forward until reaching the resting point and then resting. The termination condition to leave the resting point is if the robot rested. After resting the robot moves to the drawer location, which is located at the farthest east point in the robot environment, and then moves forward. If the robot is already facing east then it will just move forward. Notice that all the behaviors are children of initiate. Initiate is given as b_0 for this recipe. In this paper we used recipes that have a root behavior and a end behavior, in this case mission_completed, that was done so we will always have a behavior that signify the success of the goal.

We emphasize that while recipes may look similar to HTN plans [19, 12], 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 agent 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

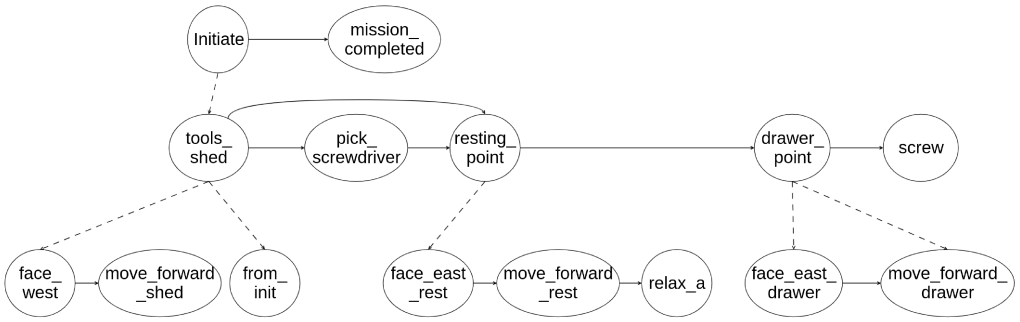


Figure 1: A recipe for a robot fixing the drawer. Dashed lines are hierarchical edges (H) while solid lines are sequential edges (N). Nodes are behaviors (B).

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 agents and robots. This type of layered-parallel execution is used in Soar [38], BITE [25], and many other systems.

Another important difference between layered recipes and HTN plans is that recipes do not assume that the agent is the only cause of change in the world (i.e., the environment is static). Indeed, two types of conditions (for both preconditions, or termination conditions). An *external* condition tests a belief that may change regardless of the robot action. In contrast, an *internal* condition tests a belief whose value may change based on an action (behavior) of the robot. A condition that is both internal and external is by definition internal.

To illustrate, a robot waiting for a stop light to change cannot do anything to change it. Thus a condition testing whether a stop light shows green is an external condition. In contrast, a robot capable of moving can change its position. A condition testing the current position is internal[2].

[2]As an aside, we note that internal conditions can be identified by the presence of their keys in the support keys of a behavior or its children.

3.2 BIS algorithm: Executing layered recipes

The BIS algorithm (Alg. 1) executes layered recipes of the type described above. The algorithm receives a recipe, a choosing mechanism for behaviors, procedures to revise the knowledge base and test conditions. The algorithm is based on BDI architect, and interleave between planning and execution. The agent 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 agent 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.

When the agent starts (by selecting the behavior b_0 for execution line 3), it first engages in hierarchical decomposition, until a complete hierarchical path through the behavior graph—b_0 through hierarchical edges, to an atomic behavior b_a is selected (lines 4–8). This path is added to the stack. During this process, the preconditions of sub-plans (which begin execution chains) are matched against the world model, to determine whether subplans are selectable. The robot chooses among alternative decompositions. Execution of the selected behaviors then commences (lines 9–10). The termination conditions of all running behaviors are continuously matched against the world model, which is itself continuously updated by the perceptual processes, and optionally by the behaviors themselves writing to internal state variables (lines 12–15). Once one or more of the behaviors signals it is ready for termination, its execution stops, as well as that of its running children (lines 16–20). The robot then chooses from enabled behaviors next in the sequence (if any)(line 24), or goes back to the parent behavior which is still running (line 11).

Algorithm 1 BIS

Require: Plan $P = \langle B, H, N, b_0 \rangle$
Require: Knowledebase W
Require: Choice Procedure CHOOSE
Require: Condition Testing Procedure TEST
Require: Belief Update Procedure UPDATE
Require: Belief Revision Procedure REVISE
Require: Start Execution Procedure START
Require: Stop Execution Procedure STOP

1: $S \leftarrow \emptyset$ ▷ New execution stack
2: $b \leftarrow b_0$
3: PUSH(S,b)
4: **while** $\exists n$, where $(b, n) \in H$ **do** ▷ b can be decomposed
5: $A \leftarrow \{n | (b, n) \in H\}$ ▷ children of b
6: $C \leftarrow \{a | a \in A, \text{TEST}(\text{preconds}(a), W)\}$
7: $b \leftarrow \text{CHOOSE}(C, P, S, W)$ ▷ Choose among C behaviors
8: PUSH(S,b)

9: **for all** $s \in S$ **do** ▷ Start execution of all behaviors in S
10: **if** s **not running** START(s)

11: $E \leftarrow \emptyset$
12: **while** $E = \emptyset$ **do** ▷ E is the set of terminating behaviors
13: $K \leftarrow \text{UPDATE}(W)$
14: $W \leftarrow \text{REVISE}(W, K)$
15: $E \leftarrow \{a | a \in S, \text{TEST}(\text{termconds}(a), W)\}$ ▷ Check termination conditions

16: **while** $E \neq \emptyset$ **do** ▷ Stop and pop all terminating behaviors and descendants
17: $e \leftarrow Pop(S)$
18: STOP(e)
19: **if** $e \in E$ **then**
20: $E \leftarrow E - \{e\}$

21: $A \leftarrow \{n | (e, n) \in N\}$ ▷ sequential followers of e, topmost terminating behavior
22: $C \leftarrow \{a | a \in A, \text{TEST}(\text{preconds}(a), W)\}$
23: **if** $C \neq \emptyset$ **then** ▷ There are potential followers
24: $b \leftarrow \text{CHOOSE}(C, P, S, W)$ ▷ Choose among C behaviors
25: **Goto** 3
26: $b \leftarrow \text{PEEK}(S)$ ▷ No potential followers, continue with parent
27: **if** $b \neq \emptyset$ **then**
28: **Goto** 11
29: **Halt**.

4 Predictive Monitoring of Recipes

4.1 Predicting Execution Possibilities

During execution, the agent may eliminate potential execution paths that are no longer feasible, given its current beliefs. Trivially this means ruling out behaviors whose preconditions cannot be met.

However, looking ahead can give more information when choosing the next action. In cases where an action can have undesirable or irreversible effects that prevent the agent from reaching its goal, it is not just useful but necessary. For example, a robot has a task of fixing a drawer with a loose screw. First it needs to go to a toolbox and take a screwdriver, then it walks to the drawer and screws the screw in. In the middle of the way the robot needs to rest since its motors get too hot. If the screwdriver is taken by someone at that point, the robot needs to go back to the toolbox and take a new one. Without execution monitoring the robot will find out that the screwdriver was taken only when it reaches the drawer.

While there are previous works that add this capabilities to robot decision making in general [32] and in particular to agents using layered hierarchical recipes [44], they where done using HTN planning. While there are noticeable similarities between hierarchical and layered recipes, there are also important differences.

Looking ahead in a recipe is a challenge. Recipe behaviors have only partial effects in the form of a termination conditions, but no way of predicting which termination condition will occur. The support keys hint at what beliefs might change but not how. A simple traversal of the recipe paths does not yield a complete simulation of the the changes to the world state, as it may do in plans. Moreover, higher level behaviors in recipes can affect the state of the world in parallel to their sub-behaviors, unlike HTN where only the primitive actions, the leafs of the network, can effect changes in the world. This means that there are more possible intersection in the search that changes the state, thus adding more complexity to the search.

Predictive execution monitoring begins with (i) a recipe, (ii) the current execution state in the recipe (that is, which behaviors are currently running), (iii) the current knowledge-base of the agent (iv) the last behavior that terminated. It then deduces, given the knowledge-base, whether any future behaviors can be shown to be *un-selectable*, even given potential changes to the beliefs of the agent, by behaviors possibly preceding this future behavior

in the execution. To do this, the monitoring system considers possible paths in the graph, from the current vertices, and projects potential changes to the beliefs, which may prove a behavior's preconditions to be false in all settings, hence the behavior is unselectable in the future, given the current knowledge of the agent.

Algorithm Lookahead (Alg. 2) 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 knowledge-base 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 knowledge-base revisions. Thus each search iteration results in multiple discrete points in the search space, to be considered. We describe the process in detail below.

Algorithm 2 Lookahead

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

1: $Q \leftarrow$ EMPTYQUEUE()
2: $successful\_paths \leftarrow \emptyset$
3: $visited \leftarrow \emptyset$
4: $p_{start} \leftarrow \langle S.pop, W \rangle \downarrow \langle S.pop, W \rangle \downarrow ... \downarrow \langle S.pop, W \rangle \downarrow \langle b_c, W \rangle$
5: ADD($\langle b_c, W, p_{start}, termCheck \rangle, Q$)
6: **while** $Q \neq \emptyset$ **do**
7: $q \leftarrow$ POP(Q)
8: **if** CHECKIFLEAF($plan, q.b$) **then**
9: ADD($q.path, successful\_paths$)
10: **Goto** 6
11: $E \leftarrow$ EXPAND($q, P, Revise, Test$)
12: **for all** $e \in E$ **do**
13: **if** $e \notin Visited$ **then**
14: ADD(e, Q)
15: ADD($e, visited$)
16: **return** $successful\_paths$

4.2 Searching Possible Future Executions

The algorithm receives the following functions:

- REVISE: Belief revision procedure described in Section 4.6
- EXPAND: This function returns the next states, i.e search nodes, in accordance with the test type of the current search node. The functions to expand search nodes can be found at Section 4.4
- VISITED: This function decides which search nodes is considered as visited and will not be added to the queue(Section 5)

The algorithm proceeds by iterating over a queue of execution traces to be considered. Each element in the queue is a *search node* $\langle b, w, p, c \rangle$, where b is the current behavior (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 behavior b 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). The algorithm halts when the queue is empty.

Alg. 2 stops expanding a search node and adds its execution path to the successful paths list, when it reaches a terminal behavior in the recipe graph: a behavior that has no sequential followers, no hierarchical children, and whose hierarchical ancestors do not have any sequential followers. We assume that a recipe has at least one such behavior.

In addition, notice that p is never considered when deciding how to expand, thus it has no sway over the expansion. The reason the path is part of the state is in order to keep the history of how we got here, since we are not looking for one path to a behavior, but rather all paths to the terminal behavior.

The expansion of the search occurs in lines 10–14. First (line 10), the algorithm asks for the set E, all possible expansions of the current search node q, by structural and belief revisions (the EXPAND method is explained in section 4.4. This set is then checked against the already visited search nodes (line 12) , to reduce the number of such expansions (this key step is the subject of Section 5). Then, the new nodes are put on the queue and marked as visited, so they do not get expanded again (line 14).

The search bears some similarity to a BFS search through a graph. However, it does not stop when we found a single path to a target behavior, but continues examining other paths, to other behaviors. Indeed if there is no precondition elimination then we will traverse all the behaviors and all the edges (hierarchical or sequential) of the recipe graph. 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.

4.3 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 agent's beliefs and behaviors when Alg. 2 was called. The execution path contains a sequence of behaviors selected for execution by the executive (BIS), 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. We denote hierarchical decomposition by $\downarrow$ and sequential edges by $\rightarrow$. 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, we look at the BIS recipe in Figure 1. An example execution trace p in this recipe may be $(Initiate|w_0)\downarrow(tools\_shed|w_0) \rightarrow (Initiate|w_0)\downarrow(pick\_screwdriver|w_1) \rightarrow (Initiate|w_0)\downarrow(resting\_point1|w_2)\downarrow(face\_east\_rest1|w_2)$. In this case we have three execution elements, which took place in sequence:

- First, the executing agent executed behaviors *Initiate*, concurrently with its child behavior *tools_shed*. At the time, the agent had specific beliefs collected in knowledge base w_0.

- Then, the beliefs of the agent have changed (resulting in w_1), and the

agent terminated *tools_shed* and selected a different child of $Initiate$, called *pick_screwdriver*.

- Finally, the beliefs of the agent changed again (w_2), resulting in the selection of the behavior $resting\_point1$, which itself has an executing child $face\_east\_rest1$.

We denote $last(path)$ to be the last execution element in this path (i.g $last(p) = (Initiate|w_0)\downarrow(resting\_point1|w_2)\downarrow(face\_east\_rest1|w_2)$). We also define subtraction between execution element of a path and a behavior in the path. The difference is the element until the last place where the behavior appeared. For example if we take $last(p)$ and subtruct $resting\_point1$ we get: $last(p)\backslash(resting\_point1) = (Initiate|w_0)$. A subtraction of a behavior from a path is done in the same way. It will be reduced until the first time in the path that this behavior was encountered.

Notice there is a difference between an execution path and a graph path. A graph path (from this point on, denoted *gpath*) is a sequence of behaviors where each behavior is connected to the previous one either by hierarchical edge or sequential edge (i.e., a path in the recipe graph). An execution path (from this point on, denoted *xpath*) is a graph path with the addition of the knowledgebase holding when each behavior was selected for execution.

4.4 Simulating 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 (if not previously visited). 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 *xpath* p, the behavior b to be expanded, and the knowledgebase w assumed to hold currently.

Each expand type will rely on at least one of these two procedures:

1. REVISE, which generates a new knowledgebase w' from the existing w and a set of new beliefs B. For example, by overiding belief values in w with new values from B.

2. TEST, which carries out the matching of the preconditions of behaviors f against the revised w'.

(i) *PreCheck*: Select hierarchical child. Given that b was selected for execution, one or more of its hierarchical children may be selected for execution. Algorithm 3. describes the process. The precoditions of all children behaviors (reached by following a single hierarchical edge from b) are tested against w (lines 2–4). In actual execution, only one would get selected. But as we are simulating all possible executions, each possible matching child b_i (indeed, each possible combination of matching conditions, for each matching child) would be a possible expansion of the current *xpath*. This is done by generating a new search node for each match: a node in which w is the same, but the *xpath* was amended to include $b{\downarrow}b_i$ at the end of the last element (line 4). Finally, the behavior b must also be expanded as it modified its beliefs during its own execution (remember, b 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) in line 5.

Algorithm 3 Expand PreCheck.

Require: Current search node $q = \langle b, p, w, c\rangle$
Require: The plan $P = \langle F, B, H, N, b_0\rangle$
Require: Condition Testing Procedure TEST
1: $E \leftarrow \emptyset$
2: **for all** $\{h|(q.b, h) \in H\}$ **do**
3: **if** TEST$(q.w, \text{preconds}(h))$ **then**
4: $E \leftarrow E \cup \langle h, q.w, q.p{\downarrow}\langle q.w, h\rangle, preCheck\rangle$
5: $E \leftarrow E \cup \langle q.b, q.w, q.p, inCheck\rangle$
6: **return** E

(ii) *InCheck*: Simulate revisions by the behavior. When b begins execution, it may directly revise the beliefs in w. A simulation of its execution requires us to predict such revisions. Algorithm 4 describes the process. 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(b) is set to *unknown* (line 4). In addition, the behavior b may also terminate, and so we also set the expansion type set to *TermCheck* (line 5).

Algorithm 4 Expand InCheck.

Require: Current search node $q = \langle b, p, w, c\rangle$
Require: The plan $P = \langle F, B, H, N, b_0\rangle$
Require: Condition Testing Procedure TEST
1: $E \leftarrow \emptyset$
2: $newkb \leftarrow q.w$
3: **for all** $\{key|\forall b \in last(p), key \in \text{support}(b)\}$ **do**
4: $\quad newkb \leftarrow \text{REVISE}(newkb, \langle key, unknown\rangle)$
5: $E \leftarrow E \cup \langle q.b, newkb, current.p, termCheck\rangle$
6: **return** E

(iii) TermCheck: Simulate behavior termination. A final set of expansions of b simulates the effects of its termination. Algorithm 5 describes the process.

When b terminates, then the termination conditions termconds(b) are true. Thus in any *TermCheck* expansion of q, new nodes must have a revised knowledgebase w' where the termination conditions hold. In the common case where termconds(b) are arranged as a disjunction (i.e., any one condition may indicate termination), this means that each combinations of the beliefs in termconds(b) (loop, line 6) generates a new w' (line 7). We denote the power set of a group T of termination conditions (that is all combination of termination condition) as $\mathcal{P}(T)$

In addition, there are two ways in which execution continues after b terminates. First, its parent may terminate given the new knowledgebase w' (line 8). Second, any behavior f that *follows* b (i.e., edge $(b, f) \in N$, loop in line 9) may be selected, should its preconditions hold in w' (line 10). Each f must replace b as the last executing behavior in the path, with knowledgebase w' (lines 11–13).

We distinguish between internal and external termination conditions. In the case of an external condition the termination condition can always be changed to true, without a single child of the behavior executed. In the case of an internal condition, the child need to first be executed before we can

change the condition value to true. Thus we need to treat this two types of condition differently when simulating possible executions.

To that extent we assume a list of keys for each behavior that holds all the keys that the behavior and all is children will possibly change. This list can be obtained by going over all the behaviors of a sub-tree of each behavior and collect the support keys for each behavior.

We use this list to attain a new list of termination conditions. These are the termination condition we want to revise to be true before continuing forward. This list will be consistent with all the termination condition that are either external or internal and there value is true or unknown (lines 3–5). Thus if the condition already holds in the database there will be no change, if it is unknown there will be no change. On the other hand if its an external condition that is currently false in the knowledgebase it will be revised to true in the new knowledgebase.

The internal condition which at the time of the parent are false will be checked with new values when the parent is expanded again after the child finished (line 8).

As there are often multiple follower behaviors 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 b is either b or a behavior that precedes it in execution. We note that this type of expansion necessarily revises the knowledgebase; when b 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 b to itself in the recipe graph can result in a combinatorial number of expansions to the earlier behavior.

If we enter a cycle in the recipe graph for the first time with a certain knowledgebase, by the end of it we most likely get a different knowledgebase. If we do the cycle again starting with a this new knowledgebase, we again most likely get a third different knowledgebase. That can be repeated again and again, until there are no new knowledgebase that can be created by this cycle. Thus it is perfectly acceptable, and sometimes even necessary, to repeat a cycle in the recipe a number of times. For that reason a cycle in the recipe graph does not translate directly to cycles in the search space.

Algorithm 5 Expand TermCheck.

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

4.5 Starting Node

The first search node added to the queue, depends on when the placement of the Alg 2 in the BIS algorithm (Alg 1). If the call is made before line 4 of Alg 1 then the first search node will have type check of *preCheck*. Since we know that the next part of the algorithm will select hierarchical decomposition, and this is exactly what the expand preCheck considers.

