universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Shaping AI Behavior: A Q-Learning Driven Approach to Automatic Behavior Tree Creation“

verfasst von / submitted by

Ralph Dworzanski BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2023 / Vienna, 2023

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Acknowledgements

I want to thank Univ.-Prof. Dr. Helmut Hlavacs for his constant and neverending support. His feedback, ideas, and guidance made this thesis possible.

I also want to express my gratitude to my girlfriend, Jennifer. Her encouragement and unwavering patience helped and motivated me throughout the most challenging times.

Finally, I want to thank my family and friends, who have shown incredible belief in me during this journey.

Abstract

Video games have gained immense traction today, ingraining themselves deeply within popular culture and captivating a global audience. These interactive digital experiences have become an integral part of the fabric of modern society. Notably, the financial success of top-tier video game products now surpasses even the most lucrative film productions, underscoring the industry's exceptional commercial prowess and dominant market position. With the rising complexity of video games, we also find a rising complexity of tools required to develop high-fidelity graphics or believable and challenging Non-Player-Characters. Specifically, Behavior Trees have emerged as popular tools to model the tasks of such a Non-Player-Character as a network of hierarchical nodes. Addressing the issues of combinatorial explosion, which were a common problem in complex behaviors modeled as state machines, Behavior Trees solve this issue. However, hand-making Behavior Trees can still be difficult, complex, and error-prone. Thus, the rising popularity of Behavior Trees also brought widespread interest in ways to generate them automatically. Fully automatic creation is commonly solved using Evolutionary Algorithms, while machine-learning-backed approaches typically focus on improving existing Behavior Trees. In this thesis, we propose a novel solution to generate Behavior Trees automatically and autonomously from reinforcement learned autonomous agents. We developed a Capture The Flag-style game in which two teams compete to win. The Behavior Trees are generated from the knowledge of agents competing in this game by extracting this knowledge and parsing it into a Behavior Tree format. The proposed algorithm can generate these trees with comparable performance to the autonomous agents. Additionally, the generated trees are shown to be versatile, adapting to constraints not considered while they were created.

Keywords

Behavior Tree, Reinforcement Learning, Automatic Creation, Video Game, Non-Player-Character

Kurzfassung

Videospiele haben heute immense Popularität erlangt und sind ein fester Bestandteil unserer Kultur und sind zu einem integralen Bestandteil der modernen Gesellschaft geworden. Bemerkenswerterweise übertrifft der finanzielle Erfolg führender Videospielprodukte mittlerweile sogar die Einnahmen der teuersten Filme, was die außergewöhnliche kommerzielle Stärke und dominante Marktposition dieser blühenden Branche unterstreicht. Mit der wachsenden Komplexität von Videospielen geht auch eine zunehmende Komplexität der zur Entwicklung hochwertiger Grafiken und glaubwürdiger sowie herausfordernder Nicht-Spieler-Charaktere erforderlichen Werkzeuge einher. Hierbei haben sich insbesondere als beliebte Modelle zur Abbildung der Aufgaben solcher Nicht-Spieler-Charaktere Behavior Trees etabliert. Sie lösen das Problem der kombinatorischen Explosion, das bei der Modellierung komplexer Verhaltensweisen als Zustandsmaschinen häufig auftritt. Das händische Anfertigen von Behavior Trees kann jedoch nach wie vor schwierig, komplex und fehleranfällig sein. Daher ist mit deren steigender Beliebtheit auch ein weitreichendes Interesse an der Methoden entstanden zur automatischen Generierung entstanden. Die vollständige und automatische Erzeugung wird üblicherweise durch evolutionäre Algorithmen realisiert, während Ansätze auf Basis maschinellen Lernens sich in der Regel darauf konzentrieren, bestehende Behavior Trees zu verbessern. In dieser Arbeit wird ein neuartiger Ansatz vorgeschlagen, der eine vollständig automatische und autonome Generierung Behavior Trees aus autonomen, durch maschinellen Lernens erworbenem Wissen ermöglicht. Hierfür wurde ein "Capture the Flag"-Spiel entwickelt, bei dem zwei Teams um den Sieg konkurrieren. Die Behavior Trees werden durch Extraktion und Analyse des Wissens der Agenten generiert, die in diesem Spiel antreten. Der vorgestellte Algorithmus ermöglicht die Generierung von Behavior Trees die zu den autonomen Agenten vergleichbar leistungsfähig handeln. Darüber hinaus zeigen wir, dass die generierten Behavior Trees sich an Einschränkungen anpassen können, die während ihrer Erstellung nicht berücksichtigt wurden.

Contents

Acknowledgements i

Abstract iii

Kurzfassung v

List of Tables ix

List of Figures xi

List of Algorithms xiii

Listings xv

1. Introduction 1
- 1.1. Motivation . . . 1
- 1.2. Research Questions . . . 2
- 1.3. Synopsis . . . 3

2. Related Work 5
- 2.1. Brief History . . . 5
- 2.2. Creation Methods . . . 5
  - 2.2.1. Evolutionary Algorithms . . . 6
  - 2.2.2. Reinforcement Learning . . . 7
  - 2.2.3. Others . . . 7
- 2.3. Applications . . . 7
- 2.4. Limitations . . . 8

3. Background 9
- 3.1. Behavior Tree . . . 9
  - 3.1.1. Definition and Structure . . . 9
  - 3.1.2. Execution . . . 12
  - 3.1.3. Mathematical Definition . . . 13
  - 3.1.4. Extensions . . . 14
- 3.2. Reinforcement Learning . . . 15
  - 3.2.1. Overview . . . 15
  - 3.2.2. Parameters . . . 16
  - 3.2.3. Limitations . . . 17

4. Methodology **19**
4.1. Design and Implementation . . . 19
4.1.1. Architecture . . . 19
4.1.2. Behavior Tree Creation Algorithm . . . 26
4.1.3. Pipeline . . . 30
4.2. Objectives and Assumptions . . . 31
4.2.1. Objectives . . . 31
4.2.2. Assumptions . . . 32

5. Experimental Setup **35**
5.1. A Capture-The-Flag Simulation . . . 35
5.1.1. Architecture . . . 36
5.1.2. Game Rules . . . 36
5.1.3. Participants in the Game . . . 37
5.2. Learning Phase . . . 39
5.2.1. States of an NPC . . . 40
5.2.2. Reward Structure . . . 42
5.2.3. Opponent Selection . . . 44
5.2.4. Training Methods . . . 44
5.3. Adaptiveness and Performance . . . 45
5.3.1. Map Sizes . . . 45
5.3.2. Game Length . . . 45

6. Results and Discussion **47**
6.1. Comparison of Opponent Selection . . . 47
6.1.1. RL vs Static . . . 47
6.1.2. RL vs RL . . . 50
6.2. Capability to Adapt . . . 53
6.2.1. Map Sizes . . . 53
6.2.2. Game Length . . . 55
6.3. Impact of Training Policies . . . 58
6.3.1. Knowledge Transfer Strategies . . . 59
6.3.2. Reward Table Structure . . . 60
6.4. Results . . . 61

7. Conclusion and Future Work **63**
7.1. Conclusion . . . 63
7.2. Limitation and Future Work . . . 64

Bibliography **65**

A. Appendix **73**

List of Tables

4.1. Example of Max-Q table 27

5.1. Damage calculation based on distance (d) and Max Fire Distance (M) . . 38
5.2. Explanation of flag and teammate states 41
5.3. An abbreviated version of reward table. $x = any, N = None, L = Low, M = Medium, H = High$ 43

6.1. Default CTF settings 47
6.2. Performance of "Team Reinforcement Learning" and the resulting Behavior Tree 48
6.3. Performance of the "RL vs RL" learning method and the resulting Behavior Tree 51
6.4. Performance of automatically created BTs on differently sized maps 54
6.5. Performance of automatically created BTs in games of various tick lengths. 56
6.6. Performance of automatically created BTs in games of score to win reduction. 57
6.7. Comparison of two knowledge transfer policies 60
6.8. Results after omitting reward table and offline learning. 60

A.1. Score summary of training and BT generation against static team. 76
A.2. Score summary of pure reinforcement learning opponents and generated BT. 76

List of Figures

2.1. Grammatical expression of a behavior tree, as seen in [35] 6

3.1. Class diagram of internal and external nodes 11
3.2. Simple BT showing one selector-, two sequence-, two condition- and three action nodes 13
3.3. Action-Reward feedback loop during training 16

4.1. Class Diagram of the Behavior Tree library implementation 20
4.2. Sequence of behavior creation and execution 23
4.3. The behavior tree shown in Listing 4.1 visualized as tree 24
4.4. Class Diagram of the Q-Learning library implementation 26
4.5. Atomic unit as described in [2] 27
4.6. Algorithm output using the sample Max-Q table 28
4.7. Overview of the BT synthesis pipeline 30

5.1. Reward-Action cycle in a multi-agent setting. 40

6.1. Learning Phase Evaluation (RL vs. Static) 48
6.2. Behavior Tree Evaluation (RL vs. Static) 49
6.3. Result progression of all games in the learning phase. 50
6.4. Learning phase against static opponent first episode. 51
6.5. Learning phase against static opponents; Last episode. 52
6.6. Score Progression over all games during Behavior Tree evaluation (static opponents). 53
6.7. A BT created from experiences gained competing against the static team . 54
6.8. Learning Phase Evaluation (RL vs. RL) 55
6.9. Behavior Tree Evaluation (RL vs. RL) 56
6.10. Score Progression over all games during Behavior Tree evaluation (RL vs. RL). 57
6.11. Generated BT of the “RL vs. RL” experiment. 58
6.12. Summary of results on a 150×150 map 59

A.1. Class diagram of the application and its usage of the AI controller libraries 73
A.2. Activity Diagram of the Learning Phase 74
A.3. Second BT created from experiences gained competing against the static team 75

List of Algorithms

1. Pseudocode of a selector node with N child nodes 12
2. Pseudocode of a sequence node with N child nodes 12
3. Algorithm detailing the training loop using an ϵ-greedy policy 17

Listings

4.1. Sample BT as Json 21
4.2. Registration of Tasks and Instantiation of BT 22
4.3. Sample Task in C++ native code 24
4.4. Q-Learning functionality to update Q-Value 25
4.5. Behavior Tree Generation Algorithm 29
4.6. Generating atomic units 29
4.7. Merging common guards 29

1. Introduction

The field of Artificial Intelligence (AI) in video games has come a long way since its early days of simple rule-based systems and scripted behaviors for Non-Player Characters (NPCs). As video games' complexity has grown in areas such as real-time rendering, physics systems, and networking, there is a corresponding need for advancements in AI systems that control Non-Player Character (NPC) behaviors. However, creating intricate, intelligent behaviors with rudimentary tools not explicitly designed for this task can be laborious, error-prone, and challenging. Therefore, game developers have been exploring tools like Behavior Trees (BTs), Finite State Machines (FSMs), and Goal-Oriented Action Planning (GOAP).

One of the early public appearances of BTs addresses the issue of combinatorial explosion in FSMs, a popular design method to create simple behaviors. This phenomenon refers to the exponential increase in complexity that occurs with each added state in a state machine, leading to a decrease in maintainability and readability. BTs were developed to respond to this challenge. They offer a hierarchical and structured way to organize and prioritize behaviors in autonomous agents (self-governing entities), facilitating complex decision-making in response to varying conditions. Today, BTs are extensively used in video games and robotics for designing a range of behaviors, from simple tasks like movement and combat to intricate social interactions. This has made them a subject of wide-ranging research in academic and industrial fields.

Over time, the execution model of BTs, their internal mechanics, and layout details have undergone many refinements. This includes the addition of more intricate and specialized nodes to facilitate the creation of complex sub-trees using simplified structures and adapting traversal methods to meet the runtime demands of video games and industrial robots. Despite these advancements, the core principles and values that have made BTs a highly-regarded tool have remained intact and continue to define all improvements and adaptations.

1.1. Motivation

A popular and well-researched topic is improving behavior trees through restructuring, automatically generating them using autonomous means, or validating the safety and correctness of a behavior tree. The motivation behind automatic generation is often linked to applying the BT. For instance, in safety-related contexts like autonomous driving or crewless aerial vehicles, an autonomous agent driven by a behavior tree must

comply with specific safety protocols and guidelines. In contrast, in entertainment, more emphasis is placed on displaying complex and believable behaviors, sometimes resulting in intricate tree structures. Authoring these trees manually is often tedious and challenging, necessitating at least basic technical knowledge.

Past research has usually focused on generation methods that require an existing behavior tree to improve upon, either by manually building one or randomly generating one that is then enhanced using genetic algorithms, a way of optimization that mimics the process of natural selection. Our study deviates from this trend, instead investigating a method that allows the complete synthesis of a behavior tree without needing a pre-existing handmade or artificial input tree.

The algorithm proposed in this thesis aims to synthesize an NPC behavior tree in a video game context. This means that the accuracy and performance of the behavior shown aren't necessarily the most critical factors. The experiment utilizes a simple "capture-the-flag" scenario, a standard game mode in which two teams compete. Our system was developed with traditional video game elements in mind. Using reinforcement learning, a type of machine learning where an agent learns to make decisions by trial and error, the agents acting as players are trained to play and win the game. The knowledge obtained during this training phase is then used to synthesize a behavior tree.

We then test the validity and performance of these automatically created trees in the same game environment they were trained in. While generating behavior trees is a prevalent and thriving research topic, complete synthesis using reinforcement learning is a novel concept, as very few studies have explored this method.

1.2. Research Questions

The proposed method for the autonomous creation of behavior trees aims to reduce the complexity of developing behaviors for Non-Player-Characters by leveraging tabular reinforcement learning methods, such that these behavior trees can be used in game-like environments, outperforming handmade behaviors and policies. Furthermore, since using several different behavior trees in a video game for several characters of the same type is often required, we explore the effectiveness of using the proposed approach with a multi-agent reinforcement learning approach.

Finally, we explore which conditions need to be placed on the reinforcement learning algorithm to create a behavior tree successfully.

These goals are evaluated by answering the following research questions:

- **RQ1:** Can the proposed algorithm automatically create Behavior Trees that outperform manually created behaviors and policies?
- **RQ2:** Can the Behavior Trees generated by the proposed algorithm adapt to and perform well in unexpected scenarios not explicitly planned for during the creation phase?
- **RQ3:** How do parameter adjustments during the reinforcement learning phase affect the performance of the created Behavior Trees?
- **RQ4:** Can the proposed algorithm successfully generate Behavior Trees that facilitate cooperative behavior in a multi-agent setting?

1.3. Synopsis

This thesis is structured as follows:

Chapter 1 provides an overview of the problem addressed in this thesis and discusses our motivations for pursuing this research.

Chapter 2 offers an in-depth understanding of behavior trees, including their technical concepts, history, and design philosophies. This chapter also comprehensively introduces the reinforcement learning method used in our implementation.

Chapter 3 examines previous work on the automatic creation of behavior trees in industry and research contexts. This includes exploring how this topic is approached in the entertainment industry and robotics.

Chapter 4 presents our proposed algorithm for autonomous and automatic behavior tree creation. Further, we examine the architecture of the libraries we developed for our study.

Chapter 5 discusses the experimental setups and evaluation methods used in our experiments.

In Chapter 6, we delve into the results of our experiments and offer a thorough analysis of our findings.

Finally, Chapter 7 concludes our work. Here, we reflect on our results, discuss the limitations of our research, and suggest potential avenues for future exploration.

2. Related Work

This chapter describes previous approaches in the literature about the autonomous creation of BTs from various techniques to synthesize one entirely or improve upon an existing one. The analysis in this chapter is not limited to the autonomous creation of BTs in the video game industry.

2.1. Brief History

While the Behaviour Tree (BT) emerged as a tool developed by video game developers [18] to simplify Non-Player Character (NPC) development by providing a replacement for the widely used Finite State Machine (FSM), the first steps toward BTs were made even earlier by Mateas and Stern [26] who created a language to express characters' behaviors in video games. Unfortunately, tracking the exact origins and historical evolution of BTs difficult due to the nature of the video game industry. Conferences in this domain rarely publish peer-reviewed or academic work. This is also noted by Iovono et al. in their extensive survey paper about behavior trees [16]. The development and evolution of behavior trees are sparsely documented, and popular literature mentions motivations for their advancements only as side notes [36], making it difficult to establish an accurate timeline.

Earliest journal papers, such as the work in [13] and [12], were released several years after their initial proposal. The expressiveness of BTs was later discovered by scientists in the field of robotics. The tool was quickly adapted to new use cases such as combat simulations [3] or home applications [4], as well as industrial- and social robots [10], and vehicles [31]. Behavior trees have thus been a popular research topic, adding an event-based execution method [1], adding utility theory [28], improving asynchronous execution [6] or using them as the basis for knowledge transfer in multi-robot-systems [45]. The work of [19] proposes the addition of a unique node to incorporate emotion into the decision-making process of a behavior tree.
Initial methods for automatic creation were attempted by Lim et al. in [23], which has sparked broad interest in the subtopic of automatic creation.

2.2. Creation Methods

This section provides an overview of previous research on the automatic creation of BTs, divided by the creation method. While offering a great degree of modularity and thus maintainability, extensive BTs suffer from increasing complexity. Ensuring a behavior

tree acts well under certain constraints is often necessary for safety-related contexts. At the same time, in the video game industry, it is often essential to author a large variety of BTs for a wide variety of NPCs. Their modularity makes them thus popular for several approaches for both fully automatic synthesis and hybrid methods, where hand-made behavior trees are enhanced or otherwise changed.

2.2.1. Evolutionary Algorithms

As previously mentioned, the earliest works for automatically creating behavior trees in video games featured improving an AI agent in a commercial game [23]. The automatically created tree was able to outperform the original hand-coded AI. Improving the performance of an AI agent was further explored using the Mario AI Benchmark in a competition in which teams submit autonomous agents to play on a previously unseen level of the popular side-scrolling platform game *Mario*. TheBT was composed of actions Mario can perform. Conditions of the tree were mapped to matrices surrounding Mario, populated with information about the character's environment. The team expressed the Behaviour Tree using grammar rules and used Genetic Programming (GP) to improve its performance. The placement and overall performance strengthen the idea that BTs can be significantly enhanced using automatic approaches [35]. The same benchmark was used to further the development and research on automated creation using similar GP approaches [8, 30]

The work of [40] has contributed explicitly to video game development by reducing the manual labor required to design artificial intelligence agents. Genetic Algorithms, in conjunction with BTs, have also been used to test the viability of difficulty management in video games by generating a diverse range of behaviors [32]. Additional efforts include the automatic generation of AI opponents that proved capable of defeating human opponents but had difficulty beating traditional AI bots [15].

```
<BT>        ::= <BT> <Node> | <Node>
<Node>      ::= <Condition> | <Action>
<Condition> ::= if(obstacleAhead) then <Action>;
            | if(enemyAhead) then <Action>;
<Action>    ::= moveLeft; | moveRight; | jump; | shoot;
```

Figure 2.1.: Grammatical expression of a behavior tree, as seen in [35]

In [39], Scheper, et al. used an Evolutionary Algorithm (EA) approach to evolving a BT in a simulated environment. The resulting BT was transferred to a real-world hardware platform. The evolved BT slightly outperformed the manually designed behavior, suggesting that moving the knowledge gained in simulations to the real world and that a generated tree proves to be a valid alternative over human-generated behaviors is possible. Other applications suggest improving BTs with GP is a viable approach in environments

where a high degree of fault tolerance is required [17]. When applied to behaviors of swarm robotics, it was found that Behaviour Trees provide an attractive alternative to traditional approaches but do not necessarily produce better results than FSM [22], despite proving themselves viable in other swarm-like applications [20, 21].

2.2.2. Reinforcement Learning

The idea of improving or authoring behavior trees by Reinforcement Learning (RL) methods is novel. The earliest works include that of Dey and Child in [11], in which an existing behavior tree was improved by reordering nodes to achieve better behavior for the autonomous agents. In [34], the concept of learning nodes was researched. These nodes use RL methods to reorder sub-nodes within composites and allow action nodes to choose different actions based on the learned knowledge.

Similarly, the concept of a *LearningSelector* node that uses a RL algorithm to reorder nodes of a selector based on states and rewards was also researched in[14]. Exploring the combination of RL and BTs in safety-related contexts has shown that under certain design constraints, machine learning approaches can be used in these domains as well [42].

