DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2019

Integrating Reinforcement Learning into Behavior Trees by Hierarchical Composition

MART KARTAŠEV

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Integrating Reinforcement Learning into Behavior Trees by Hierarchical Composition

Computer Science and Engineering
Degree project
Second cycle, 30 credits

Student: Mart Kartašev (kartasev@kth.se)
Supervisor: Petter Ögren (petter@kth.se)
Examiner: Joakim Gustafsson (jkgu@kth.se)

July 2, 2019

Abstract

This thesis investigates ways to extend the use of Reinforcement Learning (RL) to Behavior Trees (BTs). BTs are used in the field of Artificial Intelligence (AI) in order to create modular and reactive planning agents. While human designed BTs are capable of reacting to changes in an environment as foreseen by an expert, they are not capable of adapting to new scenarios. The focus of the thesis is on using existing general-purpose RL methods within the framework of BTs. Deep Q-Networks (DQN) and Proximal Policy Optimisation (PPO) were embedded into BTs, using RL implementations from an open-source RL library. The experimental part of the thesis uses these nodes in a variety of scenarios of increasing complexity, demonstrating some of the benefits of combining RL and BTs. The experiments show that there are benefits to using BTs to control a set of hierarchically decomposed RL sub-tasks for solving a larger problem. Such decomposition allows for reuse of generic behaviors in different parts of a BT. By decomposing the RL problem using a BT, it is also possible to identify and replace problematic parts of a policy, as opposed to retraining the entire policy.

Sammanfattning

Den här uppsatsen undersöker sätt att utvidga användningsområdet för förstärkningsinlärning (RL) till beteendeträd (BT). BT används inom fältet artificiell intelligens (AI) för att skapa modulära och reaktiva planerande agenter. BT skapade av människor kan reagera på förändringar i en värld på ett sätt som har förutsetts av en expert, men de är inte kapabla att anpassa sig till nya scenarier. Fokuset i den här uppsatsen ligger på att använda existerande RL-metoder inom ramverket för BT. Djupa Q-nätverk (DQN) och Proximal Policy Optimisation (PPO) har bäddats in i BT, där RL-implementationerna kommer från ett bibliotek som har öppen källkod. Experimenten visar att det finns fördelar med att använda BT för att kontrollera en mängd av hierarkiskt uppdelade RL-deluppgifter för att lösa ett större problem. En sådan uppdelning tillåter mer allmänna beteenden att återanvändas i olika delar av ett BT. Genom att dela upp RL-problemet med hjälp av ett BT så är det också möjligt att identifiera och ersätta problematiska delar av en policy, till skillnad från att träna om hela policyn.

-Translation by Aron Granberg

Contents

Acronyms

A3C Asynchronous Advantage Actor-Critic. 8

AI Artificial Intelligence. 4, 8

BT Behavior Tree. 4–9, 13, 15–17, 21–25, 28–30, 32, 34, 38, 39, 45, 52, 53, 56, 61–64

DQN Deep Q-Network. 7, 8, 20, 21, 30, 38, 43, 58, 61, 62, 67

FSM Finite State Machine. 7

HRL Hierarchical Reinforcement Learning. 8, 22, 24

MDP Markov Decision Process. 8, 17, 18, 24

MLP Multi-Layer perceptron. 33, 42, 47, 58, 67

MRP Markov Reward Process. 17

NPC Non-Player character. 26

PPA Postcondition-Precondition-Action. 15, 39, 64

PPO Proximal Policy Optimisation. 8, 30, 38, 61, 62

RL Reinforcement Learning. 5–8, 16–25, 27–30, 32, 34, 38, 39, 45–47, 51, 52, 58, 60–64

RL-BT Reinforcement learning in behavior trees. 6, 8, 16, 23, 24, 33, 38, 60

SMDP Semi-Markov Decision Process. 8, 24

1 Introduction

This chapter will present the primary topic and describe the methodology used in the thesis. It also contains background information as well as a short overview of other related works in this area of research.

1.1 AI, Behavior trees and learning

The primary purpose of Artificial Intelligence (AI) is to automate tasks that would otherwise often require human operators. Over time, the problems that AI is expected to solve have become more complicated, requiring the consideration of extended sequences of actions ahead of time. This type of problem solving is more generally known as planning.

There are two primary aspects to planning problems. The first is to determine the best course of action. This can be done in a variety of ways, ranging from different types of search and simulation methods all the way to hand-crafted designs. The second problem concerns the execution of a plan once it has been determined. Following the original plan without disruption can be possible in some simple scenarios, but it is rarely possible in dynamic applications or in the real world. There is often a need to handle a large set of possible behaviors to ensure the proper execution of plans when faced with unexpected changes in the environment. This increases the complexity of scripted agents which, if left unmanaged, will make them very difficult to maintain and modify. Over the years the different methods, such as finite state-machines or decision trees have proven to be inefficient, unresponsive or too tightly coupled to manage the ever increasing complexities required of modern agents. [16]

Behavior Trees (BTs) are a specific way to construct, structure and control the task related code in the planning algorithms of an artificial agent. One of the primary aspects provided by BTs is the effective management of a large codebase through hierarchies and modularity, as well as emphasising human readability and reusability. Consider Figure 1 as an example of a simple BT. Even without knowing the full semantics of behavior tree structure, it should be possible for a layman to roughly comprehend the high-level plan of such an artificial agent. The purpose of the agent given in Figure 1 is to grab a bottle and remove its cap, finally filling it with water. The details of such an execution and the tree nodes used in this example will be revisited in Section 2.1.

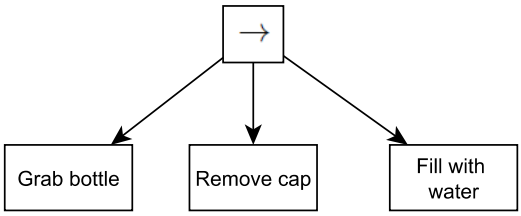

Figure 1: Example of a simple behavior tree.

From a functional perspective, BTs provide reactivity to a changing environment to an agent executing its plan. This allows the programmer to define complex behavior specifications in an expressive way by reusing already existing lower-level implementations. The reactivity arises from the task switching nature of the control structure. By repeatedly iterating through this structure, the agent can detect changes in environment conditions, allowing it to redirect the control flow to perform another task. It also makes the behavior robust to failures, allowing some parts of the plan to be executed repeatedly if necessary. It is worth noting that while BTs help define such agents, care needs to be taken in the design of individual nodes in order to guarantee the desired reactivity and robustness.

This is where learning algorithms can potentially help simplify and enhance the design process. In some cases we might not want to only react, but also adapt to changes in the environment. For example, the robot that is designed to fill bottles might be specialised to handle a specific type of bottle. Let's say the robot was to be used in a similar setting with different bottles. The robot can face an array of issues even with this relatively trivial change of scenario. What if the bottle is in a shape that the robot's gripper cannot grasp or the cap needs to be removed in some specific way? Such things could be learned if the tree was augmented with learning capabilities, allowing for continuous refinement of already existing behaviors as opposed to manually rewriting all the actions. There are also behaviors that might be beyond the ability of the engineer to define, which learning might be able to solve by trial and error. The idea for this thesis project is to investigate ways to embed this type of learning capability into BTs, which can theoretically be used to enhance both actions and control flow.

There are many types of learning algorithms, but the scope of this thesis will focus on utilising a branch of learning algorithms known as reinforcement learning (RL). The problem that RL tries to solve is finding an optimal way to behave in an environment given a space of possible actions that the agent can take and a space of observations that the agent can obtain from the environment. RL algorithms define a reward signal which is an additional scalar feedback signal in conjunction with the observations that the robot receives from the environment to know how well it is doing. The outcome of RL is a policy, determining how the agent should behave in different situations. A more elaborate description of RL will be given in Section 2.2. One might note that there are certain similarities between RL and BTs. When designing a behavior tree, the primary development effort is focused around defining and taking good actions, guided by observations from the environment for activation and execution. This is very similar to how an action space is defined and utilised in a RL problem. In essence, both methods are used to represent a policy for a behavior, the difference being that behavior tree policies are often created manually and RL policies are learned incrementally by acting in the environment. In fact, we will see that the two methods are similar enough, that in some cases it is possible to convert behavior tree nodes to a RL representation without any changes to the space definitions.

There are numerous motivations for embedding learning into BTs. It would allow the behavior tree based agent to keep improving the actions that have been defined. The control flow nodes which determine how the agent executes its plan could also be learned. This means that the agent could learn to not only improve the execution

of actions, but also the decision making capabilities regarding what actions to take. More generally, it would allow the agent to deal with situations that have not been planned for by the designer. The agent can instead learn to behave by trial and error and incorporate new experiences into its behavior. In addition, it allows a larger problem to be split into many smaller sub-tasks. This can reduce the complexity of the problem as the composition of smaller problems might be easier to learn than the entire problem at once. In pure RL scenarios it is hard to reuse sub-routines that are part of a larger task. This approach would also allow for greater reusability of learned behavior.

The approach of RL-BT would primarily be useful for software developers and engineers. BTs are primarily a pattern for development, so there would be no immediate use for a wider audience, as the low-level conditions and actions require expert knowledge. In the long-term, if developing with BTs became more mainstream and certain development principles were adopted, then the modular nature of BT nodes would allow for different domain specific libraries to be developed. This means that specialists in fields could create and share low-level implementations that could be reused in different scenarios. As BTs are human readable and have concrete structures, it could be relatively easy for laymen to program their own control algorithms using visual tools with pre-programmed nodes.

All of this could potentially be dangerous, especially when incorporating learning into the behaviors. Depending on the field of application, errors in algorithms can have disastrous effects. Examples of this could be robotic arms acting erratically or self-driving vehicles moving in an unpredictable manner. This is also one of the reasons why it is reasonable to try out algorithms in simulations, as this thesis does, before considering them for application in real scenarios. One of the benefits of BTs is the possibility to assure safety on the level of individual behaviors and by utilising BT design structures. This can be doubly useful when incorporating learning algorithms, as it is possible to assure that the learned algorithm can only execute actions when it is safe to do so.

1.2 Thesis topic and approach

This section will first present the primary research question of the thesis. The question is then broken down into a set of more detailed sub-questions

The primary question (PQ) of the thesis is formulated as follows:

PQ How can existing behavior trees based agents be enhanced with reinforcement learning?

Thus the intention behind the thesis is to evaluate the potential possibilities that arise when incorporating RL into BTs. The thesis will investigate the following sub-problems in relation to the primary question:

Q1 What types of RL algorithms are suited for use in BTs?

Q2 Is it possible to use the elements of normal BT design for learning purposes?

Q3 Are the approaches suggested in the existing literature feasible in more complex scenarios?

Q3A Is it feasible to train action nodes that execute an RL behavior directly?

Q3B Is it feasible to create control flow nodes that execute sub-trees based on a learned behavior?

Q4 Does such a decomposition into sub-problems also simplify design and code management? Can it be used for identifying individual problematic parts of the composite behavior during debugging? Is it possible to retrain unsuccessful behavior on the level of single sub-behaviors?

Q5 Can a learning based sub-tree in a BT that was initially trained for dealing with a specific type of object in a given task, be duplicated and re-specialised for another type of similar object in the same problem? Is it useful to train a set of similar trees specialised for different types of objects or situations?

Q6 Is it feasible to combine multiple basic RL nodes in a single BT at different levels in the hierarchy?

The focus when dealing with each of these questions is on exploring how RL can be incorporated into existing designs, without having to rework the agent. In the interest of clarity, it is important to emphasise that the focus of this thesis is not on developing RL algorithms themselves, but rather on investigating the utilisation of existing algorithms in order to enhance BTs.

The problem will be approached in an experimental manner. An overview of existing work and theory on the subject will be examined and presented. The focus will then be on adapting some of the existing ideas, implementing them and testing their validity by experiment. The results will be used to improve other experiments of increasing complexity, with the ultimate goal of expanding the current principles of RL usage in BTs. As each experiment is intended to expand upon the knowledge gained from previous ones, each will be presented in a dedicated section featuring the problem definition, results and discussion.