However if the call is made before choosing a sequential follower for a behavior (line 21 of Alg 1) then the check type should be *termCheck*, since we are now considering which follower to select. However this expansion is a little different, since we know which termination condition of the last behavior happened, this is reflected already in the knowledgebase, there is no need to check all the termination conditions of the current terminated behavior. This means that we do not need the loop over $\mathcal{P}(T_{new})$, only the loop over the followers and the addition of the parent behavior.

4.6 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 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 below and examine them in two important aspects:

- Soundness - Any *xpath* returned by lookahead is ***indeed feasible*** at the time of the call. This means that lookahead only returns an *xpath* if it is absolutely sure that the path will not fail.
- Complete - Lookahead returns ***all feasible*** *xpath*. However some of the *xpath* it returns may not actually be feasible.

4.6.1 Optimistic Testing is Complete

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 $\langle k, unknown\rangle \in W$ then the precondition holds. Trivially, we can see that optimistic testing gives us complete but not necessarily sound matching: 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.

That is because the test treats unknowns has an accepted precondition, this latter can be found to not be the case. We can guaranty that a path that deemed to not be feasible will indeed be infeasible unless an outside effort will be made to make it feasible.

Theorem 4.1. *Lookahead with optimistic testing is complete.*

Proof. Let us assume for contradiction that there is such a *xpath* p that was deemed infeasible, but is indeed feasible. This means that there exists a behavior b that the algorithm decided to not explore further.

Let us denote the set of all possible combination of termination condition and support conditions created knowledgebases that led to this behavior M.

This means that $\forall m \in M, optimisitc - test(\text{preconds}(b), m) = False$. Since we treat every unknown has upholding the preconditions then $\forall m \in M, \exists k \in \text{preconds}(n)$, such that $m(k) = True \vee m(k) = False$ since otherwise the $optimisitc - test(\text{preconds}(h), m) = True$.

Let us assume w.l.o.g that $\forall m \in M, \exists k \in \text{preconds}(b)$, such that $m(k) = True \wedge \text{preconds}(b)(k) = False$ then this means that that no behavior that led directly to this behavior caused a change in the keys value to match the precondition.

If preconds(b) is an external condition, then Algorithm 6 will return true and therefore the path will not be eliminated. Contradiction.

Thus preconds(b) must be an internal condition. But this means that its value could not have been changed to $False$ by a previously selected behavior thus the path is infeasible since there is no other behavior changing it to the correct value, again leading to contradiction.

□

Algorithm 6 Optimistic Test.

Require: Precondition P_b
Require: Knowledgebase W

```
1: for all ⟨k, v⟩ ∈ P_b do
2:     if W[k] ≠ v ∧ W[k] ≠ unknown ∧ k ∉ internalCond then
3:         return ⊥
4: return ⊤
```

4.6.2 Pessimistic Testing is Sound

The inverse of optimistic testing is pessimistic testing, where unknown values *do not pass* the test. By definition, $\forall v, \langle k, v\rangle \neq \langle k, unknown\rangle$. Inversely from the optimistic testing, pessimistic testing gives us sound solutions, but is potentially incomplete.

Theorem 4.2. *Lookahead with pessimistic testing is sound*

Proof. Let us assume for contradiction that there is such a *xpath* p that was deemed feasible, but is indeed infeasible. Then let assume that the behavior b is the behavior that the algorithm decided is selectable but is indeed unselectable. This means that in the knowledgebase w that we entered with has

the property $\forall k \in \text{preconds}(b), (w(k) = True \wedge \text{preconds}(b)(k) = True) \vee (w(k) = False \wedge \text{preconds}(b)(k) = False)$ this means that either no behavior before touched any of the keys in preconds(b) or that they where changed by the termination condition. If no other behavior changed any of the keys then this means that the only thing changing the keys to no longer match the precondition is an exogenous event that makes the path no longer feasible, but this is not an event known at the time of the call to lookahead. If the termination condition changed the key then this means we know that this execution path has gone through that behavior and the value changed to make the path feasible, in contradiction to the path not being feasible. □

We found pessimistic testing to be ineffective in practice, since it almost invariably predicts complete plan failure within a few iterations of Algorithm 2. This is because we most likely after a few iteration have at least one of the precondition be unknown, and the path eliminated. In the experiments, we therefor use optimistic testing. Each *xpath* we keep is a path that is feasible. That is because we only took *xpath* that their precondition held with the incomplete knowledge we have. Complete knowledge should not change that. However, we will not get all the feasible *xpath*. We will eliminate a *xpath* that is feasible because we did not have sufficient knowledge to confirm it.

Algorithm 7 Pessimistic Test.

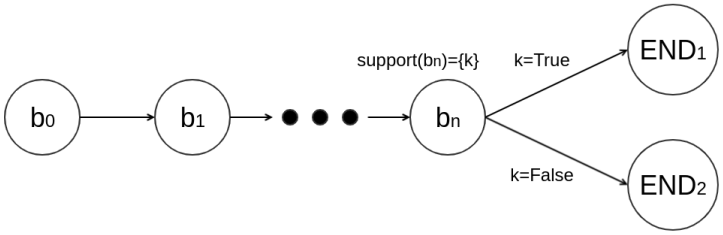
Require: Precondition P_b
Require: Knowledgebase W

```
1: for all ⟨k, v⟩ ∈ P_b do
2:     if W[k] ≠ v then
3:         return ⊥
4: return ⊤
```

4.6.3 Sound and complete testing is generally impossible

Theorem 4.3. *There does not exist a general test method that give both sound and complete output when unknown values exist in the knowledgebase, without additional knowledge.*

Proof. Let us assume a test method $test_{opt}$ that is both sound and complete. Let us look at the following recipe:



Where $k \in \text{support}(b_n)$,$k = True \in \text{preconds}(END_1), k = False \in \text{preconds}(END_2)$ and $k \notin \text{termconds}(b_n)$ Let us first assume that the only feasible path is $p_1 = b_0 \rightarrow b_1 \rightarrow ... \rightarrow b_n \rightarrow END_1$ this means that key k need to have value true to choose END_1. We know that b_n changes the value to unknown. We know that $test_{opt}$ returns all feasible paths and only feasible paths. This means that in this case it returns only p_1. Let us now look at the case where the feasible path is $p_2 = b_0 \rightarrow b_1 \rightarrow ... \rightarrow b_n \rightarrow END_2$. We know then that $test_{opt}$ will return only p_2. However, in both cases $k = unknown$ just before the choice of END_1 and END_2 there is no difference in the search but $test_{opt}$ manged to choose the right behavior each time. This means that it add additional knowledge outside of the knowledgebase and the precondition and termination condition and support keys in contradiction to the theorem basis.

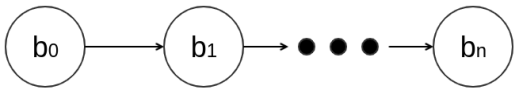
□

4.7 Complexity of the Base Algorithm

We analyze the run-time complexity of Algorithm 2. 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 recipe* $G = (B, H, N, b_0)$ *is a simple recipe when* $\forall b \in B, deg_H^-(b) = deg_H^+(b) = 0 \wedge \forall b \in B \backslash \{b_0, b_n\}, deg_S^-(b) = deg_S^+(b) = 1 \wedge deg_S^-(b_0) = deg_S^+(b_n) = 0 \wedge deg_S^+(b_0) = deg_S^-(b_n) = 1$ *and each behavior has no support keys.*

A simple recipe with n behaviors will look as follows:



Theorem 4.4. *Let G be a simple recipe with $|B| = n$ where $n \geq 2$ and each $b \in B$ has t termination conditions. Then G has at most $(2^t)^{n-1}$ xpaths.*

Proof. Let us prove by induction. Base case, $n = 2$: Let us denote the second behavior in B as b_1. Using the TermCheck expansion function on b_0 will produce the search nodes $\langle b_0, \textsc{Revise}(w_0, k) = w', \langle b_0, w_0\rangle \rightarrow \langle b_1, w'\rangle, PreCheck\rangle$ for $\forall k \in \mathcal{P}(\text{termconds}(b_0))$. Notice that each Revise call produces a different knowledgebase, due to the different termination conditions. Thus we have $(2^t)^{n-1} = (2^t)^{2-1} = (2^t)^1$ *xpaths*.

Induction step: Assume the theorem holds for $k-1$, and show true for $n = k$: Let us have a simple recipe $G = (F, B, H, S, b_0)$ with $|B| = k$ 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 $G_{k-1} = (B \backslash b_k, H = \emptyset, N \backslash (b_{k-1}, b_k), b_0)$. We know by the induction that G_{k-1} has at the most $(2^t)^{k-1-1} = (2^t)^{k-2}$ *xpath* already in the queue. This means that if we add the behavior b_k to the end of G_{k-1} then the expand function will be called for all $(2^t)^{k-2}$ search nodes created for each *xpath*, because now b_{k-1} has a sequential follower. For each such call the expand will produce 2^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 conditions do not contradict, we have $(2^t) \cdot (2^t)^{k-2} = (2^t)^{k-1} = (2^t)^{n-1}$ *xpaths*. □

Therefore, the worst case run-time complexity of the algorithm on a simple recipe $G = (B, H, N, b_0)$ is $O(2^{t \cdot |B|})$.

Definition 2. *A recipe $G = (B, H, N, b_0)$ is a flat acyclic recipe when $\forall b \in B, deg_H^-(b) = deg_H^+(b) = 0$ meaning $H = \emptyset$ and each behavior has no support keys.*

For example:

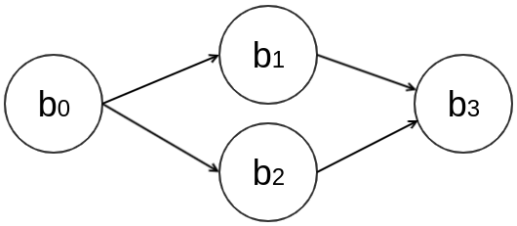


Figure 2: A flat recipe example

Theorem 4.5. *Let $G = (B, H, N, b_0)$ be a flat recipe. We denote the number of all gpaths as g. The worst case time complexity of lookahead on G is $O(g \cdot 2^{tl})$ (where l is the length the longest gpath).*

Before proving this formally, here is an intuition for the correctness of the theorem. We decompose the graph in to its component simple recipes, and add up there complexities.

Proof. For every *gpath* p, we create recipe $G_p = (B_p, H, N_p, b_0), B_p = \{b | b \in B \wedge b \in p\}, N_p = \{(b, b') | (b, b') \in p\}$. We know that for the each G_p the worst time complexity is $O(2^{t|B_p|})$, notice that in the case of the simple recipe the number of behaviors is also the length of the single path in the recipe. Thus the time complexity is the combine passes over each path that is $O(\sum_{i=0}^{g} 2^{t \cdot |B_i|}) \leq O(\sum_{i=0}^{g} 2^{t \cdot l}) = O(g \cdot 2^{t \cdot l})$. □

We can see this in the Figure 2 above. The graph is composed of two simple recipes $G_{b_0 \to b_1 \to b_3}$ and $G_{b_0 \to b_2 \to b_3}$. Since this is true for each path separately then for all the recipe we have $O(2^{t \cdot 2}) + O(2^{t \cdot 2}) = 2 \cdot O(2^{t \cdot 2})$.

Anther way of looking at this is as follows. In a DAG there are at most $2^{|B|-2}$ simple paths between two nodes (that is because in a DAG we can do a topological ordering of the vertices and then its a choice whether the vertex is in the path or not where the start node and end node are always chosen thus the -2). That means that in the case of a flat acyclic recipe we have at most $2^{|B|-2} \cdot 2^{tl}$ that is $O(2^{|B|+tl})$ where l is the length of the longest *gpath*.

Next we analyze the effects of the hierarchical edges on the complexity. As with sequential edges, we start by analyzing the complexity over a simple hierarchical recipe. That is a recipe $G = (F, B, H, S, b_0), S = \{(b_0, END)\}, \forall b \in B deg_H^+(b) = 1$. That is a graph with one hierarchical decomposition.

Theorem 4.6. *Let $G = (F, B, H, S, b_0)$ be a a recipe with $S = \emptyset, \forall b \in B deg^+_H(b) = 1$ and $|B| = n$ where $n \geq 2$ and each $b \in B$ has t termination conditions. G has at most $(2^t)^{n-1}$ search nodes.*

Proof. Let $G = (F, B, H, S, b_0)$ be a a recipe with $S = \emptyset, \forall b \in B, deg^+_H(b) = 1$ and $|B| = n$ and each $b \in B$ has t termination conditions. We denote the behaviors in the recipe as $b_0, b_1, ..., b_n$ where b_1 is the hierarchical child of b_0, b_2 is the hierarchical child of b_2 and so forth. Lets look at the run of Alg 2 on this recipe. The first search node will be $\langle b_0, W_0, b_0, preCheck\rangle$. This search node will produce the search node $\langle b_1, W_0, b_0{\downarrow}b_1, preCheck\rangle$ and so forth and so forth until we reach $\langle b_n, W_0, b_0{\downarrow}b_1{\downarrow}...{\downarrow}b_n, preCheck\rangle$. This in turn will go through inCheck and then go to termCheck. Since b_n does not have a sequential follower the only search nodes that the termCheck will produce will be $s_n = \{\langle b_{n-1}, \textsc{Revise}(W_0, t) = w', b_0{\downarrow}b_1{\downarrow}...{\downarrow}b_n, TermCheck\rangle | t \in 2^t\}$. Each $s \in s_n$ will produce the search nodes $s_{n-1} = \{\langle b_{n-2}, \textsc{Revise}(w', t), b_0{\downarrow}b_1{\downarrow}...{\downarrow}b_n, TermCheck\rangle | t \in 2^t\}$. Thus we already have $2^t \cdot 2^t$ search nodes. This search nodes will create similar search nodes for the behavior b_{n-3} and so forth until we reach b_0 again and the search will stop. Thus we have $\underbrace{2^t \cdot ... \cdot 2^t}_{\text{n-1}} = 2^{t(n-1)}$ □

We now know the *xpath*s created for a single path through one path along hierarchical edges. If we have $S = \emptyset$ we have a tree, i.e there is a single path between the root and a leaf. Thus the number of paths is the number of leafs. Thus such a recipe will have $\sum_{i=1}^{number\_of\_leaf}(2^{tl_i})$. Where l_i is the length of the path to leaf i.

Now we are ready to deal with the general case where we have both sequential and hierarchical edges, though we still assume there are no cycles in the recipe. We saw that whether its a sequential edge or an hierarchical edge we multiple the search nodes created by that edge with any other edge. Meaning all edge added to a graph path adds 2^t execution paths. This means that if all execution paths are feasible, in the base lookahead we will traverse all this execution paths. Let us then look at all *gpath* in graph G, where a path is an ordered sequence of behaviors where every two subsequent behaviors are either connected with an hierarchical edge, a sequential edge or the second behavior is a follower of an ancestor of the first behavior. We denote the number of all *gpath* as *gp* then we will have at the most $\sum_{i=1}^{gp} 2^{t \cdot i_l}$ termCheck search paths, where l_i is the length of the ith *gpath*. Notice since we are still discussing a recipe with no cycles, this means that each node can

appear only once in each *gpath* thus we still have $O(\sum_{i=1}^{2^{n-2}} 2^{t \cdot l_i})$. This is not a tight bound, since some of the 2^{n-2} paths cannot exist since only one child of a behavior can be chosen and the same goes for followers of a behavior.

5 Improving Efficiency by Pruning the Search

A key component in algorithm 2 run time complexity is the need to find all feasible *xpath*. Each search node is a tuple $\langle b, w, p, c\rangle$, where b is the current node in the graph, w is the current knowledgebase, p is the *xpath* that led to it as explained in Section 4.3, and c is the type of expansion to do. In principle, the lookahead algorithm can check the same node multiple times (see below), and this adds very much to the running time. By introducing methods for testing whether a node has been visited, the search space can be pruned and the running time improved.

In classic search on a graph, the only thing we check is if the node in the graph as already been seen before. When it comes to planning, the check is to compare the search node, that is the action and the state of the world. Since classical planning has one phase for each action, the action happens and the world changes according to effects, this is enough. However in our case we need to consider the actual execution state of the behavior and the parallel choosing of its children behavior, this means we need all three phases of c to accrue, we know that a search node $\langle b, w, PreCheck\rangle$ is not the same as $\langle b, w, InCheck\rangle$, as they will yield different search nodes upon expansion. Thus we need to compare at least these 3 elements.

The fourth element of the search node is the *xpath* p. p does not effect the expansion of the node, nodes $s = \langle b, w, p, c\rangle$ and node $s' = \langle b, w, p', c\rangle$ will produce the same amount of search nodes, and this search nodes will be the same except for the path.

5.1 Naive approaches for visited check methods

From the above we can see two kind of compression for two search node $s_1 = \langle b_1, w_1, p_1, c_1\rangle$ and $s_2 = \langle b_2, w_2, p_2, c_2\rangle$.

1. $NAIVE_1 : b_1 = b_2 \wedge w_1 = w_2 \wedge c_1 = c_2 \wedge p_1 = p_2 \Rightarrow s_1 = s_2$
2. $NAIVE_2 : b_1 = b_2 \wedge w_1 = w_2 \wedge c_1 = c_2 \Rightarrow s_1 = s_2$

The question then is why do we need to compare *xpath* if it does not effect the expansion. Let us then explore this two methods of comparison in respect to two important factors:

1. Is lookahead complete with this visited method, that is it returns all feasible *xpath*
2. Does lookahead halts with this visited method

5.1.1 Completeness

We examine the first question: is lookahead complete with $NAIVE_1$ or $NAIVE_2$.

Theorem 5.1. *Lookahead is complete with $NAIVE_1$ and optimistic testing*

Proof. Let us assume for contradiction that there is such a *xpath* p that was deemed infeasible, but is indeed feasible. Then let assume that the behavior b is the behavior that the algorithm decided to not explore further. We already saw that optimistic testing is complete, this means that there exits at least one *xpath* p' that is the prefix of p until b and manged to change the keys value to match the precondition. Thus if this path was not returned then $NAIVE_1$ eliminated all the search nodes $n = \langle b, w, p', c\rangle$, where w' is the knowledge base created by traversing p'. This means that n was put on the queue at least once to be considered visited, contradicting b not being explored further. □

However the second method $NAIVE_2$ is not complete. We can see this in Figure 3 if we start with a knowledgebase of $w_0 = have - b = true; have - c = true; have - d = false$ we will lose either the possible path $\langle A, w_0\rangle \rightarrow \langle B, w_0\rangle \rightarrow \langle D, w'\rangle$ or the path $\langle A, w_0\rangle \rightarrow \langle C, w_0\rangle \rightarrow \langle D, w'\rangle$ depending on whether B or C is entered to the queue first. Once we get to D the second time, we will discard the search node, since will have the same behavior, same knowledgebase and same check ($PreCheck$) and thus we will not get all feasible paths. This means we need to be able to keep track of the *xpath* we traversed.

