BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

ADVANCED INTENTION SCHEDULER FOR BDI SYSTEMS

POKROČILÝ PLÁNOVAČ ZÁMĚRŮ PRO BDI SYSTÉMY

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. ONDŘEJ MISAŘ**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. FRANTIŠEK ZBOŘIL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

164364

Institut:	Department of Intelligent Systems (DITS)
Student:	**Misař Ondřej, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Intelligent Systems
Title:	**Advanced intention scheduler for BDI systems**
Category:	Artificial Intelligence
Academic year:	2024/25

Assignment:

1. Study the approaches to selecting intentions to execute for BDI systems. Next, learn about the intent progression problem and its current solutions.
2. Extend current approaches to solving the problem to be feasible in BDI systems based on the language of first-order predicate logic. Focus on planning intentions by implementing projections using, for example, the MCTS method.
3. Implement such a scheduler for the FRAg multi-agent system in a suitable language and link it to the FRAg system.
4. Compare the performance of this scheduler against existing schedulers in FRAg and evaluate the results.
5. Create an A1 poster demonstrating the essential parts of this project.

Literature:

- Michael Wooldridge. 2009. An Introduction to MultiAgent Systems (2nd. ed.). Wiley Publishing.
- Brian Logan, John Thangarajah, and Neil Yorke-Smith. 2017. Progressing Intention Progression: A Call for a Goal-Plan Tree Contest. In Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems (AAMAS '17). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 768–772.
- Michael Dann, John Thangarajah, Yuan Yao, and Brian Logan. 2020. Intention-Aware Multiagent Scheduling. In Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS '20). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 285–293.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Zbořil František, doc. Ing., Ph.D.**
Head of Department:	Kočí Radek, Ing., Ph.D.
Beginning of work:	1.11.2024
Submission deadline:	21.5.2025
Approval date:	31.10.2024

Abstract

This thesis presents current approaches to intention scheduling in Belief–Desire–Intention (BDI) systems. Specifically approaches that incorporate the Monte Carlo Tree Search (MCTS) method with Goal–Plan trees (GPT) to represent agents' intentions with top-level goals. These methods are modified to be compatible with systems that are based on the AgentSpeak(L) language. In order to use Goal–Plan trees in such a system efficiently, a late variable binding strategy is introduced. The MCTS method with action level interleaving and online learning variant of MCTS method are implemented in Python language and integrated into the FRAg system. With two alternative architectures of integration with FRAg system being proposed. Finally, their performance is evaluated and compared against existing intention scheduling methods within FRAg system.

Abstrakt

Tato práce představuje současné přístupy k plánování záměrů v systémech typu Belief–Desire–Intention (BDI). Konkrétně se zaměřuje na přístupy, které zahrnují metodu Monte Carlo Tree Search (MCTS) ve spojení se stromy typu Goal–Plan (GPT) pro reprezentaci záměrů agenta s hlavními cíli. Tyto metody jsou upraveny tak, aby mohly být použity v systémech založených na jazyce AgentSpeak(L). Pro efektivní použití stromů Goal–Plan v takových systémech je zavedena strategie *late variable binding*. Metoda MCTS s prokládáním na úrovni akcí a varianta MCTS využívající zpětnovazební učení jsou implementovány v jazyce Python a integrovány do systému FRAg. Jsou navrženy dvě alternativní architektury integrace se systémem FRAg. Nakonec je porovnána jejich výkonnost s existujícími metodami plánování záměrů v rámci systému FRAg.

Keywords

intention selection, intention scheduler, BDI agent, Monte-Carlo tree search, MCTS, Goal-plan tree, AgentSpeak(L)

Klíčová slova

výběr záměrů, plánovač záměrů, BDI agent, Monte-Carlo tree search, MCTS, Goal-plan strom, AgentSpeak(L)

Reference

MISAŘ, Ondřej. *Advanced intention scheduler for BDI systems.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. František Zbořil, Ph.D.

Rozšířený abstrakt

Tato diplomová práce se zaměřuje na vývoj pokročilého plánovače záměrů pro systémy typu Belief–Desire–Intention (BDI). Hlavním přínosem je integrace metody Monte Carlo Tree Search (MCTS) do systému FRAg, který využívá jazyk AgentSpeak(L) pro definici programů agentů. Tradiční metody plánování záměrů, jako jsou Round Robin a náhodný výběr, neberou při plánování záměrů v potaz kroky, které mohou následovat po vybraném kroku. Proto se využívají metody MCTS, které používají dopředný pohled pro plánování dalších kroků. Tato metoda byla upravena pro práci se stromy Goal–Plan (GPT) s použitím strategie *late variable binding* pro efektivní reprezentaci GPT bez nutnosti předčasně vytvářet všechny možné prvky podle hodnot v prostředí. Dále byla rozšířena základní metoda MCTS o zpětnovazební učení pomocí state-action stromu, což zlepšilo využití historických dat o krocích v simulaci pro rychlejší nalezení kvalitní cesty.

Integrace navrženého plánovače záměrů do systému FRAg vyžadovala propojení Python implementace MCTS s Prolog implementací jádra FRAg. Byly navrženy dvě architektury pro integraci plánovače. První z nich je integrovaná, která spouští MCTS plánovač pomocí Python funkce volané přímo z Prologu. Druhá architektura je externí a je založena na spuštění vedlejší služby, která MCTS plánovač obsluhuje. V tomto případě je z Prolog kódu volána Python funkce, která s touto vedlejší službou komunikuje. Pro externí variantu architektury byly implementovány tři MCTS metody, sekvenční varianta MCTS, paralelní varianta MCTS a varianta MCTS využívající zpětnovazební učení.

Metody pro plánování záměrů byly následně vyhodnoceny na dvou experimentech. První experiment vyhodnocoval rychlost plánování mezi MCTS metodou implementovanou v Prologu a dvěma navrženými architekturami. Výsledky ukázaly, že pro větší výpočetní rozpočet jsou implementace v Pythonu rychlejší než implementace v Prologu. Také se ukázalo, že paralelní varianta je při vyšším výpočetním rozpočtu výrazně rychlejší než sekvenční varianta. Mezi navrženými architekturami pro seknveční variantu MCTS nebyl zaznamenán výrazný rozdíl v rychlosti plánování.

Druhý experiment vyhodnocoval kvalitu plánování několika plánovačů. Konkrétně šlo o metody náhodného plánování, Round Robin plánování a MCTS plánovač, který je již implementován v systému FRAg. Tyto byly porovnány s novými MCTS plánovači implementovanými v Pythonu. Výsledky ukázaly, že MCTS plánovače dosahují lepších výsledků než náhodné a Round Robin plánovače. Také se ukázalo, že mezi výsledky MCTS plánovačů implementovaných v Pythonu a v Prologu není výrazný rozdíl. Plánovač MCTS využívající zpětnovazební učení dosahoval v průměru lepších výsledků než ostatní MCTS plánovače.

Práce demonstruje, že plánovače záměrů využívající dopředné pohledy dosahují lepších výsledků než tradiční plánovače. Zároveň také ukazuje, že je možné tento typ plánovačů použít v systémech, které využívají jazyk AgentSpeak(L), a to při použití strategie *late variable binding* pro vytváření GPT. Budoucí práce by se mohla zaměřit na zrychlení nových plánovačů, a to buď přepracováním implementace do jiného programovacího jazyka, nebo postupnou optimalizací stávající Python implementace. Dále je možné se zaměřit na přidání nových plánovačů záměrů, které využívají metodu MCTS, ať už pro podporu rozšířených typů systémů s nejasnými cíli, nebo s uživatelskou preferencí hlavních cílů. Dalším vhodným rozšířením by byla také integrace plánovačů do multiagentního prostředí.

Advanced intention scheduler for BDI systems

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Doc. Ing. František Zbořil, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Ondřej Misař
May 21, 2025

Acknowledgements

I would like to thank my supervisor, Doc. Ing. František Zbořil, Ph.D., for his guidance and encouragement throughout the work on this thesis. I am thankful for his generosity with time and his willingness to have frequent consultations, which greatly contributed to the results of this thesis. I would also like to thank my family for their continuous support and encouragement throughout my entire academic journey.

Contents

List of Figures

Chapter 1

Introduction

Agent-based systems are a popular paradigm for implementing intelligent systems. Belief-Desire-Intention (BDI) systems are one of the well established designs for implementing rational agents that can make decisions in dynamic environments. One of the challenges that BDI systems face is intention scheduling, which determines how the agent should prioritize resources to achieve its desires. The fundamental question is whether there are multiple intentions that the agent can pursue. What is the most efficient prioritization of these intentions.

The Monte Carlo Tree Search (MCTS) method has emerged as a promising approach to intention scheduling. Mainly by using the random roll out over the Goal-Plan tree (GPT) in order to explore potential future payoff of different intention scheduling. Different variants of the MCTS method have been explored for the use of intention scheduling for single agent or multi agent systems. These new schedulers can be used by agents for selecting intentions, but most of these are not designed to work in systems based on first-order logic language, such as AgentSpeak(L). The main problem that needs to be addressed is the number of GPT nodes that are generated for every substitution based on the agent belief base, with the default behavior of early binding.

The main goal of this thesis is to study advanced intention scheduling methods. Mainly focusing on methods that use MCTS for its ability to incorporate look ahead. Study the late variable binding strategy that allows the MCTS method to be used efficiently in the FRAg system that is based on the AgentSpeak(L) language. Mitigating the problem of generating GPT nodes for every possible substitution. This approach promises a significant reduction in the size of GPT and hence improves the performance of the MCTS method. Implement and integrate a studied variations of the MCTS intention scheduling into the FRAg system. Compare the newly implemented schedulers to the schedulers that are already included in the FRAg system.

The thesis is structured as follows. Chapter 2 explores the current state of intention scheduling using the MCTS and its variation. Chapter 3 introduces the late variable binding strategy for efficient representation of Goal-Plan tree in AgentSpeak(L) based system and outlines the design of integrating new reasoning methods into an FRAg system. The implementation of the proposed methods is described in Chapter 4. Chapter 5 presents the results of the comparison of the newly implemented scheduling method with the intention scheduling methods that are included in the FRAg system. Finally, the thesis result is summarized in Chapter 6 with an outline for potential future work.

Chapter 2

Intention progression in Belief-Desire-Intention systems

This chapter describes a problem of intention scheduling in Belief-Desire-Intention systems. Then formally defines intention progression in the Goal-Plan tree. Next, a basic method of intention scheduling using the Monte Carlo tree search is described and subsequent advanced modifications that either introduce intention scheduling in different systems or better performing intention scheduling. Lastly, a application of Monte Carlo tree search in multi-agent systems is explored.

2.1 Belief-Desire-Intention architecture

Belief-Desire-Intention (BDI) is a framework used to build reactive agents [8, 9]. Here, the architecture is defined as three components. The belief represents what information agents currently know about the environment. These could be only partial or incorrect. Desire is defined as the state where the agent wants to achieve some objective. Here, we further specify the goal as a desire that is chosen by the agent and is consistent with other selected goals. Intention is a plan the agent has committed to doing. Where the plan contains actions and sub goals that include other plans. Intention itself is again compatible with the agent's goal. This means that the plan is only chosen if the agent believes that it will lead to the achievement of some desired goal.

Figure 2.1 shows the architecture of the BDI agent. Where, based on information from environment, a belief base is updated while potentially updating agents goals. Plans are usually specified by the developer and represent how goals can be achieved. Intentions are then choose plans that the agent has committed to doing in order to achieve goals. Where executed intentions update the environment agent is in. The interpreter is responsible for working with beliefs and goals, choosing potential plans that can be used to achieve goals, and scheduling which intention should be currently selected. Together, these represent a BDI agent.

The algorithm 1 shows the way in which the interpreter works with these components in each cycle, which is called the deliberation cycle. That is, split into three phases. The first phase entails choosing which event or events should be processed and updating belief base and goals accordingly. The second phase works with plans and selects plans that are applicable to achieve a specified goals and updates intentions based on these. During the

last phase, the interpreter chooses which intention should be selected and executes one or more steps of this intention.

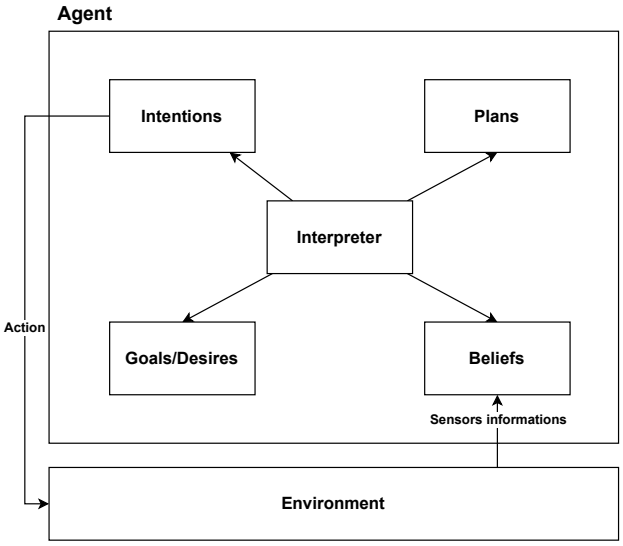

Figure 2.1: Belief desire intention architecture based on [1].

Algorithm 1 Deliberation cycle for BDI agent

Require: Initial beliefs, goal library, plan library

1: **while** agent is active **do**

Phase 1: Process event and update beliefs with goals

2: $Events \leftarrow$ PerceiveEnvironment

3: $B \leftarrow$ UpdateBeliefs$(B, Events)$

4: $G \leftarrow$ UpdateGoals$(G, Events)$

Phase 2: Plan selection and intention update

5: $ApplicablePlans \leftarrow$ FindApplicablePlans(B, G)

6: $I \leftarrow$ UpdateIntentions$(I, ApplicablePlans)$

Phase 3: Intention selection and execution

7: $i \leftarrow$ SelectIntention(I)

8: ExecuteIntention(i)

9: **end while**

2.2 Intention scheduling

BDI agents may have multiple desires and, more specifically, goals. In order to achieve a goal, the agent needs to define a plan or plans that will lead to the completion of the corresponding goal. In this way, we create an intention for each goal. However, if we have a set of intentions, how do we choose which intention and its plan should the agent pursue. The process of selecting the desired intention is that of intention scheduling [15]. Intention scheduling itself can include multiple criteria that are used for selection of intention. For

example, some of the criteria could be the importance of each intention, how many goals is selected intention achieving, minimization of conflicts between each intention, optimization of interleaving plans of each intention, or how committed should the agent be to selected intention. Because of this there are a lot of different ways a intention scheduling can be implemented, for example, Procedural Reasoning System uses meta level plans that help guide which plan should be selected. Due to the agent having limited resources, it is important to consider the performance of selected intention scheduling. Similarly, because agents may operate in dynamic environments, it is important for the scheduler to consider changes in beliefs that the agent may hold. Or if agent operates in more static environment it is beneficial for scheduler to be able to continue in already planned scheduling without recalculating the whole scheduling. Thanks to all of these aspects, the research around intention scheduling is still evolving to further optimize how BDI agents choose to execute their plans for intentions to achieve their goals.

Several agent frameworks implement a more simplistic approach to intention selection, such as First-in-Firs-Out. Where intentions are then selected based on order they were received, meaning that agent will not stop pursuing intention until it is not possible to continue, after which the intention is moved back in the queue or completed, and next intention is selected. Because of this, there is no interleaving between each intention. Another approach is Round Robin, where a fixed number of steps for each intention are executed in a cycle. In this way, we allow for some level of interleaving between each intention. But because this interleaving is done based on a fixed number of steps, it can lead to more conflicts between each intention. Another possible approach is to use coverage [14], which indicates how applicable a plan is in the current state. If a plan has a big coverage, it means that it is more applicable on a current goal. If the coverage is lower, it means that the plan is less applicable, meaning that the possibility of failure is greater. Lower coverage has a higher priority because there is a higher probability that we will not be able to achieve this intention later. Thanks to this a coverage-based intention selection prioritizes which intention should be persuade, and because we check the coverage of next applicable plans we are performing a simple look ahead.