1.3 Related Work

This section will give a brief overview of previous work on BTs and RL. The theory and further details on these topics can be found in Section 2.

The use of BT arose out of the need for modularity [3, 4], in the complex AI agents developed for representing in-game characters. In academia and the rest of the industry, BT originated as a replacement for Finite State Machines (FSMs) in the field of robotics [26]. In the past years a solid theoretical framework for developing BTs has been created [15]. While much of the theory is relatively new, there are already results showing how BTs can be used to generalise previously used approaches such as the teleo-reactive approach, decision-trees or finite state machines [16].

One of the better known approaches in the field of RL is Deep Q-Network (DQN) [6], popularised by their success in playing Atari games. Trust region policy optimisation

(TRPO) [8], a policy gradient based method, is another algorithm which focuses on the policy rather than the value function (like DQN), in order to optimise the behavior. Policy gradient based RL has produced another approach known as proximal policy optimisation Proximal Policy Optimisation (PPO) [22], which leans on the successes of earlier algorithms like Asynchronous Advantage Actor-Critic (A3C)[12] and TRPO. PPO combines ideas like trust regions and experience replay. Originally developed with the goal of being both simple to implement and configure, PPO claims results that are better or on par [21] with other RL methods. Another branch of algorithms known as actor-critic (AC) has several recent sub-types [12, 23, 13], each with different approaches in order to solve different problems within the paradigm.

The combination of learning or RL with BTs is not an entirely new concept. Some initial frameworks [5, 7] have been proposed which have shown interesting results. The common approach is to embed learning as a small RL problem in what is called a learning node [7] which substitutes an action with an entire RL model. There are also examples of learning control flow nodes [7], which decide when to execute action nodes or even entire sub-trees of a behavior tree. Learning based nodes can be tied to the Semi-Markov Decision Process (SMDP)[2] which can be used to analyse properties of such systems. Breaking a RL problem down into smaller sub-problems and solving those individually is known as Hierarchical Reinforcement Learning (HRL). Such a decomposition is directly applicable to BTs and can be viewed as a generalisation of the Markov Decision Process (MDP) in what is known as the option framework [2]. Similar methods have been applied in an approach known as Hierarchical Deep Q-Learning [11]. An approach to guarantee safety when using neural networks [25] inside BTs has also been proposed, which might be extendable to other types of learning.

2 Background

This section will give an overview of the underlying theory behind BTs (Section 2.1), RL (Section 2.2) and the combination of RL and BT (Section 2.3) which will be referred to as RL-BT.

2.1 Behavior Trees

BTs were originally developed in the video game industry with an emphasis on the maintainability and reuse of the code used for AI agents. The aspect of modularity was the key to improving the development process of such agents, meeting the needs of shared code ownership principles often applied in large software development projects. Since their utilisation in the field of robotics and research in general, more rigorous formulations have been developed. The theory described in this section follow the formalism presented in the book Behavior Trees in Robotics and AI: An introduction [15]. The remaining parts of this section list some of the most common patterns that are likely to be utilised in this project.

2.1.1 Behavior Tree Fundamentals

As stated in Section 1.1, the concept of BTs is primarily a way to structure the code of an agent such as to maximise modularity and re-usability of code in the interest of human readability, expressiveness and ease of design.

Formally, BTs are a special kind of tree structure. Like other types of trees, there is a node with no parents, known as the root node, which is the starting point of execution in a BT. In order to operate the tree a query signal known as a tick is generated at the root. The tick is propagated through the structure of the tree until a status of Running, Success or Failure is returned by a node. The Running status represents an ongoing process in a node that has not yet finished. The Success and Failure statuses represent the corresponding result of the node after it has finished executing. Once a value is returned from a node it is propagated back to its parent. Depending on the type of flow designed in the parent, it can either query another node or keep propagating the result back up the tree. A result will eventually be propagated all the way back to the root node, where a new tick is generated. This creates a cyclical loop, continuously repeating the execution of the tree.

The cyclical nature of the tick propagation is what allows BTs to be responsive to the environment. As every condition is checked again in every cycle, the control flow can change if anything has changed since the last iteration. Note that this is only possible if the tree is able to execute its actions in an incremental fashion. When an action is running, it is impossible for the tree to respond to other stimuli. This is the primary function of the intermediate Running status. It allows an action to take incremental steps towards the Success condition. This way the tree will take multiple iterations over the same action in order to reach a Success or Failure condition, while at the same time allowing the control flow to redirect control in response to a changed condition.

In a BT there are two primary types of nodes: control flow and execution nodes. The following two Sections, 2.1.2 and 2.1.3, will deal with execution and control flow nodes respectively. There is an additional type of node known as a decorator, which allows custom extensions of the basic node behaviors. Decorators will be further discussed in Section 2.1.4.

2.1.2 Execution nodes

Execution nodes are the leaf nodes of the tree. They are further separated into action and condition nodes as seen in Figure 2.

(a) Action node (b) Condition node

Figure 2: Graphical representation of execution nodes.

An action node is where all the low-level behaviors that can be utilised for controlling the agent are defined. These nodes contain the code specific to some sub-task that the tree needs to perform. Action nodes can have internal conditions for execution, which can also be made explicit using certain design structures (see Section 2.1.6). An action node will return failure if its execution conditions are not met or if the action fails in some other manner. If the process started is a longer process it can return Running to indicate that it has not failed nor finished yet. Only a finished execution of an action will return the Success status.

A condition node represents a predicate that is checked when the node gets ticked. Success or Failure is returned, representing the truth value of said predicate. Condition nodes always return either Success or Failure. Conditions are usually modelled based on the observations that the agent gets from the environment, but can also represent some internal state of the agent's memory. This allows the agent to decide, with the help of control flow nodes, which actions to execute in reaction to the changing world around it.

2.1.3 Control flow nodes: Sequence, Fallback and Parallel

Control flow nodes or simply control nodes are non-leaf nodes which have at least one child, usually more. They can be used to represent different relationships between nodes as well as sub-branches of the tree. In diagram form, the children of a control flow node are usually hierarchically displayed on the next layer, with all children of a single node sharing the same layer. Control flow nodes can have children of any type, including other control flow nodes. Control flow nodes are commonly separated into sequence, fallback and parallel nodes. It is commonly assumed that the children of a control flow node are executed from left to right in graphical representations.

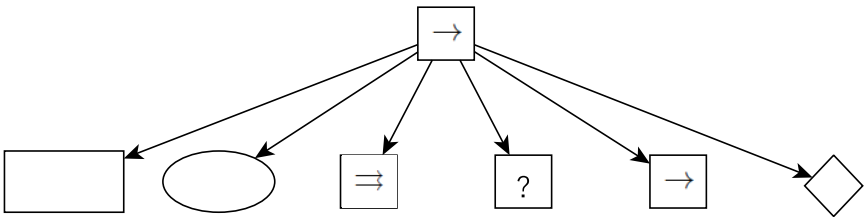

Figure 3: Graphical representation of a sequence node with six children.

Sequence nodes iterate in order until a Failure or Running status is returned. That is to say, they keep iterating over their children as long as they return Successes. The only way that a sequence node returns a Success to its parent is if every child has returned a Success otherwise the first Failure or Running status gets propagated back to the parent that ticked it. The primary purpose of a sequence node is to represent tasks that need to be taken as a series of consecutive actions. The formulation allows the agent to switch between branches and ignore the rest of the actions in a sequence if one of the prerequisite actions has failed. The algorithm of a sequence node is given in Algorithm 1 and a graphical representation of a generic sequence node is

given in Figure 3.

```
for child ∈ children do
    status = tick(child);
    if status = Running OR status = Failure then
        return status;
    end
end
return Success;
```

Algorithm 1: Pseudocode for sequence nodes.

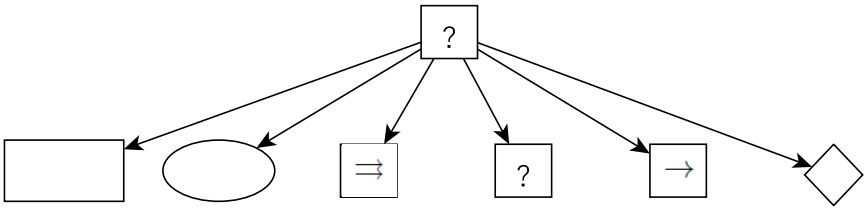

Figure 4: Graphical representation of fallback nodes.

Fallback nodes have a functionality inverse to that of the sequence nodes. They iterate in order until a Success or Running status is returned from a child node. In other words, they keep executing as long as nodes return Failures. The only way that a fallback node returns a Failure to its parent is if every child has returned a Failure, otherwise a Success or Running status is propagated back to the parent that ticked it. The intuition behind the use of this node is to guarantee that an action, a fallback, does get executed if one of its predecessors fails. A fallback node can be graphically represented as seen in the Figure 4 and Algorithm 2 shows the pseudocode of the algorithm for its operation.

```
for child ∈ children do
    status = tick(child);
    if status = Running OR status = Success then
        return status;
    end
end
return Failure;
```

Algorithm 2: Pseudocode for fallback nodes.

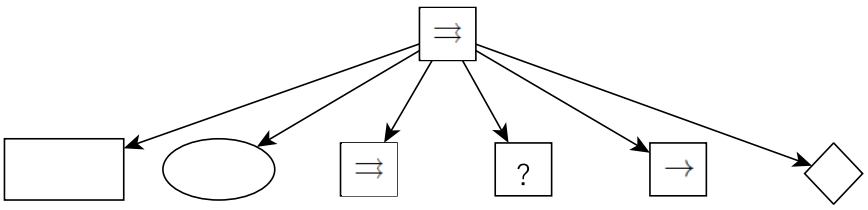

Figure 5: Graphical representation of parallel nodes.

Parallel nodes always execute all of their children. By default, parallel nodes will return running to the parent that ticked it. They return a Success if all children return a Success or a Failure if any child returns a Failure. Parallel nodes can also be modelled with a threshold parameter, which determines the number of Success or Failure statuses necessary in order for the node to return a corresponding status to its parent. Eponymically, parallel nodes are intended to be used for processes which are executed in parallel. This is another way of saying that the children of a parallel node have no precedence relationship to each other. Graphically, they can be represented as seen in Figure 5.

for $child \in children$ **do**
  $statuses \leftarrow tick(child)$;
end
if $\sum_i statuses(i) = Success \geq M$ **then**
  return $Success$;
end
if $\sum_i statuses(i) = Failure > N - M$ **then**
  return $Failure$;
end
return $Running$;

Algorithm 3: Pseudocode for M threshold parallel node with N children.

2.1.4 The Decorator node

A decorator node is essentially a mix between an execution and control flow node. It defines some custom policy that determines when the child node gets ticked, what the return value is or both. Decorators are useful for describing custom logic that cannot be easily achieved using the other node types. Definitions of the decorator can vary and are usually dependent on specific implementations. A common usage is to create an inversion node, which inverts the return status of the node from success to failure and vice versa. The graphical representation of a decorator node can be seen in Figure 6.

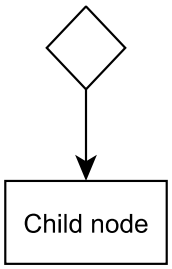

Figure 6: Graphical representation decorators.