5.1.2 Halting

Let us then look at the second factor: does lookahead halt with $NAIVE_1$ or $NAIVE_2$.

Theorem 5.2. *Lookahead is halting with $NAIVE_1$, if the recipe P has no cycles.*

Proof. A graph with no cycles has a finite number of *gpath*s. Let us assume w.l.o.g that our knowledge base has k number of keys with at most v different values. Then our knowledgebase has at most v^k possible states. This means that for each node in a path we have v^k possible knowledgebases. Then for each path *gpath* we have $v^{k^{|p|}}$ possible *xpath*s to explore. This is a finite number and thus lookahead will halt. □

However $NAIVE_1$ will not be halting when the recipe has cycles. In the proof above we used the fact that an acyclic graph has finite number of *gpath*. This is not true for a cyclic graph, a cyclic graph has infinite number of *gpath* since we can repeat vertices in a cycle infinite times, thus there will be an infinite number of search nodes.

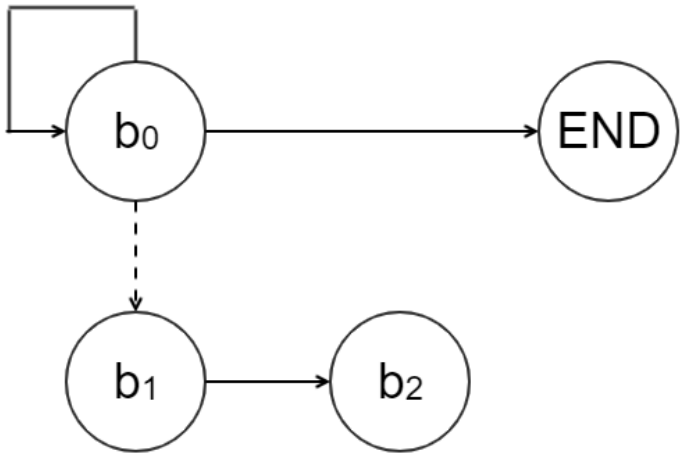
Let us look at $NAIVE_2$, this method of comparison will be halting since we have a finite number of knowledgebases (as shown in the proof above) and finite number of behaviors, thus we will have at the most $3 \cdot n^{v^k}$ possible checks to make.

5.2 How to address cycles in the graph

Notice that since we are looking for all feasible paths then since a cyclic graph has infinite number of *gpath*, if we want the algorithm to be complete (returning every feasible *xpath*) including *xpath*s that repeat the same behavior and knowledgebase tuple multiple times it cannot be halting. However, what we are looking for is not all *xpath*, for us the cycle is important only if it changes the possibility of reaching the goal. If we have two *xpath*s p_1 and p_2 where p_1 repeats a cycle n times and p_2 repeat a cycle m times,$n < m$ and this is the only difference we do not gain any additional information from p_2. It does not add possible edges or knowledgebases that help us achieve the goal that p_1 did not already give us. That is if a cycle is repeated again and the knowledgebase is the same when we leave the cycle as we started then doing the cycle over again will not give us anything new that will eliminate the possibilities to reach the goal that we did not already encountered. Thus for us the repeating of a behavior in an execution path is only important if it is with a new knowledgebase, that might resolve a future failure or create a condition where a future failure can arise.

For this reason we need lookahead to return not all feasible paths but all feasible *simple* paths, that is *xpath*s where all execution elements are distinct, do not contain an execution element more then once. This is similar

to definition of a simple path in graph theory, where any vertex in the path is distinct. However in the case of *xpath* the equivalent of vertices are the execution elements. Why a distinct execution element and not a tuple $\langle b, w\rangle$? Notice that a behavior can be reached with the same knowledge base multiple ways, including a decomposition that started with a different knowledgebase and in then became the same. For example if we have the recipe:



An *xpath* $\langle b_0, W_0\rangle\downarrow\langle b_1, W_0\rangle \rightarrow \langle b_0, W_0\rangle\downarrow\langle b_2, W'\rangle \rightarrow \langle b_0, W''\rangle\downarrow\langle b_1, W''\rangle \rightarrow \langle b_0, W''\rangle\downarrow\langle b_2, W'\rangle \rightarrow \langle END, W_{END}\rangle$ is a feasible *xpath* that we want to return, even though $\langle b_2, W'\rangle$ repeat twice, since it gives us the information that b_0 can also start with W'' and reach the end. Thus having the same tuple can be needed.

We then demand that lookahead will be complete in respect to simple *xpath*, that is return all feasible simple *xpath*. Notice that this does not invalidate our proof of completeness from before, because if an algorithm returns all feasible *xpath* then it also returns all feasible simple *xpath* since this is a subset of the former.

We now present visited methods that will be both halting and complete.

5.3 Successful Visited

First let us continue with $NAIVE_1$ as the base of our compression of equality between two search nodes. When a path p from successful paths contains a tuple $\langle b, w\rangle$ it means that $\langle b, w\rangle$ was already expended with a $PreCheck$,

notice that a new tuple $\langle b, w\rangle$ is added to a path of a search node the check type of that node is always $PreCheck$ (in line 4 in Alg. 3 and line 13 of Alg. 5). Thus if a search node $\langle b, w, p', PreCheck\rangle$ is encountered after p was added to the successful path list, the only difference between the search nodes can be the execution path p' . If $p = p'$ then $NAIVE_1$ will cover it and we will not add this search node to the queue. Let us then consider the case where $p \neq p'$, notice that since we already had $\langle b, w, PreCheck\rangle$ in the queue, all possible checks after this node was encountered where made, we already showed that the execution path does not effect the type of checks made. Thus we do not need to continue such expansions. However since we know there is a successful path from $\langle b, w\rangle$ to the end, p' should be in the successful path list with the suffix of p after $\langle b, w\rangle$. Thus we can add p' to the successful path.

Successful visited (Alg. 8) derives from the successful paths list a set of successful visited behaviors and the knowledgebase they started with $Successful\_visited = \{\langle b, w\rangle | \forall p \in Successful, \langle b, w\rangle \in p\}$. For each new search node $s = \langle b, w, p, PreCheck\rangle$ the method checks if $\langle b, w\rangle \in Successful\_visited$. If this is true then s is considered visited and not added to the queue and p is add to successful paths.

Algorithm 8 Successful visited

Require: Successful paths list S
Require: Search node $e = \langle b, w, p, c\rangle$
Require: Visited list V
1: **if** $e \in V$ **then** ▷ checks with $NAIVE_1$
2: **return** $True$
3: $S_{nkb} \leftarrow \{\langle b', w'\rangle | \forall p \in S, \langle b', w'\rangle \in p\}$
4: **if** $\langle e.b, e.w\rangle \in S_{nkb} \wedge e.c = PreCheck$ **then**
5: ADD$(e.p, successful_paths)$
6: **return** $True$
7: **return** $False$

In addition we add a call to Alg. 9 to the end of Alg. 2 before line 15. This algorithm iterates over all paths in successful path list (line 2) for each path if it includes the end behavior it adds it to the list as is (line 4–5). If not this is a partial path added in line 5 of successful visited. We want to add the appropriate suffixes. For that reason we extract all the paths that include one of the tuples of the last execution element of the partial path (line 7) and then add their suffixes after that last execution element (line 8). Notice

that SUFFIX$(p', \langle b, w\rangle, last(p))$ is a function that return the suffix of the path p after $\langle b, w\rangle$, this function can also changes the parents of $\langle b, w\rangle$ according to the knowledgebases in $last(p)$ for consistency. We saw in previous sections that when a tuple match, not all the execution elements have to match.

Algorithm 9 Successful visited processing.

Require: Successful paths list S
1: $S_{new} \leftarrow \emptyset$
2: **while** $S \neq \emptyset$ **do**
3: $p \leftarrow S.Start$
4: **if** $END \in p$ **then**
5: $S_{new} \leftarrow S_{new} \cup \{p\}$
6: **else**
7: $sf = \{p' | p' \in S_{new} \wedge \exists \langle b, w\rangle \in last(p) \Rightarrow \langle b, w\rangle \in p'\}$
8: $s_{new} \leftarrow S_{new} \cup \{p'' | \forall p' \in sf, p'' = p + \text{SUFFIX}(p', \langle b, w\rangle), last(p)\}$
9: $S \leftarrow S \backslash S.start$
10: **return** S_{new}

Theorem 5.3. *Lookahead using optimistic testing and successful visited is complete.*

Proof. Let us assume there is a simple *xpath* p that is a feasible but was not returned by the algorithm. This means there was a search node $s = \langle b, w, p_p, c\rangle$ where p_p is a prefix of p until $\langle b, w\rangle$ and s was eliminated thus p was not returned (otherwise because lookahead is complete with optimistic testing we will have returned p). We know from the completeness of optimistic testing with $NAIVE_1$ that line 1 in Successful visited (Alg. 8) will not remove it. Thus we know line 4 returned true. First this means that $c = PreCheck$. Next, according to line 5 we added p_p to the successful path list. This means that in order for p to not be returned the suffix of p after $\langle b, w\rangle$ did not exist in any of the paths in S at the end of the run of lookahead. Let us denote this suffix as p_{suf}.

In addition since line 4 returned true we know that there exist an *xpath* p' where $p' \in S \wedge \langle b, w\rangle \in p'$. In order for p' to be in successful path, there needed to be a search node $s' = \langle b, w, p'_p, PreCheck\rangle$ where p'_p is the prefix of p' until $\langle b, w\rangle$, otherwise it will not be on a path in successful path list with this tuple. Let us assume w.l.o.g that s' is the first search node that

included $\langle b, w\rangle$ (notice the first node to include any tuple that is in the path has check type $PreCheck$ since it is the only time we add to the path).

Let us look at s', since s' has a $c = PreCheck$ we will expand it according to Alg. 3. This means that we will create the the search nodes $S'_h = \{\langle h, w, p'_p{\downarrow}\langle w, h\rangle, PreCheck\rangle | (n, h) \in H \ \wedge \text{TEST}(h, w) = True\}$ and an additional search node $s'_{in} = \langle n, w, p'_p, InCheck\rangle$.

Let us look at the node s. If we where to expand it, we will have created the search nodes

$$S_h = \{\langle h, w, p_p{\downarrow}\langle w, h\rangle, PreCheck\rangle | (n, h) \in H \ \wedge \text{TEST}(h, w) = True\}$$

and an additional search node $s_{in} = \langle n, w, p_p, InCheck\rangle$. So far the suffixes of the paths are the same.

s'_{in} will produce $s'_t = \langle n, w', p'_p, TermCheck\rangle$. In turn, s'_t will produce the search nodes

$$s'_f = \{\langle f, w'', p'', PreCheck\rangle\}$$

where $\forall (n, f) \in N, \forall t \in 2^{\text{termconds}(n)}$ such that

$$w'' = \text{REVISE}(w', t)$$

and

$$\text{TEST}(f, w'') = True$$

and

$$p'' = p'_p + (last(p'_p)\backslash b){\downarrow}\langle f, w''\rangle$$

and the search nodes

$$s'_p = \{\langle parent(n), w'', p'_p, TermCheck\rangle\}$$

where $\forall t \in \text{termconds}(n) \Rightarrow w'' = \text{REVISE}(w', t)\}$.

On the other hand s_{in} will have produces the search node $s_t = \langle n, w', p_p, TermCheck\rangle$ where $w' = \text{REVISE}(w, \text{support}(n))$. s_t will produce the search nodes

$$s_f = \{\langle f, w'', p'', PreCheck\rangle\}$$

where $\forall (n, f) \in N, \forall t \in \text{termconds}(n)$, $w'' = \text{REVISE}(w', t)$ and $\text{TEST}(f, w'') = True$ and $p'' = p_p + (last(p_p)\backslash b){\downarrow}\langle f, w''\rangle$. Also, $s_p = \{\langle parent(n), w'', p_p, TermCheck\rangle\}$, where $\forall t \in \text{termconds}(n)$, $w'' = \text{REVISE}(w', t)$.

Notice all the suffixes produced by s different then s' only by the reduction of b from the last execution element of p_p and p'_p respectively. This is taken care of by the post processing that will change the elements to be the same. This means that if a suffix will produced by s it will also be produced by s'. This means the p_{suf} was produced by s' thus in successful paths. □

Alg. 2 with successful visited is not necessarily halting. Notice that if we enter a loop that repeat itself infinite number of time, without being able to reach an END behavior, then the algorithm will keep expanding every search node of the loop, since no path will be in the successful path list.

5.4 Cycle Avoidance

Deriving from the definition of simple *xpath* we can detect a recurring execution element and ignore it, this what the cycle avoidance Alg. 10 do. This visited method still uses $NAIVE_1$ as basis but this time in addition to comparing all the elements of the search node it also checks if the execution path already contains $\langle b, w\rangle$. We reduce the last execution element from the possible behaviors since this last execution element was created in this expansion, thus $\langle b, w\rangle$ will always be present in the last execution element. In addition, we do this check only if the node has a check type of $PreCheck$ since we only add tuples to the execution path in the expansion of $PreCheck$ thus it is the only expansion that was already made just with a smaller path.

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

Algorithm 10 Cycle avoidance

Require: Search node $e = \langle b, w, p, c\rangle$
1: **if** $e \in V$ **then** ▷ checks with $NAIVE_1$
2: **return** $True$
3: $p_{nodes} \leftarrow$ GET_NODES$(e.p \backslash LAST(e.p))$
4: **if** $\langle b, w\rangle \in p_{nodes} \wedge c = PreCheck$ **then**
5: **return** $True$
6: **return** $False$

Theorem 5.4. *Lookahead with cycle avoidance and optimistic testing is complete.*

Proof. We saw in the proof for successful visitedthat if we have a path the includes the tuple $\langle b, w\rangle$ 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 $\langle b, w\rangle \in (p - last(p))$ 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. □

Theorem 5.5. *Lookahead with cycle avoidance and optimistic testing is halting.*

Proof. Remember that we have v^k possible knowledgebases. This means that an *xpath* can have a behavior b, v^k times in it. If a search node $s = \langle b, w, p, PreCheck\rangle$ where b appears in $p\backslash last(p)$ v^k times then s will be eliminated. If all search nodes with $PreCheck$ of b are eliminated then there will be no more search nodes with $InCheck$ of b. This means we also eliminate all search nodes created in line 4 of Alg. 4. This leaves us with the search nodes created in line 8 of Alg. 5. These search nodes are created by the hierarchal children of b. However, since there are no more search nodes with $PreCheck$ of b then we stop creating search nodes with the children of b. Since this only happens in Alg. 3 with b, the search nodes will not be created. Thus we have a finite number of times that each search node will be added to the queue thus the algorithm is halting. □

5.5 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 the *xpath*. 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 (we so that in the proof of successful visited in Section 5.3) So a new search node duplicating this check need not be added to the queue.

Remember that our problem with $NAIVE_2$ was not that it was not halting but rather that it was not complete, since we lost execution paths. For that reason we want to save execution paths and still not make the same

checks more then once. To do this we use a map allowing us to record search nodes which are already on the queue with different execution path. The *keys* of the map are tuples $\langle b, kb, c\rangle$ where b is a behavior, kb is a knowledgebase when we reached b, 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. For example for each search node $\langle b, w, p, PreCheck\rangle$ we create a search node $\langle b, w, p, InCheck\rangle$ at line 5 of Alg. 3. If we will not expand the latter node, we will not consider the effect of running b and we will lose some paths. The *value* of each key is a set that holds all the paths that leads to the key tuple. For each search node s we check if its tuple $\langle b, w, c\rangle$ exist in the keys (line 1). 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(lines 2–3). If not then we add that tuple to the map and also the path to that keys corresponding set of paths and say the node has not been visited (lines 5–6). In the end of Algorithm 2, for all $\langle b, w\rangle$ in any of the successful paths we add all the paths in the map under the entry $\langle b, w, PreCheck\rangle$. Then we use Alg. 9 to derive all the full paths.

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 11 Merge paths.

Require: Map of expanded nodes M
Require: Search node $e = \langle b, w, p, c\rangle$
1: **if** $\langle b, w, c\rangle \in M$ **then**
2: ADD$(p, M[\langle b, w, c\rangle])$
3: **return** $True$
4: **else**
5: ADD$(\langle e.b, e.w, e.c\rangle, e.p, M)$
6: **return** $False$

Theorem 5.6. *Lookahead with merge paths and optimistic testing is reduction complete.*

Proof. Let us assume there is a path p that is feasible and not in successful paths. This means there is a tuple $\langle b, w\rangle$ in the path that was the last node

tuple we reached and then its search node was deemed visited. This means that there is a search node $s = \langle b, w, px, c\rangle$, where $px = p\backslash\langle b, w\rangle$, meaning the prefix of p until $\langle b, w\rangle$. Let us denote the suffix of p as $sx = p - px$ and the first node in sx as $\langle b_k, w_k\rangle$. We will prove by induction that sx is in successful paths.

Base case length sx is 1: This means that b_k is a leaf. Let us assume w.l.o.g that b_k is a sequential follower of b. Since s was pruned, there is a different search node $s_2 = \langle b, w, px_2, c\rangle$ that was added to the queue previously. When s_2 was popped out of the queue then since s_2 has the same behavior, same knowledgebase and same check type as s it will expand the same search nodes, with one difference which is the path in them. Thus one of the nodes that will be expanded will include the path $px \to b_k$. When this node will be expanded then it will be added to successful paths, since it reached a leaf. Thus the suffix is in successful paths. This also means that px will be in successful paths since there is a path in successful paths that contains $\langle b, w\rangle$ and so all the paths that where added to that enter in the map, including px, will be added to successful paths.

Case length sx is k: Let us assume that all suffixes of length $\leq k-1$ of p are in successful paths. We will prove for sx of length k. w.l.o.g we assume that b_k is a sequential follower. Since s was pruned, there is a different search node $s_2 = \langle b, w, px_2, c\rangle$ that was added to the queue previously, and it is the first time the tuple $\langle b, w, c\rangle$ where encountered. When s_2 was popped out of the queue since s_2 has the same behavior, same knowledgebase and same check type as s it will expand the same search nodes, with one difference which is the path. Notice that this includes the search node $\langle b_k, b_k, px_2\rangle \to \langle b_k, w_k\rangle, c_k$. This node will either be pruned or it will be added to the queue. If it is added to the queue then when it will be popped out we will have a search node with the next node in p since this node contains a suffix of p of length $k-1$ we know that either it will be pruned but eventually added to successful paths or it will continue, either way it will be in the successful path in the end, thus sx will be there too. If it is pruned then form our induction assumption we know that this suffix will be in the successful paths and so when we add this suffix to the successful paths at the end we will have the edge that include b_k in the successful paths and thus this prefix will be added too. Thus we get all the suffix in the successful list. □

5.6 Relations between the visited methods

Let us compare the visited methods.