One of the more recent and most important contributions to this thesis is the work of Banerjee [2] in which a novel algorithm was presented, allowing the autonomous acquisition of a BT by using experiences gained from a reinforcement learning model. The work was later verified and extended in [25]. While previous approaches have only used Reinforcement Learning algorithms to enhance or improve existing trees, the algorithms presented in these works leverage reinforcement learning for behavior tree creation.

2.2.3. Others

Colledanchise et al. propose a method in [5] utilizing *Linear Temporal Logic* to construct BTs in polynomial time that correctly executes their assigned tasks.
Learning from demonstration refers to a technique in which a human actor takes the role of the autonomous agent and performs its tasks. The actions the human actor takes are then used to generate a BT. This approach removes the necessity to introduce AI designers to complex tools to model behaviors and does not require a behavior tree as input. [37, 38]

Case-based reasoning (CBR) is a method to find solutions to new problems by evaluating previous experiences. This technique was successfully applied to expand behavior trees dynamically [13] and later to dynamically query behaviors to assemble a complex behavior tree from a set of simple behaviors [12].

2.3. Applications

The modularity and expressiveness of behavior trees make them a popular tool for designing and modeling artificial behaviors in various scenarios. In the domain of video games,

they are used in multiple genres and applications such as navigation of platformers [30, 35], control of agents in marine-based-games [27], and automatic difficulty management in real-time-strategy games [15].

In addition, the strengths of BTs have also been leveraged in home applications [4], swarm robotics [33, 45] and autonomous driving [44]. Furthermore, in [46], automatic creation for unpredictable environments was successfully achieved.

2.4. Limitations

During our research, we discovered that most approaches for behavior tree synthesis rely on either an input tree or genetic algorithms to artificially create random input trees as initial population and use Genetic Programming (GP) to improve these populations. Other approaches require an existing tree to generate an improved version. Overall, only a little work conducted in complete from-scratch synthesis attempts to create a working and performant behavior tree without previous input or intermediate steps.

An exhaustive review of Behaviour Trees in robotics and Artificial Intelligence (AI) in [16] revealed that the current research shows a slight overlap between learning approaches and the video game AI domain. As explained earlier, this is likely due to the nature of the entertainment industry, in which innovations are often seen as trade secrets and are only shown at industry-centered conferences. However, video games and game-like simulations are often used as testing environments for automated creation technologies before they are applied in real-world scenarios. Of 31 papers reviewed in the video game domain and 37 focused on BT synthesis, only nine articles appeared in both categories [16]. Despite this, BTs remain a popular tool in the video games industry, and research regarding improving frameworks continues to drive forward [41].

3. Background

This chapter covers the background for the tools developed in this thesis. We first cover behavior trees, their definition, design philosophy, and mathematical formulation. Then we discuss the fundamentals of Reinforcement Learning (RL) and the *Q-Learning* algorithm. This tabular reinforcement learning approach enables autonomous agents to learn optimal policies to achieve a given task. These methods, in conjunction, provide the basis of the proposed algorithm to facilitate the automatic creation of Behaviour Tree (BT)s.

3.1. Behavior Tree

A Behaviour Tree (BT) is a decision-making and actuation architecture for controlling autonomous agents, such as autonomous robots or Non-Player Character (NPC) in video games. A BT defines the behavior of an autonomous agent by hierarchically modeling the actions it can perform and the perceptions about the agent's environment.

Behavior trees emerged as a tool to replace the widely used FSM in video games. For complex behaviors involving numerous tasks and transitions, FSMs can suffer from a combinatorial explosion. This rapid increase in complexity leads to an exponential growth in the number of states and transitions, making them increasingly difficult to manage and understand [9, p. 24]. Furthermore, the larger the FSM, the more computational resources it consumes, which can be a significant concern in time-sensitive applications like video games or robotics. By offering a more structured and manageable approach, BTs address this issue, making them a valuable tool for designing and implementing sophisticated behaviors in autonomous agents.

3.1.1. Definition and Structure

BTs model behaviors as a hierarchical network of tasks arranged as a rooted directed acyclic graph (DAG). Internal nodes are used for control flow and decision making while leaf nodes represent executable tasks[1] or conditions. A common approach is to let nodes in the tree implement an interface that defines the node's specific behavior. This means a node must implement some function that enables the behavior and returns the status of the task upon completion, indicating the success or failure of the behavior. In our work, the interface to be implemented is an *update()*-function.

[1] Comparable to states in a Finite State Machine

Status indicators are often implementation-defined and tailored to the needs of the specific domain. However, three commonly found status indicators are:

1. *Success* - The node has finished its task successfully.
2. *Running* - The node has not yet finished its task.
3. *Failure* - The node could not complete its task.

A base set of internal and external nodes have been defined in various previous works such as [29, p. 334–351] and [36, 24]. These nodes provide the fundamental architecture for BT construction:

1. **Action Nodes** (or Task Nodes) are leaf nodes executing some behavior. Their update functionality is typically used for the actuation of the agent, such as moving toward a point or picking up an object. Returns *Success* or *Failure* upon task completion and *Running* otherwise.
2. **Composites** are a type of container node. These nodes are non-leaf nodes and hold 1 to N child nodes. Composite nodes are further defined by their internal execution policy and typically return the status of one or all child nodes.
3. **Conditions** are a restricted type of Action Node used for querying data and providing decision-making information to their parent node. These nodes are also referred to as *Guards* as they are commonly used before action nodes to preemptively ensure the actuation nodes' behavior can be executed successfully. These nodes typically only return *Success* or *Failure*.
4. **Decorators** is a composite node that holds one child node and defines a pre-condition or execution policy of its child node. A common example is the repeater node[2] and the inverter node[3].

Composite nodes are the most commonly used type of internal node and define the order of tasks and actions ranked by importance in a left-to-right fashion. The three most common subtypes of Composite Nodes are Selectors, Sequences, and Fallbacks. Other extensions exist but are commonly seen in domain-specific use cases.

1. **Selectors** (also called Fallback nodes) attempt to find the first successful child node by executing the tasks of its children in sequence until a child node returns a *Success* state, upon which the Selector node returns a *Success* state. If all children of the Selector node fail to execute their task, the selector node will return a *Failure* state. Selector nodes in this thesis are notated as **?**
2. **Sequences** can only be successful if all its child nodes succeed. The selector node returns a *Success* state if all of its child nodes completed their task successfully, and it returns a *Failure* state immediately when a child node is unsuccessful. The Selector node is notated in this thesis using an arrow: $\longrightarrow$

[2]A repeater ticks its child node more than once before returning the node status to its parent.
[3]Inverters invert the status of a node, such that success will become a failure and vice versa.

3. **Parallel** nodes execute all child tasks simultaneously. A success policy of the parallel node defines how many child tasks are needed at least or most to succeed or fail for the parallel node to return success or failure to its parent. Most literature uses the ⇒ symbol to indicate parallel nodes.[4]

Figure 3.1 presents a class diagram outlining a possible implementation of Task, Composite, and Action nodes in C++. User-defined nodes are separated into a package different from the core implementation. These nodes implement the base class's virtual interface and define the node's behavior by overriding the *update* function.

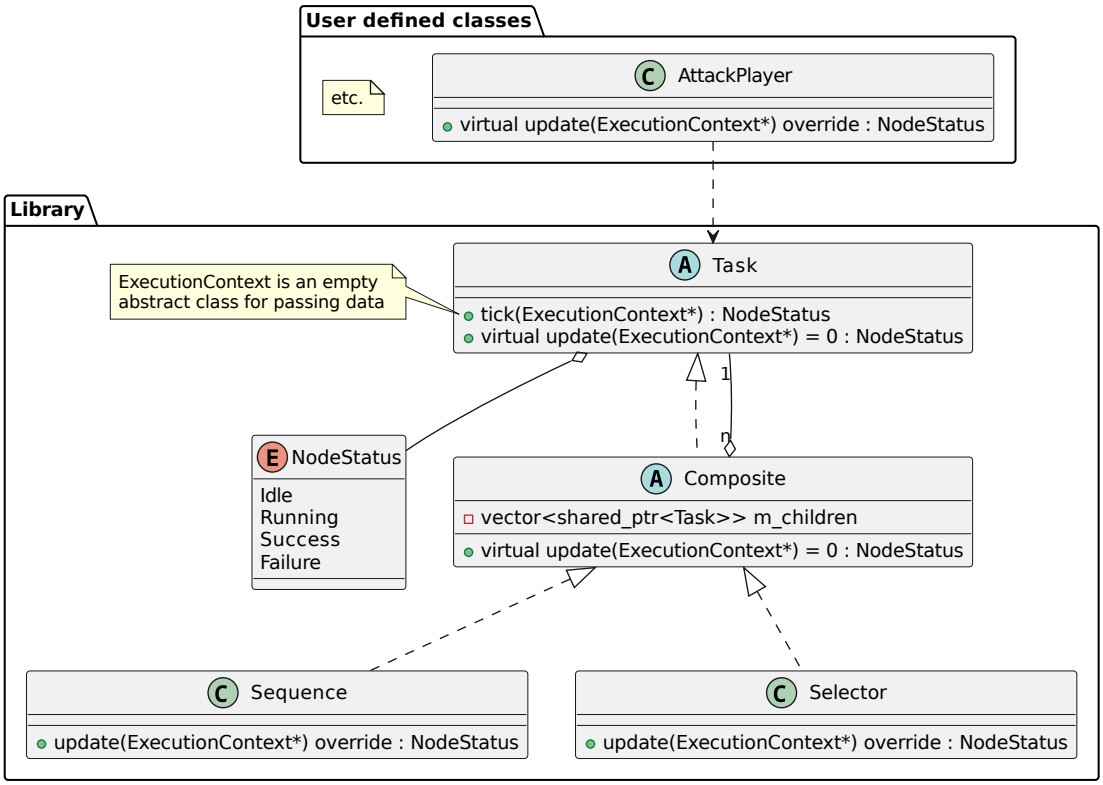


Figure 3.1.: Class diagram of internal and external nodes

The parameter passed to the function provides an interface with the autonomous agent and facilitates access to relevant application-wide data. This is useful when implementing task nodes that direct agent movement in scenarios that require framerate-independent movement speed calculations.

Notably, this implementation approach renders nodes stateless, enabling the reuse of behavior tree instances across multiple agents and contributing to efficiency and flexibility.

[4]The exact policy is implementation-defined. It should also be noted that "parallel" in the context of this node does not necessarily imply the asynchronous execution of its child tasks.

```
for i from 1 to N do
    childStatus ← Tick(child(i));
    if childStatus is running then
        return running;
    else if childStatus is success then
        return success;
return failure;
```

Algorithm 1: Pseudocode of a selector node with N child nodes

```
for i from 1 to N do
    childStatus ← Tick(child(i));
    if childStatus is running then
        return running;
    else if childStatus is failure then
        return failure;
return success;
```

Algorithm 2: Pseudocode of a sequence node with N child nodes

3.1.2. Execution

The standard execution model of a behavior tree follows a well-defined process characterized by the propagation of discrete updates called *tick* signal from the tree's roots to its leaves in a depth-first traversal. The term "tick", is conventional in the field, deriving from real-time systems and game loops where actions are updated every "tick", of the game clock. In the context of BTs, each node's internal behavior or logic is encapsulated in this "tick" function. The tick signal initiates the execution of each node's associated behavior.

When a node is "ticked", it executes its associated behavior, returning the result to its parent node in the tree's hierarchical structure. This result-propagation forms a recursive chain, ensuring the systematic execution of behaviors throughout the tree.

Composite nodes, or control structures, maintain the overall execution flow. When a composite node is ticked, it disseminates the tick signal to its child nodes. This downward propagation guarantees that child nodes are sequentially visited and executed based on the traversal order dictated by the BTs structure.

Once the tick signal returns to the root node, it marks the completion of the current task step, signaling the successful execution of the associated behavior. The root node must be ticked again to move to the subsequent behavior phase. The frequency at which the root node is ticked determines the decision-making and behavior execution rate of the autonomous agent.

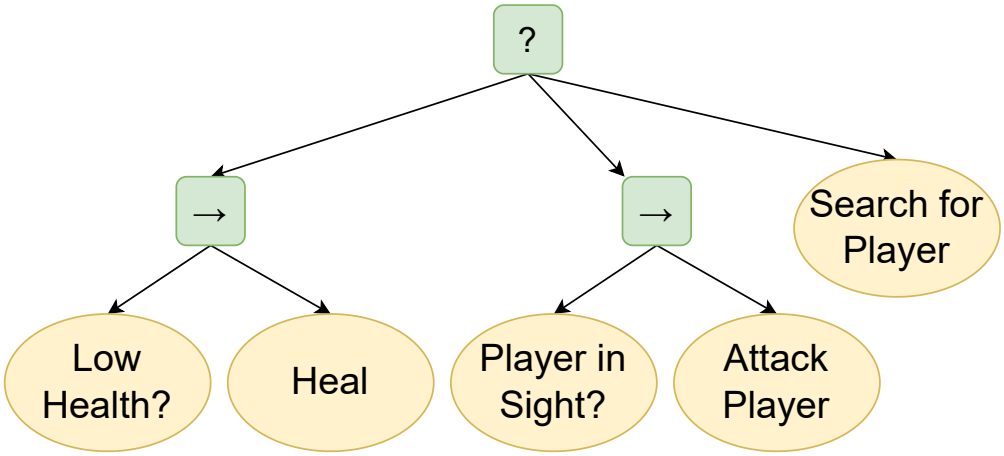


Figure 3.2.: Simple BT showing one selector-, two sequence-, two condition- and three action nodes

Figure 3.2 provides an illustrative example of a simple BT, modeling a NPCs patrolling behavior in a video game. The behavior tree's structure prioritizes the defensive "Heal" task over the offensive "Attack Player" task when the NPC's health is low, even if the player is within sight.

3.1.3. Mathematical Definition

A thorough formal mathematical definition of BTs was described by Colledanchise et al. in [7]. First, a Behavior Tree can be defined as a tuple:

$$\mathcal{T} = \{f_i, r_i, \Delta t\} \tag{3.1}$$

where $i \in \mathbb{N}$ represents the tree's index, $f_i : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ a node's behavior or functionality, r_i is a node's return status and Δt is the time step. The execution model can be defined as:

$$x_{k+t}(t_{k+1}) = f_i(x_k(t_k)) \tag{3.2}$$

$$t_{k+1} = t_k + \Delta t \tag{3.3}$$

under the stipulation that the execution is done using discrete time steps. Furthermore this formulation, a Behavior Tree is composed of three regions: A *Running*-, *Success*- and *Failure*-Region, which is defined as:

$$R_i = \{x : r_i(x) = R\} \tag{3.4}$$

$$S_i = \{x : r_i(x) = S\} \tag{3.5}$$

$$F_i = \{x : r_i(x) = F\} \tag{3.6}$$

Finally, nodes such as Conditions can be then defined as $\mathcal{T}, R_i = \emptyset$, and action nodes are nodes that satisfy Equation 3.2 without calling any subtrees. Using this definition, N behavior trees can be composed into a new behavior tree using a composite operator, such as the sequence or selector operators:

$$\mathcal{T}_0 = selector(\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_N) \tag{3.7}$$

This formal and functional description of Behavior Trees enables verification of behavior tree constructs. This is typically useful in environments with low fault tolerance and little margin for error, such as autonomous driving or uncrewed aerial vehicle operations.

3.1.4. Extensions

Since their introduction, BTs have undergone substantial refinements to increase their efficiency and versatility. Key among these improvements is the evolution from a tick propagation execution model to an event-driven model. The traditional method of updating Behavior Trees involves consistently propagating a tick, which could lead to performance degradation, particularly when long-running tasks or complex structures are involved. Modern implementations, however, have adopted an event-driven approach, leveraging schedulers for these long tasks and initiating full tree traversals only when necessary. This optimizes performance and allows for more precise control over behavior execution. The implementation of BTs in the Unreal Engine provides a noteworthy illustration of this approach.[5]

In addition to these structural improvements, specialized nodes, particularly composite nodes, have been introduced to enhance the functionality of Behavior Trees further. For instance, Memory Nodes, which store the number of child functions called during an iteration, allow for a more streamlined continuation of tasks, reducing the tree traversals required to locate the previously executing behavior. Yet, this added efficiency necessitates additional measures to verify that previously checked condition nodes remain valid - an objective achieved through an event-driven approach. Other novel additions to the model include nodes specifically designed for event monitoring [1], leading to more responsive behaviors.

Furthermore, enhancements such as Random- and Stochastic Composite Nodes have been developed to add complexity to child node visitation. Random Composite Nodes introduce unpredictability by implementing a non-deterministic order of child traversal, adding a more diverse range of potential behaviors. In contrast, Stochastic Composite Nodes adapt over time by recording the success rate of each child node and adjusting their order accordingly, prioritizing successful behaviors.

[5] `https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreesOverview/` Last accessed: 18.06.2023

These advancements in the design of Behavior Trees enhance their functionality and allow for more dynamic and responsive behaviors, with applications extending across various domains.

3.2. Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning and a subset of AI that is particularly suited for problems that can be modeled as a Markov Decision Process (MDP), often formulated as a tuple $\langle S, A, P, R \rangle$. In this tuple, S represents the set of states an autonomous agent can observe, A represents the set of all possible actions that the agent can take, and R represents the rewards, which quantify the desirability of transitioning from one state to another when the agent performs a specific action. The transition probability P denotes the likelihood of transitioning to another state, given the current state and action.

3.2.1. Overview

Q-Learning is a value iteration method and a type of tabular RL which allows an autonomous agent to determine an optimal strategy or policy to achieve its goal in a dynamic environment. It is classified as a type of model-free learning, as it does not require a specific environment model.

The agent determines an optimal policy by maximizing cumulative rewards over time. In Q-Learning, a table, known as a *Q-Table*, is maintained, which stores *Q-values* for each state-action pair of the agent. These Q-values represent the expected total reward for taking action a in a particular state s and following a particular policy.

Initially, all Q-values are set to some constant value. During the learning process, an agent explores the environment by taking actions and receiving feedback through reward values. As the agent receives feedback, the Q-values in the Q-table are updated using the Bellman equation, as shown in Equation 3.8. This equation combines the immediate reward with the discounted future rewards of the next state. [43, p. 131 – 132]

$$Q^{new}(s_t, a_t) \leftarrow \left(1 - \underbrace{\alpha}_{\substack{\text{Learning} \\ \text{Rate}}}\right) \times Q(s_t, a_t) + \alpha \times \left(\underbrace{r(s_t, a_t)}_{\substack{\text{Reward} \\ \text{Value}}} + \underbrace{\gamma}_{\substack{\text{Discount} \\ \text{Rate}}} \times \underbrace{\max Q(s_{t+1}, a_t)}_{\substack{\text{Optimal} \\ \text{Q-Value}}}\right) \tag{3.8}$$

Through an iterative process of exploration and exploitation, the agent gradually refines the Q-values in the Q-table, intending to converge to the optimal values that maximize the cumulative reward. This process is often guided by an exploration strategy such as an ϵ-greedy policy. As the agent accumulates more experience, it becomes increasingly

efficient at selecting actions that lead to higher rewards, ultimately learning an optimal policy for decision-making.

Training of autonomous agents happens over multiple episodes, which can occur online, where the agent actively interacts with a real environment, or offline, during which Q-values are updated using a fixed dataset of experiences, known as a replay buffer or experience replay. The replay buffer helps to break the correlation between experiences and stabilize the learning process by randomly sampling experiences for learning instead of learning from consecutive experiences. Figure 3.3 illustrates the interaction between an agent and its environment during the training process.

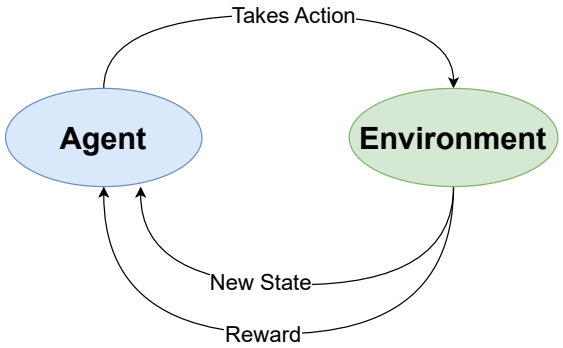


Figure 3.3.: Action-Reward feedback loop during training

3.2.2. Parameters

The Bellman Equation, used to update Q-values, relies on several components: the reward function, current Q-value, and maximum future Q-value, as well as two parameters—learning rate and discount factor—that influence the agent's learning process.