2.1.5 Summary of nodes

A summary of the basic types of nodes in a behavior tree is given in Table 1. These basic building blocks allow for the construction of more complex structures and can be augmented with additional types of nodes, but in practice this is rarely necessary. With the proper level of abstraction in the tree, most scenarios should be solvable by applying good design patterns. Note that while BTs are excellent at both providing structure as well as visualising it, they still need an expert to define the details of actions, conditions and the tree structure. The only thing that changes is the structure of the code, by enforcing a specific control paradigm that helps organise the development of the agent. With this information at hand, we can now turn back

Icon	Node type	Success condition	Failure condition	Running condition
Action	Action	Action completed	Action can not complete	Action underway
Condition	Condition	If true	If false	N/A
◇	Decorator	Custom	Custom	Custom
→	Sequence	All children succeed	A child fails	A child returns running
?	Fallback	A child succeeds	All children fail	A child returns running
⇉	Parallel	M or more successes	N-M or more failures	Less than M successes or N-M failures

Table 1: Summary of common node types

to the example in Figure 1. It should now be clear that it is simply a specification of a sequential ordering of actions. That is, the agent tries to grab the bottle. If it succeeds, it will try to remove the cap, otherwise it will try grabbing the bottle again in the next iteration. If the cap is removed, it will finally fill it with water. It is important to note that a failure does not mean that the agent gives up for the rest of the execution. The action can be tried again and again until it succeeds, given that other conditions remain the same. As an extension one could also imagine another branch of the tree, which is able to execute some other behavior if the cap removal fails.

2.1.6 Explicit success conditions

Explicit success conditions are one of the most common design elements that arise out of the need for visualisation and human readability. Explicit success conditions pull the final conditions out of the action nodes and always represent them in the tree as a condition node or sometimes multiple condition nodes. This clarifies the goal of the action as well as guarantees that an action has a well defined execution condition. When following this pattern rigorously, it makes it less likely that an agent gets stuck repeatedly executing a particular action in a scenario that the designer did not expect.

As an example, consider Figure 1 again. While this instance of the BT might have some implicit notion of when an action is successful, it is not necessarily clear what

happens if an action succeeds or what happens next. This structure makes it clear under which conditions the sequence proceeds or returns to an earlier action.

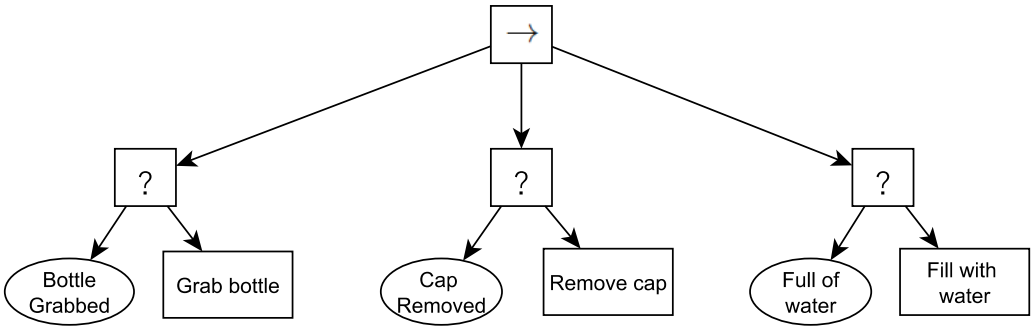

Figure 7: Example of explicit success conditions.

Now consider Figure 7. In this format it is possible to determine exactly when an action is to be executed as well as guaranteeing that an action will not be executed if its potential goal is already achieved. In addition to clarifying the structure of the tree and guaranteeing that unnecessary actions are not taken, this can also yield performance benefits depending on the design of individual action nodes.

2.1.7 Implicit sequences

Implicit sequences are a way of improving reactivity by prioritising the goal as opposed to having a sequence of consecutive steps. We achieve this by reversing the order of the actions and using a fallback over sequences with explicit success conditions. This pattern can also be applied to the previous example in Figure 7. Note that in implicit sequences, the conditions are shifted due to the reorganisation of the control flow.

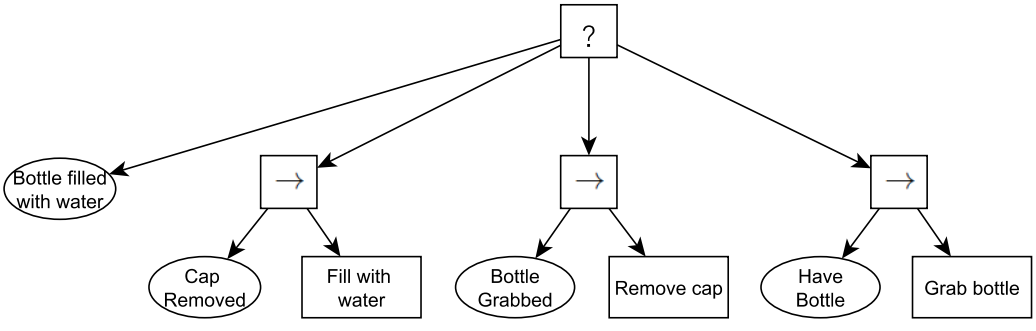

Figure 8: Example of implicit sequences.

The benefit over the previous design is that each action is implicitly tied to achieving the state where the primary goal can be executed. In Figure 8 the agent is trying to fill the bottle as its ultimate goal. First of all, we can check if this is even necessary and forgo the whole execution of the sequence if the goal condition is already met.

Subsequently, If the cap is removed, we need not check all the other conditions and actions to fill it and so on. Assuming that we fill the bottle by placing it on the floor anyway, there is no need to grab it again if the cap has been removed. By utilising this design principle we can avoid unnecessary actions entirely.

Note that each action could be replaced with an entire sub-tree of nodes, as long as it satisfies the condition required by the sequence. This allows modular expansion of the tree without changing other parts of the structure. The implicit sequence approach does not only improve reactivity and performance, but also matches a type of intuition of how humans actually lay out their tasks. We first try the obvious approach and then rely on fallback mechanisms if certain inhibiting conditions are met, instead of trying to iterate through every possible thing that could hinder our progress before continuing with the next step.

2.1.8 Back-chaining

Back-chaining is a continuation of the final idea of the previous section. What if the tree was to start out with a simple condition that we try to achieve? Based on a premise that each action can be tied to a precondition and a postcondition, we could simply enumerate all the possible actions with corresponding conditions. We could then dynamically expand the BT to generate the structure based on the pre-determined relationships between actions and conditions. This principle is known as Postcondition-Precondition-Action (PPA).

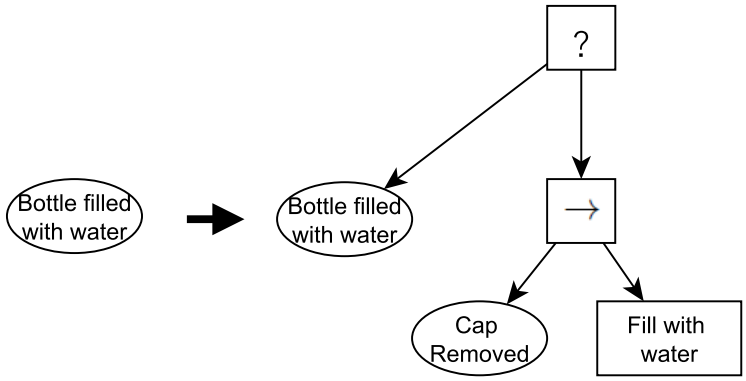

Figure 9: Example of back-chaining.

An example, based on the earlier ones, is given in Figure 9. The underlying assumption is that we have an enumerated set of actions with their post- and preconditions. If we were to start out with a simple condition such as *Bottle filled with water* we could find the necessary actions by their preconditions iteratively. In this example, the original condition is the postcondition for the action *Fill with water*, with its own precondition *Cap Removed*. If this new precondition is not true, we could expand the tree again until we get something that we can execute.

Once the tree has been created using PPA and back-chaining, it can be executed repeatedly. As long as there is a way to resolve it the tree does not have to be

modified and can be used as long as necessary. This does not prevent expansion of it later.

This pattern could also be of special interest for the specific problem of RL in BTs. One could potentially learn a new action as a response to a condition that cannot be satisfied with the current set of actions.

2.1.9 Other design considerations

There are more ideas that can be utilised in order to improve behavior tree designs. Artificial agents, especially in robotics, often have critical behaviors which can have potentially disastrous effects if they are executed incorrectly. The large state and action spaces often encountered in RL make agents harder to test thoroughly, and liable to exhibit unwanted behavior. Luckily the very structure of BTs overcomes many of these problems by applying the right design.

Safety can be guaranteed by using conditions as safeguards in combination with sequences. By adding an additional condition before a sequence, we guarantee that it only gets executed if some specific criteria is met. This is also of potential use in combination with RL. While we might not always know what we want the learned behavior to do, we might want to set up specific limitations regarding what we do not want it to do. This can help guarantee that the agent will not behave in an undesirable manner. Decorator nodes can also be useful for meeting safety requirements.

Even though the fundamentals of BTs are relatively simple to understand and implement, one of the essential questions when designing behaviors should be about the level of abstraction at which actions should be implemented. As suggested in [15], the best way to assess proper granularity is by utility. If there are parts of an action or condition which could be reused in other parts of the behavior tree, it should be broken down into multiple parts. If there are specific parts of a sub-tree that are only ever used in one specific scenario, it might be reasonable to combine them into a single node. It is also useful to consider how many different internal phases an action could have. If a node requires multiple internal steps or even a state-machine to represent, it is probably better to split it up. This also matches the idea that it is better to avoid states held in memory, as it can reduce responsiveness. Ideally, the only state that the BT needs can be observed from the environment. The level of abstraction will also be relevant for RL-BT as the formulation of a RL problem will be tied to the granularity of actions available in the BT.

Outside of design and structuring considerations, there are also purely implementation related questions that should be answered. A naive implementation of BTs might propagate the ticks sequentially until an executable action is queried and executed in the same thread. It is suggested in [15] that in many cases a parallel implementation of the ticks and actions is preferable. This allows the control structure to be queried independently of the action executions, allowing for greater reactivity. Contrast this with the situation where a single action that takes a long time to execute blocks the execution of the rest of the tree and hinders the reactivity of the agent. Parallelism is one way to work around that problem. Another solution might be to ensure that

each action can be executed in a step-wise fashion every time it is queried, but that might not always be possible.

As a closing word on BTs, it is worth noting that none of the aforementioned designs are a complete solution, nor do they supersede each other. Each of the suggested patterns are useful in different scenarios and careful consideration should be taken to determine the best one for a given situation.

2.2 Reinforcement Learning

As briefly mentioned in the introduction, the RL problem is about maximising rewards by acting in an unknown environment. That is to say that there are two primary parts to the problem: the agent and the environment. The agent acts in the environment by choosing actions. The environment provides observations and rewards to the agent. Rewards are scalar values, indicating how good it is to transition to a particular state.

A RL problem can be modelled as a history H of actions A, states S and rewards R. The return G in RL is defined as the combined reward of all future time-steps, discounted by the discount factor γ. The discount is a value between 0 and 1. The discount factor gives a preference to short-term reward and increasingly discounts the further we look into the future. This is used for modelling uncertainty, bounding the maths and other various reasons specific to application scenarios.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-2} R_T \tag{1}$$

The descriptions above can be formalised by introducing what is known as the Markov Reward Process (MRP) and Markov Decision Process (MDP). First of all, a Markov Process is defined as a process where the probability of ending up in the next state is only dependent on the current state S_t as opposed to the whole history. Formally this is expressed in Equation 2.

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, .., S_t) \tag{2}$$

The Markov Reward Process is the underlying principle for the idea of a value function in RL. The Markov Reward Process is summarised as by adding a reward for transitioning to each state and maximising that reward for the next as well as all future states as in the form of a Markov Process. This can be formally expressed through the Bellman equation (3). Note that there are many forms of the Bellman equation which are derived from the same basic principle based on a similar recursive approach, combining the reward from the immediate next step with all future rewards.

$$\begin{aligned} v(s) =& E[G_t|S_t = s] \\ =& E[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ =& E[R_{t+1} + \gamma v(S_{t+1})|S_t = s] \end{aligned} \tag{3}$$

Equation (3) simply states that the return from a given state s is the sum of the reward from that state and the discounted sum of rewards for all future states. This gives the recursive definition of a value function v. Concisely, the value function is the expected future reward we can gain by acting according to a certain policy.