Theorem 5.7. *Every search node that will be marked as visited by cycle avoidance will also be marked as visited by merge paths.*

Proof. Let $s = (b, w, p, c)$ be a search node that is pruned by cycle avoidance. This means that $\langle b, w\rangle$ exist in $p \backslash LAST(p)$. Let us notice that there are 2 possible places in the algorithm that a behavior is added to the execution path: In line 4 of algorithm Expand PreCheck (Alg. 3) and lines 11-13 of algorithm Expand TermCheck (Alg. 5). In both cases the search node created has the check type of $Precheck$. Let us notice that if we have the search node $s' = \langle b, w, p', Precheck\rangle$ we will also produce the node $s'' = \langle b, w, p', InCheck\rangle$ since this is produced in line 5 in Alg. 3 which is the procedure that will be called when s' will be taken from the queue. Notice that $s \neq s' \wedge s \neq s''$ since cycle avoidanceremoves the last execution element before checking if $\langle b, w\rangle$ exist in the path. This means that s', s'' where created by the previous iteration where $\langle b, w\rangle$ where met. Thus we know that prior to s we had s', s'' so if $c = Precheck \vee c = InCheck$ we have $\langle b, w, c\rangle$ in the merge map and thus this node will also be pruned by merge paths. We are left with the case of $c = Termcheck$. If this is the case then this means that the node was considered visited by line 1 of cycle avoidance, this means that there was a previous search node $\hat{s} = (b, w, p, c) = s$ this means that the map will already have (b, w, c) in it thus this node will be marked as visited by merge pathshas well. □

Theorem 5.8. *Every search node that will be marked as visited by successful visited will also be marked as visited by merge paths.*

Proof. Let $s = (b, w, p, c)$ be a search node that is pruned by successful visited. This means that $\langle b, w\rangle$ exist in a an execution path $p_s \in Successful-paths$. Let us notice that there are 2 possible places in the algorithm that a behavior is added to the execution path: In line 4 of algorithm Expand PreCheck (Alg. 3) and lines 11-13 of algorithm Expand TermCheck (Alg. 5). In both cases the search node created has the check type of $Precheck$. Let us notice that if we have the search node $s' = \langle b, w, p', Precheck\rangle$ we will also produce the node $s'' = \langle b, w, p', InCheck\rangle$ since this is produced in line 5 in Alg. 3 which is the procedure that will be called when s' will be taken from the queue. Let p' be the prefix of p_s until $\langle b, w\rangle$, we know such search

node existed since p_s is in the successful path list. This means this node was already put on the queue to latter expand and add p_s to the successful path list. This means that when this node was created $\langle b, w, PreCheck\langle$ was put in the map. The same goes for s''. Thus this search node will be deemed as visited by merge paths. If $c = TermCheck$ then this means the node was considered visited by line 1 of successful visited, this means that there was a previous search node $\hat{s} = (b, w, p, c) = s$ this means that the map will already have (b, w, c) in it thus this node will be marked as visited by merge paths has well. □

The opposite of theorem 5.7 and 5.8 is not true. For example if we have the following recipe:

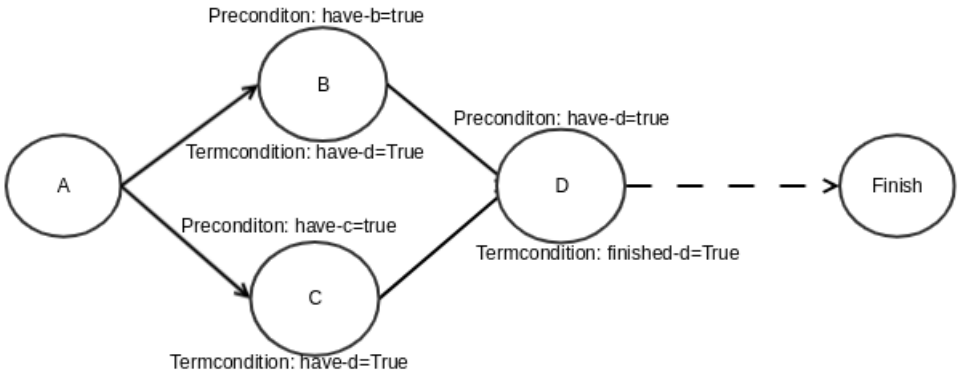


Figure 3: BIS recipe example 2

Where behaviors B needs resource b and behavior c needs resource c to achieve the same knowledgebase (B and C are different behaviors that achieve the same goal). If we have only resource b then both algorithms will return only the $xpath\ A \rightarrow B \rightarrow D \rightarrow ... \rightarrow finish$ without eliminating any search node. However, if the agent has both resources we will have a different outcome. Table 1 shows the state of the queue for each iteration of the Alg. 2.

After the fifth iteration is where merge paths will differ from both cycle avoidance and successful visited. Notice that in cycle avoidance and successful visited the node $(D; have - b = true; have - c = true, have - d = true; A \rightarrow C \rightarrow D; PreCheck)$ will be added to the queue. On the other

hand, in merge paths this node will be pruned since we will have the key $(D; have - b = true, have - c = true, have - d = true; PreCheck)$ in the merge map. Thus if the suffix $D \rightsquigarrow finish$ with the starting knowledgebase $have{-}b = true, have{-}c = true, have{-}d = true$ will exist both $A \to B \to D$ and $A \to C \to D$ will be in successful path without needing to do it twice for each prefix.

Iteration number	Queue			
	node	knowledgebase	path	check type
0	A	have-b=true;have-c=true;have-d=false	A	*PreCheck*
1	A	have-b=true;have-c=true;have-d=false	A	*InCheck*
2	A	have-b=true;have-c=true;have-d=false	A	*TermCheck*
3	B	have-b=true;have-c=true;have-d=false	$A \to B$	*PreCheck*
	C	have-b=true;have-c=true;have-d=false	$A \to C$	*PreCheck*
4	C	have-b=true;have-c=true;have-d=false	$A \to C$	*PreCheck*
	B	have-b=true;have-c=true;have-d=false	$A \to B$	*InCheck*
5	B	have-b=true;have-c=true;have-d=false	$A \to B$	*InCheck*
	C	have-b=true;have-c=true;have-d=false	$A \to C$	*InCheck*
4	C	have-b=true;have-c=true;have-d=false	$A \to C$	*InCheck*
	B	have-b=true;have-c=true;have-d=false	$A \to B$	term-check
5	B	have-b=true;have-c=true;have-d=false	$A \to B$	*TermCheck*
	C	have-b=true;have-c=true;have-d=false	$A \to C$	term-check
5	C	have-b=true;have-c=true;have-d=false	$A \to C$	*TermCheck*
	D	have-b=true;have-c=true;have-d=true	$A \to B \to D$	*PreCheck*

Table 1: Example run of lookahead on BIS RECIPE example 2.

6 Improving the Run-Time of Repeated Calls

In the first run of the algorithm we collect a lot of information. Specifically we get all feasible *xpath*s from the initial executing behavior. These paths include the knowledgebase we need to have in order to choose each node in this path. When we later run the algorithm on the terminated behavior in the stack, we will reproduce some if not all those execution paths again, since our algorithm is complete there can not be an execution path that is feasible that was not returned. In addition our checks might also include checks that we already did in the previous runs that led to a dead end.

This means that we can reuse the data we already collected. In this section we present an algorithm that does just that; it uses the data from previous runs, instead of doing the same checks again. We can go over all the paths given by the previous run and see if any of them is no longer feasible.

Alg. 12 is the process with which we go over all the previously produced successful *xpath*s. We start with the current knowledgebase W and the last terminated behavior b_c. We go over all the successful *xpath*s. Each time we check first if the path includes b_c (line 3), any path that does not include the current terminated behavior is no longer a possibility since we know our execution went through that behavior. We then find the first place that b_c appears in the path, and we consider only the suffix of this path from b_c (line 4).

6.1 Extrapolating the changes in the path

Each *xpath* element represent the stack at a certain time. It also include the knowledgebase that we started each node in the stack with. Thus an execution path encodes in itself the changes to both the stack and the knowledgebase within the "run" of the execution path. We can derive this changes by going over the execution path and extrapolate the differences between each consecutive execution element. Let us look at two consecutive execution elements e, e' of an *xpath* p.

First we can extrapolate the behaviors chosen from the sequential and hierarchical edges after each execution element. This can be done by taking all the behaviors that exist in e' but not in e. Notice that if a behavior exist in both e and e' then this is a parent behavior that was not terminated in the next execution element. Thus the group of behaviors $\{b|b \in e' \wedge b \notin e\}$ are the behaviors chosen in the next execution element, this behaviors represent a

parent behavior and all its children chosen in the hierarchical decomposition stage right after the parent was chosen (thus their all direct hierarchical children with a direct hierarchical edge from the parent to them) with the first behavior being the parent behavior. All of whom where chosen together with the same knowledgebase (notice that when adding hierarchical edges in the *PreCheck* stage we do not change the knowledgebase for the children). Thus we get all the behaviors added to the stack in lines 4–8,24 of Alg. 1.

As stated above all these new behaviors added to the stack had the same knowledgebase when they where added. Thus we can take the knowledgebase of the last behavior in each execution element. Let us now look at this knowledgebases, denoted by $e.w, e'.w$ for e, e' respectively. We look at the keys that there value changed between $e.w$ and $e'.w$. Since there value has changed then something has changed that value, this can only occur in the $InCheck$ phase or the $TermCheck$ phase. This changes where made by all the behaviors in e, meaning the previous sequential behavior but also there hierarchical children. We know that all the support keys of the previous execution element will be represented in this changes, but they will also reflect all the termination conditions that where chosen in this execution path. Notice that the difference between $e.w, e'.w$ gives us the changes that where done by e and not the changes that where done by e', since it is the difference between when e started and e' started, which is when e ended. If we want the changes that e' caused we need the difference between e' and its next consecutive execution element.

Alg. 13 uses that above extrapolation of the changes between execution elements to create an element that represents the new selected behaviors and the changes affected by these behaviors . Line 1 extrapolate the new behaviors added to the stack, while line 6–4 extrapolate the changes in the knowledgebase. In this algorithms we wanted the returned tuple to include the added behaviors and the changes they caused, thus we used the next consecutive behavior to extrapolate the knowledgebase.

6.2 The checks in repeated calls

First we need to decide what are the checks made against. In our path the knowledgebase of the next behavior after b_c is what was needed for these behaviors to be selected. This means that if the current knowledgebase and the expected knowledgebase (the one in the path) are the same, then lookahead will do the same checks as in the previous call, thus this path was

already deemed feasible. On the other hand, if we have differences between the current knowledgebase and the expected one, then new checks need to be made. However, the if there are keys with the same value in the two knowledgebases then since the plan is static, the changes on this values will be the same, again they where checked in the previous call. Thus the only keys that interest us are the keys whose values are different between the current knowledgebase and the expected one.

Thus in line 6 of Alg. 12 we extract only the changes between the current knowledgebase and the knowledgebase we expected to start this suffix path. This is done by taking the difference between the current knowledgebase and the knowledgebase saved in the next execution element after b_c.

The next step is to check if the difference between the expected knowledgebase and the given knowledgebase can eliminate the path. Remember while our original algorithm was complete it was not sound, some paths might not be feasible but we do not know that because we don't have the full effects of the behaviors.

For this reason we go over the execution elements of the path (line 7) and extract the selected behavior of that element and the changes this execution element did (line 11). Remember that in the original lookahead algorithm (Alg. 2) we had three types of checks for the three phases of a behavior, PreCheck to check for preconditions and select hierarchical children, InCheck to address the support keys of the behavior and the changes they make and TermCheck to change the knowledgebase according to the termination conditions and choose the followers that are selectable. In repeated calls this phases needed to be treated differently, since this time instead of creating new knowledgebases we are checking if the changes in the real world do or do not have effect over the choices made in the *xpath*.

PreCheck We first need to check if the behaviors we previously thought can be selected can still be selected, i.e if there preconditions are still met in the current knowledgebase (line 12). Notice that for this check we only need to check the keys which values are different between the current knowledgebase and the expected one, since we know that any value in the expected knowledgebase did not violate the preconditions (because then this path will not be part of successful paths) thus keys whose values are the same in the two knowledgebases correspond with the preconditions. This PreCheck is made in line 8 where the $TEST$ only returns $False$ if the key of one of

the behaviors precondition exist in Δk but the value does not match the precondition.

If a key does not exist in Δk then we ignore this precondition, since if the key is not in Δk then its value was either the same both in the expected starting knowledgebase (of the execution element right after b_c) and the real current knowledgebase or this value was changed during the TermCheck phase or InCheck phase.

InCheck In the original algorithm InCheck changed the keys in support to unknown, this means that any key in support of any of the new nodes that where selected will be changed to unknown. Thus the values of these keys in expected knowledgebase will be changed to unknown, and in this case ignored by the $TEST$ function in the original algorithm. In addition, these keys being in the support means they are going to be changed by this selected behavior and thus the values they hold in the current real knowledgebase are not valid for further checks of this path. For that reason in line 14 we remove the support keys from δk since there real value is not meaningful, its going to change, and there expected value in the next knowledgebases is going to be unknown and thus ignored.

TermCheck This phase in the original algorithm changed the knowledgebase to actual values, not unknown, according to the termination condition. To find out which termination condition where used to create the next knowledgebase we use the changes we found between the current execution element and the next execution element. Any key in this difference that its value is not unknown was changed by the termination condition. The $xpath$ we are currently checking is built on this changes, thus we are checking this $xpath$ has if this is the changes made to the knowledgebase, meaning this termination condition "happened". This means that the values of the keys in the current real knowledgebase will change to the values that correspond to the selected termination conditions. Thus the values of these keys in the current real knowledgebase no longer effect this path, we already checked in the previous call if this changes eliminate the path, and they did not. For that reason we can remove this keys from δk (line 15) since they are no longer effecting the path.

We do this for each element of the path until we either: eliminate the path when preconditions are discovered to no longer be true (lines 12–13),

we reached the end of the path and not eliminated it yet (line 16) or the differences between the current knowledgebase are mitigated by the changes made by the behaviors and are no longer a concern (line 8). If one of the latter reason accrued then we can concur that this path is still feasible.

Algorithm 12 Repeated Calls

Require: The Recipe $P = \langle B, H, N, b_0 \rangle$
Require: Current behavior b_c
Require: Knowledgebase W
Require: Successful paths list *paths*

1: $successful \leftarrow \emptyset$
2: **for all** $p \in Paths$ **do**
3: **if** $b_c \in p$ **then**
4: $p' \leftarrow \text{SUFFIX}(p', b_c)$
5: $W_s \leftarrow second(p')$
6: $\Delta k \leftarrow \{(k, v) | (k, v) \in W_s \wedge (k, v') \in W \wedge v \neq v'\}$
7: **for all** $(prev \rightarrow curr \rightarrow next) \in p'$ **do**
8: **if** $\Delta k = \emptyset$ **then**
9: $successful \leftarrow successful \cup p'$
10: **Goto** 2
11: $b_{selcted}, W_{diff} \leftarrow \text{PATHDIFF}(prev, curr, next)$
12: **if** $\exists b \in b_{selcted} \text{TEST}(\text{preconds}(b), \Delta k) = False$ **then**
13: **Goto** 2
14: $\Delta k \leftarrow \Delta k \backslash \{\text{support}(b) | \forall b \in b_{selcted}\}$ ▷ correspond to *inCheck*
15: $\Delta k \leftarrow \Delta k \backslash W_{diff}$ ▷ correspond to *termCheck*
16: $successful \leftarrow successful \cup p'$
17: **return** $successful$

If in the previous calls all *xpath* were feasible and the only things changed in this knowledgebase from the last call are keys that do not appear in the recipe then we will go over all *xpath* again, thus the worst case run-time is the same has the regular lookahead. However, if we eliminated a path in previous calls that had one of the precondition of a behavior violated at least this path is not checked again until the point of elimination, thus saving some checks. In addition, if δk consists of keys that are all latter changed in the path, we might not have to get to the end of it to decide it is still feasible, saving us further checks. Moreover, if the changes in the knowledgebase and the expected knowledgebase do not exist we will save the check of the path

Algorithm 13 Get Changes

Require: The previous execution element PREV
Require: The current execution element CUR
Require: The next execution element NEXT
1: $nodes \leftarrow \{n | n \in cur \wedge n \notin prev\}$
2: $W_{cur} \leftarrow \text{LAST}(cur).w$
3: $W_{next} \leftarrow \text{LAST}(next).w$
4: $W_{dif} \leftarrow \{(k,v) | (k,v) \in W_{cur} \wedge (k,v') \in W_{next} \wedge v \neq v'\}$
5: **return** $nodes, W_{dif}$

entirely since we know it is already feasible. For that reason the run-time of repeated calls can reduce significantly the run-time it takes to monitor the execution.

In section 4 we distinguished between internal and external conditions. We relay on this distinctions and our ability to determine what are the internal condition and external condition for repeated calls as well. If what was an internal condition was changed by an external force, repeated calls might not return all feasible paths. An example for this can be seen in our experiment with the Nao robot in section 8.3.

7 Experiments with random plans

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.

7.1 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 2 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 [34] to well over a hundred [45, 51], 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 2: 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 visited 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 with $NAIVE_1$ as the visited method. However, even smaller recipes timed out, even with 3 hours of CPU time given to them to run. We thus focused on the visited method. We ran the 5 chosen knowledgebases on each of the examples with the visited methods and their combinations, that is: merge paths (M), cycle avoidance (C), cycle avoidance and successful visited (C+S), merge paths and successful visited (M+S), merge paths and cycle avoidance (M+C) and all 3 pruning methods together (ALL). successful visited without some sort of cycle avoidance or merge paths proved to be as bad as the base algorithm and thus we decided to run it as part of a combination of visited methods.

7.2 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 4 and Figure 5. 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 5 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 5 that cycle avoidance 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 avoidance total runtime jumps significantly more then the others in the corresponding graph in Figure 4. successful visited with cycle avoidance does slightly better then cycle avoidance 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 paths 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.

7.3 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 7, 8 and 9. 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 7 is the total run time on plans with $d = 1$ and $b = 5$. Figure 8 is the total run time on plans with $d = 3$ and $b = 3$. Figure 9 is the total run time on plans with $d = 3$ and $b = 5$.