The learning rate α controls the influence of newly acquired information and overrides existing Q-values during the Q-value update process. It determines the balance between exploiting prior knowledge and incorporating new experiences. A higher learning rate allows the agent to quickly adapt to new information, while a lower learning rate results in a more conservative learning process.

The discount rate γ determines the relative importance of future rewards compared to immediate rewards. The agent can balance decision-making between short-term gains and long-term benefits by discounting future rewards. A discount rate of 0 indicates that the agent is myopic and only considers immediate rewards, while a value of 1 implies equal importance is given to immediate and future rewards.

A policy in Q-Learning guides the agent's action selection strategy during the learning process. A greedy policy always picks the action associated with the highest Q-value

for a given state. This strategy, while efficient, might limit the agent's exploration of the environment, potentially causing it to miss more suitable actions in certain states.

An ϵ-greedy policy is often employed to address this exploration-exploitation trade-off. This policy alternates between exploiting known high-value actions and exploring lesser-known actions. With a probability determined by the parameter ϵ, the agent occasionally selects a random action instead of the one with the maximum Q-value. This mechanism allows the agent to explore alternative paths, learn more about the environment, and potentially discover better policies or avoid suboptimal results. Balancing exploration and exploitation is crucial for the learning stability and efficiency of the agent. Algorithm 3 details the training loop using an ϵ-greedy policy.

$state \leftarrow initialState()$;
while *state not terminal* **do**
  $actions = QTable.GetAllActions(state)$;
  if $RandomValue() < \epsilon$ **then**
    $action \leftarrow actions.getRandomAction()$;
  else
    $action \leftarrow actions.getBestAction()$;
  end
  $reward, nextState = agent.doAction(action)$;
  $QTable.updateValue(state, action, nextState, reward)$;
  $state \leftarrow nextState$;
end

Algorithm 3: Algorithm detailing the training loop using an ϵ-greedy policy

3.2.3. Limitations

Despite its effectiveness and simplicity, Q-Learning has limitations, making it challenging to apply to certain problems. One such limitation is its suitability to discrete state spaces, while many real-world scenarios involve continuous state spaces. Utilizing Q-Learning in these environments often requires discretization or approximation techniques, introducing issues such as information loss and the "curse of dimensionality" [11], which refers to the exponential increase of the observed state-space, with each additional state or space of the agent. The state space size also influences both performance and the applicability of tabular methods. An overly large state space can lead to increased runtimes and ineffective training outcomes, as the agent may never visit some states, making it difficult to determine the optimal action in these situations [29]. Chapter 5 details our approach to reducing the state-action space to ensure the Reinforcement Learning phase of our pipeline, described in the next chapter, is manageable in terms of both execution time and memory requirements.

Another limitation stems from the delicate balance between exploration and exploitation in Q-Learning. Too little exploration can lead to suboptimal policies, while excessive exploration can prolong learning times and negatively impact performance. Standard methods to address this trade-off include epsilon-greedy policies, where the rate of exploration decreases over time, or more advanced strategies, such as the Upper Confidence Bound (UCB), which considers both the estimated value and uncertainty of each action. Despite these strategies, achieving the ideal balance between exploration and exploitation remains challenging in Q-Learning.

Lastly, the performance of Q-Learning is often sensitive to the choice of several hyperparameters, such as the learning rate, discount rate, and parameters related to exploration. These hyperparameters must be carefully tuned to achieve optimal performance, and finding the correct values can often require a combination of manual experimentation and optimization techniques, adding to the complexity of implementing Q-Learning solutions.

4. Methodology

In this chapter, we delineate the methodology employed in our research, detailing the architecture and underlying principles of the components we developed for autonomous agent control and the novel algorithm we propose for automatic behavior tree generation. We also set forth our study's primary objectives and the assumptions underlying our approach. Please note that details regarding the training environment for the agents, which forms an integral part of our experimental setup, are covered in Chapter 5.

4.1. Design and Implementation

We first look at the design and implementation of the libraries developed for our research. Primarily, our application is built upon two significant components: a behavior tree implementation and an implementation of the tabular model-free reinforcement learning method, Q-Learning. These components were constructed as part of a library developed in C++ that we use to control our autonomous agents. We also utilize Python scripts to bridge the gap between these two components. These scripts generate behavior trees from knowledge acquired through the Q-Learning process and import them for further evaluation into the simulation detailed in Chapter 5.

4.1.1. Architecture

The architectural design of our application draws from several strategic considerations for its structure and implementation. The system is composed of two packages: the *BT_Library* and the *QL_Library*. While both are designed to drive the behavior of autonomous agents and NPCs, they do so using different methodologies. The *BT_Library* leverages behavior trees for decision-making and actuation, and the *QL_Library* employs Q-Learning to guide agent actions.

We chose C++ as the development language for the *BT Library* and the *QL_Library* given its widespread usage in game development and its performance benefits, particularly in computationally intensive scenarios such as Reinforcement Learning. In contrast, Python scripts were utilized for connecting these components and generating behavior trees due to their flexibility in parsing and generating text.

An important architectural decision was the conscious effort to minimize external dependencies in our libraries. This choice promotes a lean and streamlined codebase, making incorporating these libraries into diverse systems easier. Although the *BT_Library* and the *QL_Library* are standalone entities, they share certain components, enabling seamless

integration and interaction. These shared features are developed to support our novel algorithm for generating behavior trees.

Behavior Tree Library

The BT portion of the library was implemented using the Non-Virtual Interface (NVI) pattern. This design pattern separates the public interface of a class (non-virtual methods) from its implementation details (virtual methods). Specifically, it wraps a purely virtual *update()* function into a non-virtual *tick()* function. The primary advantage of this approach is that it encapsulates the shared functionality like logging in the non-virtual method, reducing the need for users to manually add these details in their implementation of the virtual method. Figure 4.1 depicts the implementation details of the BT library.

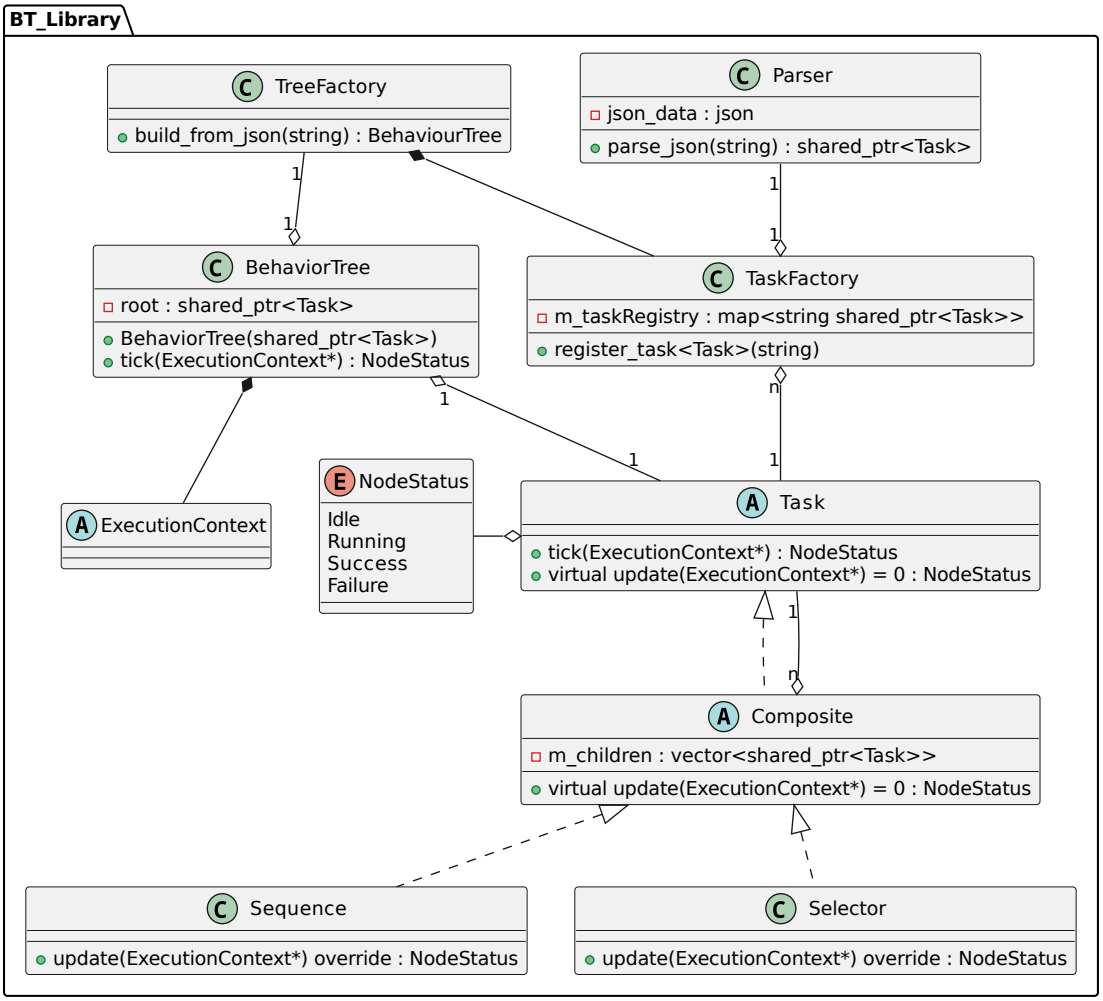


Figure 4.1.: Class Diagram of the Behavior Tree library implementation

The behavior trees are represented using the JSON format. This choice allows the behavior trees to be configured at runtime, offering increased flexibility in the behavior of autonomous agents. Moreover, since JSON inherently follows a hierarchical data format, it maps well to the structure of behavior trees.

The JSON layout follows a specific format where non-leaf nodes contain an array element called *children*, and all nodes have a *name* and *type* specifier. The *name* field aids debugging, while the *type* specifier reflects the internal class name of the node.

```
{
  "behaviortree": {
    "name": "root",
    "type": "Selector",
    "children": [
      {
        "name": "Carry crate home",
        "type": "Sequence",
        "children": [
          {
            "name": "carrying",
            "type": "carryingCondition"
          },
          {
            "name": "move home",
            "type": "moveHomeTask"
          }
        ]
      },
      {
        "name": "Get a crate",
        "type": "Sequence",
        "children": [
          {
            "name": "move to crate",
            "type": "moveTask"
          },
          {
            "name": "pickup",
            "type": "pickupTask"
          }
        ]
      }
    ]
  }
}
```

Listing 4.1: Sample BT as Json

The example above provided through a JSON structure demonstrates a simple behavior of an autonomous agent: carrying crates in its environment to a home position. In the behavior tree represented by this JSON structure, a selector node will seek the first successful child, while a sequence node demands success from all its children.

Consequently, the agent interprets the depicted behavior as: "If I am carrying a crate, I will bring it home. Otherwise, I will move to a crate and pick it up"

A visual representation of this BT can be seen in Figure 4.3. Please note that the visualization style of the tree in this thesis differs from the one used in the previous chapter. The new style organizes the tree from left to right, and from top to bottom. This shift in visual organization does not affect the execution of the behavior tree, as the execution order depends solely on the tree's inherent structure and not on its visual representation. We have adopted this visualization style for the rest of the thesis for its clear depiction of nodes' hierarchy and their relationships.

The library provides Selector and Sequence type nodes. Any additional task nodes must be registered with the *NodeFactory* before the library can use them. When initiating a behavior tree, a *Treefactory*, which includes a JSON parser utilizing an external JSON parsing library[1], is responsible for assembling a behavior tree from an input JSON.

To create a new node, a user inherits either the abstract *Task* base class for leaf nodes or the abstract *Composite* class for non-leaf nodes. An implementing node therefore must:

1. Derive from the public Task-interface and implement the virtual update function.
2. Be registered with the node factory.

The following code example demonstrates how a task can be created, registered, and loaded into the BT library using native C++:

```
struct PickUpTask : public Task
{
    PickUpTask();
    NodeStatus update(Node::ExecutionContext*) override;
};

void RegisterTasks()
{
    Node::TreeFactory().register_task<PickUpTask>("pickup");
}

// ... later

RegisterTasks();

BehaviourTree tree{TreeFactory().build_from_json("myNPC.json")};
```

Listing 4.2: Registration of Tasks and Instantiation of BT

When the *tick()* function of the behavior tree is invoked, the signal is propagated from the root of the BT to its children, each calling their respective *tick()* function. These functions, in turn, call the overridden virtual *update()* function. The *tick()* and *update()*

[1] https://github.com/nlohmann/json

functions take a single parameter named *ExecutionContext*. This parameter provides the node with the necessary data for execution. This approach encourages stateless task execution - a beneficial property in behavior trees as it permits the reuse of nodes and trees across multiple agents. The stateless nature of nodes also aids in reducing memory usage and enhances scalability when dealing with numerous agents. The *Execution Context* is an empty abstract base class that users can use to add additional information. This information can be a reference to the NPC currently traversing the tree.

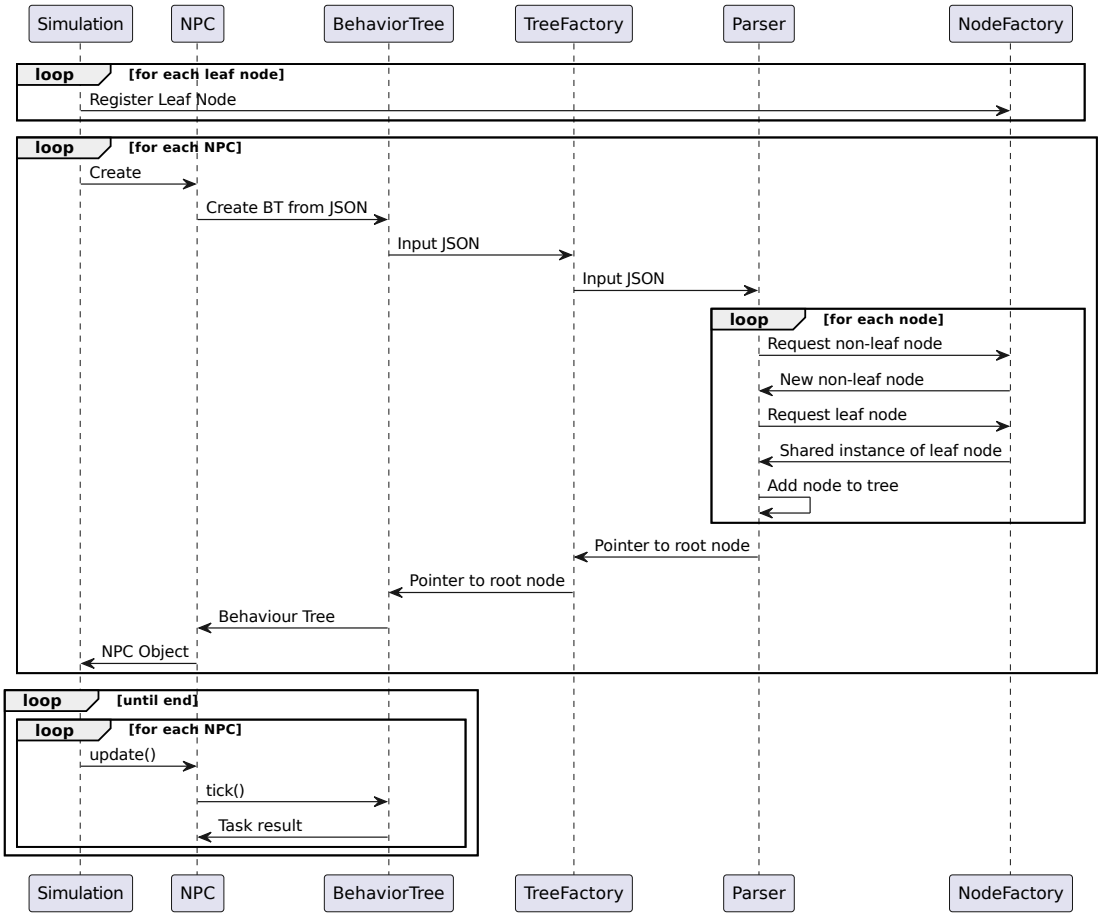


Figure 4.2.: Sequence of behavior creation and execution

The following code sample shows an implementation of the *Carrying Crate Condition* task depicted in Figure 4.3.

```
struct Memory : public ExecutionContext
{
    Memory(NPCObject& o) : npcObject(o) {}
    NPCObject& npcObject;
}

struct CarryingCrate : public Task
{
  CarryingCrate() : Task("carryingCondition") {}
  NodeStatus update(ExecutionContext* ctx) override {
    if (static_cast<Memory*>(ctx).npcObject.isCarrying) {
      return NodeStatus::SUCCESS;
    } else {
      return NodeStatus::FAILURE;
    }
  }
};
```

Listing 4.3: Sample Task in C++ native code

Figure 4.2 depicts a sequence diagram visualizing the steps involved in a simulation for first creating N Non-Player Characters (NPCs) from the JSON of a behavior tree. After creation, each NPC is updated in succession by invoking the *tick()* function of the NPC's behavior tree.

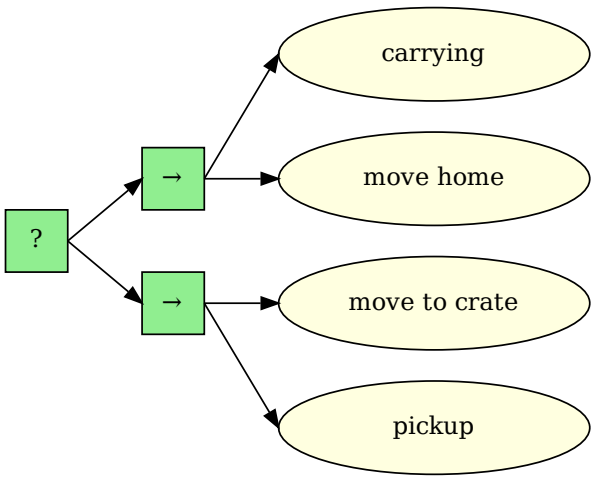


Figure 4.3.: The behavior tree shown in Listing 4.1 visualized as tree

Q-Learning Library

The reinforcement learning portion of the library is responsible for agent decision-making and actuation, like its BT counterpart. Additionally, it provides further functionality to allow offline and online learning of agent behaviors to improve their performance over time. It utilizes the Q-Learning approach by creating tuples of agent states and tasks, leveraging the Tasks stored in the node factory. Figure 4.4 shows the class diagram of the Q-Learning library, as well as its dependency on the BT library.

Users of the Q-Learning portion of the library can configure the learning- and discount rate, the ϵ-greedy strategy, and a reward table. The reward table is configured by calling the *setReward()*-function for each state-action tuple that a reward should be applied to[2]. To add or subtract a bonus to and from a reward value, the *learn()*-function responsible for updating the Q-Table takes an optional bonus value applied once when the function is called. The following snippet from the Q-Learning library shows the implementation of the Bellman equation for updating the Q-Table:

```
template<Hashable trait>
void q_policy<trait>::learn(trait old_trait_, trait new_trait,
                            std::shared_ptr<Task> task, double bonus)
{
    double reward = getReward(old_trait_, task);
    double Q = getQValue(old_trait_, task);
    double maxQ = getMaxQValue(new_trait);
    Q = (1 - alpha) \times Q + alpha \times (reward + bonus + gamma \
    times maxQ);
    storeQValue(old_trait_, task, Q);
}
```

Listing 4.4: Q-Learning functionality to update Q-Value

The parameters of the *learn()*-function are defined as follows

1. The previous state of the agent before taking an action.
2. The new state of the agent after taking an action.
3. The action the agent took.
4. The optional bonus to be applied.

The template parameter trait represents the agent's state and should be implemented as a separate state class. The state class representing the agent's current state must provide a hashing function to be usable in the Q-Table. For example, during the learning phase of an agent, at each tick, the agent should fetch the task with the highest Q-value using the *getBestTask()*-function, perform the task and call the *learn()*-function detailed above, to update the task's Q-Value.

[2]Note that reward refers encompass both positive and negative values, representing the reinforcement or discouragement of specific actions in the learning process.

The ϵ-greedy is used when calling the *getTask()*-function from the library. When requesting a task from the Q-Table, the policy dictates that the best suitable task will be returned, which is the task with the highest Q-Value for a given state. The ϵ value dictates the chance of selecting a random action from the table. It will also select a random action from the table if the given state has not been visited yet, making all values for a given state-action tuple zero.