2.3 Goal-Plan Tree intention progression

Currently, one of the popular approaches to intention scheduling is the incorporation of goal-plan trees (GPTs) in intention progression. Firstly, it is important to formally define the GPT structure and terminology used in the intention progression competition [6], because most of the new methods of intention progression are based on this specific GPT structure. GPT is then defined as a tree structure that has either AND or OR nodes and contains a goal or sub-goal node, a plan node, and an action node. Where the root node is a top-level goal that the agent wants to achieve. Every goal or sub-goal node needs to have at least one plan node as its child node, and these are OR nodes. Meaning, if goal or sub-goal node has multiple child plan nodes, we can choose any plan because all of the available plans should end up completing this goal. The plan then contains child nodes of either action node or sub-goal node, where these are AND nodes. That is because if we choose to perform a plan, we need to eventually do all the steps specified by the plan. Figure 2.2 shows an example of bigger GPT based on this specification, where G_0 is a top level goal of intention, for which one of three plans P_0, P_1 or P_3 can be selected. Completing any of these plans should lead to achieving the top level goal G_0. In order to complete the selected plan, every child node

needs to be performed. For example, by selecting plan P_1, action A_3 needs to be executed and goal G_1 needs to be achieved. Here we continue the same way for goal G_1.

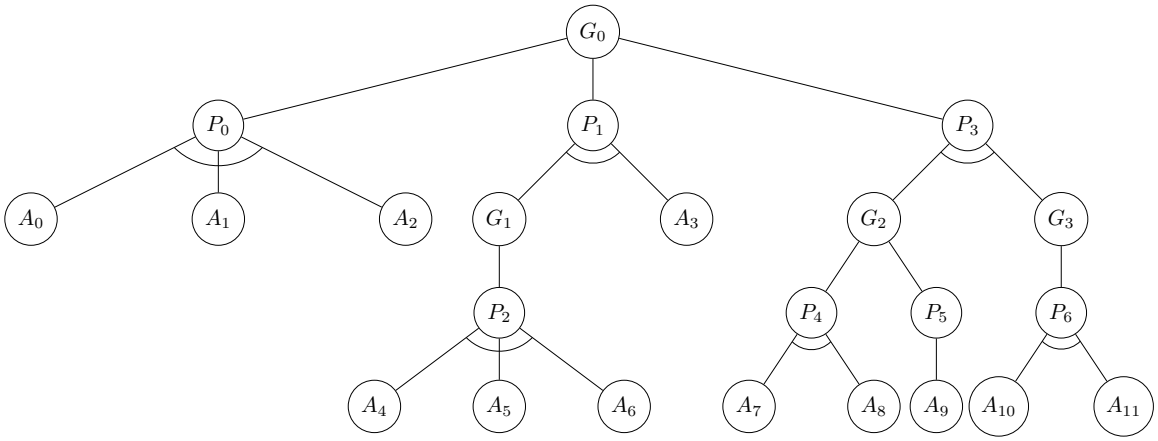

Figure 2.2: Example diagram of goal-plan tree structure based on specification from [6].

The next important part that is formally defined in the intention progression competition [6] is the intention progression within a scope of GPTs. Here, the set of intentions is represented as a set of GPTs $T = t_1, ..., t_n$ and a set of pointers to the current step for each GPT $S = s_1, ..., s_n$. In order to define the progression of steps for an intention, first $next(s_i)$ is defined as a step that should be after the current step s_i for GPT t_i, meaning either action or sub-goal. If current step is a sub-goal, as a next step a plan needs to be chosen, and the first step from this plan is selected. The next step can be selected only if the precondition for action holds and if a plan can be chosen for a sub-goal. The progression of current steps for GPT is then defined as $prog(S, s_i) = (S \setminus s_i) \cup next(s_i)$. Intention progression then means going through a path in GPT to achieve a top level goal. Where intention progression is defined as a policy that chooses which current step should be progressed based on a specified utility function $U^{\Pi'}(T, S$, which could be defined differently depending on what outcome is more beneficial. Formally defined as:

$$\Pi(T, S) = s_i \text{ and } \nexists \Pi' \text{ s.t. } U^{\Pi'}(T, S) > U^{\Pi}(T, S).$$

2.4 Monte Carlo Tree Search for intention progression

The Monte Carlo tree search (MCTS) method and its modification are one of the main methods that are being explored for intention progression in the GPT scope. Before being applied for intention progression, MCTS was used to solve one-player games. Here, this variant is called the Single Player Monte Carlo Tree Search (SP-MCTS) [10] and is used as the foundation for the SP-MCTS application for the progression of the intention. Before we look at SP-MCTS variant for intention progression, let us first look how is SP-MCTS defined for one player games and where it differs compared to MCTS. During MCTS, a tree is built in a way where each node represents a state of a game and a score value. For leaf nodes, a Monte Carlo simulation is started to calculate the value of a node. MCTS is then split into four stages: selection, play-out, expansion and backpropagation, where these stages a repeated until we reach resource limits, in this case a time. The main difference between MCTS and SP-MCTS is in the selection phase, where a node in each level starting

from a root needs to be selected based on some strategy. For SP-MCTS a modified version of Upper Confidence Trees (UCT) is used.

$$v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} + \sqrt{\frac{\sum r^2 - n_i \times v_i^2 + D}{n_i}}$$

Where the average value of the game v_i and the number of visits to the parent n_p or child n_i node are part of an original UCT. The third part is added to help explore promising nodes with a high variance of value, which considers the results achieved at the child node r^2 and the expected results based on the average value $n_i \times v_i^2$. Lastly, two constants C and D are defined. The main differences between one-player games and two-player games is the range of outcome values and the need to optimize based on uncertainty of the turn of other players. For the first difference, to solve the range of values, a solution of setting constants C and D is used. Where different possible values for the constants were explored in experiments [10]. The values of $0,5$ for constant C and 10000 for constant D are the best to balance exploitation and exploration. For the second difference, we can include not only the average value, but also the best value.

In the play-out phase, a simulation starts that randomly selects next moves from an MCTS node position until a game ends, after which a result is produced that represents how successful was the play out. This can be represented as a single score value. For the expansion phase, a strategy is used that creates a new node of position that is not yet present in a tree. In the last phase backpropagation, the result of the play-out phase is propagated back to the root node. After the SP-MCTS search is finished a next move is selected based on the value of child nodes, different strategies are possible like best average score or top value. The modification of the selection phase in SP-MCTS is the most important part for intention progression in GPTs. Other phases like expansion and play-out need to be modified for use in GPTs.

There are two basic applications of SP-MCTS for intention scheduling with GPTs, for a single agent. The first is a version that uses interleaving at the plan level [19]. The second is an expanded implementation that uses interleaving at the action level [18]. Both of these methods modify the deliberation cycle of the agent described in Section 2.1 by merging steps, where during the intention scheduling, applicable plans are chosen for goals and the intention that should be executed is chosen.

Implementation that uses interleaving on the plan levels modifies GPTs in a way where actions are abstracted away [19]. This means that pre-conditions and post-conditions of action are moved to the plan they are part of. Due to this, conflicts between individual actions in plans are resolved, but conflicts that can arise between plans still need to be handled. The input of the method are GPTs T with pointers to the current step S as described in Section 2.3 and the current environment represented as a set of conditions E. SP-MCTS nodes are modified to contain current information about steps S, environment E, in-conditions that are active, number of times this node was visited, highest value that was simulation able to achieve, and lastly sum of all the simulations that were performed for this node. Next, the phases of MCTS are similar as above, selection, expansion, simulation, and back-propagation. Here, the selection phase is basically the same, meaning that nodes are chosen based on modified UCT value until the leaf node is reached. The UCT calculation can also be further modified by removing the third term if the program does not yield high enough rewards that would require capturing the variance of each node. During expansion phase, a new nodes are created based on next steps that are possible from selected node,

because actions are abstracted away a new steps is going to be a plan in any intention. That can be selected only if pre-conditions and in-conditions hold. After that, one of the created nodes is randomly selected as the current new node. During the simulation phase, a simulation is repeated defined number of times. Where for each simulation, a random step that can be performed based on pre-condition and in-condition is selected until either all top-level goals are achieved or no next step can be selected. The simulation that achieved the highest number of top-level goals is selected as a result. Lastly, for back-propagation phase, the results of the simulation phase are propagated to the nodes on a path that leads to root. Here, Algorithm 2 shows an example pseudo-code for one cycle of selecting the next step.

Algorithm 2 Return a next plan that should be executed

Require: Goal-plan trees T with theirs current position S, current state of environment E, number of iterations α, number of simulations β

```
Initialize root node with T, S and E
while Iterations budget α is not exhausted do
    Select leaf node n_e based on UCT
    Expand selected node n_e
    Select random child node n_s for selected node n_e
    while Simulation budget β is not exhausted do
        Pseudo random simulation starting at node n_s
        Add result of simulation to list of all simulations
    end while
    Select simulation with the highest number of top level goals achieved
    The number of top level goals achieved represents the value of selected simulation
    Back-propagate the value of simulation from n_s to the root node, while updating number of visits for nodes
end while
Select a child node n_b of n_e with highest value
Return the next step n_b
```

The second implementation is similar to the above implementation but uses interleaving at the action level [18]. This variant of SP-MCTS for intention scheduling is more important because it stands as a default implementation that is being extended in new methods. The main difference is that we do not abstract away the actions. Because of this, conflicts can occur between actions. In order to minimize the number of conflicts, a new mechanism of interleaving that maximizes fairness is established. That is, we want the scheduling of intentions to be interleaving actions in a way where the variance of time that it takes to achieve each intention is minimized.

Based on this, we need to know a cycle in which the agent adopts a goal of intention d_a and a cycle in which the intention is achieved d_f. These give us r_i, which represents the response time or the number of cycles to achieve an intention i. Next, a total number of intentions n are being pursued. Lastly, a $\overline{r}$ represents the average response time for all intentions. The fairness is then defined using the variance as $1/v$ and the variance is defined as

$$v = \frac{\sum_{i=1}^{n}(r_i - \overline{r})^2}{n} \text{ s.t. } r_i = d_f - d_a$$

The main goal of intention scheduling is to maximize the number of top-level goals first and the fairness of interleaving second. So, if two next steps have the same number of top-level goals achieved, the one that has a better fairness of interleaving is selected.

The algorithm itself is very similar to the one described above. In the MCTS node, additional information is stored about every cycle in which the agent adopts the top-level goal and the cycle in which the intention is achieved. The selection phase uses modified UCT in the same way as above implementation. The next change is for the expansion phase, where new child nodes are created from the next step in each GPT that is achievable based on pre-conditions and in-conditions, meaning either action node or for a goal node, a plan is chosen and its first step is selected.

The simulation phase uses the same approach with the difference that the best simulation is chosen based on the number of top-level goals achieved and the fairness of interleaving. So first the number of top level goals is compared, and if there is a tie, the best simulation is chosen based on the fairness of interleaving so the lower variance in response time between achieving each intention.

Back-propagation again works in the same way. In addition, in the end, when selecting the best child that should be the next step, a degree of dynamism of the environment is taken into account. If it is higher then threshold that is set, number of goals achieved is taken into consideration, then lower coverage of intentions, and lastly a fairness. If the dynamism is lower than the threshold, only the number of goals achieved and fairness is taken into consideration.

When comparing these two implementations of SP-MCTS for intention progression, interleaving at the action level is more beneficial. Mainly because some conflicts cannot be resolved on the plan level and can only be resolved by interleaving between actions of different intentions, allowing us to achieve more top-level goals. Because of this a SP-MCTS that uses interleaving on the action level is the method that is used as a base for more complex methods that are used for intention scheduling with GPTs.

2.5 Online learning Monte Carlo Tree Search

In this section, we look at a promising variation of the MCTS method. The action level interleaving approach of MCTS does not take into account the history of simulations that were already run. Because of this during the simulation phase, a lot of resources may be wasted on searching through a path that historically does not lead to good results. This problem is even more apparent in big GPTs. In order to incorporate the history of simulation, an online learning approach is used, more specifically a Q learning approach [11]. Because of this during the simulation, an ϵ-greedy policy is used in order to balance the exploration and exploitation. With probability $1-\epsilon$, the next action is chosen based on the Q value and with probability ϵ the next action is chosen randomly. The value of ϵ is set to $0,1$. There are two approaches to integrated learning into the simulation, the Q learning approach and the state action tree approach. Both approaches achieve similar results, with the state action tree showing slightly better results, while both variations achieve better results than a basic MCTS method with action level interleaving.

2.5.1 Q learning approach

Q learning approach works as typical Q learning, which means it uses a Q table that stores Q values for each pairing of state and action. Here, the current state is the agents

environment belief base and current intentions state, while action is the action that leads to another state. Q value can then be retrieved based on the current state and action that can be executed. Creating all the possible pairing of state and action after each simulation would lead to a large table, because of that there is limit to when the Q table is updated. Based on the reward that was achieved during the simulation roll out, if the value of simulation that is based on the reward function is higher than the average value of all the previous simulations, it is marked as worth learning, and all the pairs of state and action for every step that was done during the simulation are used for update of the Q table.

The update of the Q table uses the current Q value for a state action pair and adds a value that is represented as $\omega[\Delta + \gamma * MaxQ(s_r, a_r) - Q(s_t, a_t)]$. Where s_t and a_t are the current state action pair that is being updated, s_r represents the state after executing the action a_t and a_r is the action that can be executed for state s_r. The function $MaxQ$ is then the highest possible Q value for the state s_r and any action a_r. The Δ is the reward value that was achieved during the simulation, ω represents the learning rate set at $0,5$ and γ represents the discount factor that determines how much the current reward prioritized compared to the future reward, it is also set to $0,5$. By setting both the rates to $0,5$, the future reward and the current reward are valued in the same way, and the current Q value and new Q value are also weighted in the same way. So, there is no priority of learning more from the simulation reward and prioritization of future or current reward.

Together, during the simulation phase of MCTS, the next action is then selected based on the ϵ-greedy policy, either randomly from available actions or the action with the highest Q value. After each selection, the pair of state and action is saved and later used to update the Q table. Where the Q table is only updated if the simulation value is higher than the average.

2.5.2 State action tree approach

Another approach presented is the state action tree [11]. Instead of creating a Q table for combinations of state and action, a state action tree is created. Here, each node is represented as a current state of agents' beliefs and intentions s_i. Action a_i that was executed to get to the current state and statistics of how many times the action was executed T_i and the total value of simulations that include this action V_i.

During the update of state action tree, the statistics are updated based on the path that was taken during the simulation, and if there is a path that does not exist in a tree it is added. The simulation phase for MCTS then works similarly as in Q learning approach, with the difference in the representation of the state action pair, where the execution of action leads to current state. When the next step is chosen as the best value, the node that represents the current state is found in the tree, and the best child node with the highest value is selected as the next step. The main advantage of using a state action tree is that we do not need to generate all the possible state action pairings. If we want to optimize the possible paths even more, the state action tree can be pruned by cutting benches of certain actions.

Figure 2.3 shows an example of state action tree. The node $< s_0, \delta, 0, 0 >$ represents a root node, where s_0 is a current state of the agent, and the use of δ for action means that no action was executed to get to this state. The values of T_i and V_i are set to zero. Child nodes such as $< s_1, a_1, T_1, V_1 >$ then represent the next state s_1 that was achieved by executing the action a_1 in a state s_0, with their corresponding values for statistics. During the update of state action tree, the visit counter and the total value for each node are updated, and if

the node for state action pair does not exist, it is created and added as a child node, for a parent node that is represented by the state before executing the action.