Notice that the scale of the graphs increases when t increases. This is in agreement with the complexity we found in Section 4, 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

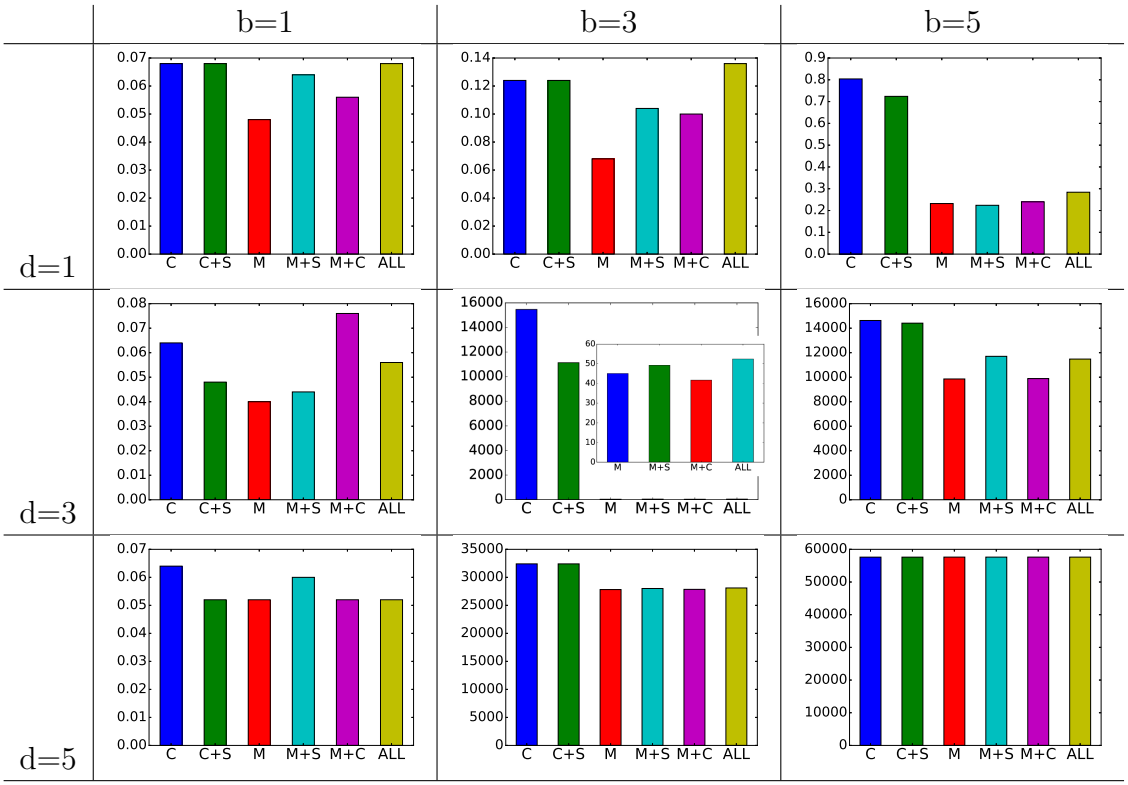


Figure 4: Total runtime for (t=1) (Lower is better)

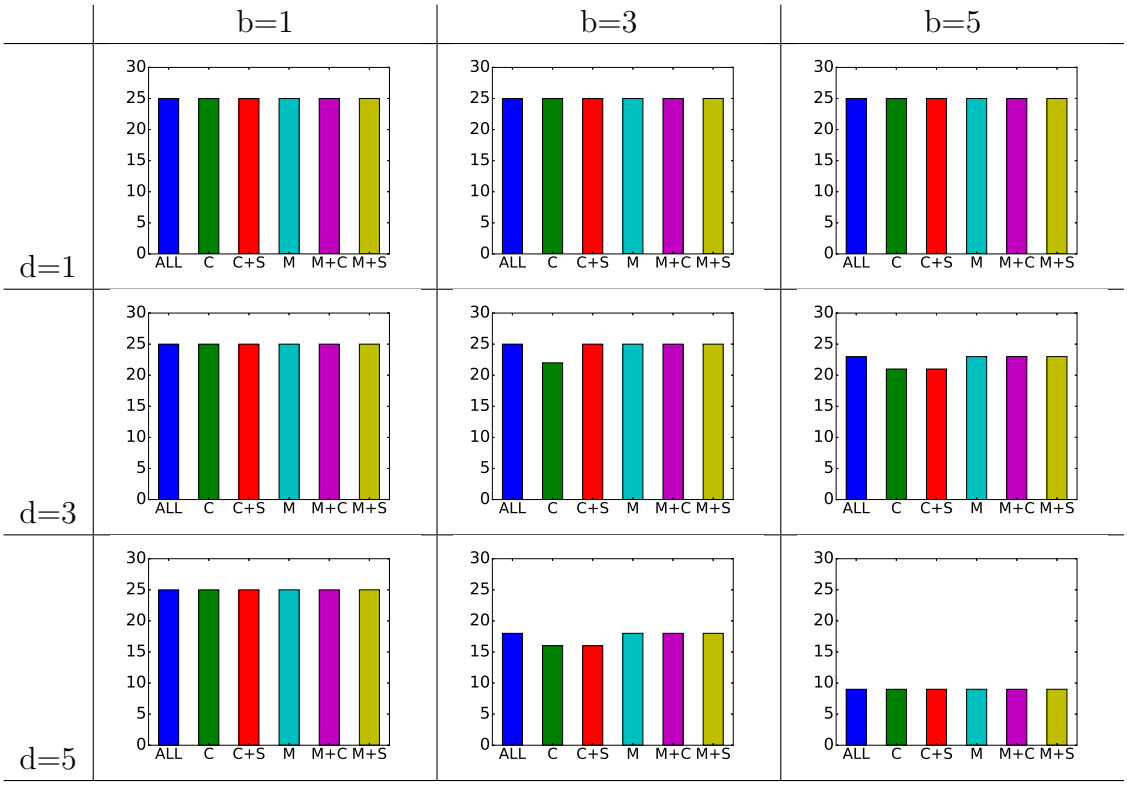


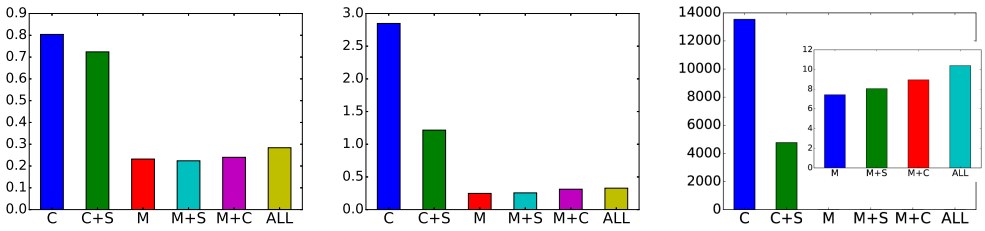
Figure 5: Number of finished runs when (t=1) (higher is better). Note the scale on the Y axis changes dramatically between subfigures.

better then cycle avoidance 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 avoidance does slightly better then merge paths alone, but not by much. This means that the best method to use is merge paths with cycle avoidance.

In general over all 675 runs for each pruning method we had:

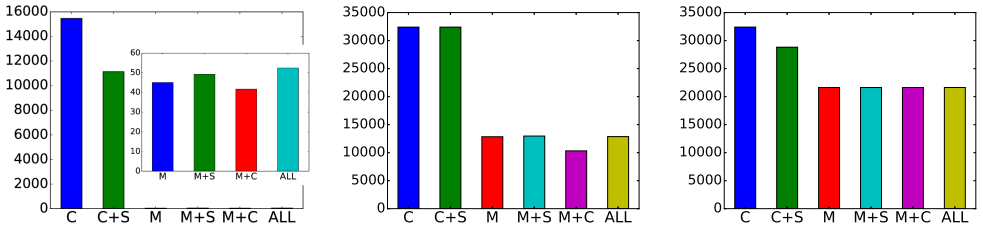
Pruning method	Finished runs	Timeout Runs
C	552	123
C+S	562	113
M+S	576	99
M+C	576	99
M	576	99
M+S+C	576	99

Figure 6: Number timeout and finished recipes for each prunning method



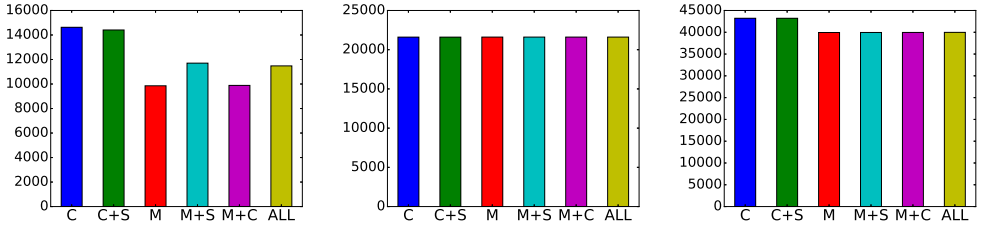
(a) Total runtime for (t=1) (b) Total runtime for (t=3) (c) Total runtime for (t=9)

Figure 7: 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)

Figure 8: 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)

Figure 9: Total runtime for (d=3,b=5) (Lower is better)

8 Experiments With a Nao Robot

8.1 Experiment Environment

In order to further evaluate the capabilities of the algorithms' we ran it on a real scenario with a robot. We used a Nao robot, a humanoid robot with 25 degrees of freedom. The scenario was has follows: The Nao robot needed to get to a toolbox and retrieve a screwdriver, from there it proceeded to a drawer standing in the opposite side and screw a screw to the drawer. Since the Nao's motors can get hot and it can stop functioning we set a resting point between the toolbox and the drawer, where it sits ad pauses before continuing. This is the same scenario we used to demonstrate BIS algorithm, thus we used the recipe presented in Figure 1. Notice that in this recipe if the robot drops the screwdriver (or the screwdriver is taken from it) after visiting the tool shed, then it will go to the drawer and will have nothing to do. This can be fixed by adding edges from the following behaviors after tool shed to tool shed, as in the recipe in Fig. 10

However, this does not solve the problem completely. If the robot ac-

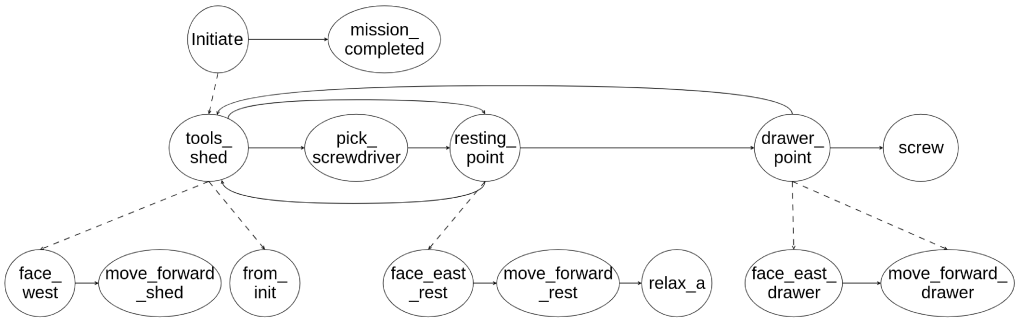


Figure 10: A recipe for a Nao robot to fix the drawer. Dashed lines are hierarchical edges (H) while solid lines are sequential edges (N). Nodes are behaviors (B).

cidental drops the screwdriver while resting and it has a preference to go forward unless it cannot. Without the monitoring algorithm it will still go to the drawer and will not be able to preform screw, it will then go back to take the screw. Of course we can make having the screwdriver a precondition of going to the resting point, but as discussed in Section 4 this solution is not scalable.

We ran this scenario in the lab. The robot goes to fix the drawer and we take the screwdriver from its hand while it is resting. Fig. 3 shows one such experiment. The top row shows from left to right: The initial position of Nao (bottom left of picture), the robot getting the screwdriver in the tool shed, the robot turning towards the resting point and the drawer. The second row (left to right): The screwdriver is taken from the robot, the robot turns to return to the tool shed to get the screwdriver again, the robot receives the screwdriver again. The last row (left to right): the robot goes to the resting point, the robot rests, the robot reached the drawer and is screwing the screw.


Initial Position
Tool Shed
Resting Point
Drawer Point

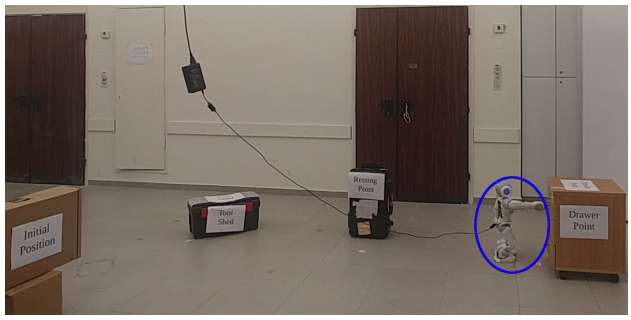


Table 3: Experiment with a Nao Robot

We ran the scenario above with BIS and our execution monitoring algorithm (Alg. 2) called each time a follower needed to be chosen. This was done on both the recipe in Figure 1 and Figure 10. Each time, we took the screwdriver from the robot while it was resting in the resting point. For the acyclic plan (Figure 1 this meant that after resting it had nothing to do, since none of the paths to the end was possible without the screwdriver. For that reason once no successful path was returned we terminated all behaviors in the stack and called BIS again from the start with the same knowledge-base that the robot already had. Notice this was possible in this recipe since restarting it will always take it to the tool shed. This is not a general solution: the problem of no possible paths is discussed further in Section 10.

We ran both recipes with the three visited methods and their combinations, that is: merge paths (M), cycle avoidance (C), successful visited (S), cycle avoidance and successful visited (C+S), merge paths and successful visited (M+S), merge paths and cycle avoidance (M+C) and all 3 pruning methods together (ALL).

We ran this experiment 50 times for each prune method on both the acyclic plan and the cyclic plan. In total we ran the experiment $7 \times 50 \times 2 = 700$. Each run was until the robot fixed the drawer, lookahead was run before we selected a behavior in line 24 of Alg 1.

Since we saw that even for a simple plan the complexity of the monitoring algorithm can be high, we also did time limit on the run of each call to lookahead (as we did in the experiments), no call could run more then 60 seconds of CPU time. If the time was finished for the call it simply returned all the possible paths it found so far.

8.2 Acyclic recipe

Let us first discuss the experiments done on the acyclic recipe (Figure 1). In these experiments we wanted to both compare the different pruning methods, but also to see how much time the lookahead adds to our run-time Notice that cycle avoidance will prune no search node since there are no cycles in this recipe, thus there will be no previous instances of a behavior we arrived at in the execution path. This means that we can look at cycle avoidance as a base line in this case, as if using only $NAIVE_1$.

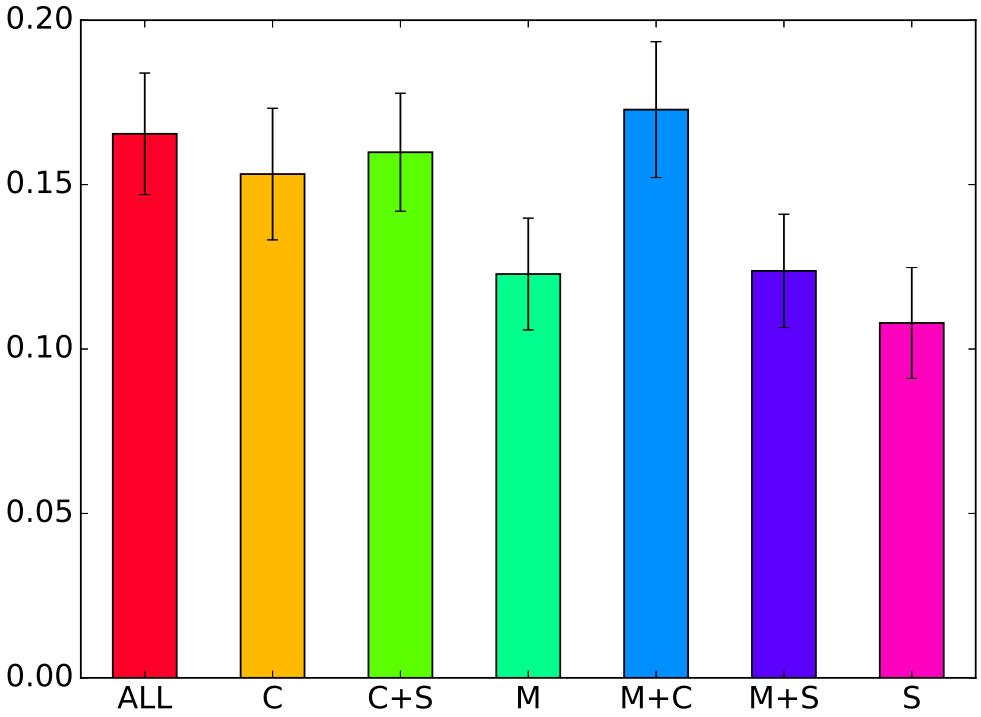
First we wanted to see the impact on the run-time, for that reason we calculated the mean total time spent in lookahead for all the duration of one run of the scenario, that is the the sum of all calls of lookahead from the initial position to fixing the drawer divided by the number of runs of the scenario (in this case 50 runs).

Figure 11a shows the mean total time spent in lookahead when Nao went from the initial position to fixing the drawer without issue. Figure 11b shows the mean total time spent in lookahead when the screwdriver was taken at the rest point in the first arrival, and no issue in the second arrival to the rest point.

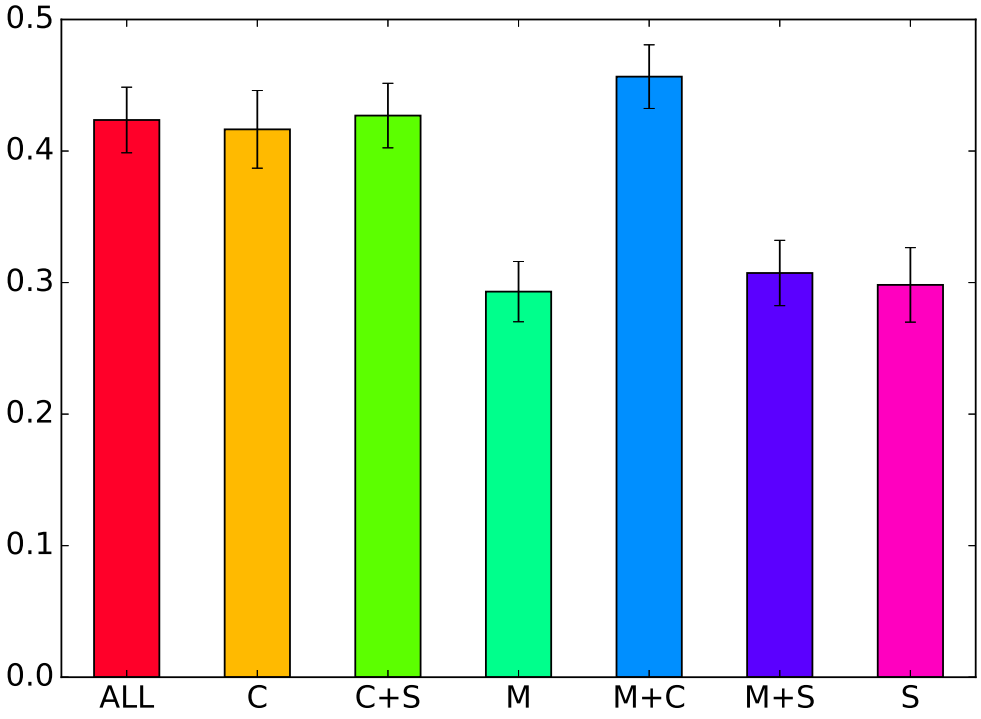
We can see in these figures that cycle avoidance and the combination of all together did the worst, cycle avoidance is after all the base line where we compare search nodes base on all there components and do not take a better look at the path, this result is as expected. The combination merge paths and cycle avoidance is one more call to a function which increases the run-time, however it will cut as much as merge paths so it surprising it did as worst. Merge paths and successful visited did the best in terms of run-time, and where there was no issue successful visited did slightly better then merge paths. This is surprising considering we showed that merge paths will eliminate at list the same search nodes as successful paths.