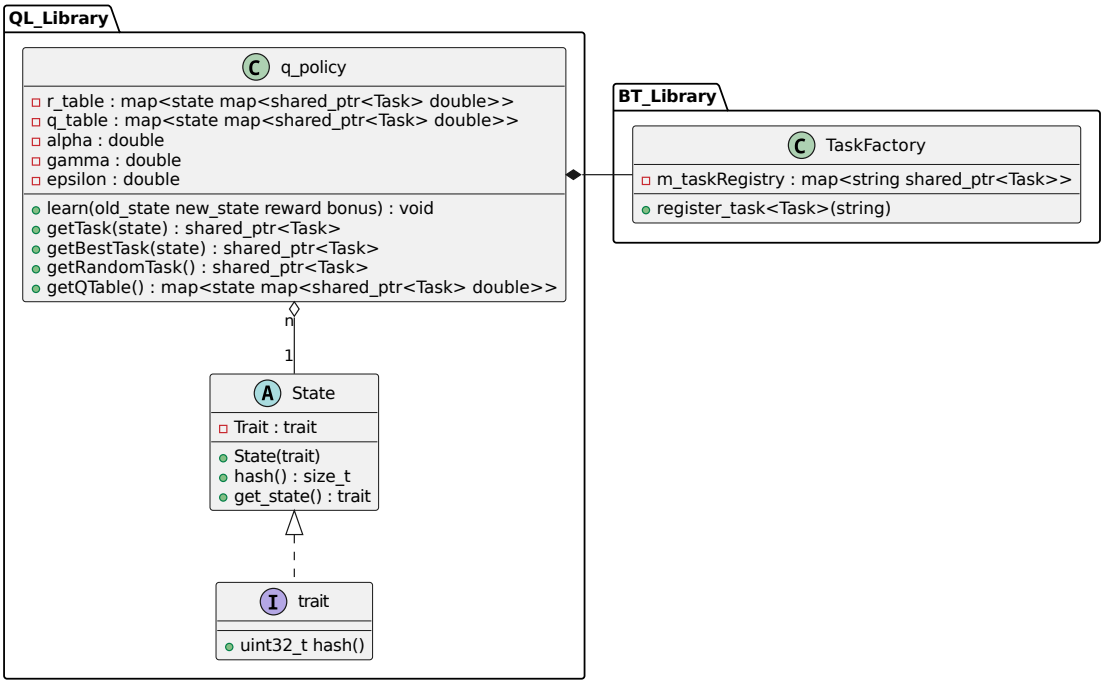


Figure 4.4.: Class Diagram of the Q-Learning library implementation

4.1.2. Behavior Tree Creation Algorithm

We now look at the algorithm we propose that allows us to create a behavior tree from the experiences gained during the episodes of the Q-Learning method. The creation algorithm relies on the concept of atomic units in behavior trees [2]. An atomic unit represents a sequence node that consists of two child nodes: a condition node, functioning as a *Guard*, and an action node.

Initially, we generate the *Max-Q*table by iterating over all states visited during the learning phase, identifying the task with the highest Q-Value. As a result, we construct a list of state-action pairs. This compiled list is subsequently used as the input for the proposed algorithm. The algorithm starts by interpreting the Max-Q table of the learned policy.

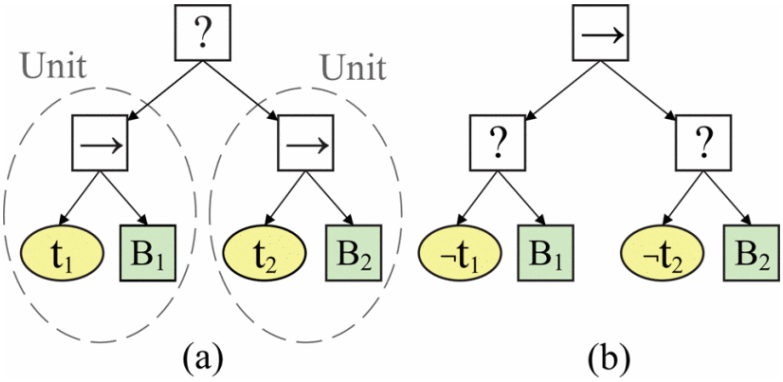


Figure 4.5.: Atomic unit as described in [2]

Contrary to the approach by Banerjee [2] and the subsequent refinements made by Marques [25], our algorithm places more emphasis on actions than states. We first establish the significance of all actions executed, with the frequency of occurrence in the Max-Q table determining their importance. It is assumed that an action performed across various states is considered more important than one executed in just a few states.

Next, we determine the state that was most prevalent for each action. Let us assume an agent's state is represented by the tuple (S_x, S_y), and its actions are represented by Action A_n. The outcome of the learning phase and thus the Max-Q-Table is depicted in Table 4.1. We can thus infer that reaching state S_1 was the decisive factor for undertaking Action A_1. Consequently, the algorithm forms an atomic unit of S_1 and A_1, where a query for S_1 serves as a *Guard* for A_1, and both are placed under a common sequence node. This step is repeated for all actions in the Max-Q table.

State	Action
(S_1, S_2)	A_1
(S_1, S_3)	A_1
(S_1, S_4)	A_1
(S_3, S_5)	A_2
(S_2, S_5)	A_2
(S_1, S_5)	A_3

Table 4.1.: Example of Max-Q table

The algorithm may have formed multiple atomic units with common guards at this stage. As our implementation does not rely on the concurrent execution of nodes, a condition node cannot evaluate differently at different stages of a single tick's traversal. Thus, the next step combines atomic units with shared guards into subtrees under a

selector node. Here, the atomic node with higher importance is placed first, and the atomic unit of lesser importance is added as a subsequent child. The *Guard* of the less significant atomic unit is replaced with its next most common sub-state to avoid re-evaluating an already assessed state in the same tick.

As a last step, the algorithm places the action that was seen the least in the list of ranked actions is used as a fallback node with no pre-condition[3]. This type of unconditional fallback behavior was suggested by previous works that combined reinforcement learning and behavior trees [34, 11]. Using the example Max-Q table, the algorithm chooses A_3 as its unconditional fallback behavior. The BT created from these steps using the example Max-Q table is shown in Figure 4.6

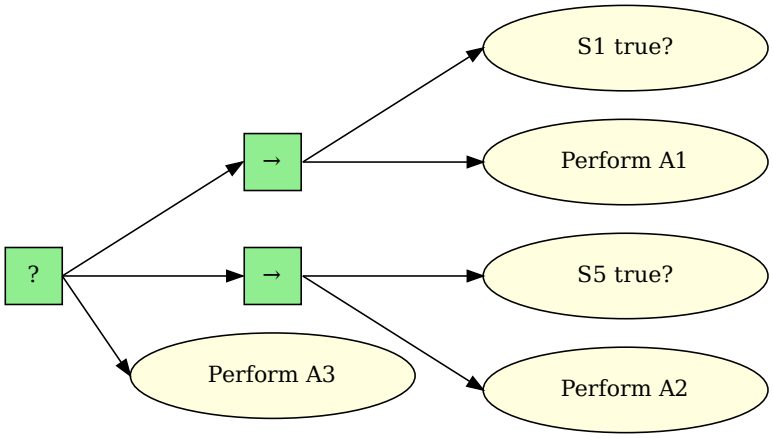


Figure 4.6.: Algorithm output using the sample Max-Q table

Our algorithm allows for creating small and compact behavior trees without building large condition nodes or reusing action nodes in multiple locations of the tree. The following snippets show the algorithm implemented in Python using a list of actions ranked by prevalance, and a dictionary of action-state pairs, the latter also ranked by prevalance, as input.

[3]A fallback node, not to be confused with the similarly named Selector-Node, is not a type of node in the BT, but a node placed as the last child of the root node, ensuring the agent acts, even if all other nodes fail

```
def create_behavior_tree(ranked_actions, action_guard_count):
    behavior_tree = {"name": "root", "type": "Selector", "children": []}
    fallback_node = ranked_actions[-1]
    del action_guard_count[fallback_node]
    ranked_actions.remove(fallback_node)
    atomic_units = create_atomic_units(action_guard_count)
    processed_units = merge_guards(atomic_units, action_guard_count)
    for action in ranked_actions:
        if action in processed_units:
            behavior_tree['children'].append(processed_units[action])
    behavior_tree['children'].append(action_node(fallback_node))
    return {"behavior tree: behavior_tree}
```

Listing 4.5: Behavior Tree Generation Algorithm

The *create_ atomic_ units()* function synthesizes atomic units from the collection of ranked actions and their guards:

```
def create_atomic_units(action_guard_count):
    atomic_units = {}
    for action in ranked_actions:
        atomic_units.setdefault(guard[0][0], [])
        action_node = action_node(action)
        condition_node = condition_node(guard[0][0])
        atomic_units[guard[0][0]].append(sequence_node([condition_node,
    action_node]))
    return atomic_units
```

Listing 4.6: Generating atomic units

Finally, the *merge_ guards()* function, generates sub-trees from atomic units with common guards:

```
def merge_guards(atomic_units, action_guard_count):
    processed_units = {}
    for key, value in atomic_units.items():
        guard_key = value[0]['children'][1]['name']
        if len(value) > 1:
            sub_units = []
            sub_nodes = sorted(value, key=custom_key)
            action = action_node(sub_nodes[0]['children'][1]['name'])
            first = sequence_node([condition_node(key), action])
            for item in sub_nodes[1:]:
                action_ = action_node(item['children'][1]['name'])
                guard = action_guard_count[action][1][0]
                sequence = sequence_node([condition_node(guard), action])
                sub_units.append(sequence)
            atomic_units[key] = selector_node([first, *sub_units])
        processed_units[guard_key] = atomic_units[key]
    return processed_units
```

Listing 4.7: Merging common guards

It is important to note that while our algorithm is written in Python, the libraries above used for controlling autonomous agents have been implemented in C++. Guard nodes

are generated from sub-states and are not known by the behavior tree library as Task nodes. To reduce manual workloads, the algorithm, while performing the steps explained above, also collects a list of guard nodes required to construct an atomic unit. This list is then used to generate C++ code for each condition node and also generates a function *RegisterGeneratedNodes()* that registers these guards with the Node-Factory component. This is not necessarily part of the implemented algorithm but further attempts to reduce manual workloads to develop these guards.

4.1.3. Pipeline

The behavior tree creation using our proposed algorithm is a three-phase process: the learning phase, the BT creation phase, and the evaluation phase, where the performance of the autonomously created BT is assessed. Figure 4.7 outlines these phases and their sequence.

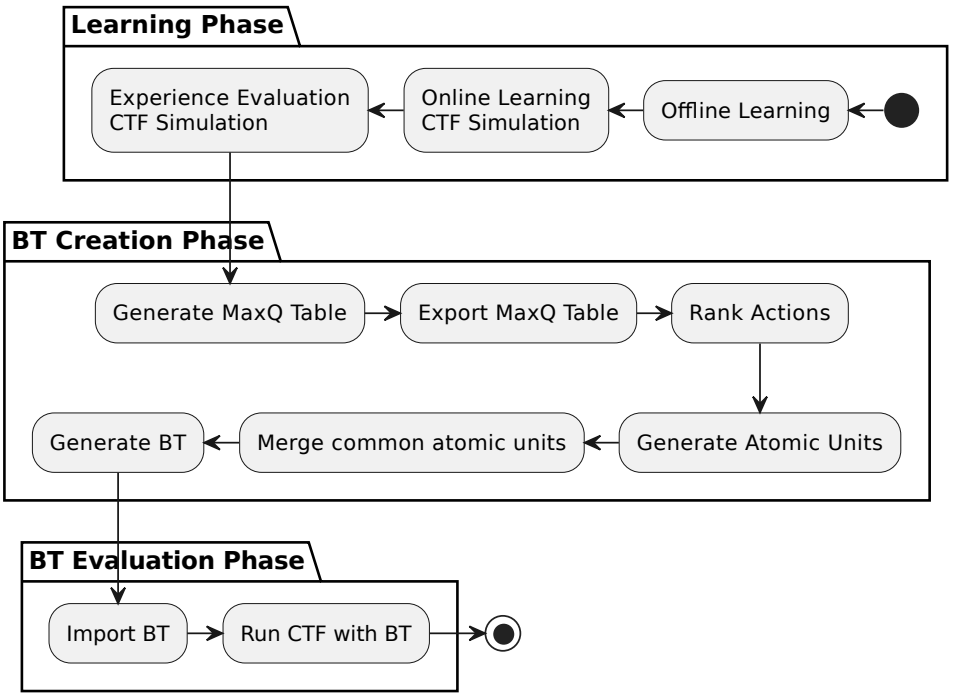


Figure 4.7.: Overview of the BT synthesis pipeline

We have implemented a pipeline linking the phases of behavior tree generation, facilitating an automatic process of BT creation. The pipeline commences with the learning phase, which consists of several steps. Initially, offline and online training of the NPCs takes place, populating the Q-Table. While optional, offline learning can expedite the online learning period by providing the agents with a non-uniformly populated Q-Table. The final step of the learning phase involves evaluating the acquired knowledge from offline and online learning. This is done by re-initializing the simulation with the ϵ-greedy

value set to 0, thereby eliminating random exploration of the Q-Table. During this phase, the Q-Table is treated as read-only, thus preventing updates to Q-Values.

The BT creation phase outputs a JSON-formatted behavior tree for each Max-Q table fed into the algorithm. In this phase, the BT generator generates the behavior tree and outputs condition nodes as task classes, eliminating the need for manual writing of these leaf nodes.

This phase generates new condition classes that were unknown at the beginning of the pipeline. Therefore, the final phase of the pipeline requires recompiling the application to include these new conditions. It is crucial to note that all nodes must be registered with the behavior tree before they can be used. Hence, the *RegisterGeneratedNodes()* must be invoked to register these generated condition nodes with the Node Factory of the library.

To conclude, the pipeline reboots the CTF simulation with the automatically created BTs to evaluate their performance. For a detailed explanation of the Capture-the-Flag (CTF) simulation, refer to Chapter 5.

4.2. Objectives and Assumptions

Like other Genetic Programming (GP) methods and Banerjee's proposed algorithm, our algorithm allows us to autonomously generate a behavior tree without relying on a behavior tree input. Other RL based ways to synthesize BTs often depend on the assumption that a task may not fail after being invoked [2, 25]. Our approach ignores this assumption and, as such, allows us to omit to store the result of a task as a separate state in the Q-table. The proposed approach will enable us to generate small and compact behavior trees while reducing possible states. Chapter 5 details the exact implementation of state configuration for an autonomous agent and shows that our algorithm can be further applied to multi-agent settings to create several cooperatively acting BTs.

4.2.1. Objectives

The algorithm we propose for automatic behavior tree creation aims to reduce the complexity of manual BT creation by leveraging a model-free reinforcement learning approach. Inversely, the automatically created behavior trees enable a visual representation of experiences gained during the learning period of such an approach and grant a better understanding of the acquired knowledge by an autonomous agent.

Autonomous Behavior Tree Generation

The primary goal of this thesis is to explore the feasibility of autonomously generating behavior trees without relying on pre-defined input trees while considering the limitations of traditional approaches. Traditional methods for behavior tree construction include genetic programming, which does not require manual intervention but often generates many suboptimal trees before finding a satisfactory one. Conversely, approaches that

utilize reinforcement learning techniques without manual intervention typically focus on improving existing behavior trees rather than generating them from scratch. To address these limitations, the proposed algorithm leverages model-free reinforcement learning techniques to automatically generate behavior trees based on the experiences gained during the learning phase of an autonomous agent. By doing so, we aim to reduce the complexity and manual effort involved in behavior tree creation, providing a more efficient and scalable approach.

Cooperative Behaviors in Multi-Agent Settings

The application of behavior trees extends beyond single-agent scenarios, as many video games involve multiple non-player characters (NPCs) that require cooperative behaviors. To address this, we extend our algorithm to a multi-agent setting, where each agent operates in an environment that necessitates collaborative behavior. By training agents in a competitive setting without relying on opponents with pre-defined strategies, we aim to create behavior trees that exhibit effective cooperative behaviors. This objective aims to enhance the gameplay experience by avoiding repetitive behaviors and fostering dynamic and interactive interactions between NPCs.

Understanding Acquired Knowledge

In addition to the autonomous creation of behavior trees, we aim to provide a visual representation of the experiences acquired by an autonomous agent during the learning process. By analyzing and interpreting the behavior trees generated by our algorithm, we can gain insights into the knowledge and strategies learned by the agent. This objective contributes to a deeper understanding of the acquired knowledge, enabling designers and developers to evaluate and refine the behavior trees based on the agent's performance and behavior.

4.2.2. Assumptions

Our research is grounded in several assumptions that shape the design and implementation of our algorithm. These assumptions guide our decision-making process and provide a basis for evaluating our approach.

Firstly, our algorithm assumes that a task can fail after being invoked, deviating from other reinforcement learning-based approaches that rely on this assumption [2][25]. By disregarding this assumption, our approach allows us to omit to store the result of a task as a separate state in the Q-table. Furthermore, unlike previous methods, we do not reuse action nodes in multiple locations within the behavior tree. This deliberate avoidance of action node reuse aims to enhance the clarity and maintainability of the generated behavior trees. Furthermore, our algorithm assumes that agents in a multi-agent setting

interact with the environment and each other. This assumption is essential for capturing the dynamics of cooperative behaviors and facilitating effective collaboration among the agents. By extending the learning phase for each agent, such that an agent receives a new state every time a different agent acts, we account for the interactions and dependencies among the agents, leading to more accurate and context-aware behavior tree generation.

However, it is important to note that the proposed algorithm does not guarantee safe behavior. While the algorithm aims to create effective behavior trees, it does not explicitly incorporate safety constraints during tree generation. Safety considerations may include avoiding actions that could lead to undesirable consequences and validating a BTs layout according to domain rules.

5. Experimental Setup

This chapter provides a detailed description of the custom simulation environment developed to test the effectiveness of our behavior tree (BT) creation algorithm. The environment represents a competitive game where two teams, composed of two autonomous agents, strive to outperform each other. Using the libraries and the pipeline introduced in Chapter 4, this simulation primarily evaluates our algorithm's performance in training autonomous agents to function as non-playable characters (NPCs) within the game.

Further, this chapter elaborates on the experimental arrangements deployed for the autonomous and automated acquisition of BTs. It also presents the criteria used to evaluate the quality of the BTs generated by our algorithm.

5.1. A Capture-The-Flag Simulation

Capture-the-Flag (CTF) is a common team-versus-team game format characterized by competitive gameplay dynamics, where teams play for victory by outscoring their opponents. In this game, each team possesses a base and an initial flag, aiming to accumulate points by successfully infiltrating the opponent's base, capturing their flag, and safely returning it to their base. Concurrently, players can engage in combat, strategically hindering the opposing team's progress or temporarily reducing their team's numerical strength.

"CTF" encompasses various rule sets, introducing distinctive constraints and gameplay dynamics. For instance, a commonly employed constraint entails that a player can only earn a point if their team's flag remains in their possession, fostering attack-and-defend behaviors and a cooperative approach among team members. Given the cooperative and strategic nature of this game type, it was selected as an experimental evaluation framework to assess the effectiveness of our algorithm.

By employing CTF as the experimental context, we aim to evaluate the performance of our algorithm under the diverse constraints and dynamics inherent in this game format. The experimental evaluation involves assessing the algorithm's ability to generate effective Behavior Trees (BTs) that exhibit adaptive and cooperative behaviors in pursuit of victory. This choice of experimental framework offers valuable insights into the algorithm's capacity to address complex decision-making challenges in dynamic, team-oriented environments. Consequently, the selected CTF setting serves as a suitable and comprehensive domain for testing and evaluating the efficacy of our algorithm's BT generation capabilities.

5.1.1. Architecture

We now turn our attention to the architecture and design decisions of the application developed to evaluate the effectiveness of the proposed algorithm. The program has been developed as a Command Line Based application that instantiates several CTF simulations sequentially, based on the parameters passed upon program start. Each CTF game is a headless instance of a game. We decided against real-time visualization of the games to allow faster execution of the training and evaluation phases.