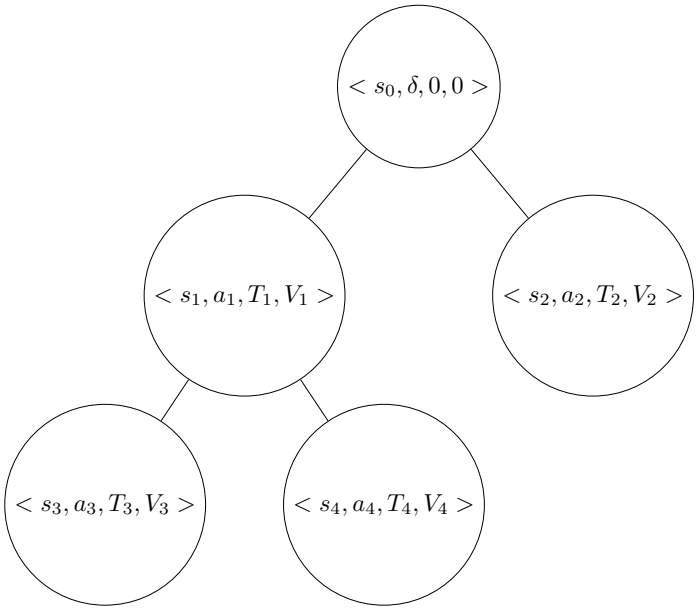

Figure 2.3: Example diagram of state-action tree that is used for storing data about state-action pairs.

2.6 Other Monte Carlo Tree Search scheduling methods

In the above section, we looked at the promising MCTS variation to improve intention scheduling. In this section, we look at different MCTS scheduling methods that expand on the basic implementation in 2.4. The focus is more on extending the potential agent systems in which the MCTS intention scheduling could be used. Where one of the modification options is the intention scheduling under uncertainty [17]. Meaning, instead of agent having all the right information about the be environment, there is some uncertainty about the information from environment. Because of this, a MCTS nodes are modified to be a pair of (q_t, q_f) where q_t is a state where action is executed successfully and on the other q_f is a state where action execution failed. Each state then contains information about the state of the environment after executing the action, the current step of intentions, a subjective probability that this state is reached P_i, the number of times this state was visited and a simulation value. Another modification to the MCTS nodes is that nodes are not connected but instead each state of a node is connected to child nodes.

Each phase of MCTS then needs to be modified to include the possibility of action failing. During the selection phase, both states are taken into account for the calculation of the UCT value. Where we take the average simulation value of a state $\overline{v_i}$ with its probability to be reached P_i and number of visits for each state k_i and a parent state k_p, so the modified UCT looks as:

$$UCT = \overline{\nu}_t \cdot P_t + \overline{\nu}_f \cdot P_f + C \cdot \sqrt{\frac{\ln \kappa_p}{\kappa_t + \kappa_f}}$$

For the expansion phase, every state in the node is expanded, where a new node again contains both states that represent success and failure. For a failed action, the plan itself is also marked as failed, so we add information about action not being done into the belief base and set the current step to parent goal of the node. If the current action is successful, the precondition and the postcondition are added to the belief base because both are now taken as existing. For a state that is the goal, each possible plan is added as a node with the first step of the plan, where probability of a plan is based on in-conditions. In the simulation phase, a simulation is run for both states of a node. Here, the next step is randomly chosen and the success is determined by the probability of reaching this step. The final value of simulation is taken as the number of top level goals achieve. After which the results of simulations for both the success state and the failure state are propagated back to the root node. Thanks to this, it is possible to use the MCTS method for intention scheduling in systems where there is uncertainty about current agents beliefs.

Another modification is the introduction of maintenance goals [16]. So we do not have only top-level goals, but other maintenance goals are added. These have a condition that must be met during the progression of the intention for the top-level goals. Each maintenance goal has a plan on how to recover its condition and a time after which the maintenance goal does not need to be considered anymore. If we can monitor the maintenance goal, we can proactively choose to recover it before the conditions are resolved. If we cannot monitor the maintenance goal, we need to recover it reactively after its conditions are broken. The main changes for the MCTS algorithm occur during the expansion and simulation phases. Here, an option is added on when and how to start the plan to recover a maintenance goal. When its condition is broken for reactive maintenance goals or for proactive maintenance goals, a strategy is added to determine when to start the recovery plan. If it is not possible to recover the maintenance goal, similar to achieving all the top level goals, the agent cannot continue, meaning a terminal state is reached. With this, the agent system can also use maintenance goals alongside the top-level goals for intention progression.

The general implementation of the MCTS scheduler for intention progression does not take into account user preference of achieving one goal over another. In order to incorporate this preference, a feedback-guided modification of MCTS scheduling is introduced [3]. The preference is created by comparing two different options for intention progression, from which the user selects one. The results of the selections are then used to train a network that is used in preference function, which returns the preference of one goal over the other. The preference function is then used in the selection phase of MCTS to help calculate the average preference of each selection, which is used instead of the average number of top level goals achieved. The process of training the preference model can then be repeated in each scheduling cycle until the user is satisfied with the results of intention scheduling. Now the selection of which intention should be pursued is not dependent only on the number of top level goals, but also on the user preference of each goal.

2.7 Monte Carlo Tree Search for multi-agent scheduling

The main goal of this work is the integration of the MCTS method for a single agent, which was explored in previous sections. In this section, we look into how MCTS methods are modified for intention scheduling in multi-agent systems. The main difference between MCTS methods in multi agents systems is how much information and what information is known about other agents.

The first method that uses MCTS for intention progression in a multi-agent system is done by having the other agents actual GPT [4]. For this implementation, we do not consider how to create these GPTs. The algorithm is built on the basis of the SP-MCTS implementation for action level interleaving that is described in 2.4. Where the main difference is in the selection phase, here we need to consider other agents. Our main goal is to maximize the number of top-level goals that we achieve, and the same holds true for other agents. So we need to add information to the selection phase that takes into account if we are allied, natural, or adversarial with other agents.

First, we need to have information about the GPTs of other agents as $\mathcal{T}_i = \{t_i^1, t_i^2, \ldots, t_i^{N_i}\}$ where $\mathcal{T}_i$ are the GPTs of agent i and t_i^k is a GPT for agent i that achieves the k-th goal and N_i corresponds to how many top-level goals agent i has. In this way, we are able to know what goals other agents achieved from GPTs that are visible to the agent.

Next, we need to have information about how other agents' goals affect us if it is positive, negative, or neutral for agent. For this purpose, a payoff matrix is created as:

$$\mathbf{P} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & \cdots & p_{1n} \\ p_{21} & p_{22} & p_{23} & \cdots & p_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & p_{n3} & \cdots & p_{nn} \end{bmatrix}$$

So for matrix P a $\mathcal{P}_{ij} = \langle p_{ij}^1, p_{ij}^2, \ldots, p_{ij}^{N_j} \rangle$ represents the payoff that agent i gets if the goals of agent j are achieved.

These definitions are now used to define a payoff based on the goals that are achieved. First, we take a binary vector $C_j = (c_j^1, c_j^2, \ldots, c_j^{N_j})$ that represents which goals did agent j achieve. Where c_j^k is 1 if a corresponding goal t_j^k from $\mathcal{T}_j$ is achieved, otherwise the value is 0. Then the payoff for an agent i is defined as:

$$\text{payoff}_i = \sum_{j=1}^{n} P_{ij} \cdot C_j$$

The payoff is then used in the selection phase, where the node is selected based on the value of UCB1 [4] and not the UCT that was used in MCTS for a single agent. This helps us to balance exploitation and exploration for the selected path. Where for exploitation, we use the average payoff value $\frac{\text{sum(payoffs)}}{n}$ and for exploration we use $c\sqrt{\frac{2 \ln N}{n}}$ where n represents the number of times the selected action has been selected and N represents the number of simulations that were performed. Lastly c is the exploration constant that was set to $\sqrt{2}$, this choice is based on the UCB1 analysis [2], where the second term of the UCB1 value is based on the Hoeffding inequality $\sqrt{\frac{4 \ln N}{2n}}$. This ensures that with high probability the true value of each action is not overestimated. The modified version of the second term is $\sqrt{2}\sqrt{\frac{\ln N}{n}}$. Other phases, execution, simulation, and back-propagation of MCTS stay the same as in MCTS that is used for a single agent.

The biggest problem with this implementation of intention progression in multi-agent systems is that the agent needs to know other agents complete GPTs. Because of that, other MCTS methods for multi agent systems try to mitigate the need of complete GPTs in order to schedule intention.

One of the options explored is the use of partially ordered GPT (pGPT) [5]. In the basic MCTS method, we have used an MCTS intention scheduling with GPT. But now

we replace the totally ordered GPTs of other agents with a pGPT. This means that we do not need to know the exact plan of other agents because we assume the program of other agents. The difference between pGPT and GPT is that pGPT drops the strict execution order of GPT, instead it uses node dependencies. Here, the dependencies are pre-conditions and post-condition. Meaning, if action has a pre-condition that is enabled by other actions post-condition, there is a dependency between these and it would make sense to execute actions after each other in this order. Importantly, this order is only partial, so one action can be enabled by either other actions or goals. Figure 2.4 shows an example of a pGPT based on the GPT from Section 2.3, where the arrows between the child nodes indicate the dependencies between each child node. For example, the action A_0 must be executed before A_1, and similarly G_3 must be achieved before G_2.

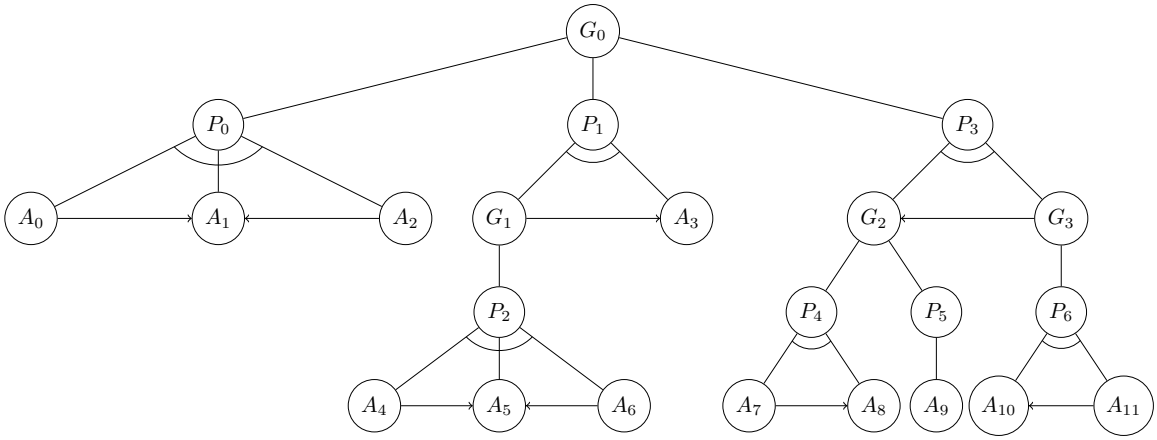

Figure 2.4: Example diagram of partially ordered goal-plan tree structure based on specification from [5] and GPT from section 2.3.

The algorithm of MCTS with pGPT is modified in the execution and simulation phases because now all the possible steps need to be calculated. So for pGPT we create all GPTs that are possible based on partial ordering, in order to create GPTs with total ordering. During the expansion phase, we need to generate all the possible next steps based on created GPTs. During the simulation phase, similarly all the possible next steps need to be considered when randomly choosing the next step. In order to calculate this set, a method is proposed where we store information for a node about what other nodes need to be executed before. After that, when a node is executed, we look at other node dependencies and update them. If all dependencies are met, the node is added to the possible next steps. The next step is then chosen if all the pre-conditions are met. So instead of agent needing to create a completely ordered GPTs for all the other agents, pGPT can be used that can be created based on the expected other agent program. But in order to create the pGPT, the other agents program, which means actions, plans, and goals need to be known or at least adequately simulated.

Chapter 3

Intention scheduler in languages of first-order logic

Advanced intention schedulers that incorporate MCTS are currently defined and studied for BDI systems that do not use languages of first-order logic. Our goal is to modify these methods in a way where they could be used with systems that are programmed in languages such as AgentSpeak(L). The following sections describe a way to modify the GPT of such systems so that it can be used efficiently in MCTS schedulers and how the method will interact with this modified structure.

3.1 AgentSpeak(L)

AgentSpeak(L) is an agent oriented programming language used to develop BDI agents [7]. This syntax is based on first-order logic language while incorporating events and actions. It allows us to specify agents environment beliefs, desires that are states that agent wants to create otherwise goals, and intentions with plans that allow us to achieve specific desires or goals. Beliefs about the environment are represented as facts such as `position(X)`. There are two types of goals, the first is an achievement goal `!goal(X)` and the second is a test goal `?goal(X)`. Where achievement goal wants to bring about a state where goals are achieved, while test goal wants to test if given goal is achieved. Where combinations of goals and beliefs with addition `+` or deletion `-` trigger events, such as `+!goal(X)`. In order to change the environment, we use actions that change the environment from one state to another, for example changing some object `change(X,Y)`. Plans are then defined in two parts, where an example of such a plan can be written as:

```
+do(X) : condition(X, Y)
       <- ?goal(X, Z);
          change(Y, Z).
```

Where head composes of a triggering event and beliefs that should hold during the execution of a plan, and body of a plan that is specified as actions that need to be executed and goals that need to be achieved in order to complete this plan. The interpreter most used for AgentSpeak(L) is a Jason[1]. Here, Jason uses round-robin as an intention scheduler.

The deliberation cycle is represented by three functions that represent each phase. These make up an agent, together with events, belief base, plans, intentions, and actions. Based

[1] https://github.com/jason-lang/jason

on the phase of the deliberation cycle, the first function selects which event should be processed. Next, a second function selects an applicable plan, based on the selected event and belief base. The last function selects which intention should be pursued by executing its plan.

3.2 Choosing Monte Carlo tree search methods

We have looked at multiple different methods that incorporate MCTS for intention scheduling. Our main goal is to modify some of these methods to be able to use them in BDI systems built on first-order logic while potentially achieving better results than currently used schedulers. Because of that, it would be beneficial to first modify a base implementation that incorporates MCTS for intention progression that we looked into in Section 2.4 that will be used as our baseline for intention scheduling using MCTS. After which we build upon this modification to incorporate the online learning variant of the MCTS method that we examined in Section 2.5 to potentially achieve better performance compared to baseline. The main focus of other advanced methods is mostly the incorporation of different types of system or improved intention selection. With online learning being an example of potential modification that could help with achieving better results in default system than a baseline implementation. Mostly because it could potentially allow us to cut the space that needs to be searched during the simulation phase of MCTS.

3.3 Goal-plan tree in languages of first-order logic

In order to use methods like MCTS for intention scheduling in BDI systems that are based on first-order logic, we need to be able to use a building structure that is GPT. The default use is by employing the early binding strategy. This leads to potentially generating a huge number of nodes, because every plan node is initiated for every possible substitution. To defer the initiation of every node based on the substitution, a late variable binding strategy can be used [12, 13, 20]. This defers the creation of every possible node by instead maintaining the context structure that holds the set of valid substitutions.

First, a context structure needs to be created for the selected plan. This is done by operation of broad unification ρU, which maps the predicate p and the predicate p', that is, from a set of predicates PS to a set of all possible most general unifiers (mgu) as $\rho U(p, PS) = \{mgu(p, p') : p' \in PS\}$. The mgu is a substitution that makes two terms syntactically identical and most general in the sense that for any other unifier θ, there exists a substitution δ such that $\theta = \delta \circ \sigma$. The result of broad unification is called a possible unifier set which is used as the context structure, where the plans and events that have a context structure are called weak plan instance (WPI) and weak event instance (WEI). The WPI contains the triggering event, the plan it self, and the context structure, the WEI contains the event, the identifier for the intention that started the event, and the context structure. The intention of the agent is then structured as one WEI with the stack of WPI. So during the execution of intention the WPI that is at the top is taken as first.