We can see that for all methods the time spent in lookahead is pretty small on this simple recipe when there was no issue. The time spent in lookahead when the screwdriver was taken away is of course longer, we had more calls to lookahead after all. But it is still way under a second.

To understand better the result of the run-time we take a look at the number of iteration each method has done. First lets take a look at the sum number of iteration all calls to lookahead in one scenario has done. Figure 12 shows us the sum of iteration over all calls of lookahead from the initial position to fixing the drawer (i.e the sum of iteration over all calls to



(a) Mean time spent in lookahead no issue.



(b) Mean time spent in lookahead screwdriver taken.

Figure 11: Mean time spent in lookahead (acyclic)

lookahead in one run of the scenario). Figure 12a shows the sum number of iteration when there was no issue and Figure 12b shows us the number of iteration when the screwdriver was taken. We do not need to do mean over all 50 runs of the scenario since the number of iteration never changed for each visited test method (what change is the time it took to do each iteration).

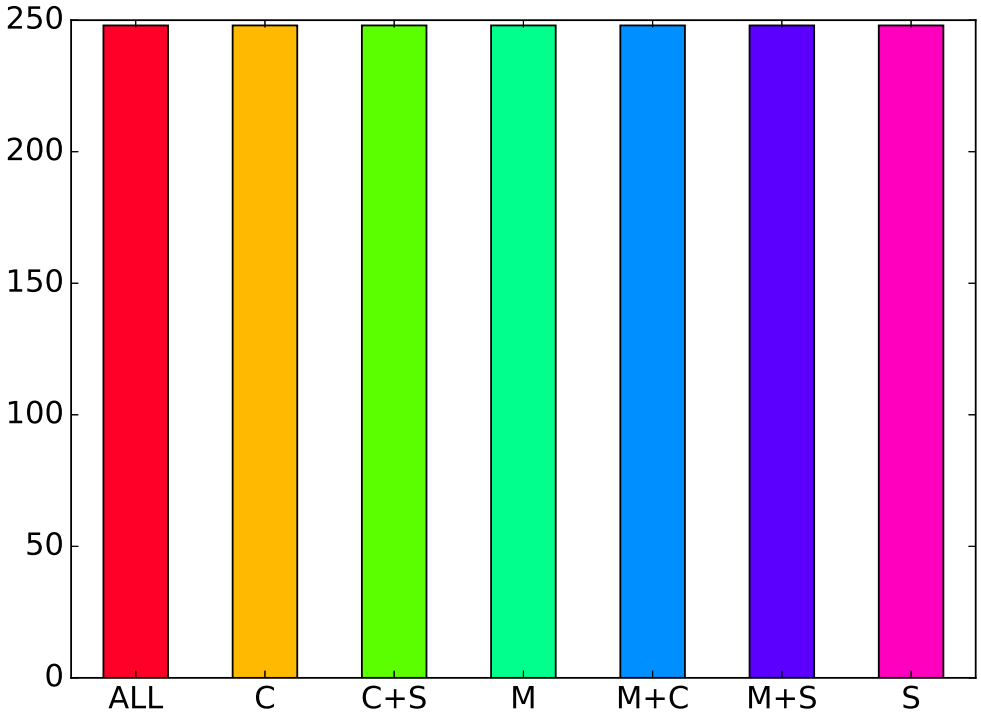
We can see in Figure 12a that the number of iteration when there was no issue is the same for all the visited test methods. On the other hand, Figure 12b does show difference in the number of iterations with the number of iteration for runs that included merge paths being lower then runs that did not include it, showing that merge paths indeed does less iterations.

Given the difference in run-time in Figure 11a over the same number of iterations we can conclude that the run-time for each iteration is the lowest for successful visited and the highest for cycle avoidance. This is further shown by the fact that indeed merge paths with cycle avoidance is slower then merge paths with successful visited. Furthermore since all the method gave the same number of iteration, running all of them together means more calls which explains why combining them all was the slowest method. In addition, the fact that successful visited had the same mean time as merge paths in Figure 11b but slightly more iteration in Figure 12b we can see that if the difference between the number of iteration is insignificant between merge paths and successful visited, using successful visited can be more beneficial.

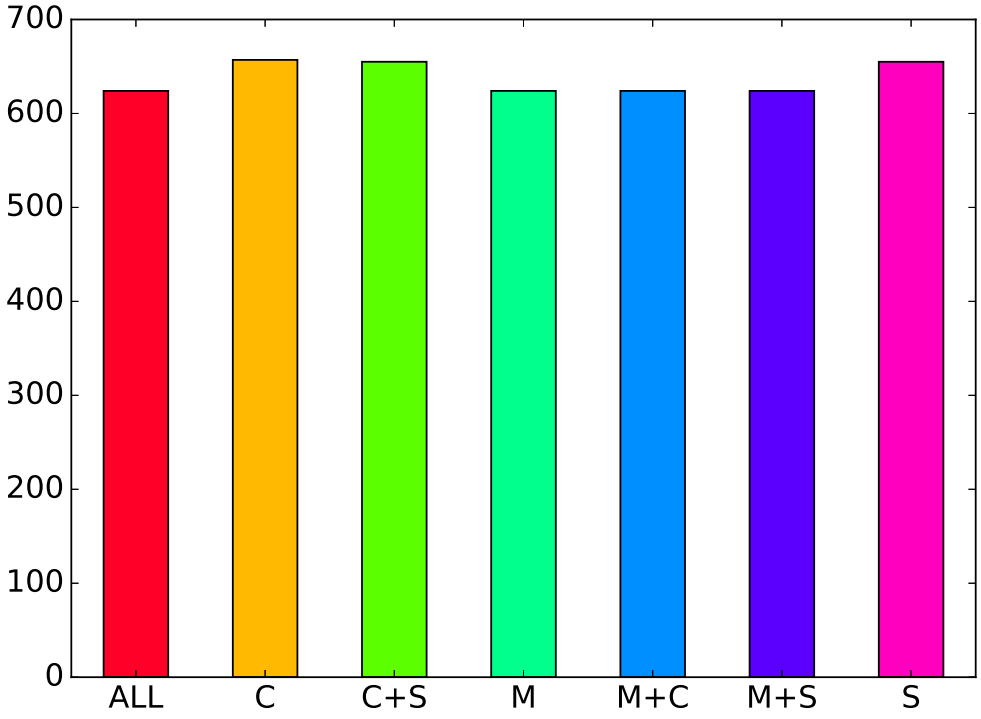
In the experiments running with the acyclic graph once lookahead found no paths that are possible we ran BIS again with the same graph. Thus we can look at the whole run of the scenario, when the screwdriver wast taken, in terms of the two runs of BIS, the first until the resting point where the screwdriver was taken, the second once we rerun BIS and Nao returned to get the screwdriver and then eventually fixed the drawer.

Figure 13 shows us the number of iteration for the first run of BIS and the second run of BIS for each visited method (splits the number from Figure 12a) over the two BIS runs). The left bar shows the first run and the right bar shows the second run. We can see in this graph that the number of iteration in the first run is identical for visited check methods. However the number of iteration for the second run is different and merge paths has the lowest number of iteration. What happen in this second run that made the difference?

For that reason we compared the number of iteration for each decision point, that is each call to lookahead. First lets look at the number of iteration



(a) Sum number of iteration no issue.



(b) Sum number of iteration screwdriver taken.

Figure 12: Sum number of iteration (acyclic)

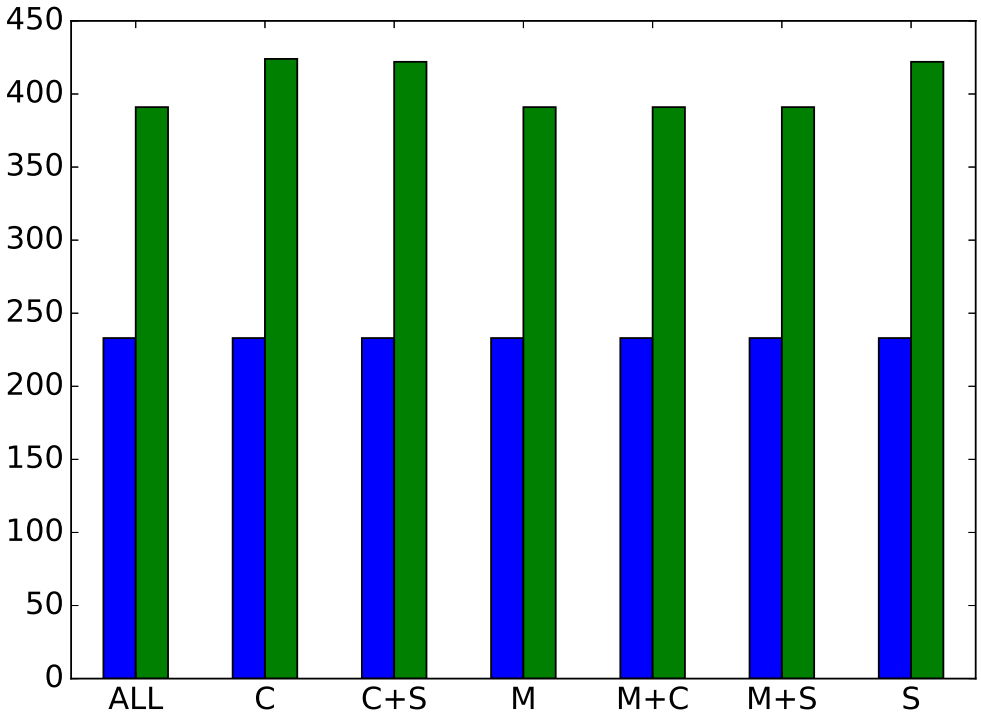


Figure 13: Total number of iteration for all calls in one scenario (acyclic).

when there was no issue. Figure 14 shows us the number of iteration for each call to lookahead ordered from the first call (when *tool_shed* ended) to the last one (when *mission_completed* ended) with only one run of BIS, since there is no issue. We can see that indeed the further we go in the recipe the less *gpath*s we have and thus less *xpath*s, the less iteration lookahead does.

On the other hand, Figure 15 shows us the number of iteration for each call to lookahead ordered from the first call (when *tool_shed* ended) to the last one (when *mission_completed* ended) when we took the screwdriver away, where $face\_west$ is the first call to lookahead in the second run of BIS.

The number of iteration is only different between the methods in $face\_west$. Notice that since the first BIS run starts from the initial position $face\_west$ is never chosen in that first run ($face\_west$ is a direct child of *tool_shed* and has preconds($face\_west$) = $\{(facing\_west, False), (at\_init, False)\}$). For that reason in the first run $from\_init$ is chosen and since when $from\_init$ finishes then *tool_shed* also end, then we have less behavior in the paths in the start of the first run then are in the start of the second run that can create the factors for the visited check method to work on. Thus we can see that indeed the number of behavior in the path effects the runtime. This still follows the conclusion from Figure 15, the further in the recipe we are the less iteration lookahead does.

8.3 Cyclic Recipe

We now consider the effects of the algorithm on a recipe that has cycles in it. This recipe solves the problem of the flat recipe, there will always be a feasible path to the end, no need to restart BIS. However, we found that just monitoring is not enough, there is a need to look in the results of the monitoring to make conclusion on what to do next. For example in this scenario if we are to do a further look in to all the possible paths gathered by lookahead we will see that all paths included the node *tools_shed*, thus we can conclude we need to go back to the tool shed to complete the mission. In a sense, this now allows full planning using the recipe, which is outside the scope of this work. We did not implement this, and further discuss this in Section 9.

We then look further on the effect of the visited methods on the algorithm. cycle avoidance is no longer a none factor, this time actually removing more nodes then just $NAIVE_1$.

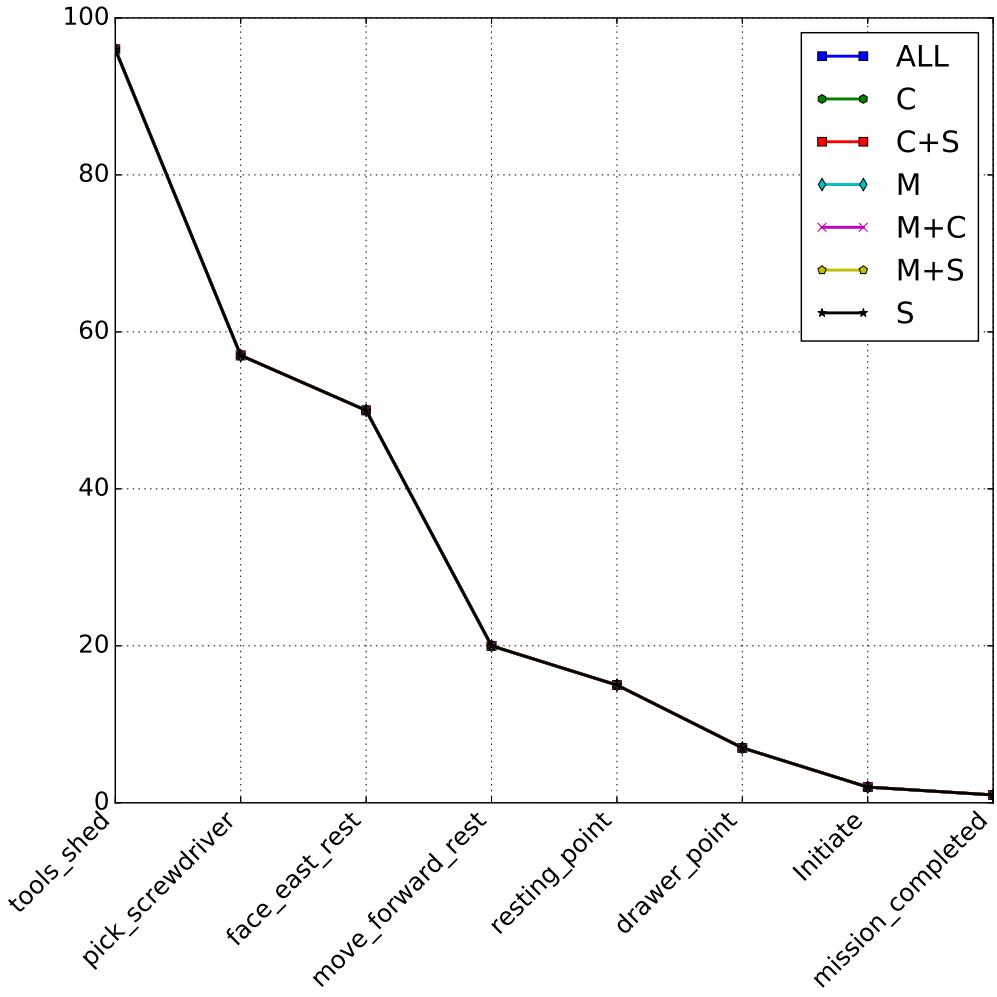


Figure 14: Iteration per decision point, flat recipe no issue.

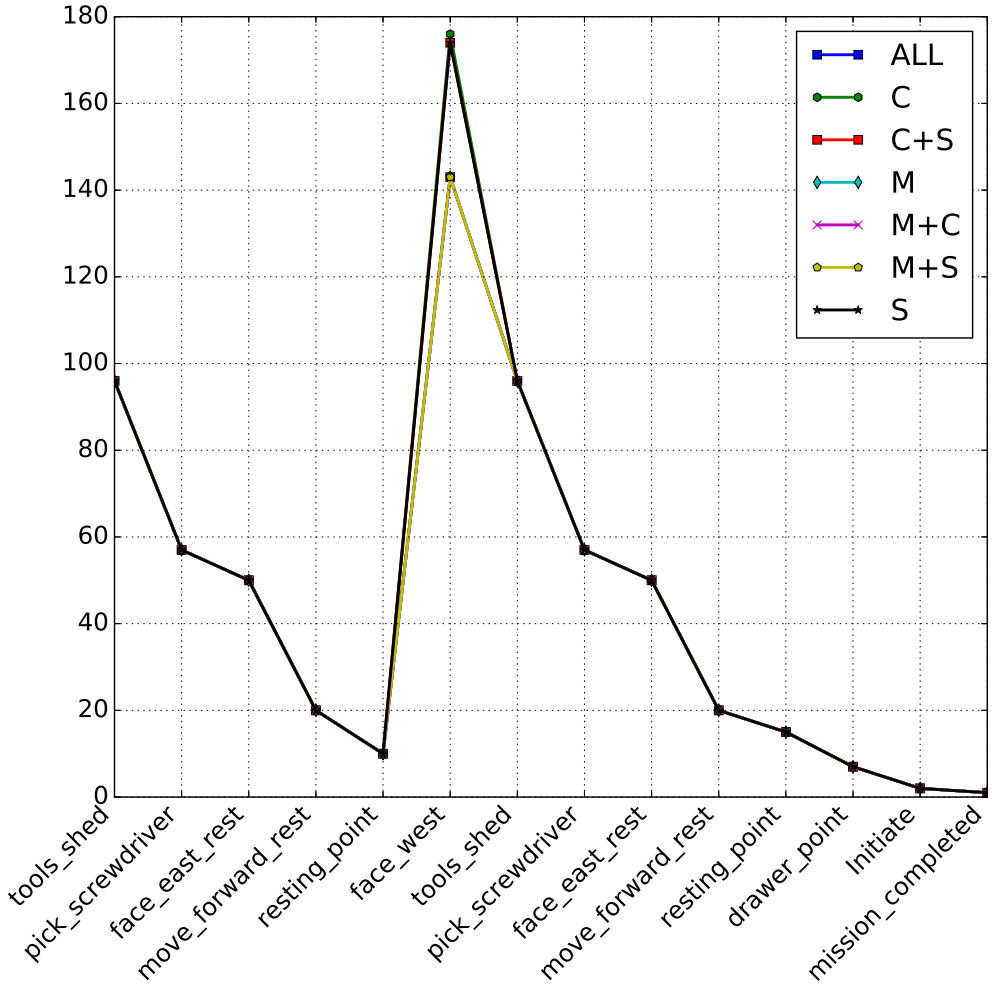


Figure 15: Iteration per decision point, flat recipe screwdrive taken.