A similar definition can be used for the Markov Decision Process MDP. The MDP introduces another useful concept known as the policy in RL. The primary difference is that in addition to states and rewards, we also take the actions themselves into account when determining the next move. A policy π defines the behavior of an agent as a probability distribution over actions with respect to a state $\pi(a|s) = P(A_t = a|S_t = s)$. The MDP can be used to give the definition of the state-value function Eq. (4) as well as the action-value function Eq. (5).

$$v_\pi(S) = E_\pi(G_t|S_t = s) \tag{4}$$

$$q_\pi(s, a) = E_\pi(G_t|S_t = s, A_t = a) \tag{5}$$

The state-value function v_π for a policy π is the expected immediate return from state s and then acting according to policy π after that. Similarly the action-value function q_π is the expected return from state s, when taking action a and acting according to policy π after that.

There are also additional forms of the Bellman equation based on the Equations (4) and (5) which are also relevant to RL. The two primary formulations are known as the Bellman Expectation Equation and the Bellman Optimality Equation.

The Bellman Expectation equation gives the recursive definition of the expected values for the state-value and action-value as seen in Equation (6).

$$\begin{aligned} v(s) =& E_\pi(G_t|S_t = s) \\ =& E[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \\ q_\pi(s, a) =& E_\pi(G_t|S_t = s, A_t = a) \\ =& E[R_{t+1} + \gamma q_\pi(S_{t+1})|S_t = s, A_t = a] \end{aligned} \tag{6}$$

The Bellman Optimality equation derives from a representation of the state and action value functions as the maximum value over all possible policies, given in Equation (7). This means that we make the best possible choice that we can make under an optimal policy at every time step.

$$\begin{aligned} v^*_\pi(S) =& \max_\pi v_\pi(s) \\ q^*_\pi(s, a) =& \max_\pi q_\pi(s, a) \end{aligned} \tag{7}$$

The Bellman Optimality Equation gives a recursive definition similar to its previous instances. It optimises the value of either the optimal action-value or state-value functions as seen in Equation (8).

$$
\begin{aligned}
v_*(s) &= \max_a q_*(s,a) \\
q_*(s,a) &= \max_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \\
&= \max_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_a q_*(s',a')
\end{aligned}
\tag{8}
$$

This gives the definition of the optimal value function as choosing the action that maximises q^* over all of the possible actions. The optimal action value function q^* is defined similarly to earlier instances of the Bellman equation. It is the sum over the immediate reward and the discounted optimal return by following the best possible policy from there. In Equation (8), $P_{ss'}^a$ represents the transition probabilities from state s to a future state s' when taking the action a. This is also the most well known form of the Bellman equation in RL.

Note that all the RL algorithms essentially replace one or more term in one of the equations above with some other more specific form, but they are all different ways of solving the same problem. As all of these functions need to be determined from unknown dynamics, the solution is typically a function approximation which can range from simple look-up tables to linear combinations and neural networks.

There are a few well known open problems to keep in mind when dealing with RL. The best known one is the exploration and exploitation problem. The core aspect of this is about the trade-off between using the information we have learned so far and exploring new areas for more potential information. In RL terms it is the choice between maximising reward given our learned knowledge and getting less reward in the short run with the potential of higher rewards in the future by exploring more of the state space. There are different ways to solve this problem. Many algorithms make use of the ϵ-greedy method, which essentially assigns a probability ϵ to taking a uniformly random action instead of what the current policy considers the best action.

Additionally it is worth noting the credit assignment problem a.k.a the blame problem. This is essentially about determining which of the actions that were taken by the agent, actually lead to the reward being received. This problem has lead to a plethora of classic approaches ranging from Temporal Difference (TD) with different time-frames (TD(0),TD(λ)) to Monte Carlo (MC) learning. The core idea behind these earlier algorithms is the choice between whether we ought to take the last step (TD(0)), some intermediate range of steps (TD(λ)) or the whole process (MC) leading up to the reward into account when adjusting our function approximator.

There is a myriad of other issues that can arise in RL, but those are usually specific to individual approaches. Many of the algorithms discussed in the rest of Section 2.2 are usually derived as a solution to one or more of the problems above.

2.2.1 Classification of Reinforcement Learning algorithms

Reinforcement learning can be broken down as model free and model based. When talking about model based RL it is important to differentiate between knowing the full dynamics of the system where the agent is operating or the situation where the agent is trying to learn a model. In the first case, deterministic algorithms and dynamic programming is usually sufficient to solve the problem. When trying to learn a model and then determine the dynamics based on the learned model, an additional step is introduced. However the final solution is still similar to the model free scenario albeit with more uncertainty. Regardless, most of the RL research is oriented towards a scenario where the model of the environment is unknown and we are trying to learn in a model free way to behave optimally.

An additional distinction can be made between on-policy and off-policy methods. On-policy methods are methods which act under the same policy which they are trying to learn. That is to say that the agent acts under the same policy that it is updating as it learns. Off-policy methods are essentially the opposite. The behavior is produced by a separate policy from the one that is being learned and updated. There are some caveats to this, as some algorithms which simply use an older version of itself but don't directly update the function they are behaving under (DQN) are still considered off-policy.

Separate from the model and on/off policy dichotomy, another set of classifications can be made between value based or policy based RL. Methods which combine the two are known as actor critic methods.

Value based learning algorithms try to predict and optimise the state-value function. Once the value function approximation is found it is usually used to act in a greedy way in order to maximise gains in the environment. The primary modern example of value based learning is DQN.

Policy based learning tries to directly approximate the optimal policy by learning a parametrisation of an approximation function that tries to maximise the action-value function q. In some cases it is more efficient to represent the policy rather than the value function. These types of algorithms tend to be more useful in continuous spaces as they handle infinite spaces better than value based algorithms which are usually more computationally expensive. They also circumvent the requirement to act greedily as the agent is optimising over long time return according to its policy. On the other hand value-based algorithms can be a lot easier and faster to train if the state space is well defined. Well known examples of policy based RL algorithms are PPO and TRPO.

An actor-critic is a combination of the previous two approaches. In this scenario, the algorithm is simultaneously learning two functions. The value function i.e the critic, is used to evaluate the performance of the policy approximation, the actor. The actor is essentially using the critic's estimate in order to update its own. There are many different versions of actor-critic, which will be discussed further in Section 2.2.3.

2.2.2 Deep Q-learning

DQN is the latest iteration of a set of similar algorithms that improves on its predecessors SARSA and Q-learning with two primary additions, experience replay and a deep network for function approximation.

Experience replay means that the agent's experiences are stored in a vector of state, action, reward, next state and added to a dataset, forming the replay memory. It is then possible to iterate over batches of stored experiences in order to increase the efficiency of the sampled states. DQN has to learn off-policy due to experience replay, so that the parameters during the update will be different from the ones used to generate the sample. Otherwise the parameters will completely determine the next data sample that the agent will be trained on.

A frozen target network means that while the behavior policy is updated, an older version of the learned policy is used for the majority of learning time. The behavior policy is updated to the latest learned one after a fixed amount of steps. DQN is considered an off-policy network as it does not directly update the policy it uses to generate its behavior.

It is one of the first general purpose methods that is able to continually adapt its behavior without any human intervention. The generality of the algorithm is definitely of interest with regards to BTs. It is also one of the more commonly used methods in available research for the combination of RL and BT [5][9][7].

2.2.3 Actor-Critic methods

One of the more common actor-critic methods is named Asynchronous Advantage Actor Critic [12] abbreviated as A3C, sometimes A2C when the asynchronous elements are not utilised. A3C differentiates itself from classical actor-critic methods by utilising multiple asynchronous workers. A worker in this scenario is like a short lived actor. It uses many fixed-length experience segments in order to compute its estimations. It also features architectures where the actor and critic networks share layers with each other. According to OpenAI, their own implementations perform better in the A2C configuration rather than the asynchronous A3C configuration [20].

There are additional versions of actor critics like Sample Efficient Actor-Critic with Experience Replay (ACER) [13] and Actor Critic using Kronecker-Factored Trust Region (ACKTR) [23]. Both of these add some additional methods on top of the usual actor-critic approach. ACKTR is, eponymically, a novel way to utilise trust regions originally proposed in TRPO. It also uses the natural gradient as opposed to the first order gradient, resulting in a better sample complexity. This means that it needs fewer samples i.e fewer steps in the environment to converge on a policy.

ACER on the other hand combines multiple ideas from other algorithms. It uses multiple workers like A3C, experience replay dervied from DQN and other ideas like trust regions, retrace for Q-value estimation and importance sampling. It has

improved performance over many other methods but is also known as one of the most complex algorithms to implement.

2.2.4 Policy gradient methods

Policy gradient methods tend to be sensitive to step size, causing the learning process to become too slow or diverge entirely due to noise. They also take millions of timesteps to train due to bad sample efficiency if approached in a naive manner. This can cause them to be infeasible even on simple tasks, if care is not taken in parameterisation and space modelling.

One of the better known improvements over classical policy gradient methods was presented in Trust Region Policy Optimisation (TRPO) [8]. The main concept of this approach is to simply constrain the policy update to some trusted (fixed) region.

Proximal Policy Optimisation was from its inception intended to be an algorithm that is easier to implement, parametrise and train by the researchers at OpenAI. PPO [22] combines ideas regarding trust region from TRPO [8] and multiple workers from A3C [12]. It uses an adaptive Kullback-Leibner (KL) penalty for training. KL is a measure of how different two probability distributions are from each other.

2.3 Reinforcement learning and Behavior Trees

The existing literature commonly defines two types of nodes, sometimes termed learning nodes, with respect to RL. The common dichotomy is between learning represented as an execution or control flow node. Such nodes are further discussed in Section 2.3.1.

The use of RL in BTs can be modelled [7] with what is known as the Option Framework [2]. In this approach, actions in a behavior tree can be viewed as options in a Hierarchical Reinforcement Learning (HRL) problem. This approach is further discussed in Section 2.3.2.

2.3.1 Reinforcement learning nodes

A learning control flow node, sometimes called a composite node [7], can be seen as an extension of the normal BT control flow node. This node uses its child nodes, usually action nodes, as the action space for an RL algorithm. Nodes like this yield the benefit of maintaining the readability and modularity of individual actions in the tree. They also allow for more complicated structures to emerge by creating hierarchies of learning control flow nodes, if allowed to decide between sub-trees in the behavior tree rather than simple actions. This type of structure also shows promise for interaction with the design principles listed in Section 2.1.

Another version of a learned fallback node is presented in [9]. In this case the children of a fallback node (selector node in the article) are used as an action space and

reordered according to action probability. The idea is to give a reward to the agent every time a condition from the original tree is triggered and then reorder the fallback node according to the Q-values for that particular state before querying through the node. This paper also proposes an interesting approach of reward propagation, which returns the reward of lower level nodes up to higher level ones alongside the BT status.

Learned execution nodes [7] have been used to embed the whole RL problem in a single action node and run it as a subroutine inside the tree. These nodes are usually less reactive and harder to control in relation to the rest of the behaviors exhibited in the tree. However with the proper level of abstraction these types of nodes have shown potential for application when learning is desirable only for relatively small and specific actions. A way to fine tune control or ensure safety of such nodes could be with the help of decorator nodes, but such experiments have not been found in the available literature.

A condition node centered Q-learning approach has also been proposed in an approach named QL-BT [5] by the authors. In this approach, the QL-BT algorithm finds all the lowest level sequence nodes inside the behavior tree and uses them as the action space for the tree. These actions are then used for training and a Q-value table is generated. Once learning has taken place, each sequence is assigned what is called a Q-Condition node. Each Q-Condition has a look-up table of states which correspond to a good utility (high reward) for its corresponding action. The tree is then sorted into descending order according to the Q-values of each action and condition pair. While the QL-BT approach is interesting from a more general RL-BT perspective, it is not so useful for this project's purposes. This thesis is primarily oriented towards using RL as part of BTs and learning new actions in addition to control flow. QL-BT additionally requires a special approach and algorithm for embedding RL in BT, which is somewhat limiting and not the purpose of this thesis.