During the execution of a plan, the context structure is modified based on the step that is being executed. This is done by using the restriction operation $\sqcap$, which returns one set of mgu that contains the substitutions that satisfy both input sets of mgu. So when updating the context with a test goal $g(t)$ a new context is created using the restriction as $ctx1 = ctx \sqcap \rho U(g(t), BB)$, where the restriction operation is applied at the current context ctx and

the mgu set created by the broad unification of the test goal and the belief base BB. The restriction itself then uses a merging operation $\ltimes$ which takes in two substitutions and maps them into one substitution if $\forall[t_1/x_1] \in \sigma_1, \forall[t_2/x_2] \in \sigma_2(x_1 = x_2 \rightarrow t_1 = t_2)$ holds, the substitutions are unified as $\sigma_1 \cup \sigma_2$ otherwise no new substitution is created. The restriction then takes all the substitutions from both sets of mgu of the current context ctx and the test goal context $ctx2$ and applies the merging operation on them like $\bigcup_{\substack{\sigma_1 \in ctx \\ \sigma_2 \in ctx2}} \sigma_1 \ltimes \sigma_2$.

If the step of a plan is action, the context is modified by using restriction, but instead of using a broad unification for the action and belief base, the function decision making (Dec) is used. The Dec function takes in the set of mgu and a predicate p and returns the ground substitution. That is, a set of substitutions where all the free variables are bound to the specific atom from the action.

The late binding strategy then allows us to have one node for a plan node. Figure 3.1 shows an example how a context for each node of a GPT would be updated by using a late variable binding strategy. Where the node of the plan P_0 has a context of possible substitution that is created by using the broad unification. After step A_0, which is the test goal is executed, the context is modified by using the restriction operator with the broad unification of the test goal. Step A_1 includes an action, which means that the context is modified by using the restriction with the ground substitution for the atom of the action. So, the context of action A_1 contains only substitutions that are grounded to the action atom. By using the late variable binding strategy, only one node for the plan is created, with the context structure that is then modified based on the steps in the plan. Instead of creating multiple plan nodes for every possible substitution.

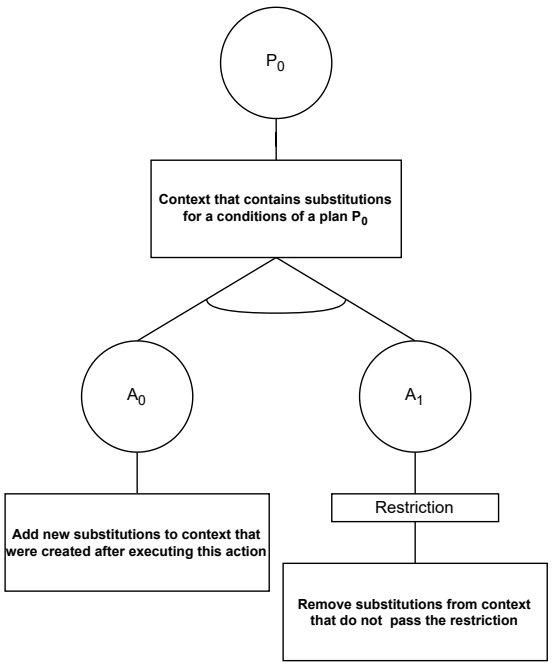

Figure 3.1: Example of a plan node that is create by using late variable binding strategy based on specification from [12].

3.4 Monte Carlo tree search in languages of first-order logic

This section contains requirements and specification on how the MCTS intention scheduler should function. The scheduler is structured as a separate package that is connected by an interface with the BDI agent. This means that BDI agent implemented in a language based on first-order logic holds the current state of environment and agents intention, which are then passed on to the MCTS scheduler through the communication interface. Because of this, scheduler will first need to parse the current state of agent, that is, represented with predicates. Meaning the GPT structure itself is not statically created before the MCTS methods starts, but rather dynamically constructed during the intention scheduling. So we have the agent belief base that is represented as predicates and the current reasoning or action that represents the state in GPT, and it contains the context structure for the use of the late variable binding strategy described in Section 3.3.

When the MCTS node is expanded, the next states of the GPT are created from the current state of environment. These are possible reasonings based on the events or actions that are for the current intentions with their corresponding substitutions. For each possible reasoning and action, a new MCTS node is created, with the context structure.

A random child node is then used in the simulation phase, where each step in the simulation is randomly chosen based on the current context of a node. So we randomly select a next step for which we select a random substitution that is possible for this step and continue until we achieve all the top goals or there is no next step that we are able to execute. Here, the value of the simulation can either be represented by the number of top level goals achieved or by the number of rewards achieved on the path. After that, the best simulation value is propagated to the nodes on the path to the root. When a computational budget is reached, an MCTS node with the highest value is selected. The next step of intention progression is returned based on the state of GPT in selected MCTS node.

This representation of a current state in GPT is then adopted for the online learning variant of the MCTS method 2.6. For the state action tree there is no need for context structure, because it is created dynamically based on actions or plans that were already selected in simulation. The structure that represents current state still needs to be modified as in processing the predicates that represent agent belief base and environment facts.

In summary, a BDI agent needs to call an intention scheduler with their current belief base and the state of environment. Here scheduler parses these predicates, and the current state of GPT is represented as a reasoning or an action with the context structure for the use of late variable binding strategy. During the expansion, the next states are created based on the current state of environment. Next, during the simulation phase, the possible actions and plans for each step are created based on the current state from which a random one is selected. After the scheduler finishes the computation, the next step is returned to the BDI agent that will use it for deliberation or to execute an intention.

3.5 Integration with FRAg system

This section describes the design of integration between the MCTS intention scheduler and the FRAg system. Here, first a FRAg system is described and integration point is identified. Next we describe a possible options of Prolog interfaces for communication with other programming languages. What are their advantages and disadvantages and which would be the best option. Finally, two system architectures are presented for how should

the MCTS method be integrated into the FRAg system. It is also important to note that our main goal is to integrate the MCTS method for a single agent, because of that we do not take into account multiple agents running in FRAg system.

3.5.1 Frag system

The main goal of this work is to integrate the advanced MCTS method for intention scheduling into an FRAg system[2]. Where the Frag system is an interpreter for creating BDI agent systems in AgentSpeak(L) language [13, 12]. The interpreter is implemented in SWI Prolog language. The FRAg system already contains the implementation of late variable binding strategy for expanding reasoning's and actions based on the current state of environment with the description of how late variable binding works in Section 3.3.

Agent program

An agent system specification is split into three parts in the FRAg system. First, is an agent program that is implemented in AgentSpeak(L) dialect that can be interpreted in Prolog. Next, a setting for a system is specified, meaning agent specific settings and environment of the system. Lastly, an environment is implemented in Prolog that specifies all the interactions and functionality of the environment, where agent is located.

One step of agent

In FRAg system one step of agent is split into four phases sensing, update models, deliberation, and acting. Figure 3.2 shows how each phase goes after each other. During the sensing phase, the agent processes changes in the environment, which is either adding or deleting some of the agents current beliefs. Next, during the update model phase for a reasoning method that is set, its model is updated based on the current agent belief. This is a phase during which the FRAg system will connect to the MCTS reasoning method, so it can be updated to reflect the current state. Meaning, MCTS is run with current agent belief base and environment and the resulting best path is stored to be later used in deliberation and acting. During the deliberation phase, all active events are processed. If there are applicable plans for an event, the plan that is going to be used is selected by calling the reasoning method. Where, based on the best path that was created by MCTS a plan is selected. Lastly, during the acting phase, an intention is selected in the same way by calling the reasoning method which selects an intention from possible intentions based on the best path that was created by MCTS during the update model phase. Then the intention plan is executed.

3.5.2 Prolog interfaces for comunicating with other languages

In order to implement the MCTS method in different languages, then a Prolog. We need to look into possible interfaces that allow us to communicate in each direction. That is because we need a way to call an MCTS method from the Prolog agent in the update model phase. Next, during the MCTS phase of expansion we need to be able to call a Prolog code for deliberation and during the simulation phase, call the roll out of MCTS node environment and selected step. Prolog has multiple built-in interfaces for commination between Prolog and other programming languages.

[2] https://github.com/VUT-FIT-INTSYS/FRAg

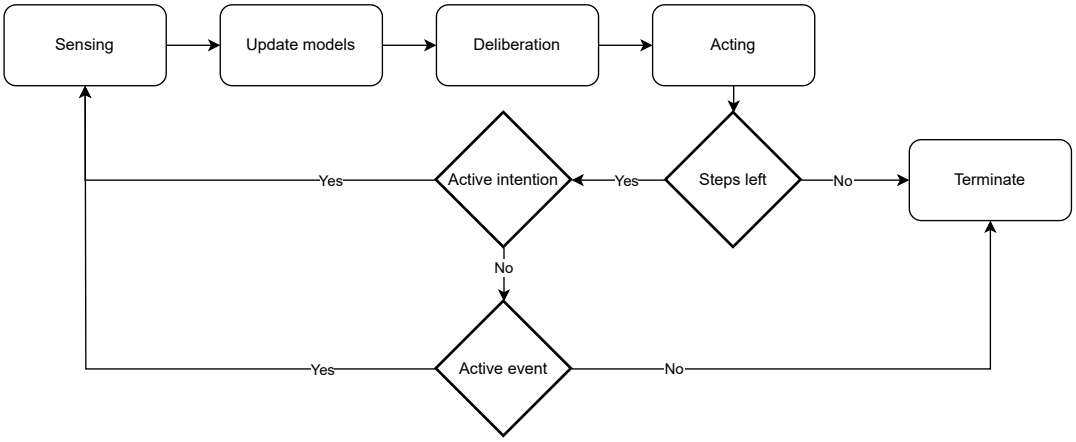

Figure 3.2: Agent deliberation cycle in FRAg system for specified number of steps.

Prolog foreign langauge interface

One of the interfaces provided in the SWI Prolog is for the C language[3]. This allows us to call C functions from Prolog by registering them as foreign predicate, see the Listing 3.1. Then in the C program we are able to call Prolog predicates, see Listing 3.1. This interface is designed for low level integration with C programs, by allowing us to manage type conversion between Prolog and C and manage memory allocation for Prolog queries.

```
//Calling prolog query from C program
qid_t qid = PL_open_query(NULL, PL_Q_PASS_EXCEPTION, p, a0);

//Reqister C function as predicate
install_t install() {
    PL_register_foreign("add_integers", 3, pl_add, 0);
}
```

Listing 3.1: Calling Prolog query from C and reqistering C function as a Prolog predicate.

C++ interface

The C++ language interface is provided as a layer around the C interface[4]. Where the interface allows us to register functions as an foreign predicates in Prolog see Listing 3.2. Also we are able to call Prolog queries from the C++ program see the Listing 3.2. Compared to the C interface, C++ requires extra resources but allows for easier integration with Prolog, by providing such things as a mapping of Prolog exceptions to C++ exceptions, wrapping of Prolog types to classes, and also easier memory management where objects manage their resources automatically.

```
//Calling prolog query from C++ program
PlQuery q(pred, PlTermv(h0), PL_Q_NORMAL);

//Reqister C++ function as predicate
```

[3] https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/clib.html%27)
[4] https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/pl2cpp.html%27)

```
PlRegister x_hello_1(NULL, "hello", 1, pl_hello);
```

Listing 3.2: Calling Prolog query from C++ and reqistering C++ function as a Prolog predicate.

Java Prolog interface

Java Prolog interface provides a bidirectional interface between Java and the Prolog language[5]. This is done by using Java native interface to communicate with the Prolog foreign language interface in C described in Section 3.5.2. In Prolog this interface allows us to create a Java object, call a Java code or fetch and set a Java field values see Listing 3.3, while in Java we are able to call a query in Prolog.

```
//Calling prolog query from java program
Query q1 = new Query(
                "consult",
                new Term[] {new Atom("test.pl")}
          );

//Call java function from prolog
jpl_call(F, setVisible, [@(true)], @(void))
```

Listing 3.3: Calling Prolog query from Java and calling Java function from Prolog code.

Janus

Janus is a bidirectional interface between Python and Prolog[6]. Where Janus module is included in the SWI-Prolog. In Prolog we are able to call Python functions and in Python code we are able to call Prolog query, see Listing 3.4.

```
//Calling prolog query from python program
janus.query_once("Y is X+1", {"X":1})

//Call python function from prolog
py_func('script', get_result(Input), Return),
```

Listing 3.4: Calling Prolog query from Python and calling Python function from Prolog code.

When calling a Python function, the first attribute represents the name of the Python package, and the second attribute represents the function that should be called in the Python package. We can pass arguments to the function and we can retrieve a return object. It is important to note that Janus handles the conversion of basic data types and Python interpreter is initialized once for the Prolog process. That is, after lunching the Prolog program, the current Python interpreter that is initialized in the path is used. If we want to containerized Python interpreter that we want to use, we need to use tools such as `conda` or `virtualenvs` to initialize Python dependencies.

[5] https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/jpl.html%27)
[6] https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/janus.html%27)

MQI

PrologMQI is a message queue interface library that allows us to launch a SWI Prolog instance and use it as a library inside many programming languages such as Python or Go[7]. Currently a package for the Python library is provided by SWI Prolog. Janus allows us to call a Prolog code from a Python program too, but the problem is that the instance of Prolog is the same as the one that started the Python program. In order to separate the instance of agent for the deliberation and roll out of MCTS in Python, we need to be able to launch a new Prolog instance. Prologmqi allows us to then create a new Prolog instance and a new Prolog thread, which is used to query in newly created Prolog instance, see Listing 3.5. Where PrologMQI does not use shared memory like Janus for Python objects, but communicates with started Prolog instance by calling the top level queries similar to how they would be called in terminal and receiving the response as a json object. The json object can then be converted to Prolog format in Python string. Because of this we are able to dynamically start a new Prolog instance and initiate a FRAg system inside it in order to use agent deliberation for expansion and roll out for simulation phase.

```
# Initiate new prolog instance
with PrologMQI() as mqi:
    # Create a new prolog thread
    with mqi.create_thread() as prolog_thread:
        # Call prolog query
        result = prolog_thread.query("atom(a)")
```

Listing 3.5: Calling Prolog query from Python code by creating a new instance of Prolog.

Process creation

We are also able to call other programs from Prolog by creating a new process[8]. So instead of an interface that allows us to directly call a different language code from Prolog, we can instead run a program as a process from Prolog. For example, we can call a Python program, like we would inside a terminal and retrieve the output of the program as a result, see Listing 3.6. This way we are able to communicate with different programming languages in Prolog and also separate the program that is being called from Prolog itself. The main disadvantage is in the processing of the result, because instead of directly calling a function that returns an object for interfaces such as Janus 3.5.2, we instead need to process the standard output of the program.

```
# Call a python program
process_create(path(python),
                  ['test.py', Input],
                  [stdout(pipe(Out)), process(PID)]).
```

Listing 3.6: Creating a new process in Prolog that calls a Python program.

[7] https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/mqi.html%27)
[8] https://www.swi-prolog.org/pldoc/man?predicate=process_create%2f3

Selection of appropriate interface

After a review of available interfaces for integration with Prolog. I have chosen to use a Python interface Janus to call the Python function and PrologMQI to integrate Prolog into the Python implementation of MCTS.

While interfaces in C and C++ allow us to integrate with minimal overhead in Prolog, the implementation of MCTS is going to be called only once for each step, so the actual overhead with using the Janus interface is not that important. The Java Prolog interface uses the Java native interface to integrate with the C interface, so the overhead is similar to Python integration. Secondly, for implementation of the actual MCTS method, using languages such as `C` and `C++` would allow for potentially better performance thanks to the nature of these languages, but it could be beneficial to completely separate the inner integration with Prolog, which can be done with PrologMQI that is currently officially available as a Python package. Another point is the ease of use, where `C` interface is low level with the need to manage the memory of Prolog calls, while the Python integration has a simplified API.

When it comes to choosing between using the Janus or creating a new process that calls Python program, the main reason why to use Janus is the easier processing of the result. So we are not depended on what the Python program outputs on the standard output, instead we directly get the result that the called Python function returns. Lastly, Python has an extensive ecosystem of libraries for intelligent systems and machine learning tasks, while in this project we do not use any of these libraries, it could be beneficial for future integration of different intention scheduling methods.