Figure A.1 in the appendix depicts the classes relevant to the experimental setup of our work and how these classes integrate the behavior tree and Q-Learning implementations detailed in the previous chapter. The package *Application* house the *Application* class that acts as a manager class during the application's lifetime. It also acts as a managing class for *Simulation*-Objects, handles configuration values for each game instance, and in the case of the learning phase, also initiates the Offline-Learning. It is also responsible for transferring Q-Tables from NPCs between game instances. The *Application* class can be seen as a framework for enabling the automatic creation of behavior trees.

The *Simulation* class represents a single CTF game instance. It is responsible for instantiating all objects required for successful game execution, upholds business rules, and provides the main feedback loop for autonomous agents driven by a reinforcement learning policy. In addition, the package *Application.Simulation* contains the following classes, which are coupled to the *Simulation* class:

- *Team* - Container storing a reference to a team's base and players.
- *Base* - Container storing flag information.
- *NPCGameObject* - Representation of one player in the game.
- *BehaviorComponent* - Controlling a player's policy in the game by calling the *update()*-function of the *AIController* class.
- *AIController* - Virtual class stored by the BehaviorComponent.
- *RL-/BT-/StaticController* - Representation of the actual policy defining the player's behavior.
- *Map* - Map-relevant information such as the location of entities in the game and provides distance calculation and pathfinding functionality.

5.1.2. Game Rules

The experimental setup employs simple rules to define win conditions and game mechanics. As previously stated, tabular reinforcement learning approaches encounter performance degradation, and learning becomes inefficient as the number of state-action tuples increases.

Therefore we deliberately implemented a simple ruleset, omitting intricate complexities. The following are the rules for the CTF game utilized in this experiment:

- The game is played in a series of ticks, each representing a full round. Within a tick, each player takes one action.
- The game is played on an N × N-sized map in a 2D space composed of tiles that include randomly placed barriers and pickups.
- Two teams participate in the game, each with two players identified by their respective team and player indices.
- Each team has a flag located at its base. The bases are at map coordinates (1, 1) and (N-1, N-1).
- If a player reaches the opponent's base while the opponent's flag is present, the player picks up the flag. The flag moves with the player.
- When players return to their base, their health and ammunition are replenished. If a player has the opponent's flag, their team is awarded one point. The flag is then returned to the opponent's base.
- The game is finished when a pre-configured amount of ticks expire, or a team can achieve a pre-configured score before tick expiration. The team with the higher score wins the game. Otherwise, the game is declared a draw.
- Every player starts with 100 health points. Health points decrease when a player incurs damage, such as when another player attacks them. If a player's health falls to zero, they drop the flag (if they have it), which is then returned to the opponent's base. The player becomes inactive for seven ticks, during which they cannot perform any actions. After this respawn period, the player reappears at their base with full health and ammunition, ready to participate again.

5.1.3. Participants in the Game

Each participant in a game with a health value greater than zero can take one of the following actions during a game tick:

- *FireAtEnemy* - This action enables a player to engage an opponent in combat. To do so, the player must have a line of sight to an opponent within half the map's length ($N/2$) and have sufficient ammunition. The action fails if these conditions aren't met, or there are no active opponents. Upon successful execution, the player's ammunition count decreases by one, and the opponent's health is reduced by a base damage value, further adjusted based on the distance. Refer to Table 5.1 for damage calculation.

- *MoveToEnemyFlag* - This action guides the player one unit closer to the opponent's base, regardless of the flag's presence. The action succeeds once the player arrives at the destination and returns a running status otherwise. The path between the player's current location and the opponent's base is determined using the A* algorithm.
- *MoveToHome* - This action leads the player one unit closer to their base. The action succeeds once the player arrives at their home base and returns a running status otherwise.
- *MoveToHealthPickup* - This action drives the player one unit closer to the nearest health pickup. The action fails if another player retrieves the pickup before the player arrives. Otherwise, it succeeds, restoring the player's health to 100. If the player has not reached the target but moved closer, this action returns a running state.
- *MoveToPowerup* - This action directs the player one unit closer to the nearest powerup pickup. The action fails if another player picks up the power-up before the player arrives. Otherwise, if it succeeds, the player's ammunition is restored to the full value, and a permanent damage bonus is granted for future attacks. If the player has not reached the target but moved closer, this action returns a running state.
- *Idle* - This action involves the player standing still. The player's position does not change when this action is taken. It always succeeds.

Distance d	Base Damage Range
$d > M$	-
$M \leq d \geq M/2$	1 - 5
$M/2 > d \geq M/4$	6 - 12
$d < M/4$	13 - 18

Table 5.1.: Damage calculation based on distance (d) and Max Fire Distance (M)

The behavioral actions that each player can perform are integrated into the node classes of the behavior tree structure. These node classes invoke the related member functions from the player's class, and the return values of these functions signify the state of the task - whether it is successful, failed, or in progress. The illustration does not include these classes to keep the application's class diagram clear and concise.

In each game tick, an updating process is conducted for the teams and players, starting with the instances created first. The order of these updates, while determined by the creation sequence, does not affect the game's eventual outcome.

All player instances can utilize the range of functionalities the game map offers, enabling them to perform pathfinding, compute line-of-sight, and calculate distances between any two tiles.

Each player's actions in the game are driven by the *BehaviorComponent*, while the specific nuances of these actions are defined by the *AIController* class. This abstract base class branches into three classes that handle agent control using different approaches: reinforcement learning, behavior trees, and a static if-else logic structure. The team's policy sets the control approach for its agents - be it reinforcement learning, behavior trees, or the static if-else structure. Within a team, all players adhere to the same policy, maintaining uniformity and avoiding a mix of different control approaches within the same team. One of the key objectives of this experiment's design is to evaluate how well a team performs when it consists solely of agents controlled via reinforcement learning.

Team Reinforcement Learning

The agents of this team utilize the *RLController* component within their *BehaviorComponent*, guiding their in-game actions. Each agent operates with an individual instance of the *QL_Library*, thus owning a unique Q-Table. This structure ensures that, although the same controller guides all agents, they each acquire distinct experiences and develop different knowledge bases. This process facilitates a multi-agent learning environment, with agents learning independently rather than sharing experiences.

Team Static

To provide a comparative measure for the effectiveness of the reinforcement learning and automatically generated behavior tree approach, a second team of Non-Player Characters (NPCs), termed *Team Static*, was developed. The agents in this team follow a static decision-making process controlled by an $if - else$ structure, akin to the predator setup detailed in Dey et al. (2013). The agents are characterized by an aggressive policy prioritizing attacking opponents over scoring through flag capture. It is crucial to note that this team's policy parallels the reward table configuration of *Team Reinforcement Learning*. However, a significant difference lies in the decision-making process - the agents in *Team Static* execute static decision-making and are incapable of adaptive behavior over time. This team thus serves as a control group to gauge the relative efficacy of reinforcement learning and behavior tree approaches.

5.2. Learning Phase

We now detail applying the Q-learning process to the reinforcement learning-based actors in the CTF simulation. Recall that the learning phase is the first step in our pipeline, and the experiences and knowledge accrued by the players during this phase form the foundation for the exported behavior trees.

The learning rate and discount factor, integral to the Q-learning algorithm, have been set to 0.9, indicating a high emphasis on both learned and future rewards. The ϵ-greedy policy, a critical aspect that controls the trade-off between exploration and exploitation, is set at 0.3. These values have been chosen based on the successful implementation of reinforcement learning parameters in the work of Dey et al. [11].

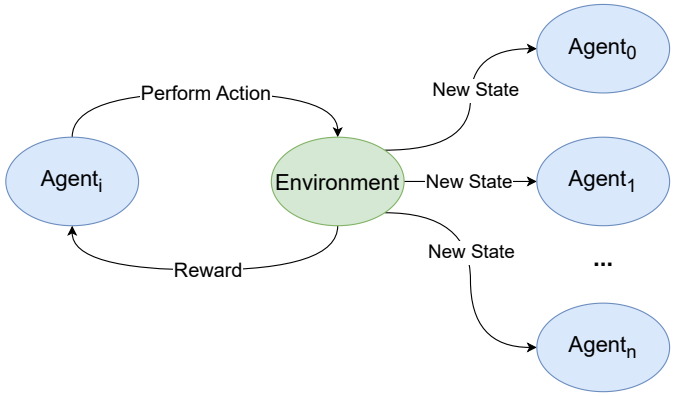


Figure 5.1.: Reward-Action cycle in a multi-agent setting.

Figure 3.3 illustrates the action-reward cycle for a single agent within the reinforcement learning process. However, in our multi-agent CTF simulation, a team comprises multiple agents, each driven by a distinct Q-Learning instance. To accurately reflect this, we extend the depicted learning cycle for each agent to receive a new state every time a different agent acts.

This is crucial due to the interconnected nature of multi-agent environments. Let us assume the agents are indexed by A_n. An action performed by an agent, A_1, might dramatically alter its state (such as health, ammo count, or status of the flag) and, consequently, the state of another agent A_n by directly or indirectly interacting with it. This highlights the necessity for every agent to adapt to the actions of others continuously.

5.2.1. States of an NPC

Each actor in the simulation consists of several attributes that collectively define its current state, such as a tuple of x, y coordinates on the map, the number of health points or ammunition left, and a respawn countdown that indicates how long the NPC must remain motionless, among others. To manage the complexity associated with the size of the state-action space, we employed a discretization approach similar to previous studies [11, 25]. This approach limits the number of states by categorizing the attributes into predefined classes.

- Health
- Ammo count

- Distance to the closest enemy
- Status of the opponent's flag
- Status of their teammate

These attributes are categorized into $None, Low, Medium, High$ values. Health, ammo count, and distance to the opponent are discretized into these categories based on thresholds. Specifically, any value above 66% is considered $High$, whereas a value below 33% is designated as Low. A value of $None$ denotes a state with health or ammo count below 1 or when the agent does not have line-of-sight to an opponent. Table 5.2 provides detailed definitions for the categories associated with the "Flag" and "Teammate" States.

	Flag State	**Teammate State**
None	Player does not have the flag & Player cannot capture the flag	Teammate has health below 1
Low	Player does not have the flag & Player can capture the flag	Teammate does not have the flag & Teammate's last action was not to get flag
Med	Player has the flag	Teammate does not have the flag & Teammate's last action was to get flag
High	-	Teammate has the flag

Table 5.2.: Explanation of flag and teammate states

This discretization yields a theoretical total of $4^5 = 1024$ potential states. However, certain limitations and constraints on these conditions effectively reduce the size of the state-action space.

Firstly, states in which the Health condition is None are excluded, as an agent with zero health cannot perform any actions. This condition eliminates a quarter of the theoretical state space, reducing the number of potential states from 1024 to 768. Next, states with High Flag conditions are also excluded, as this state is undefined in our context. This condition also eliminates a quarter of the remaining state space, reducing the potential states from 768 to 576.

Finally, several combinations of states are deemed impossible due to game logic. These are:

- States where the agent and the teammate possess the flag (Flag condition = Medium, Teammate condition = High). Since there is only one flag in the game, both agents can't have the flag simultaneously.
- States where the agent cannot capture the flag and the teammate does not possess the flag (Flag condition = None, Teammate condition = High). These states are logically inconsistent as they imply that neither agent can take action on the flag.

- States where the agent can capture the flag but the teammate already has the flag (Flag condition = Low, Teammate condition = High). These states are logically inconsistent, as an agent cannot capture the flag if their teammate already holds it.

We further reduce the potential state space by excluding these impossible state combinations. An exhaustive check of these conditions, performed by iterating through each possible state, yields 336 valid states for the NPC within our game simulation.

This state space reduction enables us to create a manageable and realistic Q-Table for the reinforcement learning process, ensuring that our agents learn based on practical and possible scenarios within the game environment.

5.2.2. Reward Structure

The reward structure shapes the learned behavior of RL agents. In the case of our RL agents in a capture-the-flag style game, this structure is pivotal for promoting actions that lead to scoring points, preserving an agent's life, and eliminating opponents.

Actions that align with these goals, such as scoring points, eliminating opponents, or preserving an agent's life, are positively reinforced with a reward of +1. Conversely, actions that potentially jeopardize these goals, for example, moving away from the home base when in possession of the flag, are negatively reinforced with a reward of -1. Particularly detrimental actions to the team's success, such as choosing to idle when possessing the opponent's flag, are strongly discouraged with a reward of -2.

Moreover, a system of bonuses is implemented that further adjusts the reward values based on specific circumstances, adding nuance to the learned behaviors. These bonuses are cumulative and applied in addition to the basic rewards from the reward table, reinforcing or negating the value of actions under specific conditions. The detailed configuration of the reward table and bonus system is discussed in this section.

Reward Table

The rewards given to the agent directly influence the structure of the behavior tree. The reward structure has been configured such that the agents strive to accumulate points and demonstrate self-preservation and aggression when necessary. This should also be reflected in the automatically created BTs. Specifically, the reward table for an agent's Q-Policy is configured such that actions that lead to scoring points, eliminating opponents, or preserving an agent's life are awarded +1. Actions detrimental to these goals are penalized with -1. If an action could potentially undermine the team's prospects (such as the agent opting for the *Idle* action while possessing the opponent's flag), a steeper penalty of -2 is enforced. An abridged depiction of the Q-Table is presented in Table 5.3. Furthermore, a bonus structure is also in place to further reward or punish certain actions in particular instances. The bonuses are cumulative and skew the overall reward value, reinforcing an action's value under a specific circumstance.

These are as follows:

- Scoring a point for the team will give the reward an additional bonus of +1.
- Dropping an opponent's health below one will give an additional bonus of +1.
- Choosing any action that results in a successful state of that tick function will grant a +1 bonus.
- Choosing any action that results in a failure state of that tick function will reduce the bonus by -1.
- Choosing one of the pickup actions grants an additional +1 bonus if a health kit or powerup was found and picked up.

	Health	Ammo	Dist. Enemy	Flag	TeamMate	Reward
MoveToEnemyFlag	x	x	x	L	N/L	+1
MoveToEnemyFlag	x	x	x	x	H	-1
MoveToHome	x	x	x	M	x	+1
MoveToHome	x	x	x	N	x	-1
FireAtEnemy	x	L/M/H	L/M/H	x	x	+1
MoveToHealth	L/M	x	N	N	x	+1
MoveToPowerup	x	N/L	N	N	x	+1
Idle	x	x	N	N	H	+1

Table 5.3.: An abbreviated version of reward table.
$x = any, N = None, L = Low, M = Medium, H = High$

Partial Reward Table

A significant facet of this research endeavor is minimizing the labor-intensive process required for generating credible and effective behaviors embodied in behavior trees. Formulating a behavior tree is an intricate process, and the same complexity extends to determining a detailed reward table. In light of this, we evaluate an automatically created BT, this time circumventing the reward table presented earlier and instead implementing bonuses awarded each tick. We supplement this setup by adding a bonus of +1, provided when the agent opts for the *MoveToEnemyFlag* action while not having the flag, as well as a bonus of +1 when the *MoveToHome* action is selected with the flag in possession. For this experimental setup, the pre-population of the Q-Table during an offline learning phase is bypassed, given that the reward table is vacated, and bonuses are exclusively computed during online learning.

5.2.3. Opponent Selection

We extend the multi-agent setting by replacing "Team Static" with a second "Team Reinforcement Learning" during the online learning period. The objective of this experiment is to evaluate the necessity of using manually crafted adversaries in a competitive setting or if our proposed algorithm can also generate well-performing behavior trees if the preceding RL agents were only able to compete against other RL agents.

The offline learning period instantiates and populates a singular Q-Table to instantiate all four autonomous agents. Each agent operated under its own Q-Policy, drawing from an individual Q-Table. The conclusion of each game saw the successful team's Q-Table duplicated in its entirety, forming the basis for the agents in the succeeding simulation. The same number of games is used as before. The automatically generated BTs compete against "Team Static" under the same constraints as the reinforcement learning agents.

The final phase of the experiment involves the evaluation of the automatically generated behavior trees. These trees, products of the reinforcement learning process, were employed by agents to compete against "Team Static" under identical constraints applied to the reinforcement learning agents.

5.2.4. Training Methods

Two additional trials were conducted within the experimental framework to explore the avoidance of local optima and evaluate different training approaches. The objective was to investigate the impact of varying the knowledge transfer policies during the learning phase of the CTF simulation. The performance of the automatically generated behavior trees was assessed using different policies for transferring knowledge to subsequent simulations.

The ϵ-greedy policy allows for random exploration rather than always selecting the best action and serves as a guiding principle to avoid local optima during the learning process. Expanding on this concept, we apply a similar notion on a global scale by defining policies that determine when knowledge should be transferred to agents in subsequent CTF simulations. The performance of generated BTs using the following knowledge transfer policies is compared.
Knowledge transfer occurs when:

1. The agents win a game or achieve a new high score.

2. The agents win a game or achieve a new high score or with a 33% chance.

The first policy can be seen as a greedy policy, favoring winning strategies, while the second policy resembles the ϵ-greedy policy employed during the online learning phase. It is important to note that both policies limit the number of training episodes, as a fixed number of games are played before behavior tree generation takes place. This investigation

aims to examine the outcomes of applying these policies and determine whether favoring certain strategies can lead to improved results compared to utilizing the maximum number of learning episodes and the impact of these policies on the automatically generated behavior trees.

5.3. Adaptiveness and Performance

By subjecting the automatically generated BTs to various scenarios, we aim to examine their robustness and resilience in navigating the challenges an adversarial team poses. The performance and effectiveness of the reinforcement learning algorithm and the subsequently automatically created behavior trees are evaluated by competing in the CTF simulation against the static team described in this chapter. The performance analysis encompasses various metrics, including wins, losses, ties, and overall success rates, providing insights into the effectiveness and competitiveness of the BTs within this dynamic environment. We further investigate the performance of the automatically created trees under settings previously not seen during the learning phase.

5.3.1. Map Sizes

The learning phase takes place for its entire length under the same constraints. During game development, it is expected that parameters like map sizes and layouts can change. Considering the overhead associated with reinforcement learning algorithms, it is not favorable to regularly restart the learning of an autonomous agent to be able to adapt to new environments. We use smaller and larger map sizes to evaluate the adaptiveness to these changing settings than those used to train the agents. The performance on map sizes not considered during learning provides an insight into the ability to adapt to the automatically created behavior trees.

Through these adjustments, we gain insights into the adaptability of the automatically generated behavior trees in dynamic environments for success in CTF simulations.

5.3.2. Game Length

Similar to the map size, the length of each game is a fixed value during the learning phase. However, our experiment aims to investigate the adaptability of automatically created behavior trees to varying game lengths. In this experiment, we explore two approaches: explicit and implicit modifications.

In the explicit approach, we adjust the tick length, controlling the overall duration of the games. Shortening the tick length increases the pace, requiring faster decision-making, while lengthening it allows for more strategic analysis. This approach examines how behavior trees adapt to different time constraints.

In the implicit approach, we modify the number of points required to win without changing other parameters. Reducing the points requires quick flag capture, potentially demanding more aggression while increasing the points encourages reward behaviors. This approach explores the adaptation of behavior trees to varying game lengths while keeping time constant.

6. Results and Discussion

In this Chapter, we investigate the outcome of the experiments described in Chapter 5, present our findings, and discuss the results of our experiments. The CTF simulation can be run under multiple configurations and allows changing various game parameters. For answering our research questions, we use a default configuration which is depicted in Table 6.1. Unless otherwise stated, this configuration is used for each experiment.

Parameter	Value
Map Size	100×100
Tick Limit	10'000
Maximum number of games	30
Score to win	100

Table 6.1.: Default CTF settings

The parameters were carefully selected to enable extended training periods for the Reinforcement Learning (RL) agents while maintaining the application's overall performance. To assess the efficacy of the autonomously generated behavior trees (BTs), we conducted evaluations using a tenfold increase in the number of games compared to the training set. This larger evaluation set helps mitigate the impact of statistical outliers and provides a more robust assessment of the BTs performance.

6.1. Comparison of Opponent Selection

In this experiment, we look at the outcome and performance of automatically created behavior trees when the previously learning agents competed against other reinforcement learning agents in the learning phase of the pipeline, as opposed to agents that learned by competing against the team with static decision-making. By comparing these scenarios, we gain insights into the effectiveness of the autonomously created behavior trees in dynamic competitive environments.