Both [5] and [9] have different details to their operation, but end up being similar in functionality. We would note that there is little difference between the proposed algorithms and simply training a model which picks the best action according to a state. Matching condition nodes to actions and then checking if the state matches the condition is the same as simply selecting the best action according to state. The primary difference is that the Q-conditions can potentially execute multiple actions from a state, if it ends up matching for more than one condition. A similar principle applies to [9], as a sorted fallback node is the same as executing the highest probability action and choosing the next highest probability if the action returns a failure. It seems that either one of the above algorithms are not necessary, as we can always simulate this by executing any number of actions from a normal RL model. This can be done according to the probabilities for each action according to state. For example having a set number of highest probability actions to execute or executing the actions according the success and failure conditions following either the fallback or sequence node principle.

2.3.2 Hierarchical reinforcement learning and Semi Markov Decision Processes

HRL has shown promise in application to large and complicated RL problems by solving them as sets of smaller problems instead of one large problem. Decomposing the problem into multiple smaller ones can reduce effects of the curse of dimensionality as the complexity of models increases non-linearly with the size of the observation and action spaces. This is done in [7] with the help of what is known as the Option Framework [2].

A problem faced with HRL during the training is that of temporal abstraction or lack thereof. The classical MDP representation of RL does not feature temporally extended actions. That means that each action is executed for one time-step after which a reward is given and another action is chosen. In a context where there might be many nodes in a BT that are learning simultaneously, this can cause learning to become unstable. If an episode is just any set of consecutive ticks, the states that the node sees could be completely unrelated.

The option framework takes an RL action and extends it in time, such that it executes for multiple timesteps. Each action usually has a termination condition upon which the action ends and a new one can be chosen. Any action can execute for any number of steps, even a single step, and receives a reward for each step even if the action has not terminated. In the context of BTs the termination condition can be tied to the status types of failure or success, while the running condition continues the execution of the same action. An option framework needs to be able to evaluate when the action terminates as well as how much reward it predicts to gain along the way before termination. This makes the implementation less general-purpose than desirable for the purposes of this thesis.

As mentioned in Section 2.1.9, it should not be necessary for an action in a BT to execute continuously and can instead do so in steps. This means that while the temporally extended options can be useful in some circumstances, it is also possible to model an episode of RL as a consecutive series of uninterrupted ticks to a node. This means that if a node is not ticked for a timestep, that is considered the end of an episode and the next time the node is ticked an new episode is started. This principle can hypothetically be used in a RL-BT learning node to represent learning in a HRL context, similarly to SMDP and the Options Framework.

2.4 Summary

BTs have been extensively researched and formalised over the past decade. This provides a comfortable theoretical base with a plethora of design patterns and implementation guidelines. Reinforcement learning is a quickly evolving field, and large research groups like OpenAI and DeepMind have provided some formality and implementations that allow experimentation with the newest methods with relative ease.

While both the RL and BT problems have strong theoretical and practical foundations,

the work on their combination is thus far very limited. There are some relatively simple formulations that have been researched, but no extensive work has been done on the subject. This project aims to experiment with different modern algorithms in order to determine their suitability for different use cases in BTs. In addition it will attempt to recreate and evaluate the learning nodes proposed in some of the previous projects and attempt new methods of implementation.

3 Experimental setup

The experimental environment is set up in Python3. The choice of programming language was informed by the extensive amount of scientific research resources available in the field of learning. Two primary external libraries were used in setting up the experiments: Stable-Baselines [24] and Project Malmö [10]. Minecraft is also necessary as a part of Malmö, which is the primary simulation environment for the experiments.

A custom module was created to facilitate the BT implementation and support easier definition of experiments. The details of the BT implementation will be discussed in Section 3.3. Additional abstract classes were implemented in order to decrease the boilerplate code that would otherwise be necessary for each individual agent. These implementations were created for both Malmö and Stable Baselines and will be further discussed in Sections 3.1 and 3.2 respectively. The python environment was set up inside an Anaconda virtual environment.

The experiments were run on a Windows 10 Home 64-bit platform with Nvidia Cuda 9.0 and Cudnn 7.2 support. The system features an Intel(R) Core(TM) i7-6700 8-core CPU running at 3.40GHz, a NVIDIA GeForce GTX 970 GPU and 16GB of physical main memory.

The previously mentioned libraries and environment are the basis for the experiments.

3.1 Minecraft and Project Malmö

In order to provide a simulation and experimentation environment an open-source project developed by Microsoft, called Project Malmö [10], will be used. Based on the video game Minecraft, Project Malmö has been used for several research projects, is well documented and provides a configurable platform for AI related experiments [18], including machine learning [19, 14].

As an interface for Minecraft, Malmö allows access to different types of observations and control over an agent that are necessary for experiments with BTs and RL. The goal-free sandbox nature of Minecraft defines no intrinsic victory condition or idea of what it means to do well in the game. In this sense Minecraft is like a small abstraction of the real world, with freedom to set your own goals within the limits of the mechanics provided by the game. This freedom for defining problem scenarios and experiments sets it as a suitable platform for the scope of this project.

Malmö was originally implemented in C++ and has to be compiled as such. However there are multiple pre-compiled versions with interfaces to different programming languages, including Python. As Python is also the environment for Baselines, it provides an ideal combination with other implementations in order to provide all the necessary elements in a unified experimentation environment. In order to facilitate easier experimentation within Malmö, the existing Python interface provided by Microsoft was augmented with abstract agent and mission classes. This was done in order to avoid the repetitive appearance of boilerplate code that is needed in most experimental scenarios, as well as simplify the interfacing of the primary libraries.

Malmö operates as a interface between an instance of Minecraft and whatever code platform is being utilised for controlling the agent. It allows setting up scenarios, known as missions, that can be customised in every aspect. It is possible to change the structure of a level as well as the characters inside it. A level can be built manually or generated randomly depending on the requirements of the user. There are a variety of other modifications that can be used, most notably changing the amount of time before the internal game environment progresses to the next state. This allows to speeding up the game-world in order to increase the pace of experiments.

The world itself is made up of discrete cubes called blocks. These blocks can be of a variety of different materials that could be encountered in the real world, such as sand, wood, dirt and stone all the way up to lava and obsidian. All blocks can be mined in order to remove them from the world. Mining is done by repeatedly clicking a block with bare hands or with specific tools which make the process faster. The different materials can be gathered after mining in order to craft items that can be used in order to boost different abilities of the agent. They can also be processed an then be placed back in the world, allowing the agent to build new structures or assist exploration. For example, blocks of logs can be crafted into planks and used for building.

While the structure of the world itself is discrete in the form of blocks, not everything inside it is. The player, or agent in terms of Malmö, moves in a continuous manner. Non-Player characters (NPCs), items and blocks that have been removed from the structure of the world, known as entities, follow the same principle. The game uses an internal coordinate system described in Figure 10. The agent also has a pitch variable that changes the camera angle up and down. Yaw is used to track the horizontal camera position. The yaw accumulates as positive or negative according to the side the agent is turning to. Note that the variables x y and z change in a discrete manner for each block in the world, where an integer value describes the bottom-left corner of a block. When entities are described in a continuous form, they will have an added real value in the range [0,1). For example the coordinates for the centre of any block could be given as $[x + 0.5, y + 0.5, z + 0.5]$, where x, y and z could be any integer valued coordinate.

Functionally, Malmö provides the same type of control to an agent that a player would have, alongside some additional features for more specific experiments, such as discrete commands for checkerboard-like movements. In terms of observations the interface is highly customisable allowing for a variety of observation types ranging

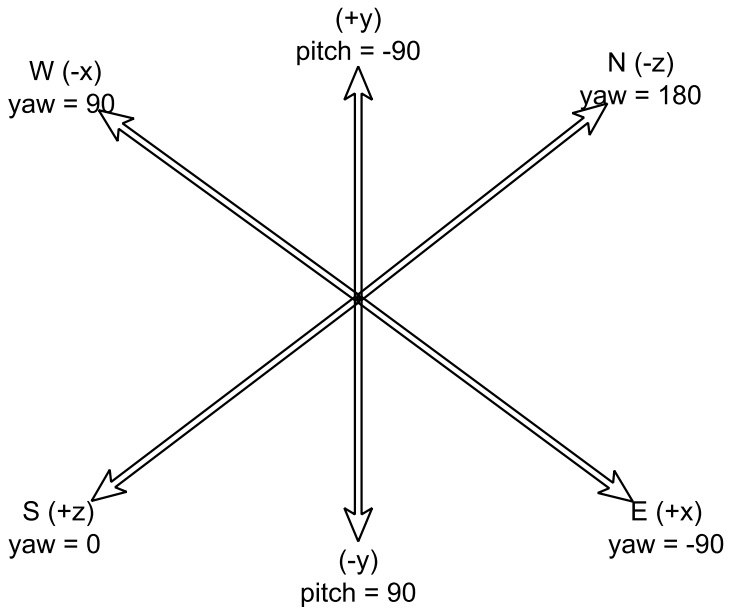

Figure 10: Coordinate system used inside project Malmö.

from the pixels being displayed on-screen to in-game information about the location and status of different entities or blocks. These observations can be set up to be gathered in a defined area around the agent, or from other areas inside the world. Due to performance implications, all the observations are disabled by default, but can be added to the agent at the beginning of a mission. There are other relevant observations such as the contents of the agent's inventory and equipped items. In addition to actions and observations, Malmö provides an interface for defining rewards that will be checked for and awarded by the game engine. These rewards will be returned alongside the rest of the observations to be utilised as desired by the designer. The usage of these is not mandatory and can be sidestepped with manual definitions of rewards based on observations. However, they provide an excellent means to define termination conditions for missions as well as rewards for damaging entities or reaching certain positions which would otherwise have to be scripted manually.

For this project, the agent generally uses continuous actions as it is the most common way to interact with the game. The specific actions and observations that will be used are described in detail for each experiment.

3.2 Reinforcement Learning implementation

The basis for the RL implementations in this thesis are the algorithms from OpenAI's open-source project called baselines [17]. There is an extension of this repository called Stable Baselines [24], which provides documentation and a convenient interface for research and experiments. Baselines has implementations for all the RL algorithms described in Section 2.2.

The Baselines and Stable Baselines packages are based on what are known as *Gym* environments. In order to use the baseline RL algorithms out-of-the-box, the environments need to implement the interface presented in Listing 1. The interface features two primary functions related to learning: step and reset. The step function is intended to manually step the environment forward, according to whatever dynamics are present in the simulation. After taking an action and stepping the environment, the method is expected to provide the next observation and reward for the previous action. This makes the interface convenient for training a single policy, or nodes in the context of this thesis, but makes it impossible to train multiple policies simultaneously. Even though training is limited to a single node, once the it is trained it can still be embedded in a BT and multiple policies can be utilised in a tree once they are trained. Additionally, the interface also features a render function for visualising the environment. In our case, this method is simply ignored as it is an additional feature that is not required for training and the simulation is already visualised in Minecraft.