3.6 System architecture

In order to integrate an MCTS based intention scheduling into an FRAg system, we propose two alternative architectures. In this section, we first describe the design requirements that are common for both architectures. Next, each architecture is described, and their potential advantages and disadvantages are explored. Our goal is to implement both of these architectures and evaluate their performance for intention scheduling.

3.6.1 Requirements

Before we introduce the architecture design for integrating MCTS into the FRAg system. It is important to establish some requirements that are taken into account.

- The run of the MCTS method needs to happen during the update model phase of agent deliberation cycle that is described in Section 3.5.1.
- It needs to be possible to separate the Prolog instance that is used in the MCTS method itself. This is mostly important because we do not want to possibly interfere with the running FRAg agent.
- It must be possible to dynamically configure the MCTS method. This mostly means that we should be able to pass all the MCTS parameters and agent belief base with environment facts directly into the MCTS method, so the MCTS scheduling is not dependent on the type of agent program that is run.
- If the MCTS run stops because of an error, the whole FRAg system should not fail.

3.6.2 Integrated Monte Carlo Tree Search architecture

Figure 3.3 shows the first architecture in which during the update model step of the agent loop a Python script is called using the Janus module. The Python script then starts the MCTS method that finds a next intention that should be scheduled. The result is then returned to the agent in the update model, where it is processed. For calling the Prolog methods in MCTS we use MQI library to initiate a new Prolog instance. If an error occurs during the run of MCTS it is cough and the Python function returns an empty result.

The main advantage of this architecture is the encapsulation of the MCTS method and no overhead of other communication. The disadvantage is that we lose a better controller of the Python service because the way Janus initiates Python instance can not be easily changed.

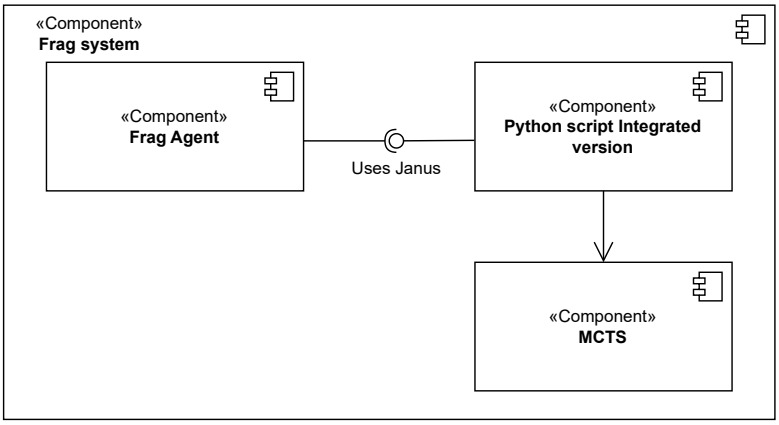

Figure 3.3: Component diagram of system architecture with MCTS integrated in Python script called by Janus.

3.6.3 External Monte Carlo Tree Search architecture

Figure 3.4 shows a second architecture where in agent update model a simple Python script is called, that then creates a request and calls a currently running Python service. Where the ruining Python service then starts the MCTS method. Thanks to this, the Python script that is initiated by Janus only creates a request and calls the other Python service. The Python service that actually runs the MCTS method runs independently. This gives us more control over the actual service that finds the next intention. If an error occurs during the communication between the Python function that is called by Janus and the Python service, it is cough and the Python function returns an empty result. The same principal applies if there is an error during the run of MCTS in the Python service, it is a cough, and the service returns an empty response.

The downside is that we add a new communication layer between Python script that is run by Janus and the other running service. The main advantage is that we are able to work more easily with parallel computation, because the service itself does not run in Prolog process. This is mainly because Janus was designed more to run simple Python functions and not for complex computation.

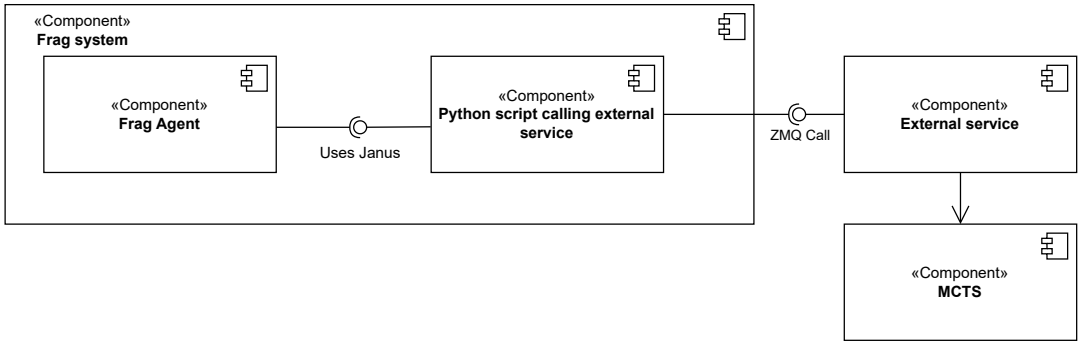

Figure 3.4: Component diagram of system architecture with MCTS in external Python service that is called by internal Python scripted started by Janus.

Figure 3.5 shows a sequence diagram of the flow in order to start the MCTS method and retrieve the result for the external architecture. First, the FRAg system is initiated, then the agent is initiated. During the update models step of the agent loop, all the parameters for MCTS are retrieved, and agent belief base with the environment facts are found. In the update model step, a Python function in a specific file is called, with all the required inputs. The function creates a request with the input attributes and calls a locally running service. Python service then receives the request, parses the body of the request, and initiates specific MCTS method, runs it, and formats the result as a response, and sends it back. The Python function then receives the response, formats it, and returns it as a result. In the update model phase, the result is retrieved and stored for later use during the deliberation or acting phase.

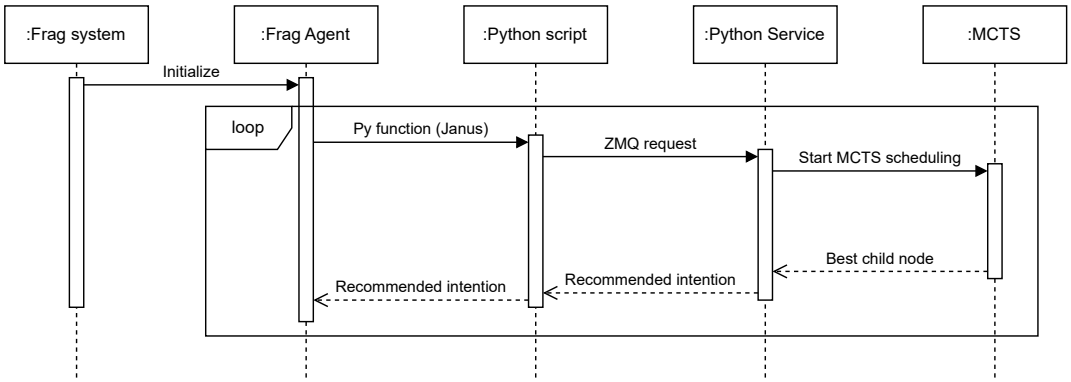

Figure 3.5: Sequence diagram of communication in external architecture between FRAg agent and external Python service.

Chapter 4

Implementation of Monte Carlo Tree Search

In this chapter, we outline the main parts of implementing MCTS method for intention scheduling and its integration into a FRAg system. Both architectures proposed in Section 3.5 are implemented. First, a basic MCTS method is described, and then the advanced MCTS method that uses online learning is described. Lastly, the usage of these methods in the FRAg system is shown.

4.1 Used technologies

Based on chapter 3.5, where possible options for communication with Prolog were discussed. The implementation uses these tools:

- Python version `3.11`.
- `SWI Prolog` for using the FRAg system.
- `Zmq` for creating the external MCTS service and for creating a request for this service.
- `PrologMQI` for creating a new Prolog instance and calling the Prolog query from Python.
- `Janus` for calling the Python script from the FRAg system.
- `Multiprocessing` for parallel calculation.
- `Loguru` for logging information.
- `Anaconda` for containerization of Python interpreter.
- Other libraries that are used are build in Python such as `time`, `json`, `random` and `math`.

4.2 Initiation of new reasoning method in FRAg

In order to initiate a new reasoning method for the FRAg system, we need to perform the following steps:

1. First, a file that encapsulates the reasoning method is created. In the file, we declare a term that specifies the name of this reasoning method as `reasoning_method(name)`.

2. Next, the new reasoning method needs to be included in the FRAg system. This is done in the `fraginit.mas2fp` file by adding the term `include_reasoning(Filename)` with the filename path where the new reasoning method is implemented.

3. Lastly the required predicates need to be implemented in the file of the reasoning method, which are:

   - `init_reasoning` that initializes all the parameters for the reasoning method.
   - `get_plan` chooses one plan from all possible plans.
   - `get_substitution` that selects one of possible substitutions and reduces it to variables.
   - `get_intention` chooses one intention from possible intentions.
   - `update_model` that creates an additional clauses that are needed for the other predicates.

   Where all of the methods except `update_model` are taken from the Prolog implementation of MCTS that is included in the FRAg system, which can be found in a file called `FRAgPLReasoningMCTS.pl`.

4.3 Communication between Prolog and Python

For communication between the Prolog implementation of reasoning method in FRAg system and the MCTS intention scheduling, two versions based on the architecture designs in Section 3.5 are implemented. It is important to note that the architectures differ only in the communication layer while the MCTS module is the same.

4.3.1 Calling Python function from Prolog

Every new reasoning method uses Janus in order to call a Python script from the Prolog code. More specifically, based on the FRAg agent step loop from 3.5, we call the Python script during the phase of `update_model`. The difference between each MCTS reasoning method is in which Python script is called, and for external architecture, we add a new input parameter that specifies which MCTS method should be used. For variant of the integrated MCTS architecture, see Listing 4.1.

```
py_func(
    'mcts_integrated',
    get_result(
        Silent_Program_String, Facts_String,
        Expansions, Simulations, Simulation_Steps,
        Environment, Absolute_Module_Name),
    Ret)
```

Listing 4.1: Use of Janus module in order to call Python script with input parameters and return object.

This code calls a Python function `get_result` in package `mcts_integrated` and the result is then retrieved in object `Ret`. In order to get values for input attributes we use build in predicates in FRAg system, each attribute is then captured like:

- `Silent_Program_String` where first the program is captured using the `take_snapshot` after which the program is converted to silent program, and lastly the program is converted to string as an input for Python function.
- `Facts_String` represents the environment facts in which the agent is situated, these are captured by using the new predicate in the environment utility, that is called the `all_facts_struct`, found facts are then converted to string.
- `Expansions` represents the α computation budget for the MCTS method, this parameter is specified in agent settings and is stored as a term `mcts_expansions`.
- `Simulations` represents the β computation budget for the MCTS method, this parameter is specified in agent settings and is stored as a term `mcts_number_of_simulations`.
- `Simulation_Steps` represents the number of steps that should be done during the roll out of one simulation, this parameter is specified in agent settings and is stored as a term `mcts_simulation_steps`.
- For external service architecture attribute that specifies which MCTS method is added.
- `Environment` represents the name of the environment in which the agent is situated.
- `Absolute_Module_Name` represents the absolute path to the environment file that is loaded.

The Python function returns a result that is captured as `Res` objects. The result is returned as a string, so first it is converted to term by using the `term_string`. Lastly, the result is asserted as recommended path that is later used in the predicates for selecting next plan and next intention that should be executed.

4.3.2 Integrated architecture

For an integrated architecture, no other separate service for MCTS is started. Because of that all of the communication implementation is in file `mcts_integrated.py` where the MCTS class is initialized with the input attributes and then the MCTS method is started. After the method finishes the result of the MCTS scheduling is formatted in the right Prolog format which is a two lists, one for reasoning prefix and second one for scheduled action like `[[{prefix}],[{act}]]`

4.3.3 External architecture

For external architecture, we need to separate the Python script that reasoning method in Prolog calls and the running service that starts the MCTS method, where these are implemented in files `mcts_service_call.py` and `mcts_service.py`.

The Python script that is called by the Janus module from the Prolog code is in the file `mcts_service_call.py`, where the function creates a ZMQ request with a body that is a

json object of input attributes. After which the result of the request is received as a json object that contains two attributes, one represents the prefix of reasoning and the second represents the intention that should be executed. These are converted to the final format that is the same as for integrated architecture `[[{prefix}],[{act}]]`.

The ZMQ service that initiates the MCTS method and then runs it is implemented in the file `mcts_service.py`. Where a ZMQ service is started in the main method of the script, so it is started automatically when the Python script is run. The service itself is started as a local service on a port `5555`. The service runs until it is stopped externally. When a message is received, the body is processed as a json object. Then based on the input type, a specific MCTS method is initiated and started. The result is then processed, and a response is created as a json object that contains two attributes of reasoning and intention.

4.4 Basic Monte Carlo Tree Search implementation

For a basic MCTS method that is described in Section 2.4, both options of the system architecture are implemented. Where the part of system that is implemented in Prolog and the MCTS method itself are the same for both options. So, the only difference is the communication between the reasoning method in Prolog and the MCTS method implemented in Python. Where the sequential variant of MCTS can be found in file `mcts.py` and parallel variant of MCTS can be found in file `mcts_parallel.py`, where file `base_mcts.py` contains the shared methods for expansion, selection, and back propagation.

4.4.1 Monte Carlo Tree Search node representation

The MCTS node is implemented in the Python file `mctsnode.py`. Each node contains the program and environment terms that represent current agent state, where every node contains the reasoning or intention that was created during the expansion of parent node. Lastly, node contains the number of visits, total value of simulations values from this node, and children nodes. The node class contains the expand method that calls the expansion predicates from FRAg agent on the Prolog thread that is passed into the method. From the expanded actions and reasoning's a new child MCTS nodes are created.

4.4.2 Selection and Expansion phase

The selection and expansion phase are connected by first initialization of the Prolog instance, by using the PrologMQI library. With created Prolog thread, the main FRAg system file `FragPL.pl` is loaded. With an initialized Prolog instance, an agent belief base (program) and environment facts need to be asserted into the Prolog instance, so the MCTS phases work with current state of agent. Listing 4.2 shows an example of what terms an agent program can contain. In order to assert these terms, the program is split into each representative term by counting the number of closures, after which, for every term, a query is created. In order to reduce the amount of queries that is called with the Prolog thread, all the assert term queries are joint together and are executed as a single query.

```
[fact(my_position(a)),
event(1,ach,go,null,[[]],active,[]),
plan(2,ach,get_stone(_),[item(_,_,_)],[act(task_maze,silently_(pick))]),]
```

Listing 4.2: Example of content that can be in agent program.

Next, an agent is created in FRAg system, by loading the environment file and by calling the initiation of agent. Because the environment is initiated to the base state, first all the terms for environment are retracted, and then the environment facts that represent current agent state are asserted in the same way as an agent program. This is then a properly initiated protolog instance that can be used for calling the agent predicates.

Now a selection phase begins, where a root node is the starting point and next node is selected until a leaf node is reached. Based on the Section 2.4 an UCT method is used for selecting the next node, with the third term removed, because the programs that are used in FRAg do not yield high enough rewards, where capturing the variance of each node value would be beneficial, see Listing 4.3. This variation of UCT is UCB1 and the constant c is set to traditional $\sqrt{2}$, the reason for this choice is described in Section 2.7.

```
def uct(
        self,
        parent: node.MctsNode,
        children: node.MctsNode,
        c: float) -> float:
    average_value = children.total_value / (children.visits * 3)

    exploration_value = c * math.sqrt(math.log(parent.visits + 1) /
                                           children.visits)

    return average_value + exploration_value
```

Listing 4.3: Implementation of UCT calculation for selecting the next node that should be expanded.

When a node is selected, before continuing, the action or reasoning of the node needs to be executed in order to update the belief base and environment to represent the new state based on the path that is selected. The execution is done by calling the predicate in the FRAg agent based on the MCTS node deliberation, so for force execution of action, a predicate `force_execution` is called, and for reasoning, a predicate `force_reasoning` is called.