We start again with looking at the total time spent in lookahead. Figure 16 shows the total time spent in lookahead using the cyclic recipe and having no issue. Figure 17 show the total time spent in lookahead using the cyclic recipe and taking the screwdriver at the resting point. They are both divided in to three graphs:

Figure 16a and Figure 17a show the run time for all the methods without and with issue respectively. We can see that cycle avoidance did the worst by a lot. Its important to note that cycle avoidance is the only method that reached the time limit, did not finish in under 60 seconds of CPU time. This is a specially significant in the case where there is no issue, where this is a big overhead on the decision process that does not effect it at the end. In the case where there is an issue, if the issue is not a no return situation, this overhead, in the case of cycle avoidance, can be the same as not detecting and going back from a dead end.

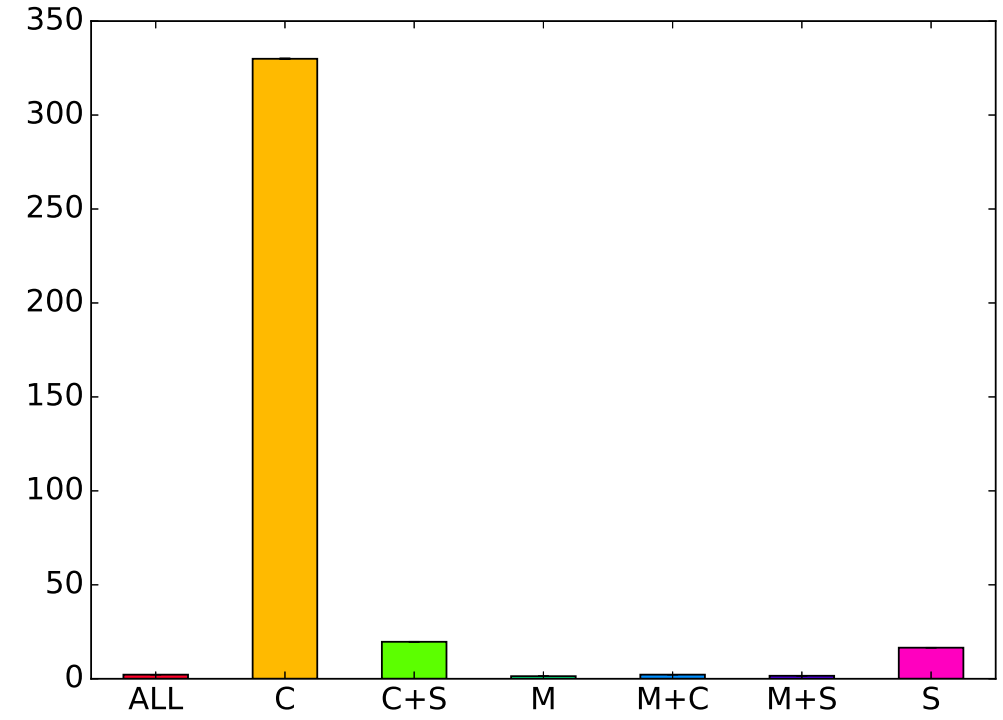
Figure 16b and Figure 17b removes cycle avoidance from the graphs without and with issues respectively, and focus on the other methods. We can see that this time successful visited did a lot worst then merge paths or any combination with merge pathsboth when there was no issue and when there was an issue. This is in contrast to the acyclic graph, where successful visited had the same runtime when needing to go back and slightly faster when there was no problem. This means that this time the difference in the number of iteration was not insignificant. In addition, we can see that while successful visited is significantly better then cycle avoidance it can still give more then 10 seconds of overhead, that may or may not save time latter.

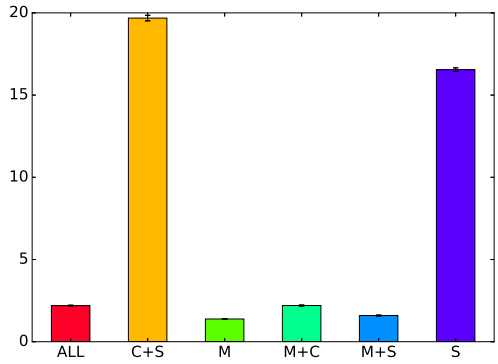
Figure 16c and Figure 17c focuses only on merge paths and the combination with it, without and with issues respectively. We can see that has before merge paths has the best runtime, followed by merge paths with successful visited (which we showed previously has the best runtime per iteration) and then merge paths with cycle avoidanceand all together. This concurs with our previous findings. We can also see that in both cases when using merge paths the runtime is still very low, and when there is no issue it is just a little over a second. When there is an issue it a almost 4 seconds, this is more significant increase but one that can save us more latter.

Remember that in Section 7 we did not include successful visited by itself since it did as bad as $NAIVE_1$. Contrary to section 7 in this experiment cycle avoidance did the worst, where successful visited did worse than merge paths but not close to cycle avoidance.

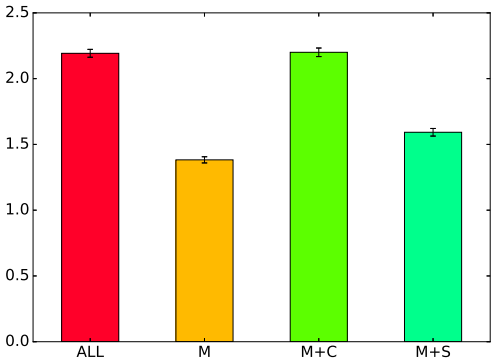
Next we look at the number of iterations each visited check method did.

(a) Mean time spent in lookahead all methods.





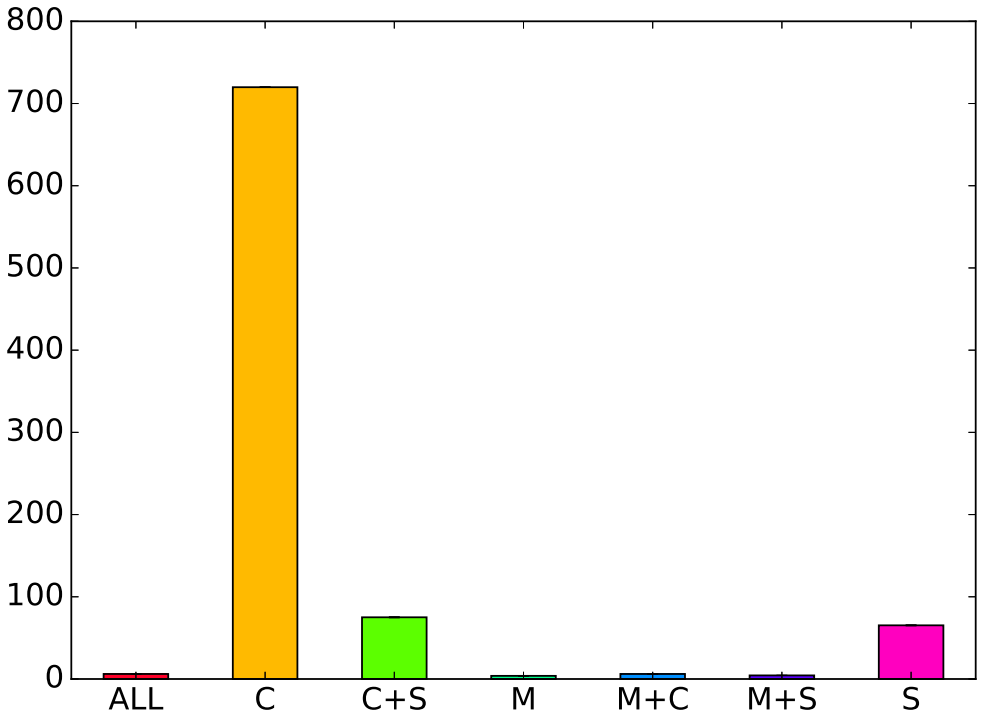
(b) Mean time spent in lookahead without cycle avoidance.

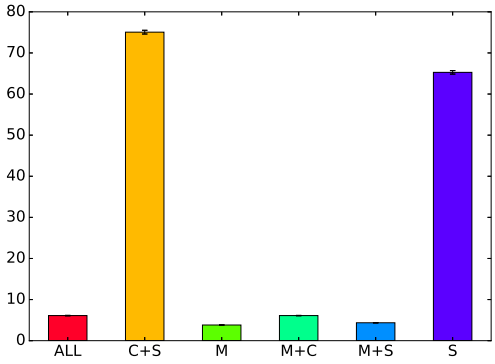


(c) Mean time spent in lookahead only method with merge paths.

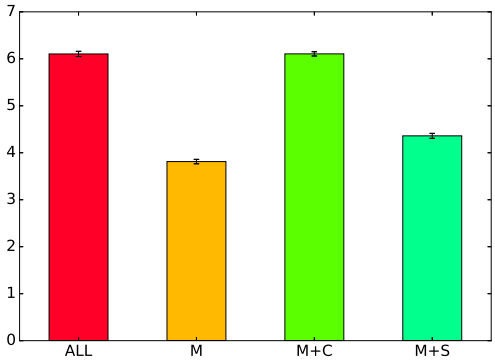
Figure 16: Mean time spent in lookahead (cyclic, no issue)

(a) Mean time spent in lookahead all methods.





(b) Mean time spent in lookahead without cycle avoidance.



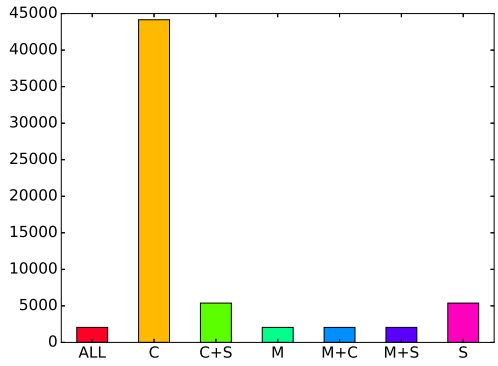
(c) Mean time spent in lookahead only method with merge paths.

Figure 17: Mean time spent in lookahead (cyclic, screwdriver taken)

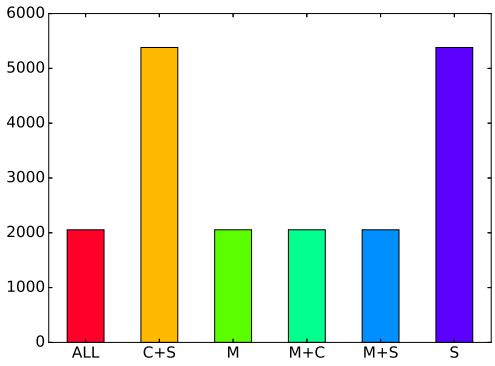
Figure 18 shows the total number of iteration done in each run of the scenario with the different visited check methods. At the top, Figure 18a we have the number of iteration all calls to lookahead done when there was no issue for all methods. Figure 18b shows a closer look at the methods without cycle avoidance since cycle avoidance did a lot more iterations (after all some of it runs where stopped at the time limit) and the scale makes it difficult to see the differences between the other methods. The bottom part of Figure 18 shows the number of iteration for of all the calls to lookahead when the screwdriver was taken. Again we separate in to two graphs, Figure 18c show the number of iteration for all methods, while Figure 18d show the number of iteration where cycle avoidance was removed. This time the number of iteration and the run-time are more correlated, merge paths and all the combinations including it are the best both in the lower run-time and in the lower number of iteration, successful visitedis next and cycle avoidancedoes the worst. Notice that in contrast to the acyclic recipe, the number of iteration in successful visitedis much higher then merge paths. We can conclude then that while the run-time of each iteration of successful visitedmight be a little faster in our implementation, in more complex recipes where the number of iteration can vary significantly between the two visited check methods merge paths is the better.

Figure 19 shows the number of iteration per at each decision point from the first decision to the last, when there was no problem. Each decision point is represented by the top terminated behavior, the behavior e at the end of the loop in lines 16-20 in Alg. 1. The top graph, Figure 19a shows us the number of iteration per decision point for all visited methods. Remember that cycle avoidance did not finish most its runs under 60 seconds. This is reflected in the graph both by the significant difference between cycle avoidance and the rest of the methods, and by the fact that cycle avoidance does not follow the same trends as the other methods, this is since it stopped before finishing thus the number of iteration is not represented of the number iteration it took to finish the lookahead like the rest of the methods, but rather the number of iteration it manged to do under 60 seconds, with the exception of three decision points ($move\_forward\_rest, initiate, mission\_completed$) which correlate to the three decision points with the lowest number of iteration in the rest of the methods as well.

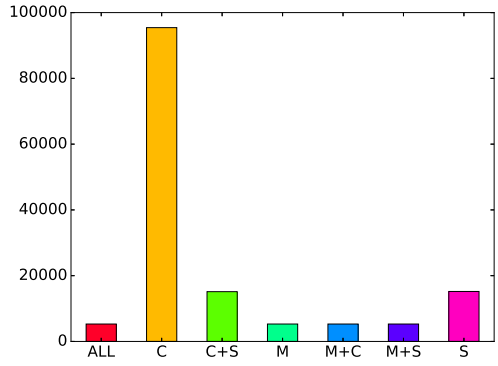
Thus we compare the rest of the methods in Figure 19b where cycle avoidance is removed and we see the number of iteration per decision point when there was no issue for the rest of the methods. In contrast to the



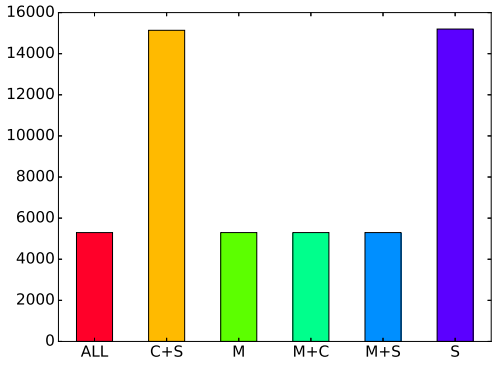
(a) Sum number of iteration all methods (no issue).



(b) Sum number of iteration without cycle avoidance (no issue).



(c) Sum number of iteration all methods (screwdriver taken)



(d) Sum number of iteration without cycle avoidance (screwdriver taken)

Figure 18: Sum number of iteration (cyclic)

number of iteration per decision point in the acyclic recipe (Figure 14) where the number of iteration was reduced the closer we got to the end, in the case of the cyclic recipe the number of iteration does not depend on how close to achieving the goal we are. This is because even thou we still have a clear end there are a lot of cycles in the graph so reaching the end is not a straight line and thus we have more possibilities.

First thing we can see in this graph is that when a behavior has more sequential edges (but outgoing and in going) lookahead does more iterations. *tool_shed* and *resting_point* has more sequential edges then *drawer_point*, and indeed lookahead when *tool_shed* and *resting_point* terminate does more iterations then when *drawer_point* terminates. The same goes for *drawer_point* and *pick_screwdriver*, where *drawer_point* has more sequential edges and lookahead does more iteration when *drawer_point* terminates then when *pick_screwdriver* terminates.

Second we can see that there is an importance on how many times a key is present in the recipe. Lookahead does more iteration when *tool_shed* terminates then when *resting_point* terminates even thou they have the same number of sequential edges. However, *pick_screwdrive* is the only behavior that changes a key in the database, *has_screwdriver* and nothing else can change that but external forces. When *tool_shed* terminates Nao still does not have the screwdriver thus lookahead has more possibilities of when *pick_screwdriver* will be in the *xpath*. However when *resting_point* terminates Nao has the screwdriver and since the precondition of *pick_screwdriver* is $has\_screwdriver = False$ and no other behavior touches *has_screwdriver* pick screwdriver will never be picked, thus there are less possible *xpath* and less iterations.

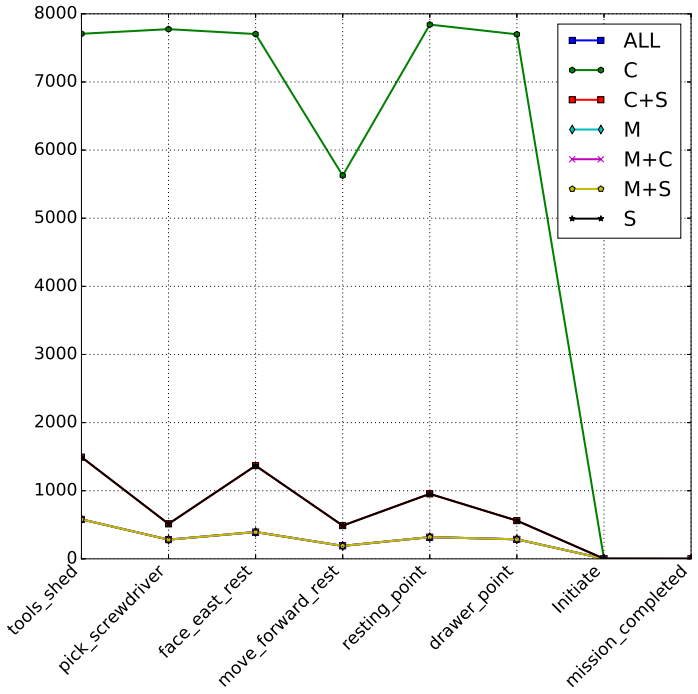
This is also an example of when repeated calls assumption are broken. *has_screwdriver* is an internal condition, *pick_screwdriver* changes it. However, when we took the screwdriver *has_screwdriver* was changed to be an external condition. In this calls for lookahead we will never return to *pick_screwdriver* since its precondition is always false. If we after taking the screwdriver we were to preform the repeated calls algorithm (Alg 12) it would have returned no path, since there was no remedy for the taken screwdriver in the paths returned from the previous call.