Listing 1: Gym interface

```
import gym
from gym import spaces

class CustomEnv(gym.Env):
  def __init__(self, arg1, arg2, ...):
    ...
  def step(self, action):
    ...
    return observation, reward, done, info

  def reset(self):
    ...
    return observation

  def render(self):
    ...
```

The main issue that might not be directly apparent from observing Listing 1 is a control conflict with Malmö. As Malmö and Minecraft are run in separate applications, there is no direct control over the simulation environment from the agent's controller in Python. This means that we cannot pause and step through the simulation, but we can still provide observations and rewards at discrete timesteps by querying the Malmö environment after taking an action inside the step function. This is usable as long as we use the interface for training a single RL model at a time, as mentioned previously. For the pattern where the Gym environment controls the flow of the mission, the step and reset functions were implemented roughly as described with pseudocode in Listing 2. However, in some cases the pattern was too restrictive and had to be reworked, as we cannot wait for the environment to progress every time we do something in a node. In order to overcome this the RL algorithm step function had to be split into an action step and and an observation step. This means that while the tree is being ticked, all the learning nodes get to take their

actions without waiting for an observation and reward. The observation and reward is instead provided to all the learning nodes simultaneously before the next tick is started.

Listing 2: Gym controlled interface pseudocode

```
def step(self, action):
  mission.take_action(action)
  observation, done = mission.get_state()
  reward = mission.get_reward()
  return observation, reward, done, _

def reset(self):
  mission.restart()
  observation, _ = mission.get_state()
  return observation
```

Baselines models are generally initiated with what are defined as spaces, which describe the abstract structure of the problem to be learned. There are 4 primary types of spaces: Discrete, Box, Multi-Discrete and Multi-Binary. Any space can be used for both actions or observations as long as it is supported by the corresponding RL algorithm. Discrete spaces are essentially enumerations of possible values. They are given a set number of possible values upon definition, which can be mapped to the environment. Box spaces are continuous vectors that are defined within a specific range. Multi-Discrete and Multi-Binary are special categories which allow to define multiple discrete or binary (as an edge case) values in a single space. They are separate primarily for performance reasons as all the spaces can be combined with special Tuple or Union objects provided by the Gym environment.

It is also possible to utilise different types of networks as a parameter to any of the models. The network itself can be chosen from default options or defined manually in Tensorflow, which is the underlying architecture that facilitates the learning. Beyond that, the RL models are highly configurable with any desired hyper-parameters specific to each algorithm. Tensorboard support was enabled to monitor the training process for all experiments.

3.3 Behavior Trees implementation

All the common BT node types described in Section 2.1 were implemented for use in the experimental environments. There is an open-source implementation of BTs in C++ [16] made available by researchers at KTH, which provided some useful insights for the Python implementation in this project.

Custom nodes were also defined to facilitate recurring functionality in the experiments. Such nodes include learning nodes like the Learning Execution node described in Section 4.1 and Learning Control Flow node in Section 5.

An additional symbol for sub-trees is used in the graphical representation of the BT diagrams in this report. Seen in Figure 11, the graphical representation is shaped

like a downwards widening action node in order to represent the expanding nature of a sub-tree. This representation is used primarily to decompose the graphical representations into smaller pieces.

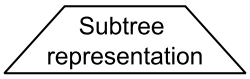

Figure 11: Graphical representation of sub-trees

Learning nodes also received their own graphical representation. Nodes that include learning are marked with a * symbol. Examples of these nodes can be seen in Figure 12

(a) Action node (b) Control flow node (c) Sub-tree

Figure 12: Graphical representation of learning nodes

4 Learning execution node

This section will focus on answering the sub-questions Q1, Q2 and Q3A.

As described in Section 2.3.1, a learning execution node is essentially an action node that is executing a RL model in order to determine its actions. It is the simplest form of RL that can be embedded in a BT as it concerns only a single node without children. This experiment will compare a manual behavior to learned DQN and PPO based ones, as they represent value and policy based algorithms respectively. Additionally, PPO also supports multi-discrete actions, meaning that it can map to multiple simultaneous actions not just the single discrete action supported by DQN. The configurations described in this section will be run with both discrete and multi-discrete actions spaces. This scenario will also investigate the difference between using BT based conditions directly as opposed to underlying continuous measurements.

4.1 Learning execution node experiment setup

The goal of the agent in this scenario is to fight a single skeleton while taking minimal damage. The basis for evaluation is a manually designed model. The manually designed tree is shown in Figure 13. The agent first moves closer to be in attacking range, but not too close in order to avoid being hit back. It then iterates over moving left and attacking.

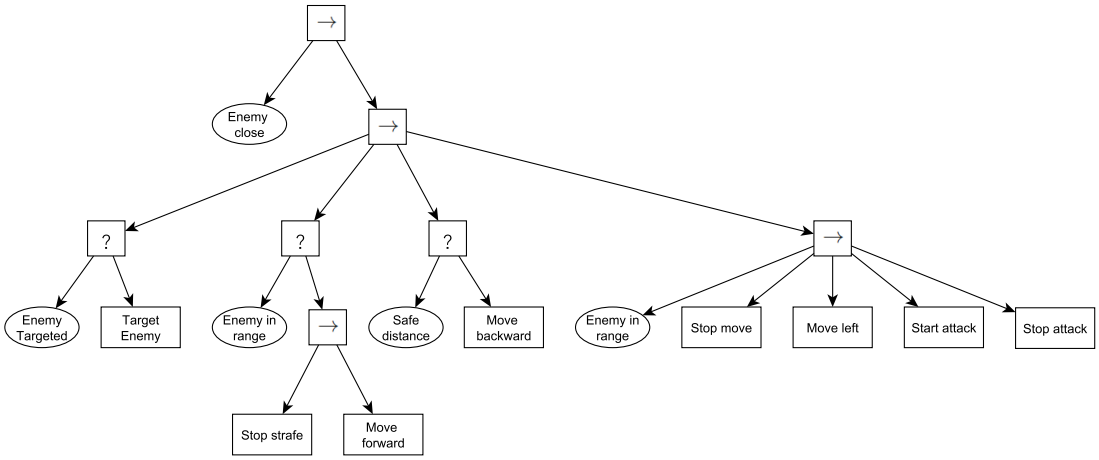

Figure 13: Manual tree definition for execution node experiment

The environment is a flat empty grassland in order to avoid problems with navigation that we are currently not interested in. The agent and the skeleton start 10 blocks apart. An example of the environment can be seen in Figure 14.

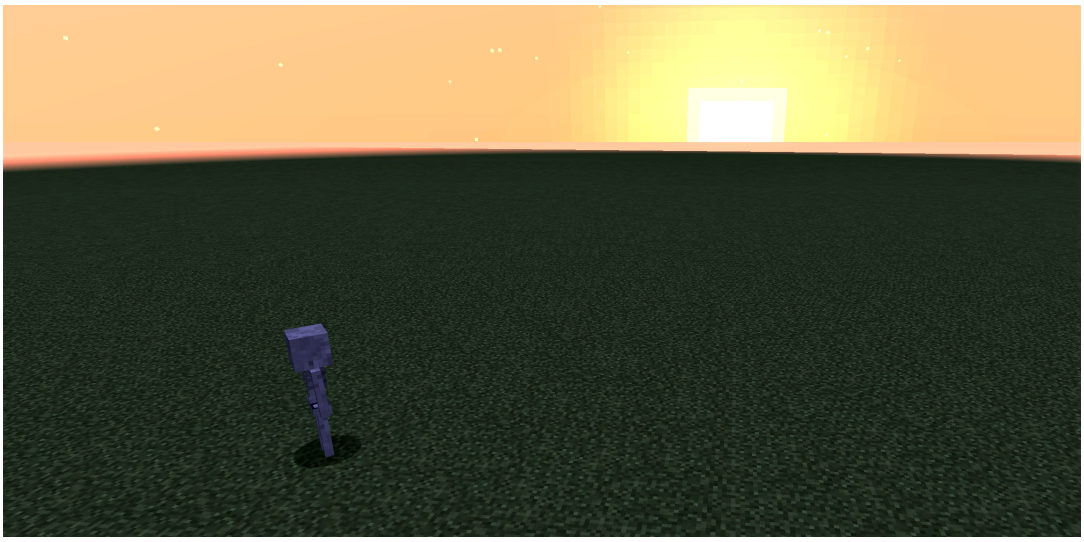

Figure 14: The environment for the execution node experiment. A single skeleton opponent in a flat grassland.

The actions and conditions in the tree are the minimal set of behaviors that were deemed necessary in order to successfully accomplish the fighting task. An important thing to keep in mind when assessing the tree is that commands in Malmö have a continuing effect, in the sense that a *Move forward* command will execute in Malmö until a *Stop move* command is given.

The actions are roughly divided between movement and the attack commands. There are three actions for lateral movement *Move right* and *Move left* and *Stop strafe*, of which only the first is not in use for the manual tree, but included in the learning node's action space. A similar set of commands exists for forward and backward movement with the actions *Move forward*, *Move backward* and *Stop move*. The

Target Enemy action is used to set the pitch and yaw of the agent so that the agent is looking at the skeleton. *Start attack* and *Stop attack* start and stop the attack animation. The reason why these are separated into two actions, is that in Minecraft the time between attacks matters, as attacking too quickly reduces the amount of damage dealt by the agent. So while separating the two is not strictly necessary for the manual behavior described in Figure 13, it could potentially be something that the learning agent is able to optimise.

In most configurations in this experiment the action space is a simple discrete space over all the possible actions as an enumeration. In the multi-discrete configuration, there are three sub-spaces. The spaces (discrete sets) are defined as follows:

- {Move forward, Stop motion, Move backward}
- {Move left, Stop strafe, Move right}
- {Start attack, No operation, Stop attack, Target enemy}

This should allow the multi-discrete configuration to behave more like the manual case, which has the technical benefit of being able to execute multiple actions in a command cycle.

The condition *Enemy close* is meant as a pre-condition for the tree, that makes the tree usable as a branch in other scenarios where enemies might be encountered. It adds no additional functionality to the tree other than stopping actions after the skeleton is defeated. *Enemy targeted* works based on ray-casting from the agents viewpoint and determines if there is anything in its line of sight. If a skeleton is detected it returns True, otherwise it is False. *Enemy in range* and *Safe distance* are based on the same internal measurement, which is the euclidean distance from the skeleton. The difference is that *Enemy in range* is True if the agent is at least 3.5 distance units from the target and *Safe distance* is True if the agent is farther than 2.5 distance units from the target. The first of these two condition is intended to guarantee that the agent is in range to hit the skeleton and the second that it will back away if it gets too close, making it harder for the skeleton to hit back. The lateral movement is in place for the same reason.

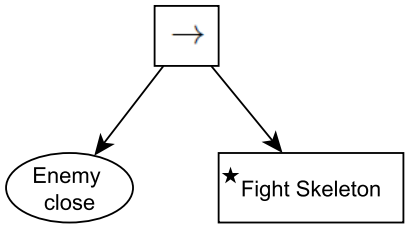

Figure 15: Learning setup for execution node experiment

In the first learning configuration, actions and conditions from the manual tree are used directly in the RL model. This means that all the control structures are removed and the actions and conditions are all tied to a single RL model inside a BT node. As illustrated in Figure 15, this node is marked with a star to indicate it as a learning node. Conditions from the manually defined tree are used as discrete observations.

The contents of actions and conditions were left unchanged. The reward was designed to give a small negative reward for each passing time-step as well as a larger negative reward for taking damage or dying. Positive rewards were given for dealing damage and defeating the skeleton. The reward function with sources and values is described in Table 2.

Source	Time step	Damaging skeleton	Skeleton health 0	Taking damage	Agent health 0
Reward	-0.10	5 to 20 *	1000	-5 to -20 *	-1000

Table 2: Reward definition for learning execution node

* As damage is proportional to attack speeds, the reward is actual *damage dealt* x 10. However as the damage range is limited for unarmed attacks, the reward range is as displayed in the table.

Another learning configuration was defined where the observations were not used directly in the form that they were in the original manual tree. In this scenario the observations were defined based on the underlying continuous values which were originally used in order to design the condition nodes. Instead of returning a boolean value by comparing the measurement to a predefined parameter, the measurement itself will be used in the form of a continuous observation.