Next, an expansion phase starts by calling the expand function for a selected MCTS node. Where two predicates from FRAg agent are used to expand the current agent state. First, expands the possible actions that can be done in current agent state as `model_expand_actions`, next a possible reasoning is expanded by calling the predicate `model_expand_deliberations`. Both results are combined and for each item a new MCTS node is created as a child node, with the deliberation action, current agent program, and current environment facts. At the end of expansion, a random node is selected as a new node that will be used in the simulation phase.

4.4.3 Simulation and Backpropagation phase

During the simulation phase, a new Prolog instance is initiated. The FRAg system setup is the same as in the chapter above 4.4.2, where one additional predicate needs to be called `set_reasoning` in order to set the method that is used for roll out, where `random_reasoning` is used, which selects next step randomly. The initiation of agent belief base and environment facts is done before each simulation in order to start each simulation from the same state, while the setup of the FRAg system is done only once before all simulations are run.

With the agent state initiated a roll out predicate is called. This predicate is specified in the Prolog file of the reasoning method as described in 4.2. It is a simplified version of the MCTS roll out that can be found in the Prolog implementation `FRAgPLReasoningMCTS.pl`. The roll out is modified to only perform one simulation and the return is direct instead of using the engine yield. The agent state is setup before hand in the Python implementation, but at the end of roll out all the terms that represent agent state are retracted.

After calling the roll out predicate the result is retrieved, which contains the number of rewards achieved in the simulation that represents the value of the simulation. The simulation value is stored in a list that contains all the simulation values. This is repeated based on the β budget that is specified.

Lastly, the backpropagation phase starts where for every MCTS node that was visited during this run, a best simulation value is added and the visit counter is increased. These phases are repeated until the α budget is reached, after which the child node with the highest value from root MCTS node is selected. Where if the first node is a reasoning node and if the node has any child nodes, a next node that represents the action is selected also. Meaning, the result can be a prefix of reasoning node and action node. This can be done because of the one step loop of agent in FRAg system that is described in 3.5.1.

4.4.4 Python interpreter issue

For the final implementation there appears to be one problem. After the Python script was called by the Janus module, when the agent thread was joined in the FRAg system, the Python interpreter that called the script was not properly finished. That is, it was still initiated but now not connected to any Prolog thread. This is mostly because Janus is designed to be mainly used with the main Prolog thread. This does not impact the implementation and results itself, but after the FRAg simulation finishes and halt is called in order to finish the program, it is not terminated because there is still a Python interpreter that cannot be properly discarded because the thread that creates it is already joined.

Based on the official documentation, there is one way to fix this problem and that is by first initiating the Python interpreter in the main thread so the interpreter is bind to it, it can still be later on used by the agent thread. In order to bind the interpreter only when needed, a new setting term is added as `python_interpreter`, that can be specified in the FRAg agent setting. If the term is set, a simple Python code is called with the Janus module `py_call(print(""))` at the start of FRAg program. Python interpreter is then bound to the main Prolog thread, and when the agent program stops and the FRAg system is then halted it terminates properly.

4.4.5 Optimization of computation budget

The MCTS method uses a tree to select the next intention that should be scheduled. Because of this, it is possible to further specify which nodes are selected or created. When scheduling intention, the goal is to create a tree of possible steps that the agent can take and select one step that could potentially be beneficial.

During the whole run of agent it is possible that at some points there is no actual benefit in trying to find the next step, that is, if there is only one possible next step. For example during the first α loop, a root node is selected, expanded, and simulations are run for one expanded child node, but if the root node has only one child node, it will always be selected as a next agent step. If that happens, it is not beneficial to continue with calculation of the tree further, because next step is already known.

So, to the basic MCTS implementation at the start of α loop a check is added if root node is already expanded and if the root node has only one child, the MCTS is stopped and the single child node is returned as the result. This is beneficial if the computational α budget for the MCTS is larger, because the intention scheduling for some steps can be done at the start and only states where agent can perform multiple steps are explored.

4.4.6 Parallel variation of Monte Carlo Tree Search

When the external architecture is used, it is possible to easily launch new Python processes. That is because the service itself is run outside the Janus Python interpreter. Because of this, it is possible to use parallel computation in order to speed up some parts of MCTS method. More specifically during the simulation phase, where each simulation is independent of each other.

We do not want to start a new Prolog instance for each simulation, so instead a new class `PrologWorker` is implemented. This starts a new Prolog instance and initiates the FRAg agent in the same way as in the section above, but it does not start the simulation, instead it waits for a new item in a job queue. Figure 4.1 shows the architecture of how the MCTS implementation and workers communicate with each other using queues for jobs and results. So, at the start of an MCTS method a static number of workers is started with a job queue and result queue, so the Prolog instance is initiated. Then during the simulation phase based on the computation budget β, a new jobs are inserted into a job queue, where job contains the program and environment of MCTS node for which the simulation should be run. The initiated workers then take a new job from the job queue, assert the MCTS node belief base and environment facts, and lastly start the roll out. The result of the roll out is then inserted into a result queue. The results are then received after the job is created in simulation phase and are inserted into the list of all the simulation results.

This way, the number of simulations is effectively split between the number of workers, instead of each simulation being run sequentially after each other. In the next section 5 the difference in speed between the sequential and parallel variant is explored in experiments.

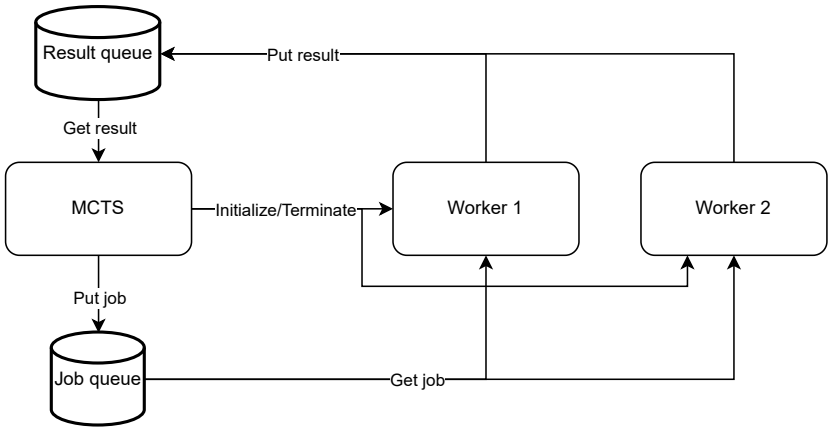

Figure 4.1: Diagram of communication between MCTS and parallel Prolog workers by using the job and result queues.

4.5 Online learning Monte Carlo Tree Search implementation

The advanced version of MCTS uses the modification that includes online learning in order to introduce learning for the simulation phase, where the method itself is described in section 2.5. For communication between the FRAg system and the MCTS method, an external architecture is used as described in 4.3. Because the architecture uses an external running Python service, in the request there is an additional attribute that specifies which MCTS method should be used. Then in the service, a specific MCTS method is initialized and run. The name of each type is then described in Section 4.6. The selection, expansion and backpropagation phases of MCTS are the same as for the basic implementation of MCTS 4.4, where the modification is done in the simulation phase. The implementation can then be found in the file `mcts_online_learning.py`.

For online learning, we use the state action tree variant described in 2.5. The setup of the simulation is the same as for basic MCTS implementation, with the difference of which reasoning method is used for roll out. Meaning, the predicate `set_reasoning` sets the roll out method to `random_reasoning_learning`. This is a new reasoning method that is created in the same way as described in 4.2.

In order to introduce an online learning into roll out, we need to first construct state action tree based on the steps that were taken in simulation. Where state action tree is constructed in Python part of implementation but after a simulation we need to find information about each action or plan that was chosen. So during the roll out in methods that determinate next plan or action, after each selection we assert a new term that describes a step taken and current state as `insert_state_action(State, Action)`. Instead of an actual agent program, we use the hash that describes it, in order to achieve a smaller footprint. At the end of roll out, all of these terms are queried and returned in roll out result.

Before using the terms to update the state action tree, they first need to be converted to a queue where instead of being a pair of current state and action taken. It is converted to state after action execution and action taken to get to that state, in order to be properly used for updating the state action tree. This can be done because the terms are sorted according to their assertion, so they represent a sorted list of steps that were taken during the roll out. The converted queue is then used to update the state action tree that is represented as a tree structure where each node contains the next program hash, action that was taken, visit counter, total value of simulations and children nodes. At the start a root node is created that is represented by currently selected MCTS node and empty action.

When adding new nodes to tree based on queried terms, we check if current state action node has some children, if not new node is added as a child, if node already has a children nodes we check if some of them match in their program hash and action taken. If any children match, the child visit counter is increased, and the child is returned. If no children match, a new node is created child node and returned. With the returned node, the update is repeated with a new queue item, this way each step of roll out is checked.

Now, the constructed state action tree can be used during the decision in each roll out step. Before the roll out, the state action tree is asserted into the Prolog instance where each node is represented as a term `state_action` with the program hash of the parent node and action that was taken. These are asserted in the same way as agent belief base and environment facts, by joining them into one query.

In the new reasoning method `random_reasoning_learning`, for predicates that chose which intention should be executed `get_intention` and `get_plan` that chose which reasoning should be pursued, a state action tree is used to help with decision. The next step for the agent is chosen with the ϵ-greedy policy. First, a random number is generated between 0 and 1 that determinate if the next step should be chosen randomly or based on the state action tree. Where the limit is set as 0.1, so if the random number is below, next step is random, if it is above, next step is chosen based on the state action tree. If the state action tree is supposed to be used, a term `state_action` with the highest value is found based on the current program and possible means of plans or intentions in this step. If such a term does not exist, the next step is chosen randomly, if such a term exits, the intention or plan of this term is chosen as the next step for agent.

4.6 Usage

This section describes the usage of implemented MCTS methods. Where we are able to run these methods in the FRAg system or separately for testing intention scheduling for a specific agent step. The complete documentation for configuration of the program in FRAg, setup, and how to run the FRAg program can be found on the official public repository[1]. This section mainly focuses on what needs to be changed or added in order to use new MCTS methods for intention scheduling in existing FRAg programs. The provided implementation contains the whole FRAg system with new MCTS methods added, if instead a new methods should be integrated into current FRAg system the new or modified files need to be added. These files can be seen in the Appendix A, where every new or modified file is shown. The folder structure is the same as for the publicly available FRAg version.

Before we change any setting for the FRAg system, we first need to set up Python interpreter. It is recommended to containerize the Python interpreter so it can be easily started in the same state every time. For containerization, we use a tool `anaconda`. The Python libraries that are needed can be either installed by using the `requirements.txt` file to install them directly with Python by calling `pip install -r requirements.txt` or by creating a new anaconda environment with the configuration file `environment.yml` by calling `conda env create -f environment.yml` in the anaconda terminal.

In order to use new implemented MCTS methods in the already defined FRAg system program. We need to change multi agent setting in `.mas2fp` file. An example is shown for the `garden` program that is implemented in FRAg examples. The entire setting file can be found in Appendix B. First we need to specify which reasoning method we want to use, this is done in `set_agents` by adding option like `(reasoning, method)`. There are four new reasoning methods that can be used:

- `mcts_reasoning_integrated_python`
- `mcts_reasoning_external_service_sequential`
- `mcts_reasoning_external_service_parallel`
- `mcts_reasoning_online_learning`

Next we need to specify parameters for the MCTS method, that is done by adding an option like `(reasoning_params, mcts_params(5,5,60))` where the first attribute specifies

[1] https://github.com/VUT-FIT-INTSYS/FRAg

the α computation budget, the second attribute specifies the β computing budget, and the last attribute specifies how many steps should be done in one roll out.

Lastly, when we use a reasoning method that calls Python scripts by using the Janus module, we need to specify setting `(python_interpreter, true)`, which sets up the Python interpreter in a way so it is connected to the main Prolog thread. The reason why this needs to be done is described in Section 4.4.4. If we want to use a reasoning method that calls an external Python service, we first need to be in a Python environment that contains all the libraries. Then we need to start the MCTS service as `python mcts_service.py`.

If we want to run one MCTS computation on some known agent state, we can do so by running the Python script `mcts_integrated.py` in order to start the integrated architecture version and `mcts_service_call.py` to start the external architecture version. In these scripts in the body of the main method, the function that is used in the FRAg system is called. We are able to specify each input attribute that should be used. Meaning, we are able to properly test the communication with the Python service and the final results. In order to run this function we need to specify these attributes `program`, `Environment state`, `Alpha` computing budget, `Beta` computing budget, `Steps` number of steps in the roll out, `Type` for the type of MCTS method, this is specific to external architecture, `Environment name` and `Environment path` is the path file where the environment is implemented. When running MCTS methods in FRAg these settings are filled in their representative reasoning method as described in Section 4.3.1.

Chapter 5

Comparison of reasoning methods

This chapter presents two experiments, with the aim of comparing the ability of intention scheduling for implemented methods from section 4 and existing methods in the FRAg system. Experiments are performed on the program `garden` that is newly implemented, but is similar to the included example task maze in the FRAg system, where the first experiment is also performed on the example task maze. First, we compared the potential overhead in terms of architecture type. Then we compare the quality of the results for every method.

5.1 Task maze program

For the first experiment, the task maze example provided in the FRAg system is also used. This example was implemented for experiments to compare the late variable binding strategy with the early variable binding strategy [12].

The agent is situated in the grid based environment, where each square has one of four material, which are gold, silver, bronze, and dust. The agent can move in the environment between squares and pick up material. If material is picked up, it degrades into lower value material, where gold is the highest and dust is the lowest. During movement, the agent can only move to the squares that are next to him. The belief base for the agent are items that he already picked up, current position with material, information about materials that are on the squares around him, and generated tasks that the agent should try to accomplish. The task is represented as a three material, where in order to complete the task agent needs to pick up each material in the specified order. When an agent completes a task, he receives a reward.

The main reason for the use of the task maze example is because compared to `garden`, the effort to achieve a top level goal, which is the selected task, is much higher. That means that the GPT that is created for the MCTS method can be larger, which can lead to potentially longer runtime.

5.2 Garden program

The garden program is newly implemented for these experiments. The agent is situated in a grid based environment that is similar to the task maze environment, where the goal is for the agent to water each zone, while only having a limited supply of water. This program is designed to test the agent intention scheduling for more continues simple task. Because the

agent has a limited supply of water, the simulation is designed to end after some time, when the agent can no longer water any zone. Where the current set of rules for environment is designed in a way to stress test the agent, meaning even with unlimited water tank it is not possible to keep the moist level for every zone high, so the agent is trying to keep the zones moist level as high as possible.

5.2.1 Definition of the garden example

The garden example itself then specifies these rules:

- The agent is situated in a grid based environment that is 3x3 in size. Where each space has its own zone. The agent starts in the upper left position that is `[1,1]`.
- Every zone has a level of moisture that can range from 0 to 100.
- At the start, each zone has a different moisture level.
- The agent can either move from one zone to another or water a zone that is located on the space where the agent currently is.
- The agent has a limited water tank that has a value of 10.
- The agent can only see zones around it and can only move to these zones, similar to the task maze example.
- Agent knows about two zones that have the lowest moisture level, these zones are marked as critical zones.
- After every action agent does in the environment, every zone degrades and its moisture is reduced by 4.
- When an agent waters a zone, the moisture level is increased by 30 points and the value of the water tank is decreased by one. If the agent has an empty water tank, he can not water any zone.
- After the agent waters a critical zone, two zones with the lowest moisture level are selected as critical zones.
- When an agent waters a zone, he receives a reward, where the value is determined by the level of moisture the zone has after watering. If the agent waters a zone that has a moisture level higher than 80, the reward is 1. If the moisture level is lower than 80 the reward is 2. In order to encourage agent to prioritize critical zone if the hydration zone is lower, rather than watering zone that is already at 100.
- The agents continued goal is to water any critical zone.