6.1.1. RL vs Static

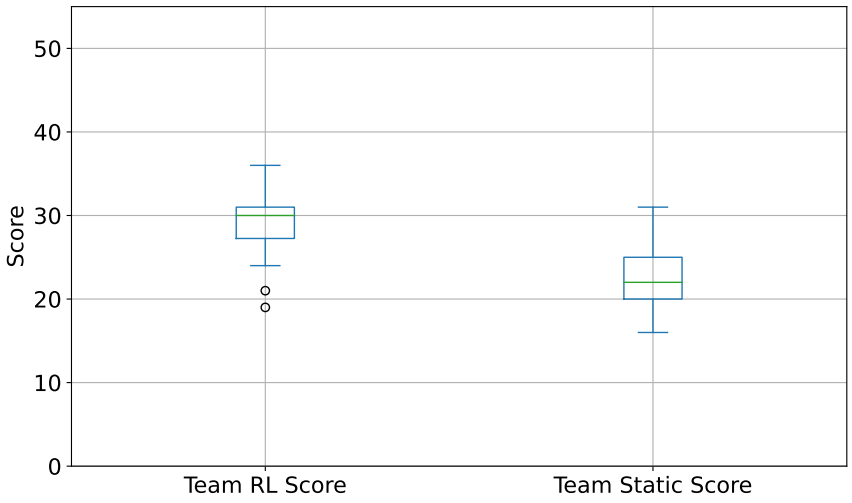
The overall results of the learning (Online Learning) and evaluation phases (Experience and BT Evaluation, respectively) are shown in Table 6.2. The success rate of 93% games won during the evaluation period of the learning phase indicates that the agents have successfully derived a winning policy from the experiences gained in the learning period.

Using the proposed algorithm to export behavior trees for each reinforcement learned agent, we can achieve a success rate of 80% showing promising results in our creation method.

Phase	Wins	Losses	Ties	Total	Success Rate
Online Learning	2	24	4	30	0.06
Experience Evaluation	28	2	0	30	0.93
BT Evaluation (BT vs. Static)	240	39	21	300	0.8

Table 6.2.: Performance of "Team Reinforcement Learning" and the resulting Behavior Tree

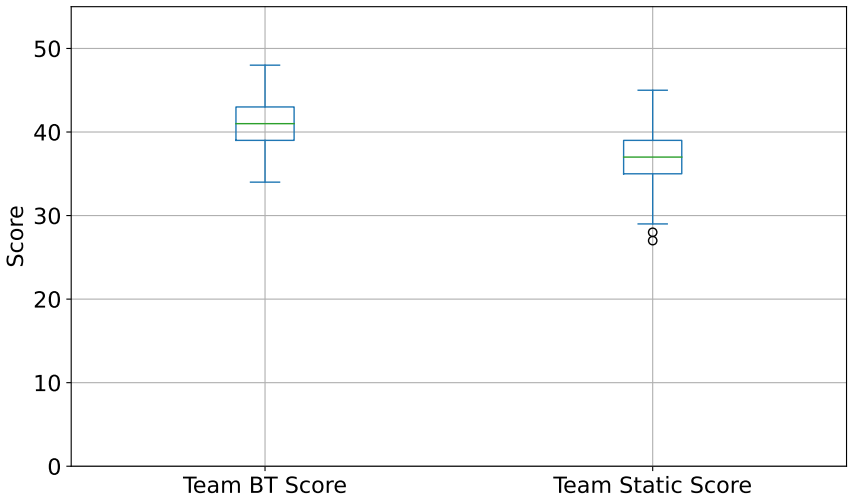
A collection of scores achieved per game after the learning period is depicted in Figure 6.1, while Figure 6.2 depicts the scores during the evaluation period of the generated BT. We compare the game results of these evaluations to gain insight into the effectiveness



	Team RL Score	Team Static Score
count	30	30
mean	29.233	22.600
std	3.748	4.320
min	19	16
50%	30	22
max	36	31

Figure 6.1.: Learning Phase Evaluation (RL vs. Static)

of the automatically created behavior trees. The team using the automatically created Behavior Trees allowed for games with overall higher final scores than the RL team in the evaluation period, with the overall performance being similar, with an equally small spread of points achieved and an overall higher mean score than the static team. The tables in this figure can also be found in the appendix.



	Team BT Score	Team Static Score
count	300	300
mean	40.977	36.757
std	2.354	2.999
min	34	27
50%	41	37
max	48	45

Figure 6.2.: Behavior Tree Evaluation (RL vs. Static)

To get a better understanding of this result, we further turn our attention to Figure 6.3, which depicts the progression of score results during the learning phase. While there is no clear indication that the team learning the game was able to adopt a significantly better strategy to point scoring over time, it is, however, visible that the agents determined a better policy of defense, as the point difference between the two teams decreased drastically when comparing the first episode of learning with the last.

The agents' increased understanding of a winning policy can be further seen by comparing the result progression of individual games. Figure 6.4 plots the progression of points per team for the first episode of learning. Figure 6.5 shows the point progression of the last episode while learning. Comparing the respective performances of each team during these episodes it is visible, that the team utilizing Q-Learning to find an optimal strategy, struggled for several hundred ticks in their first game to score a point and allowed their

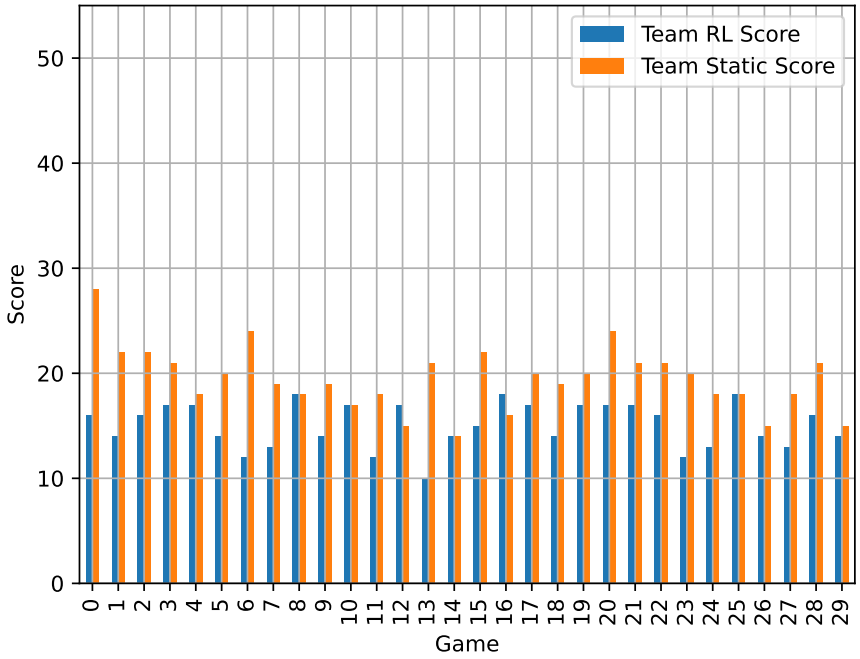


Figure 6.3.: Result progression of all games in the learning phase.

opponents to act aggressively. The point progression of the last episode shows that the RL agents have found a working strategy to score points faster than their opponents while also keeping the opponents from scoring themselves for a long time. However, the random exploration policy of the agent's Q-Learning method, and the performance of the static team, ultimately still led to a loss during the learning phase. The results obtained from this experiment show that our algorithm can be applied to multi-agent settings. Figure 6.7 and Figure A.3 show the automatically created Behavior Trees used in this experiment. The BTs possess inherently different structures from each other, but both model strategies lead to winning the game.

Finally, we turn our attention to Figure 6.6, which shows the final score of each game during evaluation of the BT. We see, that the automatically created BTs were able to outperform their opponents in a majority of cases, often by a significant margin.

6.1.2. RL vs RL

We assessed the performance of autonomous agents and behavior trees during the online training phase, wherein the team composed of reinforcement learning agents was not pitted against opponents with static decision-making capabilities. Instead, both teams consisted of autonomous RL agents. The performance of the final Q-Table and the

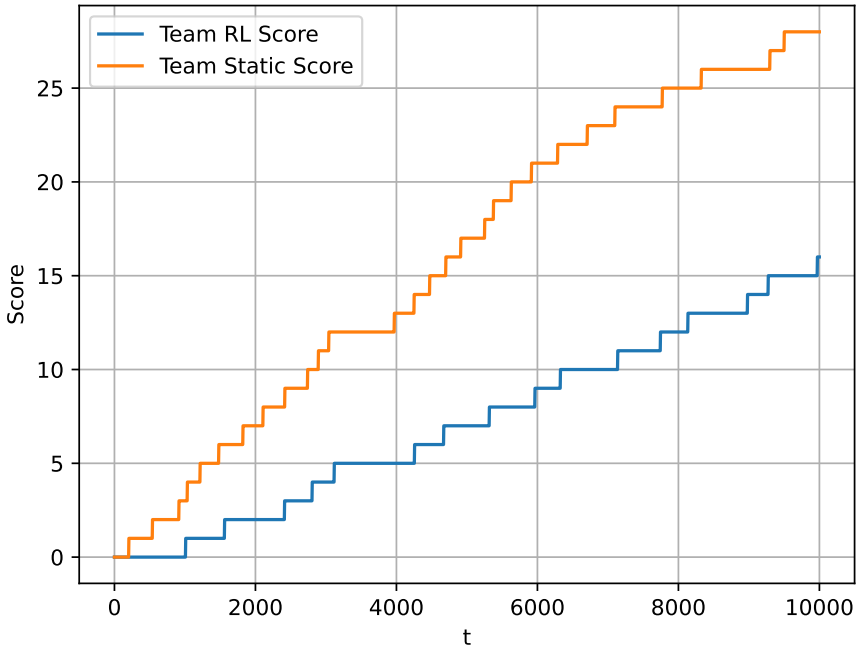


Figure 6.4.: Learning phase against static opponent first episode.

resulting behavior tree, generated automatically from this RL approach, is illustrated in Figure 6.8, which depicts the performance of the agents after learning has concluded.

Phase	Wins	Losses	Ties	Total	Success Rate
RL vs Static Evaluation	26	2	2	30	0.86
BT Evaluation (BT vs. Static)	182	94	24	300	0.61

Table 6.3.: Performance of the "RL vs RL" learning method and the resulting Behavior Tree

We see performances comparable to the previous phase, with the RL agents winning a majority of the games played, albeit less consistent. The performance of the generated behavior trees is presented in Figure 6.9. While the generated trees seemingly were more successful at scoring against the static opponents, they also allowed their opponents to score more points than the players of the previous phase, leading to narrower results and ultimately a slightly worse performance than before. The tables have also been included in the appendix.

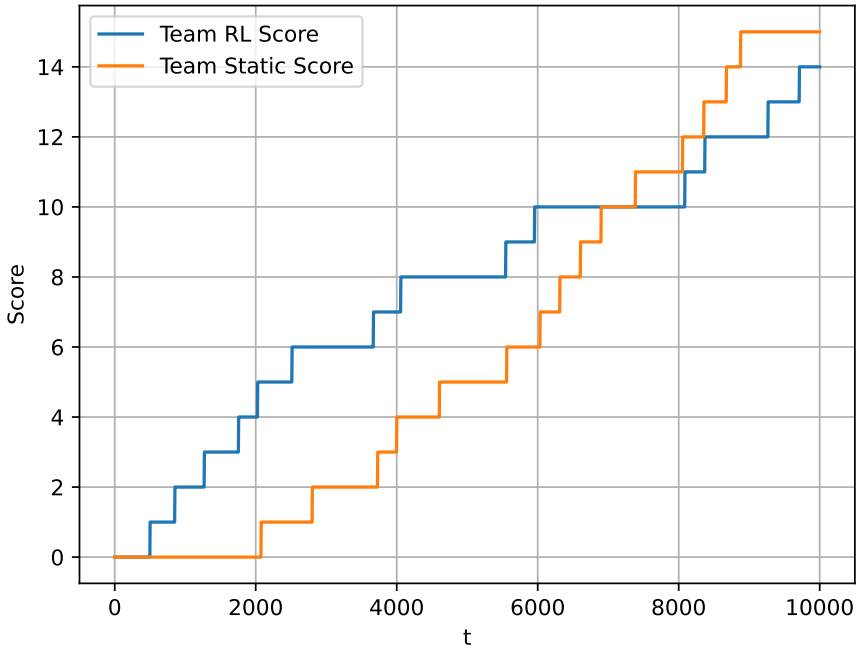


Figure 6.5.: Learning phase against static opponents; Last episode.

The outcomes of this experiment reveal promising results, indicating that the fully autonomous creation approach yields superior results compared to the methodology employed in the previous section. The result of all games played in this evaluation phase is presented in Figure 6.10. The results show that the automatically created behavior trees consistently won against their opponents, however with closer final results than before. An example of an automatically generated behavior tree of this experiment is presented in Figure 6.11. The less aggressive behavior that leads to higher-scoring opponents can be seen in this tree. The *FireAtEnemy* action is only selected when an opponent is close to the agent. As the static team has been configured to attack their opponents at the earliest convenience, even at the expense of lower damage output, it allowed the static team to score more points overall.

Upon analyzing the overall performance of the resulting behavior tree, we observed similar mean, minimum, and maximum values for the game results when compared to the preceding reinforcement learning team. However, it is noteworthy that the respective opponents of each team exhibited significantly different performances. These findings suggest that the generated behavior trees exhibited a more offensive inclination than defensive tendencies. While they achieved comparable scores to their counterparts, they seemed to encounter difficulties effectively defending their flag.

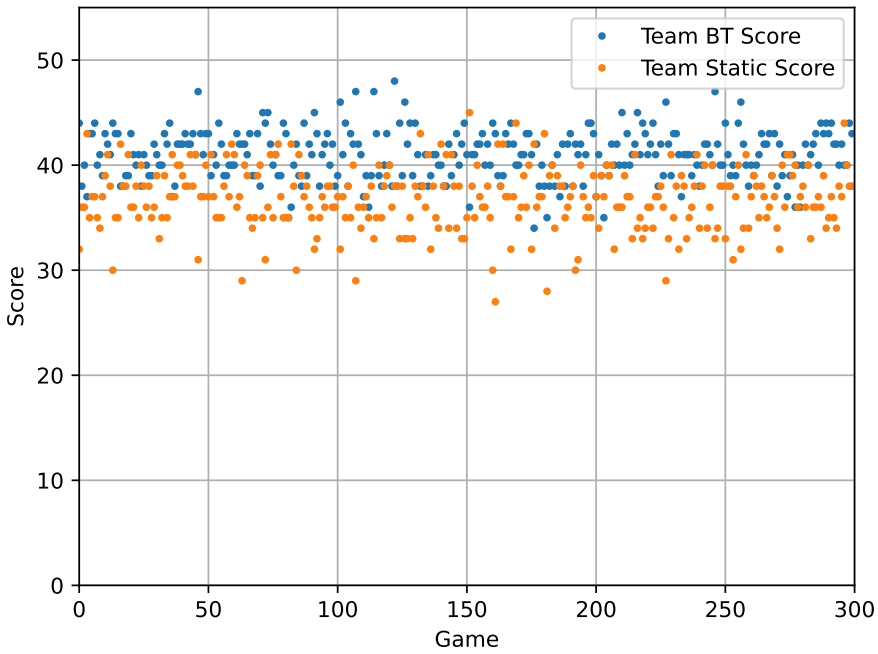


Figure 6.6.: Score Progression over all games during Behavior Tree evaluation (static opponents).

6.2. Capability to Adapt

The adaptability of the autonomously generated behavior trees was evaluated by assessing their performance in games with different map sizes and game lengths (ticks per game). The results provide insights into the effectiveness of these behavior trees in various gaming scenarios. We compare the results in this section with the performance displayed in Table 6.2.

6.2.1. Map Sizes

Changing the size of a map affects the game in multiple ways: Smaller map sizes lead to more combat situations, while larger maps lead to harsher punishments when losing the opponent's flag due to longer travel times. This section examines the adaptability of the autonomously generated behavior trees on maps of different sizes. Table 6.4 depicts the outcome of games played on maps smaller and bigger than the one used to obtain training data.

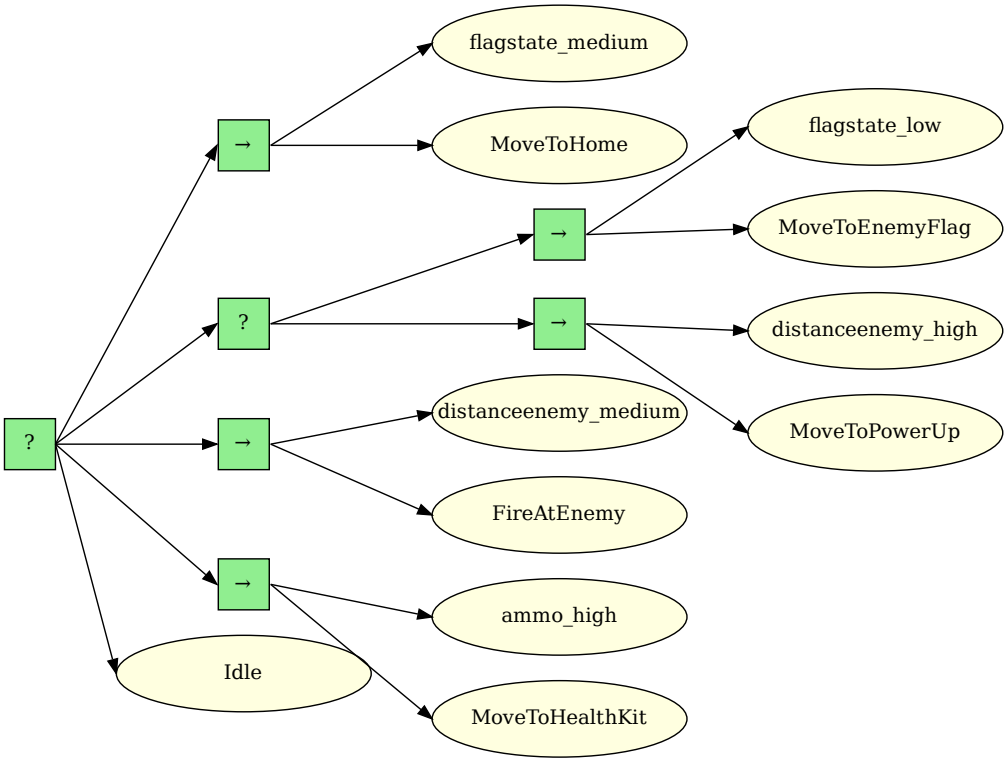
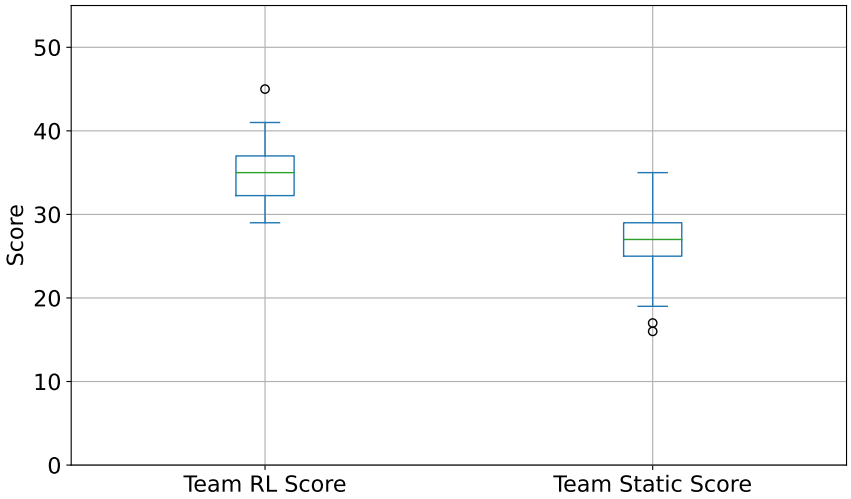


Figure 6.7.: A BT created from experiences gained competing against the static team

Map Size	75×75	100×100	150×150
Wins	182	240	226
Losses	92	39	49
Ties	25	21	25
Total	300	300	300
Success	0.61	0.8	0.75

Table 6.4.: Performance of automatically created BTs on differently sized maps

Table 6.4 depicts the outcome of games played on maps smaller and larger than the one used for training. The results demonstrate that the autonomously generated behavior trees consistently achieve a success rate above 50% across all map size configurations. This indicates that the behavior trees adapt reasonably well to varying map sizes and maintain a competitive performance.