An additional sub-configuration is defined for the previously mentioned configurations. Two additional observations are added regarding the agent's movement indicating whether the agent is currently moving and in what direction. This gives 2 observations with 3 states for both lateral and forward-backward movement. This allows the agent to move in all directions, permitting simultaneous forward and lateral movement. As this is something that is specific to Malmö rather than the overall problem of RL-BT, it will not be the primary focus of the experiment.

All configurations use the default Stable Baselines Multi-Layer perceptron (MLP) network of two layers with 64 neurons.

4.2 Learning execution node results

This section contains the results of the experiments described in Section 4.1. It will start by presenting learning data from the training period and proceed to comparisons between the resulting models.

Tensorboard was set up in order to monitor the performance of the different models during training. The best way to evaluate the convergence towards an optimal policy was the reward function as it is an universal metric between the model types. The graphs containing the per episode rewards are displayed in Figure 16.

The results are described using a set of specifiers which are presented in the list below. For clarification of these configurations, please see Section 4.1. As can be seen in Figure 16, each model was configured to run for 25000 time-steps, but due to different ending times for the last episode, the exact amount might vary slightly. Every point in the graph represents the result of one episode.

DQN - A DQN based policy with 3 discrete observations based on the original conditions from the manual tree.

DQN continuous - A DQN based policy with a continuous observation for the distance to the target.

DQN 4 obs continuous - A DQN based policy with a continuous observation for the distance to the target and additional observations for the agent's own state.

DQN 5 obs - A DQN based policy with discrete observations for the distance to the target and additional observations for the agent's own state.

DQN 8 move - A DQN based policy with discrete observations and an expanded action space with 8 movements in every direction.

DQN 8 move stop - A DQN based policy with discrete observations and an expanded action space with 8 movements in every direction. Every action is stopped before the next tick.

PPO - A PPO based policy with 3 discrete observations based on the original conditions from the manual tree.

PPO 5 obs - A PPO based policy with discrete observations and additional observations for the agent's own state.

PPO continuous - A PPO based policy with a continuous observation for the distance to the target.

PPO 4 obs continuous - A PPO based policy with a continuous observation for the distance to the target and additional observations for the agent's own state.

PPO multi - A PPO based policy that separates the action space into sub-spaces.

After the learning was completed, each model was run for an additional 50 episodes for evaluation purposes. The reward function was later used for comparison as it is an universal metric for measuring the performance between the different types of agents. Remaining health and the number of simulation steps over the duration of the mission were also recorded. The results of this comparative evaluation can be seen in Table 3. The reward function was applied as if during training. At each tick the RL models were used for determining the next step and then evaluated according to the reward function. In the case of the manual agent, the reward function was applied after each tick of the entire tree.

4.3 Learning execution node discussion

The results show that in general, the direct translation of BT conditions to discrete observations and adding all action nodes as the action space is viable - at least for problems of this size. Theoretically the size of the problem should technically not be a problem at all, as this type of node design essentially entirely isolates the RL problem from the rest of the tree. However, this does show that it is possible to learn very efficient behaviors when the state spaces are indeed small. Therefore, translating the conditions and actions from the BT structure into RL action and

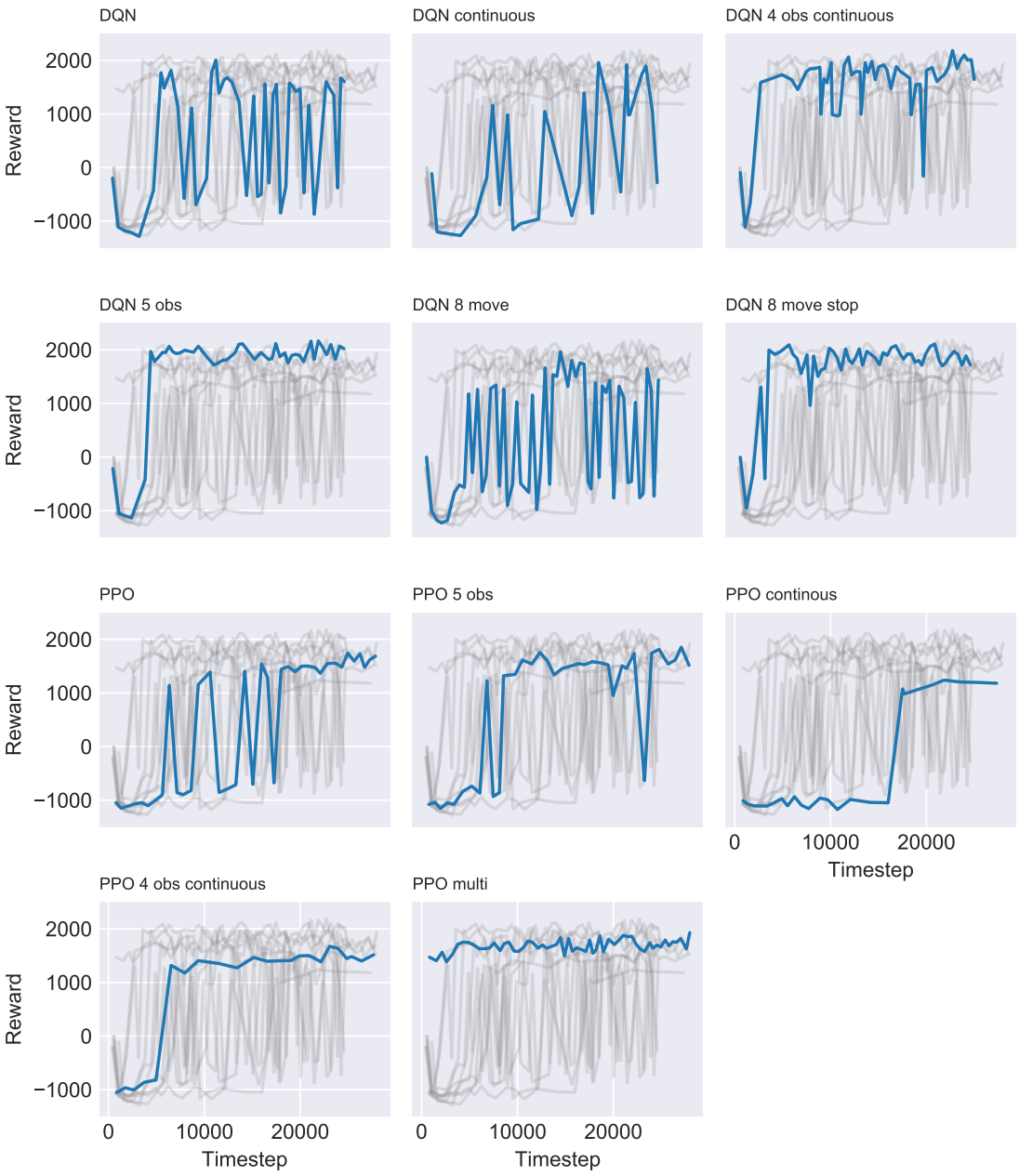

Figure 16: Execution node training reward evolution

Configuration	Reward			Health remaining			Step count		
	Max	Min	Mean	Max	Min	Mean	Max	Min	Mean
DQN	1948.36	-609.86	1506.12	15.00	0.0	6.14	500	304	397.92
DQN 5 obs	1983.14	1591.80	1809.46	20.0	19.5	19.99	537	343	453.76
DQN continuous	990.7	967.2	980.88	20.0	20.0	20.0	330	93	192.88
DQN 4 obs continuous	2564.6	1790.37	2145.75	20.0	7.0	13.57	512	349	431.4
DQN 8 move	1510.07	-890.66	-94.16	10.0	0.0	1.61	577.0	409.0	481.12
DQN 8 move stop	2108.48	1548.63	1859.6	20.0	16.83	19.53	548.0	344.0	448.42
PPO	1959.45	-576.34	1555.62	20.0	0.0	11.57	753	368	565.58
PPO 5 obs	2022.17	1024.20	1713.95	20.0	7.17	16.52	939	412	696.58
PPO continuous	1526.00	-1206.92	689.50	20.0	0.0	10.06	9160	206	4442.48
PPO 4 obs continuous	2027.20	943.40	1639.80	20.0	15.00	19.19	2176	150	1215.46
PPO multi	1934.77	1504.93	1716.9	20.0	7.0	16.49	601	328	456.04
Manual	1790.32	1349.80	1651.56	20.0	20.0	20.0	814	396	562.52
Random	1226.21	-1121.74	-774.9	20.0	0.0	0.46	2059.0	516.0	667.58
Random multi	1795.36	1388.78	1620.33	20.0	11.0	17.33	691.0	356.0	536.42