Figure 5.1 shows the environment state at the beginning of the simulation. Where each space represents a zone and has its own moisture level. The agent is located in the top left zone `A`.

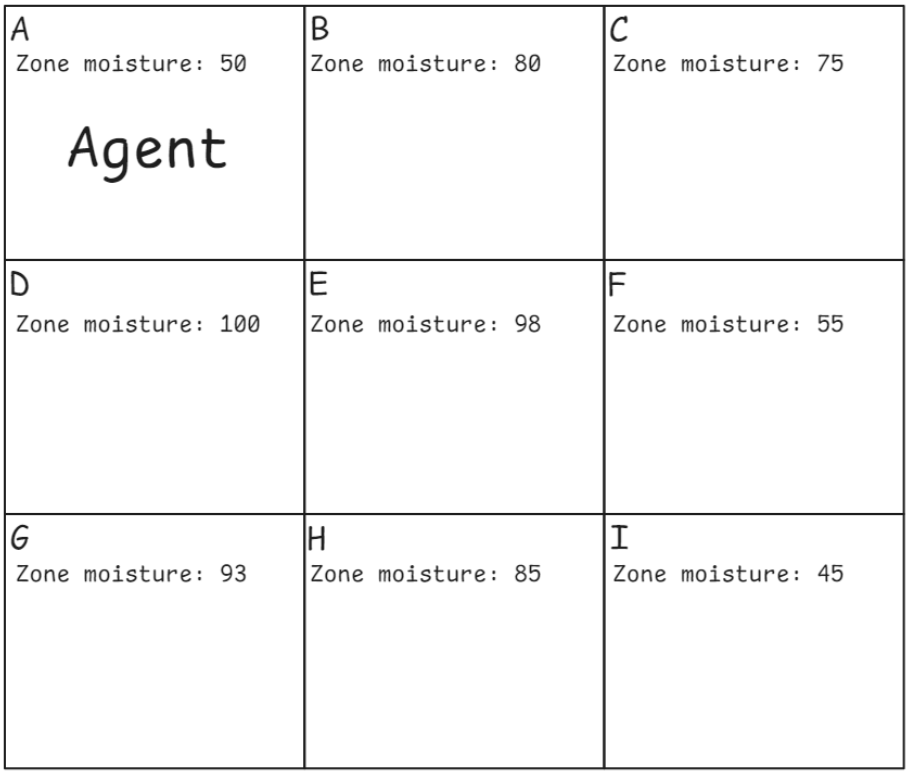

Figure 5.1: State of the garden environment at the start of simulation

Listing 5.1 shows the agent belief base, that is his current position, which zones are currently marked as critical, the remaining water in the water tank, zone where the agent is currently located, and the door that represents the zones around the agent.

```
fact(my_position(a));
fact(critical_zone(a));
fact(critical_zone(i));
fact(available_water(15));
fact(zone(a,[1,1]));
fact(door(right,b));
fact(door(down,d));
```

Listing 5.1: Agents belief base at the start of the simulation for garden program

5.2.2 Implementation of environment

The garden environment is implemented in the folder `environments`, where every example environment is located. The environment is implemented in Prolog in a file `garden.pl`. The environment is implemented in the similar way as other example environments for task maze, shop, and work shop. So the Prolog file implements these predicates, for initialization, adding the agent into the environment, cloning the environment, saving and loading the instance of environment, moving agent in the environment and watering the zone in the environment. The predicate for agent movement is the same as for task maze, where agent can move only one square in available direction, but after the movement every zone is degraded where moisture level is lowered by four. When watering the zone, it is first

checked if there is any water left in the tank. If so, the moisture of the zone is queried and increased by 30 points. Then the water in the tank is decreased and the reward value is selected. In the end, all zones are degraded in the same way as during movement and two zones with the lowest moisture level are selected, and a critical zone term is added for these.

5.2.3 Agent program and settings

The agent program is implemented in the folder `examples` for the garden in a file `garden.fap`. Because the agent moves in the environment for zones, the program is similar to the task maze agent program. One main goal `maintain_garden` and four plans are specified, the first plan is specified to solve the main goal by first creating goal `resolve_critical` for critical zones, where after resolving the critical zone a `maintain_garden` goal is again created. For resolving the critical zone, the next three plans are specified. If the current zone where the agent is located is the critical zone, the agent waters it. If the critical zone is located around the agent, the agent first moves to the zone and then waters it. If the critical zone is nowhere near the agent, the agent moves in a possible direction to find the zone.

The agent settings are specified in file `garden.mas2fp`, where the environment file for the garden is included. The agent specific setting is different based on the reasoning method that is used, and the specific settings are then described for every experiment. At the end of settings file, the agent is loaded with the specified program.

5.3 Experiment implementation

Each experiment is implemented as a Python program, in order to automate the execution of different configurations of the `garden` program. Where different agent settings that specify reasoning method is implemented in the `garden` folder for examples, where each method has its own folder where all the results are then stored. The algorithm 3 shows how each configuration of the garden program is set up and started.

Algorithm 3 Set up garden simulation, run it and store the results

Require: Experiment class attributes: FRAg program command, path to FRAg settings, path where result should be stored, number of runs for each method, name and path where the output file is created

Require: Experiment run input: name of folder where method is stored, number of steps for program, α, β, number of steps in roll out

1: Set up path of output for method that is being run
2: **for** `number of runs` **do**
3: Replace garden settings file `.mas2fp` with setting file from method folder
4: Replace reasoning parameters α, β and roll out steps for MCTS method
5: Replace the number of steps for FRAG program
6: Start measuring time that it takes for simulation to run
7: Start FRAg simulation by using the package `subprocess`
8: End measuring time that it takes for simulation to run and save it to the list
9: Save the output of FRAg simulation for the run to the output path
10: **end for**
11: Create output file that contains measured times for each run

For a better evaluation of each method, the results of simulation runs are then visualized. For creating the plots, the `seaborn` and `pyplot` libraries are used. For formatting the data for plots, a `pandas` library is used. The implementation is separated to file `plot_results.py`, where specific results can be plotted. During the second experiment, the results of each run are also automatically plotted. The final plots are created from the stored output files and are stored in the same output folder as the results of each run. The output file is split into agent steps and for each step the zone moisture level for every zone is extracted and stored, at the end all the extracted data are used for plotting the zone moisture level for every agent step.

5.4 Experiment comparing system architectures

The goal of this experiment is to compare the integrated architecture with the external architecture and to find if the potential computational overhead of the external service is significant enough to impact the time it takes to schedule intention. Or if the potential to start the service externally with a specific Python interpreter can help with the performance. Lastly, we want to find out if this potential overhead can be mitigated by using parallel optimization for simulations. The second goal is to compare the new MCTS implementation in Python to the already included MCTS implementation in Prolog to see if there is any benefit to moving some parts of intention scheduling to different languages, while keeping the option to still use first-order logic for program specification.

To explore different variations of computational budget, we test how much time it takes to get result for five steps of agent simulation. Where the setup is done for multiple different settings of α and β budgets, steps during the rollout were set to static forty. Each setting is run 10 times, where the average and lowest time is taken as a result. The experiment is written in Python and is called `epxeriment1.py`, where implementation is described in Section 5.3. The start time is taken before running the FRAg simulation and the end time is after the simulation completely ends. So we do not measure just the one step of the agent, but a whole simulation that does five steps. The goal is to see if the overhead of communication has an effect on a whole simulation. The experiment is run for both the task maze example and for the garden example. For the parallel implementation of MCTS, the number of workers is set to four.

The tab 5.1 shows the results of the experiment for the task maze example and the tab 5.2 shows the results of the experiment for the garden example. Both experiments show that for very low configuration of α and β, the MCTS implementation in Prolog is the fastest. This is expected because the computation budget is so low that the calculation of MCTS intention scheduling is so fast compared to the overhead of starting Python script. The same result is between the sequential and parallel implementation of MCTS, here the overhead of starting a persistent worker is even higher. For higher computational budget, we start to see that the implementation in Python is faster than the implementation in Prolog, where the gap starts to be even higher in a bigger computational budget. The similar case happens for the parallel implementation, where even for computation budget of 5 for α and 5 for β starts, it starts to be faster than the sequential variant. The gap becomes even more prevalent for a higher computational budget. For an experiment that was run on the `garden` program, the gap between each implementation is less significant. This could be explained because the garden scheduling is for a short run more simple, so the size of GPT in MCTS is smaller and thanks to that the computation is faster.

Configuration	Prolog MCTS	Integrated arch.	External arch. sequential	External arch. parallel
α=1, β=1 (Avg)	4.4760	9.5292	8.5173	12.9548
α=1, β=1 (Min)	3.7297	7.6638	7.8411	11.5964
α=5, β=5 (Avg)	98.1758	34.8420	36.6804	25.3186
α=5, β=5 (Min)	92.9426	33.2225	34.8548	24.3520
α=10, β=10 (Avg)	369.1806	90.5079	92.7427	43.1175
α=10, β=10 (Min)	340.9618	83.5594	89.9693	40.2057

Table 5.1: Results of 10 independent runs. With average and minimum runtime (s) for task maze example of different configurations for α and β, for different MCTS method

Configuration	Prolog MCTS	Integrated arch.	External arch. sequential	External arch. parallel
α=1, β=1 (Avg)	1.5989	6.7598	7.0687	11.3861
α=1, β=1 (Min)	1.4708	6.2643	6.6872	10.7034
α=5, β=5 (Avg)	29.2795	17.7207	18.0286	16.0244
α=5, β=5 (Min)	28.3783	16.2944	16.8605	15.3606
α=10, β=10 (Avg)	110.9685	41.3752	49.0347	28.1919
α=10, β=10 (Min)	104.9796	30.3402	40.4067	24.7958

Table 5.2: Results of 10 independent runs. With average and minimum runtime (s) for garden example of different configurations for α and β, for different MCTS method

The first goal of the experiment was to mainly compare the two different architectures, where as we can see, the integrated architecture is slightly faster, but the difference always stays around two seconds. Meaning for a higher computational budget the difference is negligible. The external architecture allows us to use a parallel variant of MCTS, which for higher computational budget shows significant improvement and is only worse for really low computation budgets of 1 α and 1 β. It is important to note that this low budget is not practical to be used for actual intention scheduling because there is no actual benefit in using such a configuration compared to methods like random selection.

The second goal was to compare the already included implementation of MCTS in Prolog, with the new implementation in Python. Here we can see that the implementation in Prolog is faster only for the really low budget of 1 α and 1 β, which, as was discussed above, is not a practical configuration which would be used for real programs. For even a slightly higher computational budget, we already start to see a speed improvement in Python implementation compared to Prolog implementation. With higher computational budget such as 10 α and 10 β, the difference starts to be significant for both sequential and parallel implementation in Python.

5.5 Experiment comparing results of reasoning methods

In the second experiment, the goal is to compare the quality of intention scheduling, for build in reasoning methods in the FRAg system and new MCTS implementations. The second

experiment is only run on the `garden` program, where different computation budgets for the α, β, and roll out steps are explored for the MCTS methods. Each run of FRAg simulation runs for 50 steps. The goal is for the agent to keep as many zones watered with a limited supply of water in the tank. The initial setup of the simulation is described in Section 5.2. The experiment is implemented in Python in a file `experiment2.py`, where implementation is described in section 5.3.

Each configuration is run five times, and each run output is saved separately. Then, the level of moisture for each zone is plotted over time. The average value of all zones is taken as the final value for the simulation run. For every configuration, the average value of every run, the highest value run, and the water tank value for the highest value run are taken as a result. For sequential variant of MCTS in Python we only use the external architecture, because, as was shown in experiment one, the difference in speed between these is really small. For the parallel implementation of MCTS, the number of workers is set to four.

The tab 5.3 shows the results for MCTS methods for different configurations of the computational budget. For every method, the results represent the average value of every run, the highest value run with the water tank level for the highest value run. The results show that almost every MCTS method performs best for low computation budget of α 5, β 5 and roll out steps of 4 or 5. For higher computation budgets, the average and the highest value start to drop off. The same happens for really low roll out steps of 3 where the results start to be more random.

This indicates that the `garden` example is a simple program where a higher computation budget with a deeper search does not yield better results because it introduces too much noise. In order for agent to achieve the top level goal of watering the critical zone, usually only small amount of steps need to be taken, depending on the agent position and the position of the zone. So by doing a deeper search in terms of expanding more MCTS nodes with agent states, it does not introduce better ways to achieve the top level goal, but rather introduces how to achieve next top level goals which can skew other nodes to have a higher value while the best possible step to achieve current top level goal gets drowned out. The same happens for the number of steps in the roll out, with more random steps, the random reasoning starts to go into paths that do not make that much sense but still yield some reward. If we want a higher computation budget to perform better, we would need to make the example `garden` more complex, this could be done by making the grid layout bigger or by introducing more complex top level goal like watering both critical zones after each other.

When comparing MCTS methods to each other, we can see that the basic MCTS method performs similar in Prolog and Python implementation. The biggest difference is that Prolog implementation of MCTS is more deterministic in how intentions are scheduled. This could be explained by slightly different implementation of the MCTS method, where the Prolog version does not choose a random child node after expansion but rather first node and for selection of node a non-expanded node is always chosen first. These make the MCTS implementation more deterministic, which, compared to the Python version, leads to a better average value but worst highest value. Implementations also differ in the amount of water used, while Prolog implementation always uses all the water, the Python implementation still has some water left in the tank. This can be viewed as both negative and positive, meaning that the Prolog variant tries to use more water aggressively, while Python implementation preserves more water while achieving similar results. In general, both methods perform on a similar level.

Slightly bigger difference can be seen for the online learning variant of MCTS, here both average and highest value are the best of all the MCTS implementations. This shows that during the simulation, the roll out is more focused on higher yielding paths, which in return leads to MCTS choosing a better next intention. On the other hand, for higher computation budget, the online learning variant starts to perform worse than other methods, this could be because the noise that is created by planning more in the future gets even stronger.

Configuration	Prolog MCTS	External arch. sequential	External arch. parallel	online learning
α=5, β=5, roll out=3 (Avg/-Max/Water tank level)	15,4/15,4/0	12,6/22,3/2	11,5/18,3/2	3,7/4,8/6
α=5, β=5, roll out=4 (Avg/-Max/Water tank level)	15,4/15,4/0	13,2/18,3/2	14,4/19/3	19,4/23,7/1
α=5, β=5, roll out=5 (Avg/-Max/Water tank level)	15,4/15,4/0	16,8/23,7/1	17,3/23,7/1	19,4/26,3/2
α=5, β=5, roll out=8 (Avg/-Max/Water tank level)	15/15,4/0	14,7/18,3/2	11,6/18,3/2	13,6/22,3/2
α=10, β=5, roll out=5 (Avg/-Max/Water tank level)	13,6/14,2/0	13,3/19/3	12,92/15/3	10,2/12,1/4
α=10, β=10, roll out=5 (Avg/-Max/Water tank level)	13,3/13,3/0	11,6/15,6/3	12,7/21,5/3	8,2/9,2/4
α=15, β=10, roll out=5 (Avg/-Max/Water tank level)	13,4/14,2/0	7,1/16/3	8,8/16/3	7,4/11,1/3

Table 5.3: Results of 5 independent runs. With average and maximal value with the final water tank level for maximal value, for a run of garden example of different configurations for α and β, for different MCTS method

The tab 5.4 shows the results for random and round-robin reasoning methods from the FRAg system, with the best possible result of each MCTS method, where all the results for MCTS methods are shown in the tab 5.3. The results show that all the MCTS methods outperform the build-in reasoning method of round-robin and random. Where random performs the worst, which is expected because there is a high probability of randomly getting stuck, is searching for the next critical zone. Round-robin reasoning performs better than random and is close to MCTS reasoning, but the average value of all runs shows that it is not possible to consistently achieve higher value. The `garden` example contains a simple top level goal, which helps intention schedulers such as round-robin. From all the reasoning methods the new online learning variant of MCTS performs the best by being able to consistently achieve the high result for multiple runs. The basic variant of MCTS in Python is also able to achieve high value for a run, but it is not able to keep it up for multiple runs. That is, the advanced MCTS method is able to achieve the highest value more consistently, while also using the same computational budget as other reasoning methods. For the evaluation of the results, only the most interesting plots are shown. All the figures for the best run of every evaluated method are shown in the Appendix C, for a quick comparison.