	Team RL Score	Team Static Score
count	30	30
mean	35.067	26.233
std	3.956	4.360
min	29	16
50%	35	27
max	45	35

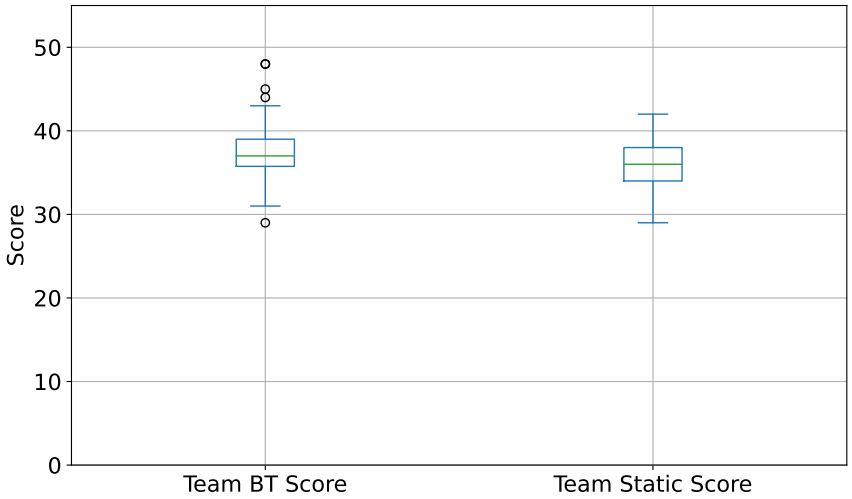
Figure 6.8.: Learning Phase Evaluation (RL vs. RL)

A summary of all games played is depicted in Figure 6.12. The similarity to the previously shown results exhibits that the automatically generated behavior trees are capable of being deployed even on various map size configurations.

Overall, the performance outcomes on different map sizes provide valuable insights into the adaptability and effectiveness of the autonomously generated behavior trees. The success rates indicate their ability to navigate diverse map configurations and maintain competitive gameplay, showcasing their potential for deployment in various game scenarios. By leveraging their learned experiences, the behavior trees demonstrate their adaptability to the specific challenges posed by different map dimensions.

6.2.2. Game Length

The duration of a game can be manipulated by adjusting the score required for victory or the maximum number of ticks allowed. In the default configuration, neither team reached the score threshold for an early game termination, necessitating the utilization of reduced score values for the experimental scenarios presented in this section.



	Team BT Score	Team Static Score
count	300	300
mean	37.373	35.557
std	2.626	2.836
min	29	29
50%	37	36
max	48	42

Figure 6.9.: Behavior Tree Evaluation (RL vs. RL)

First, we investigate the influence of modifying the number of ticks per game. Figure 6.5 presents the outcomes of this experimentation. Analogous to the adjustment of map sizes, the behavior trees generated through the automated process exhibit remarkable adaptability, showcasing no difficulties in accommodating shorter and longer games.

Ticks	**5'000**	**10'000**	**20'000**
Wins	178	240	223
Losses	77	39	64
Ties	45	21	13
Total	300	300	300
Success	0.59	0.8	0.74

Table 6.5.: Performance of automatically created BTs in games of various tick lengths.

Shorter games demand swift point-scoring and punish losing in combat situations more harshly, while longer games allow more leniency for both sides.

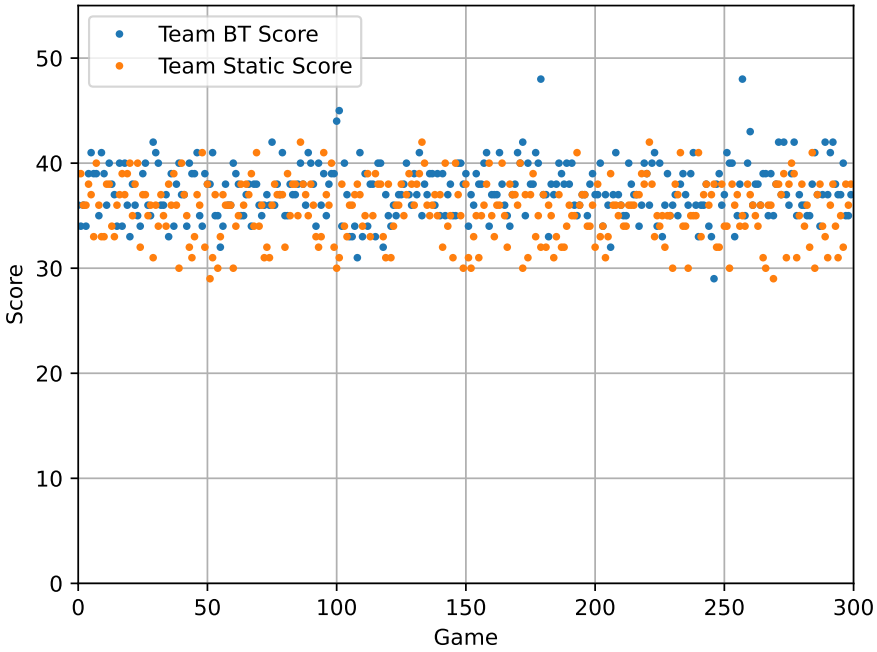


Figure 6.10.: Score Progression over all games during Behavior Tree evaluation (RL vs. RL).

Next, we modify the game length by significantly reducing the score required for victory. The results obtained from this experiment are summarized in Table 6.6, which exhibits outcomes consistent with the previous analysis.

Score	**25**	**50**	**100**
Wins	202	193	240
Losses	98	86	39
Ties	0	21	21
Total	300	300	300
Success	0.67	0.64	0.8

Table 6.6.: Performance of automatically created BTs in games of score to win reduction.

Notably, the performance of the automatically created behavior trees remains consistently promising across diverse game-length configurations. Their success rates surpass again the 50% threshold, indicating their competence in adapting their strategies to suit different gameplay durations. This underscores the robustness of the generated behavior trees, enabling them to navigate and respond to various game parameters effectively.

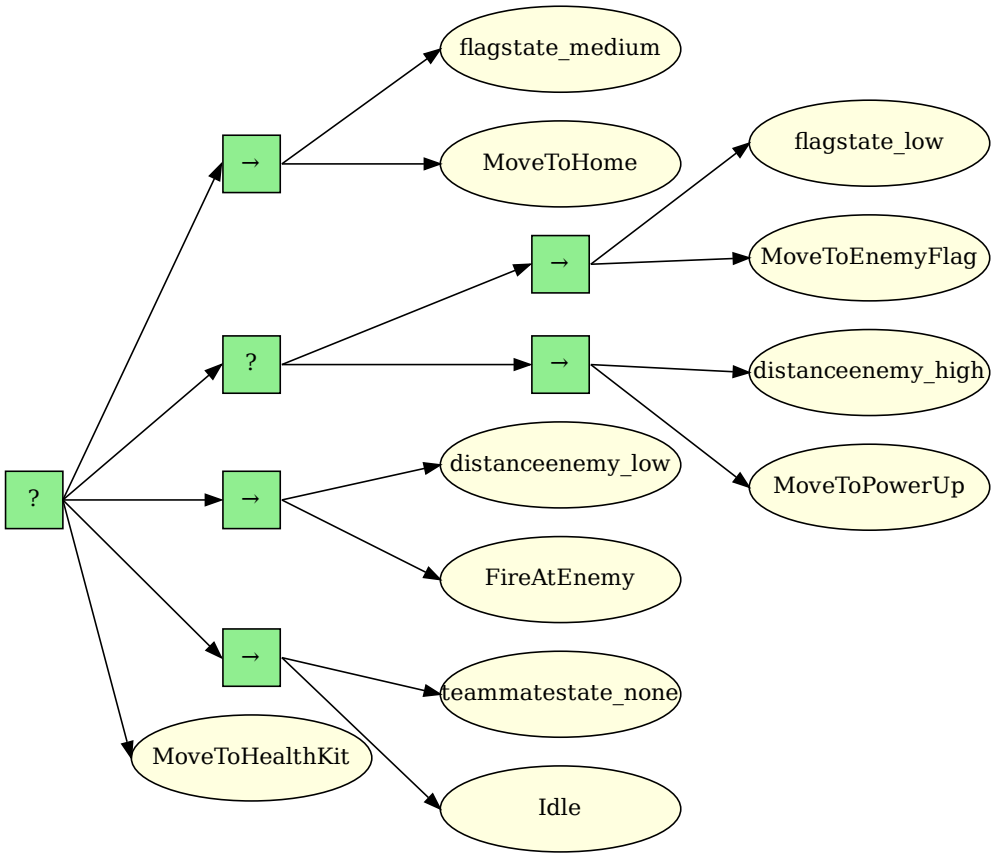
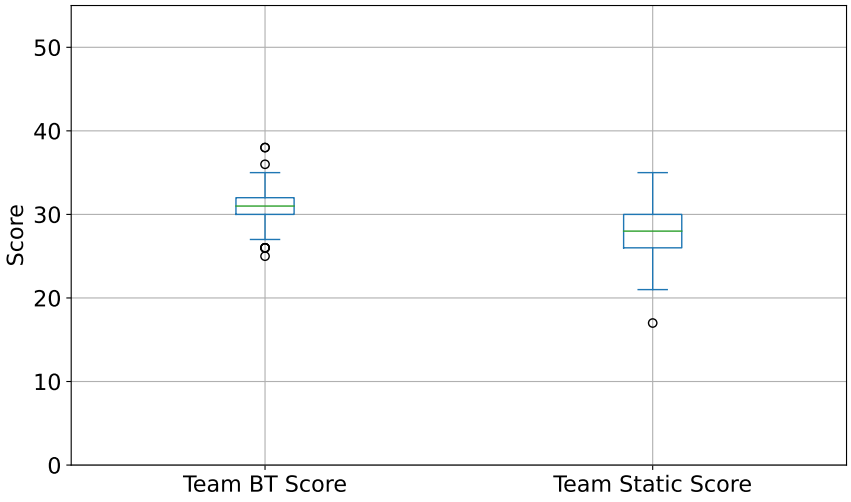


Figure 6.11.: Generated BT of the "RL vs. RL" experiment.

6.3. Impact of Training Policies

Now, we investigate the effects of fine-tuning policies on Q-Table learning in the context of the generated BT using the proposed algorithm. Our analysis focuses on two aspects: the influence of various knowledge transfer strategies between two CTF simulations during the Online-Learning phase when pitted against a static team, and the examination of the consequences of employing a partial reward table that exclusively utilizes the bonus structure within an environment comprising reinforcement learning agents.

To explore the impact of knowledge transfer strategies, we evaluate different approaches for sharing learned policies between two CTF simulations. By assessing the created BT performance during the Online-Learning against the static team, we aim to determine the most effective strategy for transferring knowledge and leveraging it to enhance the



	Team BT Score	Team Static Score
count	300	300
mean	30.977	28.050
std	2.034	2.748
min	25	17
50%	31	28
max	38	35

Figure 6.12.: Summary of results on a 150×150 map

agent's decision-making capabilities.

Furthermore, we investigate the consequences of employing a partial reward table in learning. Specifically, we focus on utilizing the bonus structure as the sole component of the reward table in an environment populated by reinforcement learning agents. By isolating the impact of this partial reward structure, we aim to assess its influence on the generated BT and understand its implications for the agent's learning and overall performance.

6.3.1. Knowledge Transfer Strategies

The knowledge transfer policies are implemented to favor winning strategies but overall limit the total amount of learning episodes for the final Max-Q table. Table 6.7 presents the results of the two knowledge transfer policies. It is important to note that the total number of games played during the learning phase was fixed, regardless of the knowledge transfer policy.

Comparing the success rates, the "Greedy" policy had a success rate of 0.49, while the "Epsilon Greedy (0.33)" policy had a success rate of 0.52. These success rates represent the proportion of favorable outcomes achieved by the BTs during evaluation.

	Greedy			**Epsilon Greedy (0.33)**	
	RL Evaluation	**BT Evaluation**		**RL Evaluation**	**BT Evaluation**
Wins	13	147		24	157
Losses	16	119		6	106
Ties	1	34		0	37
Total	30	300		30	300
Success	0.43	0.49		0.8	0.52

Table 6.7.: Comparison of two knowledge transfer policies

Interestingly, despite the intention to prioritize winning behaviors, both policies demonstrated inferior performance compared to the approach of always transferring knowledge. This is because fewer knowledge transfers in these policies correspond to fewer training episodes due to the fixed number of games in the learning phase.

These findings suggest that more frequent knowledge transfer improves performance in the generated BTs regardless of the specific criteria. It highlights the importance of knowledge sharing and accumulation throughout the learning process.

6.3.2. Reward Table Structure

Table 6.8 presents the evaluation results obtained after omitting the reward table and offline learning in the RL vs RL experiment. In this scenario, the synthesized BT is generated solely based on the experiences acquired during the learning phase, using an efficient reward and bonus structure.

	RL Evaluation	**BT Evaluation**
Wins	19	186
Losses	10	78
Ties	1	85
Total	30	300
Success	0.64	0.62

Table 6.8.: Results after omitting reward table and offline learning.

The evaluation results demonstrate the close performance alignment between the BT and the reinforcement learning agents. The BT exhibits promising results, showcasing a success rate of 0.62, closely following the success rate of 0.64 the reinforcement learning agents achieved during the evaluation period.

These findings suggest that our algorithm enables the efficient creation of well-performing BTs with minimal manual labor, leveraging an effective reward and bonus structure during the learning phase. This highlights the potential of our approach to streamline the development of adaptive and proficient autonomous agents in complex environments.

6.4. Results

The results presented in Section 6.1 provide insights into the research questions *RQ1* and *RQ4*. Firstly, the proposed algorithm effectively generates BTs in cooperative multi-agent settings, surpassing traditional manual solutions. This confirms the algorithm's potential for generating well-performing BTs in scenarios requiring agent collaboration.

The adaptability of the automatically created Behavior Trees, explored in Section 6.2, addresses *RQ2*. The algorithm demonstrates its capability to generate flexible BTs that can seamlessly adjust to parameter changes without re-creation. This inherent flexibility empowers the algorithm to operate effectively in dynamic environments, where parameter values may vary during runtime.

Section 6.3 provides insights into *RQ3*, examining the impact of parameter changes in reinforcement learning strategies on the performance of the generated Behavior Trees. The experiments highlight that prioritizing winning strategies in a fixed learning episode scenario, which reduces the overall experience gathering, results in inferior behavior tree performance. However, the experiments also demonstrate the algorithm's ability to abstract knowledge from an autonomously trained agent using Q-Learning. These findings offer promising prospects for future applications, such as debugging tabular reinforcement learning methods by generating behavior trees from them.

Finally, the experiments show that the proposed algorithm can successfully synthesize well-performing Behavior Trees using a partial reward table, implying the potential for the fully autonomous creation of behavior trees through the proposed algorithm.

The results confirm the proposed algorithm's effectiveness, adaptability, and potential in generating high-quality Behavior Trees in various settings, including cooperative multi-agent scenarios and dynamic environments.

7. Conclusion and Future Work

This chapter concludes the work presented in this thesis. We further discuss the limitations of our approach and possible future work and extensions.

7.1. Conclusion

The work presented in this thesis offers a novel and effective approach to synthesizing behavior trees for Non-Player-Character (NPC) control in video game-like settings. By leveraging the experiences gained from autonomous agents in a multi-agent reinforcement learning environment, our algorithm addresses the challenges associated with the manual creation of behavior trees. This approach proves to be a promising solution for developing intelligent NPC behaviors that can adapt to dynamic environments and exhibit cooperative interactions.

The motivation behind utilizing behavior trees for NPC control lies in their ability to provide a hierarchical structure that captures complex decision-making processes. However, the manual creation of behavior trees is a labor-intensive task that requires expert knowledge and significant effort. Our research tackles this challenge by proposing an automatic behavior tree generation algorithm that leverages the knowledge acquired through Q-Learning.

The core of our algorithm involves processing the Max-Q-Table obtained from the Q-Learning process. We distill the knowledge into a reduced representation known as the Max-Q-Table by identifying the actions with the highest Q-Values for each visited state during the learning phase. From this table, we generate atomic nodes consisting of task and task-condition pairs. These atomic nodes are then organized into sub-trees based on shared guards and assembled into a final behavior tree using task priority selection and selector nodes.

To evaluate the effectiveness of our automatic behavior tree generation algorithm, we conducted experiments in a Capture-The-Flag game scenario. In this game, two teams of two players per team aim to score as many points as possible within a specific time frame. The results of our experiments validate the capabilities of the algorithm. We observed that the algorithm successfully creates Behavior Trees automatically, reducing the manual labor required for their development. Furthermore, the automatically generated Behavior Trees demonstrated comparable performance to manually designed trees, even when operating within constraints not encountered during the learning phase of the reinforcement learning

agents. Additionally, our algorithm showcases its applicability in multi-agent settings, where it can generate distinct yet cooperative Behavior Trees.

By automating the synthesis of behavior trees and enabling their adaptability to dynamic environments, our research contributes to the field of NPC control in video games. It offers a promising avenue for efficiently creating intelligent and cooperative NPC behaviors, reducing the burden of manual tree construction, and facilitating the development of interactive and immersive gaming experiences.

7.2. Limitation and Future Work

The algorithm proposed in this work utilizes a tabular reinforcement learning method known to suffer when a large state-action space is used. We, therefore, relied on a method of state discretization due to runtime- and memory concerns, as well as the inability of tabular reinforcement learning methods to be effective with large spaces. To alleviate this, we further focussed on a simple testing setup for the proposed algorithm, which limits the number of actions an agent can take and the possible states an agent can be in.

A future extension might consider other machine-learning approaches to autonomously synthesize behavior trees to allow for a larger state-action space that does not need to be categorized into a smaller subset of discrete states. We also do not consider an upper bound of states and actions during development. An interesting future direction of this work could explore the viability of larger game scenarios.

Behavior trees these days provide a large set of leaf and non-leaf nodes that can replace subtrees with powerful expressions and provide additional mechanisms increasing a behavior tree's versatility. Additionally, as discussed, the execution model of behavior trees has largely shifted and no longer utilizes a tick-driven execution model. Instead, an event-based system is often integrated into a behavior tree's blackboard. Our algorithm uses only a small subset of pre-defined internal nodes and a simple execution model. A future extension of our work might consider an automatic creation approach that generates these "Second Generation" behavior trees that use event-based systems and more complex nodes.

Finally, we have not looked at the performance of the exported behavior trees against human opponents. Adding this component to future iterations could provide a helpful and necessary step to allow the adoption of the algorithm in the video game industry.