Lets then take a look at what happens when we take the screwdriver. Figure 20 shows the number of iteration at each decision point when we take the screwdriver while Nao is at *relax_a*. Figure 20a shows the number of iteration at each decision point, again each decision point is represented by

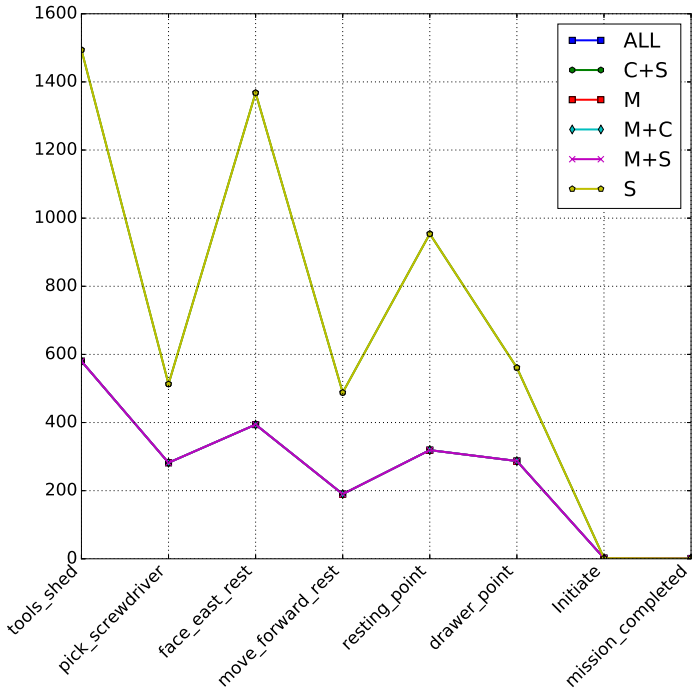
the last terminated behavior from Alg 1, for all the methods. Again cycle avoidance did not finish most its runs and thus does not give us insight over lookahead algorithm behavior.

Figure 20b shows the number of iteration for each decision point for all methods except cycle avoidance. We can see that the start is the same as Figure 19b until *move_forward_rest*, the next behavior is *resting_point* but this time Nao does not have the screwdriver when it finishes. We can see that the first call when *resting_call* terminated and Nao does not have the screwdriver does more iterations then all the rest of the calls. This time lookahead can enter *pick_screwdriver* after finishing *resting_point*, and even the shortest path to reach success is longer then what it was when Nao was without the screwdriver in *tool_shed* which is way we have more iteration then when *tool_shed* terminated.

However, in this case there is one trend that is different between merge paths and successful visited. In merge paths the number of iteration when *drawer_point* is terminated without having the screwdriver (the first instance of *drawer_point* in the graph), and the number of iteration when *tool_shed* is terminated (also without having the screwdriver) is lower with the termination of *drawer_point*. On the other hand, in successful visitedthis is the opposite. We saw two factors that can effect the number of iteration: 1. the number of sequential edges (both in and out going) 2.the length of the shortest path to success. Notice that the second factor, the shortest path to success effect more strongly the behavior of successful visited then merge paths. Successful visited needs at least one successful path before it starts eliminating search nodes that are not identical, merge paths on the other hand just need the knowledgebase, behavior and check type to be identical in order to eliminate search nodes. Thus successful visited is more effected from the length of the shortest path to success then merge paths. *drawer_point* indeed have less sequential edges but a longer shortest path to success then *tool_shed* thus it takes more iteration from the end of *drawer_point* rather from the end of *tool_shed* for successful visited to finish, but this does not effect merge paths as much and thus the number of iteration form the termination of *tool_shed* and *drawer_point* is not significant, since the this two factors are effecting in different directions.

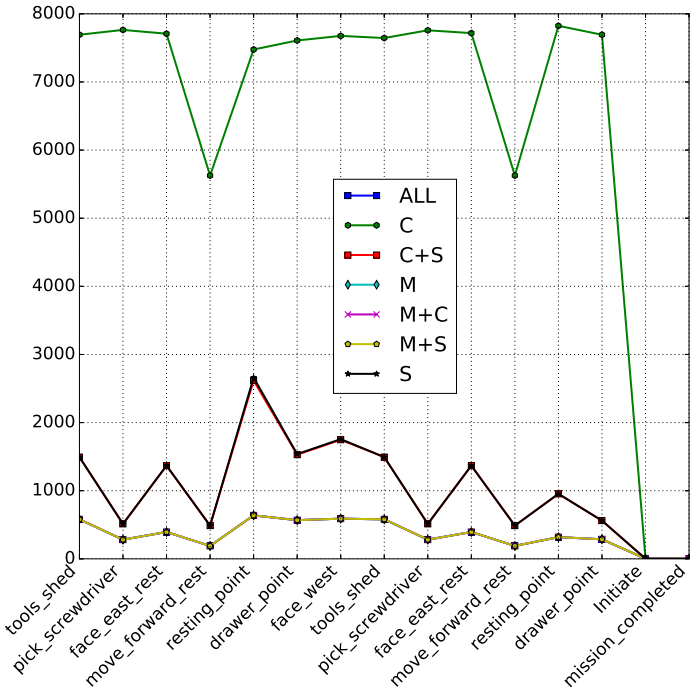


(a) Iteration per decision point (no issue).

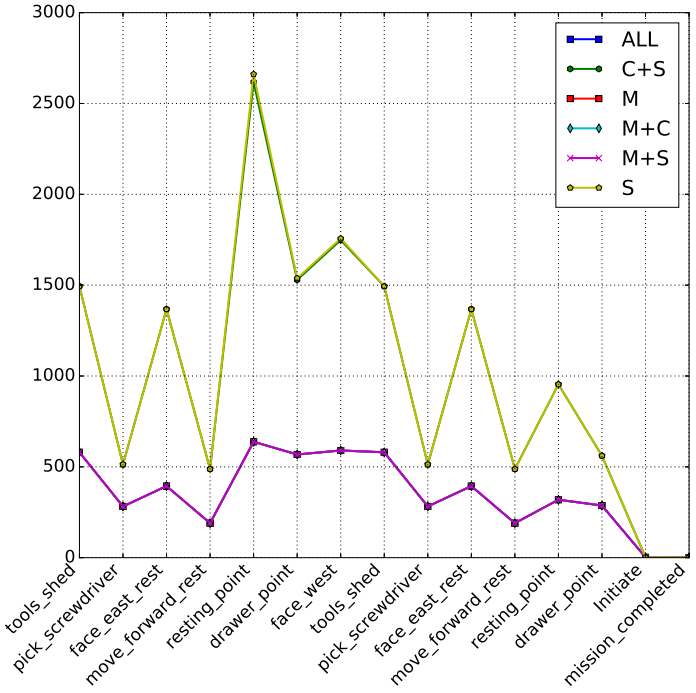


(b) Iteration per decision point without cycle avoidance (no issue).

Figure 19: Iteration per decision point (cyclic, no issue)



(a) Iteration per decision point (screwdriver taken)



(b) Iteration per decision point without cycle avoidance (screwdriver taken)

Figure 20: Iteration per decision point (cyclic, screwdriver taken)

9 Discussion

9.1 Revise Function

The algorithm presented so far has used a function $REVISE$ to change the knowlegebase in the expansion function. This function was not discussed, as it stands for a general belief revision procedure, by which a new belief is incorporated into a knowledgebase. However, there are two issues that merit further note.

First, the $REVISE$ function received by the lookahead algorithm does not have to be the same function received by BIS. This is due to the different values of different keys. Some of this values cannot be fully deduced by the precondition and termination condition. For example, let say a robot represents the value of its location by a structure that holds the x and y value. Then if the termination condition of a behavior only refers to the y value of the location, we cannot deduce the x value of the location, in this case $REVISE$ should decide whether x stays the same or become unknown. Another example for this is in numerical values and conditions with operation different then equal (greater then, lesser then etc.). The $REVISE$ function of the agent might know how to change this value to the exact number while the agent is running; however, the $REVISE$ function of lookahead will not. Thus will need to treat this revision differently then when the agent is actually running.

The second thing to note is that the algorithm assumes that $REVISE$ keeps the knowlegebase consistent, that is there are no contradictions between different key values that are connected. For example in the recipe in Fig. 1 notice that we have several keys that represent the location of the robot: $at\_init, at\_toolshed, at\_rest, at\_drawer$. We assume that the revise function will not set two of this keys to be *true* at the same time. In addition, $REVISE$ needs to know that if one of them is true then the rest cannot be unknown (have the value *unknown*) but has to be false.

9.2 No feasible paths

Another thing that should be addressed is what happens when there are no feasible paths in the recipe. In this case, there is no course of action the agent can take. For our experiment with the robot we addressed this simply by re-executing the same recipe, since we knew that it has a solution. This

is not the case for all BIS recipes.

There are two solutions for this case that need to be further explored in the future. One is the solution presented in classic BDI, having a plan (or recipe) library. If the recipe that was used failed, and there are no possible paths found, choose a different recipe from the library. Anther solution is to use a planner using the knowledge we have and the behaviors of the library, rearranging the behaviors from the recipe in a way that will yield feasible paths.

9.3 Choosing between feasible paths

Another problem that arose from the experiment on the NAO is the following. When we do have feasible paths as in Section 8.3 but some are better then others. As we saw in those experiments, if we explore the feasible paths returned we can learn that we always have to go to the tool shed in order to complete the mission. If we can pass this information to the behavior selection methods, it can make a more informed decision, preferring to go to the tool shed at the earliest possible time. Of course a simple solution to this is make the behavior selection method consider whether it has the screwdriver or not in its decision when the robot has the possibility to to go to the tool shed. However, this is equivalent to back-propagating all the precondition of following behavior to each behavior in the design of the recipe in the first place. In small recipes this is easy to do, but the more complex recipe the more this can be problematic and not at all effective.

We can see then our algorithm can be a first step not only in refining a recipe, but also in refining a plan of action for the agent that is more compatible with the world that it encounters (represented by its knowledgebase), using the same recipe for a different scenario. We can use the knowledge collected by the algorithm with further processing to deduce more information for the choose method to consider when choosing the next behavior.

10 Conclusion

In this paper we presented the problem of predictive execution monitoring of in layered hierarchal recipes; 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 recipe graph to try and find the branches that are predicted to fail. We then presented different search node visited checks methods to make the algorithm more efficient and reduce the running time. We proved that this visited method are complete.

Next we explored how to use the data already obtained by previous runs of the monitoring to reduce the running time and traverse less $xpath$s, including ones that were eliminated in the past runs.

We then showed experimental results, run over multiple 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 visited 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 visited method is not always beneficial, and in most cases using all the visited methods together is even detrimental.

Finally we showed how monitoring can effect real robot problems by running a simple scenario on a NAO robot. We showed that without execution monitoring the robot can fail its mission. In addition we ran the scenario multiple times to measure the real life effect of the time it takes the robot to execute the monitoring algorithm, with the different visited methods. We saw that merge paths can significantly reduce the runtime, from more then a minute with only cycle avoidance in the cyclic path to a little less then a 1.5 seconds. We saw that successful visited can sometimes preform better then merge paths, but only if there is an insignificant amount of iteration difference between them. Once the number of iteration became large, specifically when cycles where involved, then merge paths is better by far then the rest.

11 Bibliography

[1] J. A. Ambros-Ingerson and S. Steel. Intergrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, Minneapolis/St. Paul, MN, 1988. AAAI Press.

[2] T. Belker, M. Hammel, and J. Hertzberg. Learning to optimize mobile robot navigation based on HTN plans. In *IEEE International Conference on Robotics and Automation*, volume 3, pages 4136–4141. IEEE, 2003.

[3] A. Bouguerra, L. Karlsson, and A. Saffiotti. Monitoring the execution of robot plans using semantic knowledge. *Robotics and Autonomous Systems*, 56(11):942–954, 2008. Online at http://www.aass.oru.se/˜asaffio/.

[4] P. R. Cohen, M. S. Atkin, and E. A. Hansen. The interval reduction strategy for monitoring cupcake problems. In D. Cliff, P. Husbands, J.-A. Meyer, and S. W. Wilson, editors, *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, 1994.

[5] P. R. Cohen, R. St. Amant, and D. M. Hart. Early warnings of plan failure, false positives, and envelopes: Experiments and a model. Technical Report CMPSCI Technical Report 92-20, University of Massachusetts, 1992.

[6] L. de Silva and L. Padgham. A comparison of BDI based real-time reasoning and htn based planning. In *Australasian Joint Conference on Artificial Intelligence*, pages 1167–1173. Springer, 2004.

[7] L. de Silva and L. Padgham. Planning on demand in BDI systems. In *International Conference on Automated Planning and Scheduling*. University of Southern California, 2005.

[8] L. de Silva, S. Sardina, and L. Padgham. First principles planning in BDI systems. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems*, volume 2, pages 1105–1112. IFAAMAS, 2009.

[9] R. J. Doyle, S. A. Chien, U. M. Fayyad, and E. J. Wyatt. Focused real-time systems monitoring based on multiple anomaly models. In *International Qualitative Reasoning Conference (QR'93)*, Eastsound, WA, 1993.

[10] C. Earl and R. J. Firby. Combined execution and monitoring for control of autonomous agents. In W. L. Johnson, editor, *Proceedings of the First International Conference on Autonomous Agents (Agents-97)*, pages 88–95, Marina del Rey, CA, 1997. ACM Press.

[11] K. Erol, J. Handler, and D. S. Nau. Complexity results for HTN planning. *Annals of Math and Artificial Intelligence*, 18(1):69–93, 1996.

[12] K. Erol, J. A. Hendler, and D. S. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *International conference on AI Planning and Scheduling (AIPS)*, volume 94, pages 249–254, 1994.

[13] G. P. Farias, R. F. Pereira, L. Hilgert, F. Meneguzzi, R. Vieira, and R. H. Bordini. Predicting plan failure by monitoring action sequences and duration. *Advances in Distributed Computing and Artificial Intelligence*, 6(4):55–69, 2017.

[14] R. E. Fikes. Monitored execution of robot plans produced by strips. In *Proceedings of the IFIP Congress*, pages 189–194, 1971.

[15] C. Fritz. Execution monitoring – a survey. Available at: `https://pdfs.semanticscholar.org/199b/08d06dda4f0665d472b81f418080c0facdac.pdf`, 2005.

[16] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The belief-desire-intention model of agency. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 1–10. Springer, 1998.

[17] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *AAAI*, volume 87, pages 677–682, 1987.

[18] I. Georgievski and M. Aiello. HTN planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156, May 2015.

[19] M. Ghallab, D. Nau, and P. Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[20] Giuseppe De Giacomo, Fabio Patrizi, and Sebastian Sardina. Automatic Behavior Composition Synthesis. *Artificial Intelligence*, 196:106–142, 2013.

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

[22] A. E. Howe and P. R. Cohen. Understanding planner behavior. *Artificial Intelligence*, 76(1–2):125–166, 1995.

[23] F. Ingrand and M. Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44, June 2017.

[24] M. Kalech. Diagnosis of coordination failures: a matrix-based approach. *Autonomous Agents and Multi-Agent Systems*, 24(1):69–103, 2012.

[25] G. A. Kaminka and I. Frenkel. Flexible teamwork in behavior-based robots. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, 2005.

[26] G. A. Kaminka, D. V. Pynadath, and M. Tambe. Monitoring teams by overhearing: A multi-agent plan recognition approach. *Journal of Artificial Intelligence Research*, 17:83–135, 2002.

[27] G. A. Kaminka and M. Tambe. What's wrong with us? Improving robustness through social diagnosis. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 97–104, Madison, WI, 1998. AAAI Press.

[28] G. A. Kaminka and M. Tambe. I'm OK, You're OK, We're OK: Experiments in distributed and centralized social monitoring and diagnosis. In *Proceedings of the Third International Conference on Autonomous Agents (Agents-99)*, pages 213–220, Seattle, WA, 1999. ACM Press.

[29] G. A. Kaminka and M. Tambe. Robust multi-agent teams via socially-attentive monitoring. *Journal of Artificial Intelligence Research*, 12:105–147, 2000.

[30] E. Khalastchi and M. Kalech. On fault detection and diagnosis in robotic systems. *ACM Computing Surveys*, 51:1–24, 2018.

[31] E. Khalastchi, M. Kalech, G. A. Kaminka, and R. Lin. Online data driven anomaly detection in autonomous robots. *Knowledge and Information Systems*, 43(3):657–688, 2015.

[32] R. Lallement, L. de Silva, and R. Alami. HATP: An HTN planner for robotics. *2nd ICAPS Workshop on Planning and Robotics*, 2014.

[33] R. Lin, E. Khalastchi, and G. A. Kaminka. Detecting anomalies in unmanned vehicles using the Mahalanobis distance. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-10)*, 2010.

[34] S. C. Marsella, J. Adibi, Y. Al-Onaizan, G. A. Kaminka, I. Muslea, M. Tallis, and M. Tambe. On being a teammate: Experiences acquired in the design of robocup teams. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(1–2), 2001.

[35] C. E. McCarthy and M. E. Pollack. Towards Focused Plan Monitoring: A Technique and an Application to Mobile Robots. *Autonomous Robots*, 9(1):71–81, 2000.

[36] J. P. Mendoza, M. Veloso, and R. Simmons. Plan execution monitoring through detection of unmet expectations about action outcomes. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3247–3252. IEEE, 2015.

[37] C. Muise, S. A. McIlraith, and J. C. Beck. Monitoring the Execution of Partial-Order Plans via Regression. In *Proceedings of the International Joint Conference on Articial Intelligence*, page 8, 2011.

[38] A. Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1990.

[39] O. Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.

[40] O. Pettersson, L. Karlsson, and A. Saffiotti. Model-free execution monitoring in behavior-based robotics. *IEEE Trans. on Systems, Man and Cybernetics, Part B*, 37(4):890–901, 2007. Online at http://www.aass.oru.se/˜asaffio/.

[41] M. E. Pollack. Plans as complex mental attitudes. In *Intentions in Communication*, pages 77–103. MIT Press, 1990.

[42] M. Ramirez, N. Yadav, and S. Sardina. Behavior Composition as Fully Observable Non-Deterministic Planning. In *ICAPS*, 2013.

[43] A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. Technical Note 56, Australian Artificial Intelligence Institute, Apr. 1995.

[44] S. Sardina, L. de Silva, and L. Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 1001–1008. ACM, 2006.

[45] M. Tambe, W. L. Johnson, R. Jones, F. Koss, J. E. Laird, P. S. Rosenbloom, and K. Schwamb. Intelligent agents for interactive simulation environments. *AI Magazine*, 16(1), 1995.

[46] M. Tambe and W. Zhang. Towards flexible teamwork in persistent teams. In *Proceedings. International Conference on Multi Agent Systems*, pages 277–284. IEEE, 1998.

[47] M. M. Veloso, M. E. Pollack, and M. T. Cox. Rationale-Based Monitoring for Planning in Dynamic Environments. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 171–179, Pittsburgh, PA, 1998.

[48] A. Walczak, L. Braubach, A. Pokahr, and W. Lamersdorf. Augmenting BDI agents with deliberative planning techniques. In *International Workshop on Programming Multi-Agent Systems*, pages 113–127. Springer, 2006.

[49] D. S. Weld. Planning-Based Control of Software Agents. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 268–274, 1996.

[50] D. E. Wilkins, T. Lee, and P. Berry. Interactive execution monitoring of agent teams. *Journal of Artificial Intelligence Research*, 18:217–261, 2003.

[51] A. Yakir and G. A. Kaminka. An integrated development environment and architecture for Soar-based agents. In *Innovative Applications of Artificial Intelligence (IAAI-07)*, 2007.

12 Hebrew Abstract