Reasoning method	Average value of all runs	Value of best run	Final water tank level for best run
Random	5,1	7,6	3
Round Robin	11,6	15,4	0
Prolog MCTS	15,4	15,4	0
External arch. sequential	16,8	23,7	1
External arch. parallel	17,3	23,7	1
Online learning MCTS	19,4	26,3	2

Table 5.4: Results of 5 independent runs. With average and maximal value with the final water tank level for maximal value, for a run of garden example of different methods from FRAg system and best configuration of each MCTS method

Figure 5.2 shows the level of moisture for every zone of each agent step for the round-robin reasoning method, and figure 5.3 shows the level of moisture for every zone of each agent step for the random reasoning method. Here we can see that random reasoning gets stuck in watering zones i and b, where most of the time is spent, because of that at the 4 out of 9 zones have a moisture level of 0. The Round Robin method can spend more time actually watering, it still has a problem with watering zone i, but it can instead pursue the second critical zone. Most of the zones moisture levels are then kept higher than 0, but Round Robin uses all the available water in the tank, meaning just a few more agent steps would lead to moisture levels only falling lower.

Figure 5.2: Value of each zone moisture of each agent step for best run of the Round Robin reasoning method

Figure 5.3: Value of each zone moisture of each agent step for best run of the random reasoning method

The best runs for every MCTS method look very similar, meaning that a better average value was achieved by consistently being close to the best possible run, instead of achieving some result that other MCTS method cannot achieve. All the figures for the best run of every method are shown in the Appendix C. So instead of only comparing the best runs, we compare how the best run differentiates to a worst run.

Figure 5.4 shows the level of moisture for every zone of each agent step for online learning variant of the MCTS reasoning method, for the best run. Figure 5.5 shows the result of the worst run for the parallel version of the MCTS reasoning method, which is similar to the result of the sequential version. Figure 5.6 shows the best run result for the MCTS method implemented in Prolog.

As we can see, the online learning variant is able to keep all the zones moisture level positive. Although the worst run for parallel variation was unable to keep the moisture level in two zones positive, the Prolog implementation was not able to keep one zone positive. Between the Prolog implementation and online learning variant, one key difference is in zone i where the online learning was able to keep the zone in positive, the Prolog variant let the zone go to 0 and instead focused on other zones. During the worst run of the parallel variant, it focused on the watering zone i, but then missed an opportunity to water the zones that were close to each other. The `garden` environment does not give any punishment for letting the zone go to 0. So, for some configuration, the better option is to let the zone go to zero and water it when there is another critical zone closer. Instead of wasting the moisture of every zone by doing more movement to water only one critical zone. The implementation of MCTS in Prolog behaves in a similar way as for the new MCTS methods, but wastes some of the possible steps, which leads to a slightly lower value at the end when compared to the best runs of online learning or parallel variant.

When all three versions of MCTS are compared, we can see that the levels of moisture are very similar throughout the simulation. This is expected because there are not that many options to do things differently that would lead to marginally better results. So,

the online learning variant of MCTS performs the best because it achieves slightly better results more consistently.

Figure 5.4: Value of each zone moisture of each agent step for best run of the online learning variant of MCTS method

Figure 5.5: Value of each zone moisture of each agent step for the worst run of the parallel MCTS method implemented in Python

Figure 5.6: Value of each zone moisture of each agent step for best run of the MCTS method implemented in Prolog

The goal of the second experiment was to compare different reasoning methods for intention scheduling in a simple example. Where all of the MCTS methods show a better performance compared to random and round-robin reasoning. Where MCTS implementation in Prolog being the benchmark for the MCTS that are implemented in Python, here basic MCTS shows a very similar performance. The online learning variant of MCTS shows a slight improvement compared to the basic version of MCTS.

5.6 Summary of experiments

In this section a new MCTS implementation in Python is compared to other reasoning methods in different benchmarks. The results indicate that the MCTS implementation in Python can be faster than the implementation in Prolog, especially for a higher computational budget, where the parallel implementation has the potential to use multiple workers efficiently. Also, the results indicate that the difference between the integrated architecture and external architecture is really small and can be ignored for a higher computational budget.

When comparing reasoning methods that are included in the FRAg system, which are Random, Round Robin, and MCTS in Prolog. We can see a trend where the Python implementation of basic MCTS performs on the same level as the MCTS in Prolog, while performing better than the Random and Round Robin reasoning methods. The online learning variant shows slightly better results than the MCTS implemented in Prolog.

This shows that there may be potential benefit to moving some parts of intention scheduling to different language, while also demonstrating that methods that use MCTS may yield better results than more basic versions of intention scheduling like random or Round Robin. It also important to note that all the methods that use MCTS are significantly slower than methods such as random, this was not part of the experiments but it is apparent just from the principal of these methods, because there is no model that needs to be computed in order to schedule intention, the selection of either action or plan is done at

time of selection. This leads to almost instant selection of what the agent should do next. In a provided program `garden` the speed of intention scheduling is not important, because the main metric is the number of steps the agent takes. For different type of program, simple methods like random or Round Robin could still lead to better results if the restriction is rather time based.

Chapter 6

Conclusion

In this thesis, we have looked into advanced intention scheduling methods for Belief-Desire-Intention agents. These incorporate Goal-Plan tree (GPT) structure with Monte Carlo tree search (MCTS) in order to select which intention should the agent pursue. Each advanced method builds upon a baseline implementation that uses interleaving at action level, either by optimizing the search for intention or by modifying the method so it can be used in systems that introduce things such as uncertainty. Lastly MCTS modifications that allow us to use them in multi-agent systems are introduced.

Next, we introduced an approach that allows us to use these methods in systems based on languages of first-order logic such as AgentSpeak(L). Which is a late variable binding strategy in order to modify the GPT so it could be used in MCTS methods without the need to create all the possible nodes for every substitution. Instead a plan node holds a context structure that contains all the possible substitutions created by using the broad unification. With each execution of plan action than modifying the context structure with the use of restriction operation for the specific type action.

In order to integrate new MCTS methods into an FRAg system, different options for communication between Prolog and different languages were explored. The Python language was chosen for implementation, with adequate Janus and PrologMQI interfaces that are included in SWI Prolog. Two architectures were designed. Integrated architecture that starts the MCTS method in the Python function that is called by the Janus interface and external architecture that instead uses a running service that starts the MCTS method, which communicates with the Python function that is started by the Janus interface. With the main goal of evaluating the potential benefit of using an external running service for the parallel variation of the MCTS method.

Four new reasoning methods were added to the FRAg system, where two are basic MCTS methods, one is a parallel variation of the basic MCTS method, and one MCTS that uses online learning. The two architectures were evaluated in order to compare their speed with the Prolog version of MCTS. Next, all the reasoning methods were compared in their quality of results to different build in methods in FRAg system. The result shows that the MCTS methods implemented in Python are faster for bigger computation budget than Prolog version of MCTS. While also being able to achieve similar or better results for `garden` program. The online learning variant of MCTS is able to achieve slightly better results than all the available reasoning methods.

Future work can be divided into two different points of improvement. First, it could be beneficial to improve current MCTS methods in their speed, by either improving the current Python implementation or changing the implementation to a different language such as `C`

or `C++`. Second, the integration of different language into the FRAg system allows us to more easily implement different reasoning methods for intention scheduling. These could be different variations of MCTS or completely different intention scheduling methods. The Python ecosystem also opens up new opportunities to use libraries that could help improve reasoning. Next work could also include the integration of new MCTS methods into multi agent system. The potential way to use MCTS schedulers for multi agent systems was outlined in the text. For future works, it would be beneficial to further experiment on the online learning variant, to find programs for which the method would be best suited. Further extraction of some parts of the FRAg system to different languages could also potentially yield some improvement.

Bibliography

[1] Bordini, R.; Seghrouchni, A.; Hindriks, K.; Logan, B. and Ricci, A. Agent programming in the cognitive era. *Autonomous Agents and Multi-Agent Systems*, may 2020, vol. 34.

[2] Bouneffouf, D. Finite-time analysis of the multi-armed bandit problem with known trend. In: *2016 IEEE Congress on Evolutionary Computation (CEC)*. 2016, p. 2543–2549.

[3] Dann, M.; Thangarajah, J. and Li, M. Feedback-Guided Intention Scheduling for BDI Agents. In: *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2023, p. 1173–1181. AAMAS '23. ISBN 9781450394321.

[4] Dann, M.; Thangarajah, J.; Yao, Y. and Logan, B. Intention-Aware Multiagent Scheduling. In: *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2020, p. 285–293. AAMAS '20. ISBN 9781450375184.

[5] Dann, M.; Yao, Y.; Logan, B. and Thangarajah, J. Multi-Agent Intention Progression with Black-Box Agents. In:. August 2021, p. 132–138.

[6] Logan, B.; Thangarajah, J. and Yorke Smith, N. Progressing Intention Progression: A Call for a Goal-Plan Tree Contest. In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2017, p. 768–772. AAMAS '17.

[7] Rao, A. S. AgentSpeak(L): BDI agents speak out in a logical computable language. In: Velde, W. Van de and Perram, J. W., ed. *Agents Breaking Away*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, p. 42–55. ISBN 978-3-540-49621-2.

[8] Rao, A. S. and Georgeff, M. P. Modeling rational agents within a BDI-architecture. In: *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, p. 473–484. KR'91. ISBN 1558601651.

[9] Rao, A. S.; Georgeff, M. P. et al. BDI agents: from theory to practice. In: *Icmas*. 1995, vol. 95, p. 312–319.

[10] Schadd, M. P. D.; Winands, M. H. M.; Tak, M. J. W. and Uiterwijk, J. W. H. M. Single-player Monte-Carlo tree search for SameGame. *Know.-Based Syst.* NLD: Elsevier Science Publishers B. V., october 2012, vol. 34, p. 3–11. ISSN 0950-7051.

[11] Song, C.; Yao, Y. and Chan, S. Incorporating Online Learning Into MCTS-Based Intention Progression. *IEEE Access*, 2024, vol. 12, p. 56400–56413.

[12] Vidensky, F.; Zhoul, F. and Veigend, P. Integrating Late Variable Binding with SP-MCTS for Efficient Plan Execution in BDI Agents. In: *Proceedings of the 17th International Conference on Agents and Artificial Intelligence (ICAART 2025) - Volume 1*. 2025, p. 679–686. ISBN 978-989-758-2724-5.

[13] Vídeňský, F.; Zbořil, F.; Beran, J.; Kočí, R. and Zbořil, F. Comparing Variable Handling Strategies in BDI Agents: Experimental Study. In: *Proceedings of the 16th International Conference on Agents and Artificial Intelligence - Volume 1*. Rome: SciTePress - Science and Technology Publications, 2024, p. 25–36. ISBN 978-989-758-680-4. Available at: https://www.fit.vut.cz/research/publication/13085/.

[14] Waters, M.; Padgham, L. and Sardina, S. Evaluating Coverage Based Intention Selection. In:. January 2014, vol. 2.

[15] Wooldridge, M. *An Introduction to MultiAgent Systems*. 2nd ed. John Wiley & Sons, 2009. ISBN 978-0-470-51946-2.

[16] Wu, D.; Yao, Y.; Alechina, N.; Logan, B. and Thangarajah, J. Intention Progression with Maintenance Goals. In: *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2023, p. 2400–2402. AAMAS '23. ISBN 9781450394321.

[17] Yao, Y.; Alechina, N.; Logan, B. and Thangarajah, J. Intention progression under uncertainty. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*. 2021. IJCAI'20. ISBN 9780999241165.

[18] Yao, Y. and Logan, B. Action-Level Intention Selection for BDI Agents. In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2016, p. 1227–1236. AAMAS '16. ISBN 9781450342391.

[19] Yao, Y.; Logan, B. and Thangarajah, J. SP-MCTS-based intention scheduling for BDI agents. In: *Proceedings of the Twenty-First European Conference on Artificial Intelligence*. NLD: IOS Press, 2014, p. 1133–1134. ECAI'14. ISBN 9781614994183.

[20] Zbořil, F.; Vídeňský, F.; Kočí, R. and Zbořil, F. Late Bindings in AgentSpeak(L). In: *Proceedings of the 14th International Conference on Agents and Artificial Intelligence vol. 3*. Lisabon: SciTePress - Science and Technology Publications, 2022, p. 715–724. ISBN 978-989-758-547-0. Available at: https://www.fit.vut.cz/research/publication/12621/.

Appendix A

Content of the submitted archive with new and modified files compared to default FRAg system

```
text_xmisar01.pdf
Files to generate text
Readme.md
poster.pdf
FRAg
    core
    examples
```

```
FRAg
├── core
│   ├── environments
│   │   ├── basic
│   │   ├── counter
│   │   ├── doc
│   │   ├── garden (new)
│   │   ├── shop
│   │   ├── task_maze
│   │   ├── workshop
│   │   ├── FRAgPLEnvironmentUtils.pl (modified)
│   │   ├── FRAgPLEpisodeSteps.pl
│   │   └── FRAgPLStatsUtils.pl
│   ├── MCTS_python (new)
│   ├── reasoning_methods
│   │   ├── Basic
│   │   │   ├── FRAgPLReasoningRandom.pl
│   │   │   ├── FRAgPLReasoningRandomOnlineLearning.pl (new)
│   │   │   ├── FRAgPLReasoningRobin.pl
│   │   │   └── FRAgPLReasoningSimple.pl
│   │   ├── Joint
│   │   ├── MCTS-advanced (new)
│   │   └── MCTS-base
│   ├── base_experiment.py (new)
│   ├── experiment1.py (new)
│   ├── experiment2.py (new)
│   ├── fragagent.fap
│   ├── FRAgAgent.pl (modified)
│   ├── fragagent_fragagent.out
│   ├── FRAgAgentInterface.pl
│   ├── FRAgBlackboard.pl
│   ├── fraginit.mas2fp (modified)
│   ├── FragPL.pl (modified)
│   ├── FragPLFrag.pl
│   ├── FRAgPLRelations.pl
│   ├── FRAgSync.pl
│   ├── FRAgTerminal.pl
│   ├── mcts_integrated.py (new)
│   ├── mcts_service.py (new)
│   ├── mcts_service_call.py (new)
│   ├── plot_results.py (new)
│   ├── run.bat
│   └── stats.pl
└── examples
    ├── garden (new)
    ├── task_maze
    ├── trader
    └── worker
```

Appendix B

Example of FRAg setting file

```
include_environment("environments/garden/garden.pl").

set_agents([(bindings, late),
       (reasoning, mcts_reasoning_external_service_parallel),
              (reasoning_params, mcts_params(5,5,5)),
              (python_interpreter, true),
              (control, terminate(timeout, 50)),
              (environment, garden)]).


load("garden","../examples/garden/garden",1,[(debug, reasoningdbg),
              (debug, mctsdbg_path),
              (debug, mctsdbg),
              (debug, actdbg),
              (debug, interdbg), (debug, systemdbg)]).
```

Listing B.1: Compleate configuration file for FRAg system for new MCTS method.

Appendix C

Results of best run for every reasoning method in second experiment

(a) Round robin reasoning method.

(b) Random reasoning method.

Figure C.1: Value of each zone moisture of each agent step for the best run of Round robin and Random reasoning method.

(a) Parallel MCTS method implemented in python.

(b) Sequential MCTS method implemented in python.

Figure C.2: Value of each zone moisture of each agent step for the best run of Parallel and Sequential MCTS methods implemented in python.

(a) Online learning MCTS method implemented in python.

(b) MCTS method implemented in prolog.

Figure C.3: Value of each zone moisture of each agent step for the best run of online learning MCTS and MCTS implemented in prolog.

Appendix D

Poster