Bibliography

[1] Agis, R.A., Gottifredi, S., García, A.J.: An event-driven behavior trees extension to facilitate non-player multi-agent coordination in video games. Expert Systems with Applications **155**, 113457 (Oct 2020). https://doi.org/10.1016/j.eswa.2020.113457, https://linkinghub.elsevier.com/retrieve/pii/S0957417420302815

[2] Banerjee, B.: Autonomous Acquisition of Behavior Trees for Robot Control. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3460–3467 (Oct 2018). https://doi.org/10.1109/IROS.2018.8594083, iSSN: 2153-0866

[3] Berthling-Hansen, G., Morch, E., Løvlid, R.A., Gundersen, O.E.: Automating Behaviour Tree Generation for Simulating Troop Movements (Poster). In: 2018 IEEE Conference on Cognitive and Computational Aspects of Situation Management (CogSIMA). pp. 147–153 (Jun 2018). https://doi.org/10.1109/COGSIMA.2018.8423978, iSSN: 2379-1675

[4] Bouchard, B., Gaboury, S., Bouchard, K., Francillette, Y.: Modeling Human Activities Using Behaviour Trees in Smart Homes. In: Proceedings of the 11th PErvasive Technologies Related to Assistive Environments Conference. pp. 67–74. PETRA '18, Association for Computing Machinery, New York, NY, USA (Jun 2018). https://doi.org/10.1145/3197768.3201522, https://doi.org/10.1145/3197768.3201522

[5] Colledanchise, M., Murray, R.M., Ögren, P.: Synthesis of correct-by-construction behavior trees. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 6039–6046 (Sep 2017). https://doi.org/10.1109/IROS.2017.8206502, iSSN: 2153-0866

[6] Colledanchise, M., Natale, L.: Improving the Parallel Execution of Behavior Trees. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 7103–7110 (Oct 2018). https://doi.org/10.1109/IROS.2018.8593504, http://arxiv.org/abs/1809.04898, arXiv:1809.04898 [cs]

[7] Colledanchise, M., Ogren, P.: How Behavior Trees modularize robustness and safety in hybrid systems. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 1482–1488. IEEE, Chicago, IL, USA (Sep 2014). https://doi.org/10.1109/IROS.2014.6942752, http://ieeexplore.ieee.org/document/6942752/

[8] Colledanchise, M., Parasuraman, R., Ögren, P.: Learning of Behavior Trees for Autonomous Agents. IEEE Transactions on Games **11**(2), 183–189 (Jun 2019). https://doi.org/10.1109/TG.2018.2816806, http://arxiv.org/abs/1504.05811, arXiv:1504.05811 [cs]

[9] Colledanchise, M., Ögren, P.: Behavior Trees in Robotics and AI: An Introduction. CRC Press (Jul 2018). https://doi.org/10.1201/9780429489105, http://arxiv.org/abs/1709.00084, arXiv:1709.00084 [cs]

[10] Coronado, E., Mastrogiovanni, F., Venture, G.: Development of Intelligent Behaviors for Social Robots via User-Friendly and Modular Programming Tools. In: 2018 IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO). pp. 62–68 (Sep 2018). https://doi.org/10.1109/ARSO.2018.8625839, iSSN: 2162-7576

[11] Dey, R., Child, C.: QL-BT: Enhancing behaviour tree design and implementation with Q-learning. In: 2013 IEEE Conference on Computational Inteligence in Games (CIG). pp. 1–8 (Aug 2013). https://doi.org/10.1109/CIG.2013.6633623, iSSN: 2325-4289

[12] Florez-Puga, G., Gomez-Martin, M., Gomez-Martin, P., Diaz-Agudo, B., Gonzalez-Calero, P.: Query-Enabled Behavior Trees. IEEE Transactions on Computational Intelligence and AI in Games **1**(4), 298–308 (Dec 2009). https://doi.org/10.1109/TCIAIG.2009.2036369, http://ieeexplore.ieee.org/document/5325892/

[13] Florez-Puga, G., Gómez-Martín, M., Díaz-Agudo, B., González-Calero, P.: Dynamic Expansion of Behaviour Trees. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment **4**(1), 36–41 (2008). https://doi.org/10.1609/aiide.v4i1.18669, https://ojs.aaai.org/index.php/AIIDE/article/view/18669, number: 1

[14] Fu, Y., Qin, L., Yin, Q.: A reinforcement learning behavior tree framework for game ai. In: Proceedings of the 2016 International Conference on Economics, Social Science, Arts, Education and Management Engineering. pp. 573–579. Atlantis Press (2016/08). https://doi.org/10.2991/essaeme-16.2016.120, https://doi.org/10.2991/essaeme-16.2016.120

[15] Hoff, J.W., Christensen, H.J.: Evolving Behaviour Trees: - Automatic Generation of AI Opponents for Real-Time Strategy Games. Master's thesis, NTNU (2016), https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2405140, accepted: 2016-09-07T14:00:48Z Publication Title: 116

[16] Iovino, M., Scukins, E., Styrud, J., Ögren, P., Smith, C.: A survey of Behavior Trees in robotics and AI. Robotics and Autonomous Systems **154**, 104096 (Aug 2022). https://doi.org/10.1016/j.robot.2022.104096, https://linkinghub.elsevier.com/retrieve/pii/S0921889022000513

[17] Iovino, M., Styrud, J., Falco, P., Smith, C.: Learning Behavior Trees with Genetic Programming in Unpredictable Environments. In: 2021 IEEE International Conference on Robotics and Automation (ICRA). pp. 4591–4597 (May 2021). https://doi.org/10.1109/ICRA48506.2021.9562088, iSSN: 2577-087X

[18] Isla, D.: GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI (2005), https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai, section: programming

[19] Johansson, A., Dell'Acqua, P.: Emotional behavior trees. In: 2012 IEEE Conference on Computational Intelligence and Games (CIG). pp. 355–362 (Sep 2012). https://doi.org/10.1109/CIG.2012.6374177, iSSN: 2325-4289

[20] Jones, S., Studley, M., Hauert, S., Winfield, A.: Evolving Behaviour Trees for Swarm Robotics. In: Groß, R., Kolling, A., Berman, S., Frazzoli, E., Martinoli, A., Matsuno, F., Gauci, M. (eds.) Distributed Autonomous Robotic Systems: The 13th International Symposium, pp. 487–501. Springer Proceedings in Advanced Robotics, Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-73008-0_34, https://doi.org/10.1007/978-3-319-73008-0_34

[21] Kuckling, J., van Pelt, V., Birattari, M.: Automatic Modular Design of Behavior Trees for Robot Swarms with Communication Capabilites. In: Castillo, P.A., Jiménez Laredo, J.L. (eds.) Applications of Evolutionary Computation, vol. 12694, pp. 130–145. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-72699-7_9, http://link.springer.com/10.1007/978-3-030-72699-7_9, series Title: Lecture Notes in Computer Science

[22] Ligot, A., Kuckling, J., Bozhinoski, D., Birattari, M.: Automatic modular design of robot swarms using behavior trees as a control architecture. PeerJ Computer Science **6**, e314 (Nov 2020). https://doi.org/10.7717/peerj-cs.314, https://peerj.com/articles/cs-314

[23] Lim, C.U., Baumgarten, R., Colton, S.: Evolving Behaviour Trees for the Commercial Game DEFCON. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcazar, A.I., Goh, C.K., Merelo, J.J., Neri, F., Preuß, M., Togelius, J., Yannakakis, G.N. (eds.) Applications of Evolutionary Computation. pp. 100–110. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12239-2_11

[24] Marcotte, R., Hamilton, H.J.: Behavior Trees for Modelling Artificial Intelligence in Games: A Tutorial. The Computer Games Journal **6**(3), 171–184 (Sep 2017). https://doi.org/10.1007/s40869-017-0040-9, http://link.springer.com/10.1007/s40869-017-0040-9

[25] Marques, R., Sousa, M.d.: Agents that learn to behave with reinforcement learning and behavior trees. Master's thesis, Instituto Superior Técnico (2021), https:

//www.semanticscholar.org/paper/Agents-that-learn-to-behave-with-reinforcement-and-Marques-Sousa/e4023df328ecdda5379b61807c6b33fecb8a281f

[26] Mateas, M., Stern, A.: A behavior language for story-based believable agents. IEEE Intelligent Systems **17**(4), 39–47 (Jul 2002). https://doi.org/10.1109/MIS.2002.1024751, conference Name: IEEE Intelligent Systems

[27] Meng, F., Hyung, C.J.: Research on Multi-NPC Marine Game AI System based on Q-learning Algorithm. In: 2022 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA). pp. 648–652 (Jun 2022). https://doi.org/10.1109/ICAICA54878.2022.9844648

[28] Merrill, B.: Building Utility Decisions into Your Existing Behavior Tree. In: Game AI Pro 360, pp. 81–90. CRC Press, 1 edn. (2020). https://doi.org/10.1201/9780429055058-7

[29] Millington, I., Funge, J.: Artificial Intelligence for Games, Second Edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edn. (2009)

[30] Nicolau, M., Perez Liebana, D., O'Neill, M., Brabazon, A.: Evolutionary Behavior Tree Approaches for Navigating Platform Games. IEEE Transactions on Computational Intelligence and AI in Games **9**, 227–238 (Sep 2017). https://doi.org/10.1109/TCIAIG.2016.2543661

[31] Olsson, M.: Behavior Trees for decision-making in Autonomous Driving. Master's thesis, KTH, School of Computer Science and Communication (CSC). (2016), http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-183060

[32] Paduraru, C., Paduraru, M.: Automatic Difficulty Management and Testing in Games using a Framework Based on Behavior Trees and Genetic Algorithms. In: 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 170–179 (Nov 2019). https://doi.org/10.1109/ICECCS.2019.00026

[33] Partlan, N., Soto, L., Howe, J., Shrivastava, S., El-Nasr, M.S., Marsella, S.: Evolving-Behavior: Towards Co-Creative Evolution of Behavior Trees for Game NPCs (Sep 2022). https://doi.org/10.48550/arXiv.2209.01020, http://arxiv.org/abs/2209.01020, arXiv:2209.01020 [cs]

[34] Pereira, R.d.P., Engel, P.M.: A Framework for Constrained and Adaptive Behavior-Based Agents (Jun 2015). https://doi.org/10.48550/arXiv.1506.02312, http://arxiv.org/abs/1506.02312, arXiv:1506.02312 [cs]

[35] Perez, D., Nicolau, M., O'Neill, M., Brabazon, A.: Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcázar, A.I., Merelo, J.J., Neri, F., Preuss, M., Richter, H., Togelius, J., Yannakakis, G.N. (eds.) Applications of Evolutionary Computation, vol. 6624, pp. 123–132. Springer Berlin Heidelberg (2011).

https://doi.org/10.1007/978-3-642-20525-5_13, http://link.springer.com/10.1007/978-3-642-20525-5_13, series Title: Lecture Notes in Computer Science

[36] Rabin, S.: Game AI Pro 3: Collected Wisdom of Game AI Professionals. Computer game development / Design, CRC Press (2017), http://www.gameaipro.com/

[37] Robertson, G., Watson, I.: Building behavior trees from observations in real-time strategy games. In: 2015 International Symposium on Innovations in Intelligent SysTems and Applications (INISTA). pp. 1–7 (Sep 2015). https://doi.org/10.1109/INISTA.2015.7276774

[38] Sagredo-Olivenza, I., Gómez-Martín, P.P., Gómez-Martín, M.A., González-Calero, P.A.: Trained Behavior Trees: Programming by Demonstration to Support AI Game Designers. IEEE Transactions on Games **11**(1), 5–14 (Mar 2019). https://doi.org/10.1109/TG.2017.2771831, conference Name: IEEE Transactions on Games

[39] Scheper, K.Y.W., Tijmons, S., de Visser, C.C., de Croon, G.C.H.E.: Behavior Trees for Evolutionary Robotics. Artificial Life **22**(1), 23–48 (Feb 2016). https://doi.org/10.1162/ARTL_a_00192, https://direct.mit.edu/artl/article/22/1/23-48/2836

[40] Schwab, P., Hlavacs, H.: Capturing the Essence: Towards the Automated Generation of Transparent Behavior Models. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment **11**(1), 184–190 (2015), https://ojs.aaai.org/index.php/AIIDE/article/view/12795, number: 1

[41] Sekhavat, Y.: Behavior Trees for Computer Games. International Journal on Artificial Intelligence Tools **26** (Jan 2017). https://doi.org/10.1142/S0218213017300010

[42] Sprague, C.I., Ögren, P.: Adding Neural Network Controllers to Behavior Trees without Destroying Performance Guarantees. In: 2022 IEEE 61st Conference on Decision and Control (CDC). pp. 3989–3996 (Dec 2022). https://doi.org/10.1109/CDC51059.2022.9992501, iSSN: 2576-2370

[43] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press, second edn. (2018), http://incompleteideas.net/book/the-book-2nd.html

[44] Tadewos, T.G., Shamgah, L., Karimoddini, A.: Automatic Decentralized Behavior Tree Synthesis and Execution for Coordination of Intelligent Vehicles. Knowledge-Based Systems **260**, 110181 (Jan 2023). https://doi.org/10.1016/j.knosys.2022.110181, https://www.sciencedirect.com/science/article/pii/S0950705122012771

[45] Venkata, S.S.O., Parasuraman, R., Pidaparti, R.: KT-BT: A Framework for Knowledge Transfer Through Behavior Trees in Multi-Robot Systems (Sep 2022). https://doi.org/10.48550/arXiv.2209.02886, http://arxiv.org/abs/2209.02886, arXiv:2209.02886 [cs]

[46] Zhang, Q., Yao, J., Yin, Q., Zha, Y.: Learning Behavior Trees for Autonomous Agents with Hybrid Constraints Evolution. Applied Sciences **8**(7), 1077 (Jul 2018). https://doi.org/10.3390/app8071077, `http://www.mdpi.com/2076-3417/8/7/1077`

Acronyms

AI Artificial Intelligence. 1, 8, 15

BT Behaviour Tree. xi, 1, 5–9, 20, 22, 25, 28, 30, 31, 33, 42–45, 47, 50, 58–61

CTF Capture-the-Flag. 31, 37, 39, 44, 58

EA Evolutionary Algorithm. 6

FSM Finite State Machine. 1, 5, 7, 9

GP Genetic Programming. 6, 8, 31

NPC Non-Player Character. 1, 5, 6, 9, 13, 19, 30, 35, 36, 40

RL Reinforcement Learning. 7, 9, 15, 31, 42, 44, 47, 49, 50

A. Appendix

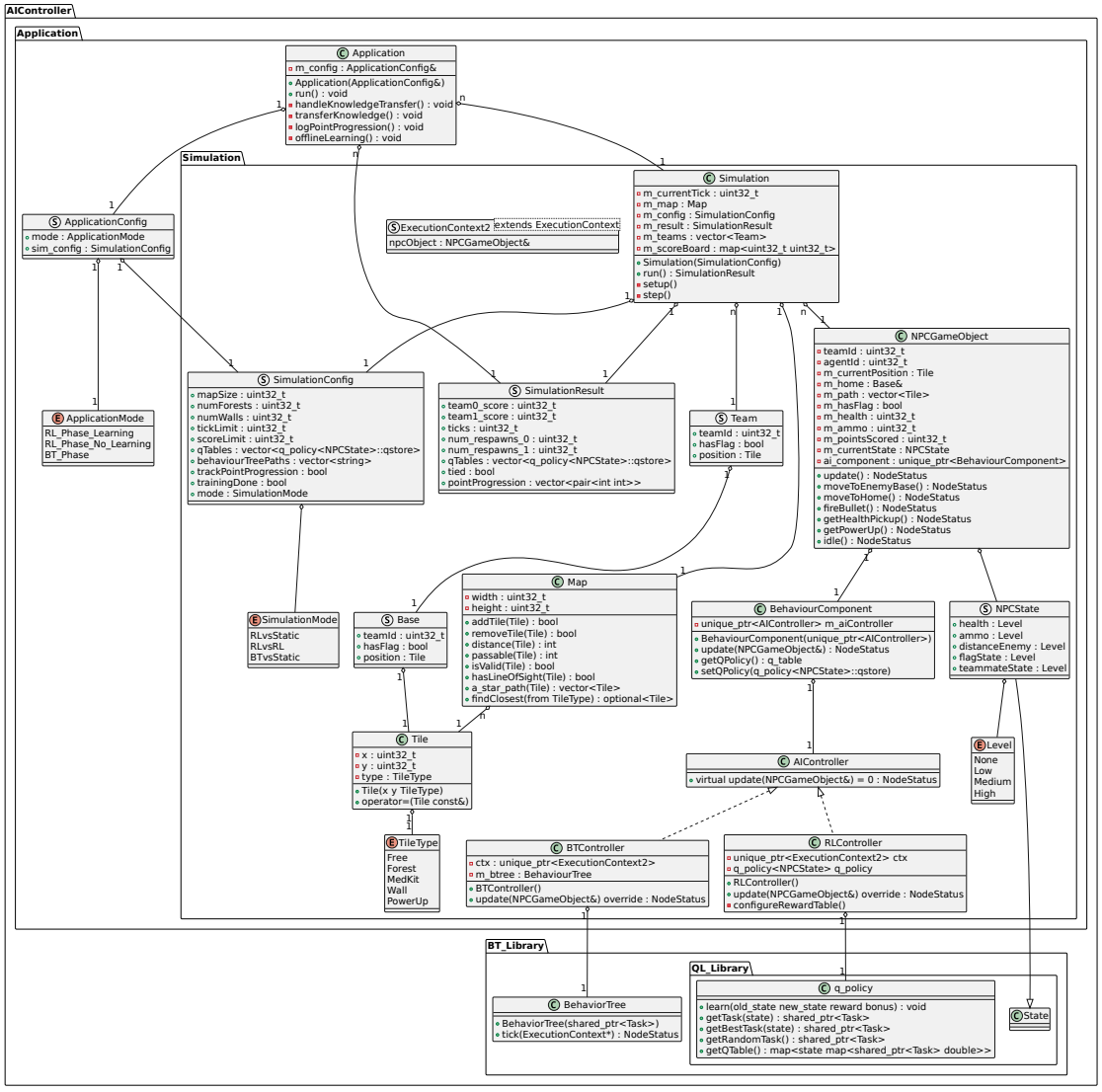


Figure A.1.: Class diagram of the application and its usage of the AI controller libraries

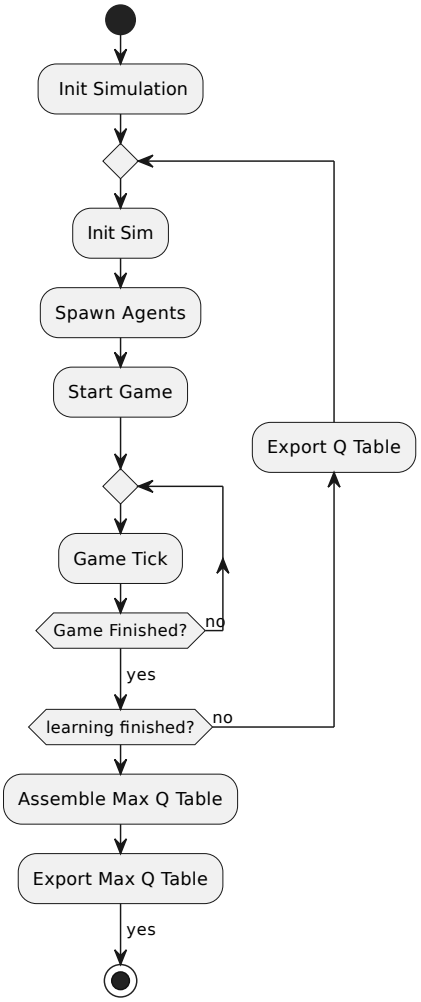


Figure A.2.: Activity Diagram of the Learning Phase

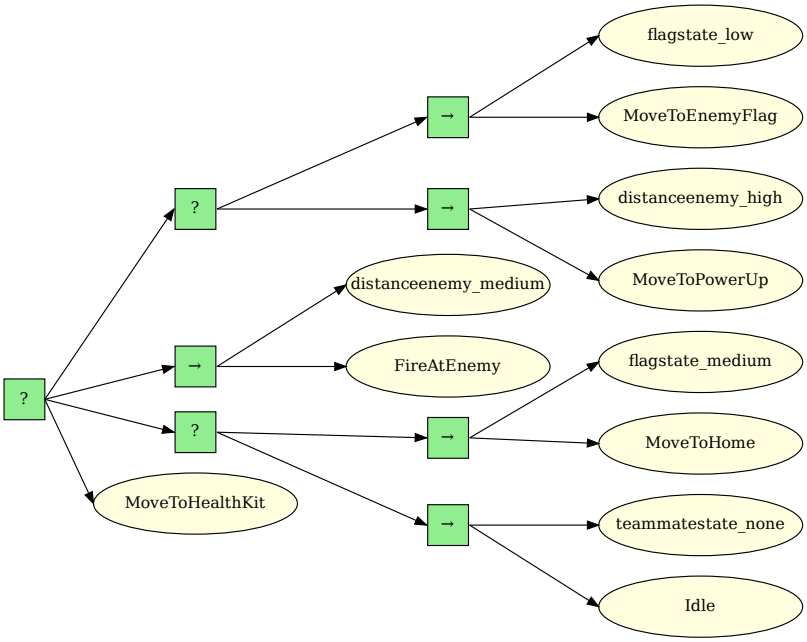


Figure A.3.: Second BT created from experiences gained competing against the static team

	Team RL Score	Team Static Score
count	30	30
mean	29.233	22.6
std	3.748	4.32
min	19	16
50%	30	22
max	36	31

(a) Table of Figure 6.1

	Team BT Score	Team Static Score
count	300	300
mean	40.977	36.757
std	2.354	2.999
min	34	27
50%	41	37
max	480	45

(b) Table of Figure 6.2

Table A.1.: Score summary of training and BT generation against static team.

	Team RL Score	Team Static Score
count	30	330
mean	35.067	26.233
std	3.956	4.360
min	29	16
50%	35	27
max	45	35

(a) Table of Figure 6.8

	Team BT Score	Team Static Score
count	300	300
mean	37.373	35.557
std	2.626	2.835
min	29	29
50%	37	36
max	48	42

(b) Table of Figure 6.9

Table A.2.: Score summary of pure reinforcement learning opponents and generated BT.