Table 3: Results for execution node evaluation over 50 episodes

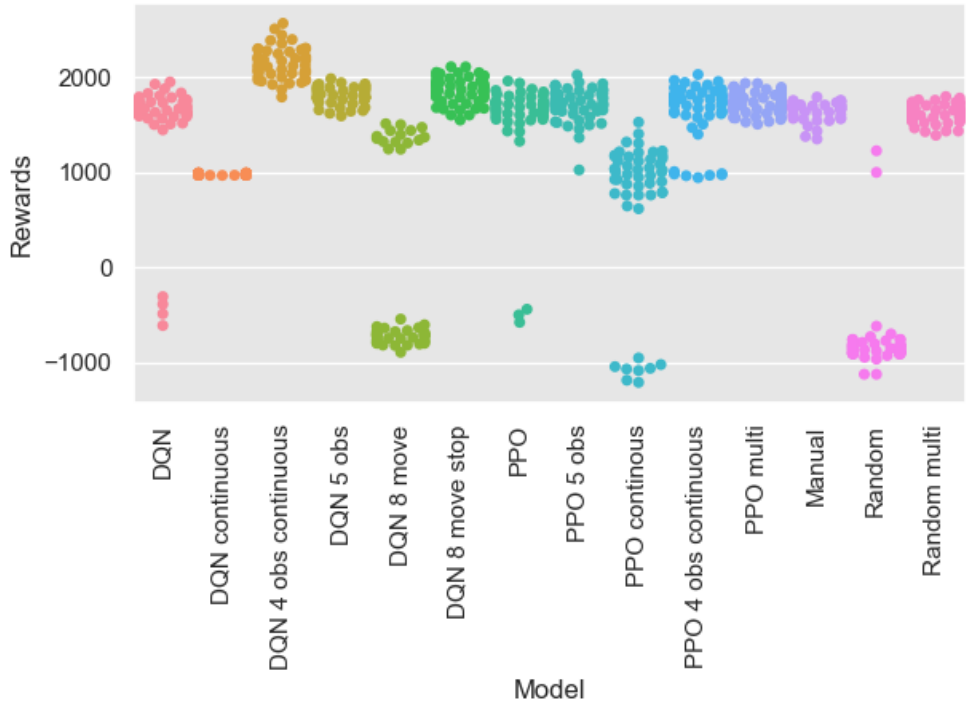

Figure 17: Rewards for execution node evaluation over 50 episodes

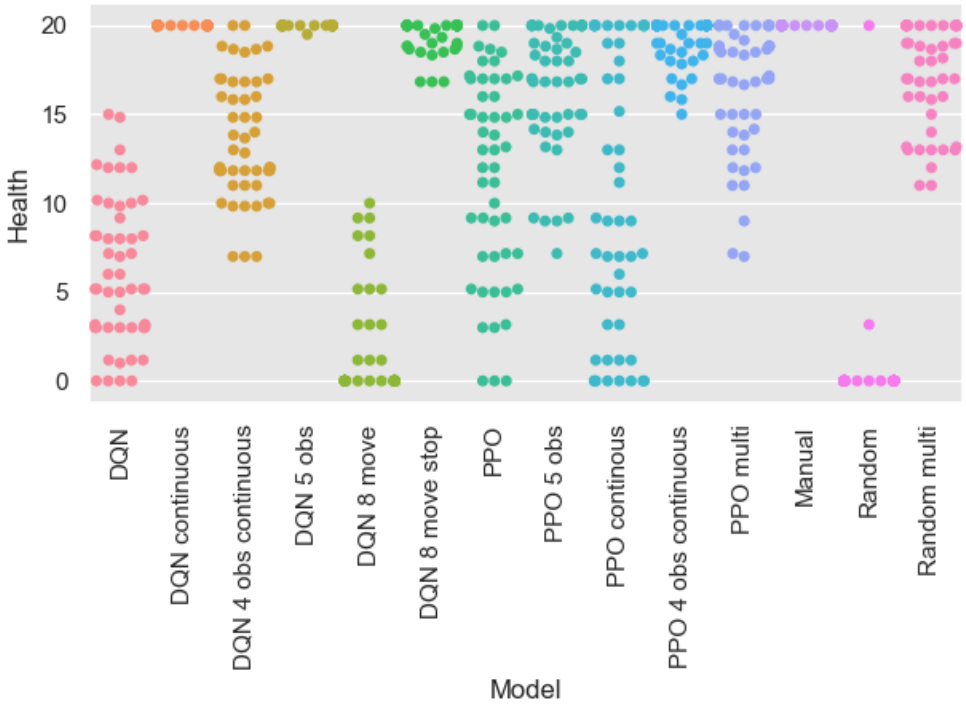

Figure 18: Health for execution node evaluation over 50 episodes

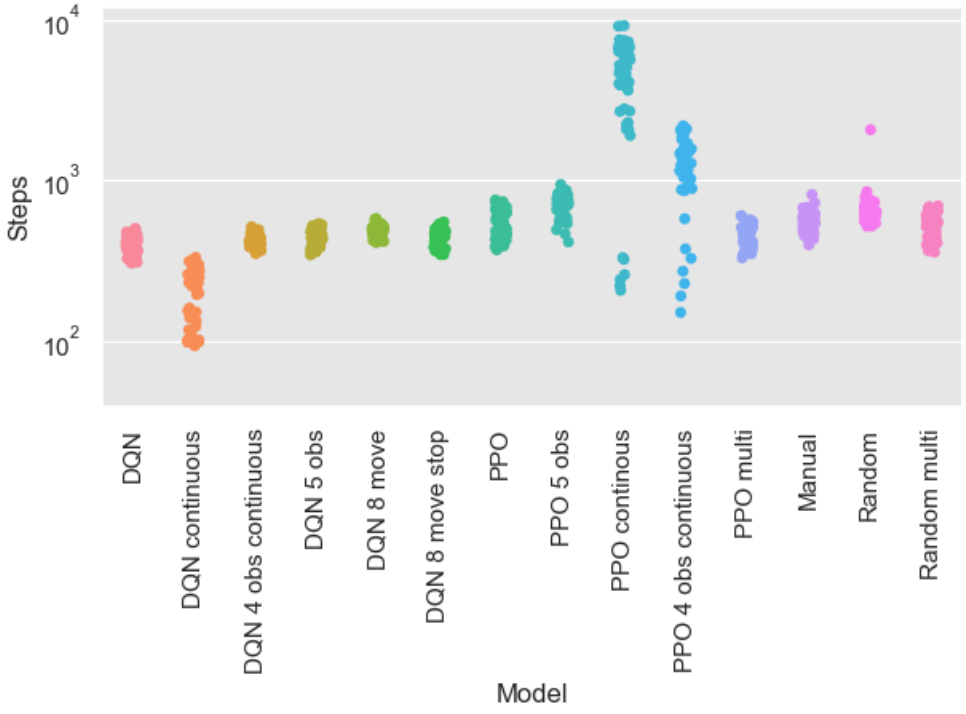

Figure 19: Steps taken for execution node evaluation over 50 episodes

observation spaces, can be a way of implementing learning without throwing away existing expert design and code. This illustrates the remark made earlier about there being similarities in the way that BT and RL are defined.

In comparison with the random baseline, it becomes clear that most configurations were able to learn meaningful behaviors that improve the agent's capability over the untrained state. More importantly, Table 3 shows that most configurations are able to learn behaviors which come close to and even surpass the manually designed agent. The cases which improve over the manually designed agent do so by taking some risks with regards to health, but end up being able to finish the fight in less time-steps. This is most notable in the case of the *DQN 5 obs* and *DQN 4 obs continuous* which achieve far higher average and maximum rewards than the manually designed agent. They do so by finding a balance between steps taken and health remaining. The best model by reward *DQN 4 obs continuous* has less health remaining than the other high scoring models, but wins out with reduced step counts.

For both DQN and PPO, the configurations with additional observations seem to have better performance than their unextended counterparts. Indeed the best rewards are achieved by the two DQN models with 5 and 4 observations. This is evident from the learning curves maximum value and stability during learning, as seen in Figure 16. The reward is also much improved during evaluation as seen in Table 3. This result shows that the continuous nature of actions was harder for the algorithms to deal with, at least on the scale of networks being used in this experiment. This has little implications on RL-BT, but it is an useful observation regarding the nature of Malmö. It can be concluded that the effect of continuous actions can be remedied by adding observations to represent them.

A notable discrepancy with the rest of the results is evident in DQN with continuous observations. Due to limitations in the observation interface in Malmö, the observations from entities - and by extension skeletons - are only provided in a certain range around the agent. As the agent is able to move faster than the skeleton, it learns that by moving away continuously, the skeleton disappears from the observations. This is indistinguishable from the case where the skeleton dies under the current reward function. Essentially, the agent learned to run away, side-stepping the problem as opposed to solving it in the intended way. Notably, when the agent was trained again over multiple attempts with the same model configuration the result did not change, which was not the case for any of the other agents. This is reflected in the lower scores for this type of model as well as the maximum step count. Indeed this is a better solution than what it was able to learn by trying to fight the skeleton, but lower than what it would have received by defeating it. That this behavior was learned over multiple attempts suggests that the space was already too complex for the network size or that the missing observations regarding the agent itself made the policy too hard to learn. Either way, the results are far improved when the continuous observation configuration is expanded with the two agent state observations in *PPO 4 obs continuous* or when using discrete observations in *PPO* and *PPO 5 obs*.

The multi-discrete case was also able to learn a decent behavior, but it could be that the division of the space made the problem somewhat over-simplistic. The agent was able to win even in the first episode as can be seen by the non-negative reward in the

first episode in Figure 16, whereas other agents did not manage their first victories before 4-6 episodes. Seeing as the differences to a random multi-discrete model are minimal, it is safe to say that this configuration made the problem relatively trivial. This does indicate that creating meaningful divisions of actions into smaller discrete spaces helps the learning further and reduces complexity. The downside is that such division requires more manual work and design than the other approaches. Some ideas for future experiments might be to group actions from one sub-branch into one multi-discrete subspace when inferring models in performance based substitution or PPA. Despite the strong start during learning this model does not achieve results as good as the ones with the additional observations. This suggests that even with the sub-division of the action space which significantly boosts performance, it is hard for models to grasp the continuous effects of actions without additional observations. Additionally, it could be said that the division of space makes the problem more restricted and therefore limits the possibilities available for the RL algorithm.

Therefore, it can be said, with regards to Q1, that it is possible to learn meaningful behavior with most algorithms, however some options are more reliable than others and might depend somewhat on the specifics of the problem. This also applies to Q3A, which is referencing the type of node used in this experiment. As the RL problem is completely isolated into a single node, this is not entirely surprising and affirms the work from earlier studies [7]. This is one of the main benefits as well as down-sides of this approach. As the node is a completely isolated RL learning problem that is embedded into the tree, it is possible to apply RL out of the box, without having to do anything different from normal RL. So with regards to Q1, there is not much difference between RL in BTs and normal RL. It also allows to transform the existing BT into RL without much work, as hypothesised by question Q2. However, it is harder to provide any guarantees for critical scenarios. The results of the DQN continuous node are a perfect example of how RL can have unexpected results inside a BT. Safeguards for such cases would probably have to be designed through some type of guard decorator node or built into the learning node itself, reducing code reuse.

5 Learning control flow node

This section will investigate the sub-question Q3B.

This experiment will feature another node described in Section 2.3.1, the learning control flow node. It is designed to show that there is no fundamental difference between learning with an embedded model in an action node as opposed to a control flow node with given leaf nodes. As in the previous experiment, the results will be compared to each other as well as a manual agent. In addition to training models with the new configurations, the trained RL models from the previous experiment will be reused for evaluation purposes to illustrate the similarity between the node types from the perspective of RL.

5.1 Learning control flow node experiment setup

The goal of the agent remains the same as in the previous experiment, described in Section 4.1. The agent must defeat a skeleton with and maximise the remaining health. There is a change to the environment in order to test whether the actions of a control-flow node can be modified without impacting the agent's performance. A separated area with obstacles is created in order to necessitate an expansion of the movement actions. Additionally, the agent starts in a separate area which makes it impossible to get to the target without jumping out first. The new environment can be seen in Figure 20.

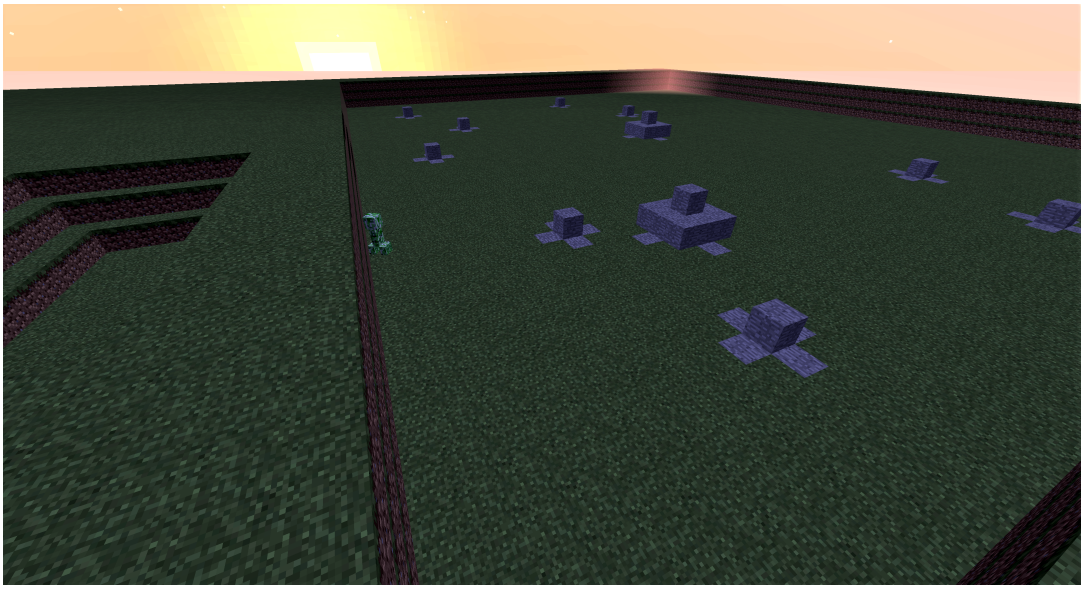

Figure 20: The environment for the control flow node experiment. A separated starting area and a set of randomly placed obstacles.

The manual configuration was also extended in order to better imitate the slightly increased complexity of the learning agents. It is an extension of the manual agent from the previous experiment (Figure 13). As seen in Figure 21, the manual agent can also jump when faced with obstacles and turns in a continuous manner as opposed to instant targeting, reusing the features added to the trees with learning capabilities. The manual agent shall be used for evaluation purposes.

The learning agent's behavior remains functionally similar, but is now constructed using a control flow node. The first configurations works with the same action and observation space as the *8 move stop* configuration from the previous experiment. In order to utilise the branching capability of a control flow node, the tree will be expanded to feature more extended behaviors. Some leaf nodes will be replaced with sub-trees. A graphical representation of the configuration is given in Figure 22. The reward function remains the same as in the previous experiment and is presented in Table 2.

The behavior described in Figure 22 is mostly the same as the one described in Section 4.1. The difference in the configuration of this experiment comes from the sub-tree that embeds all the movement actions. It allows each of the actions to

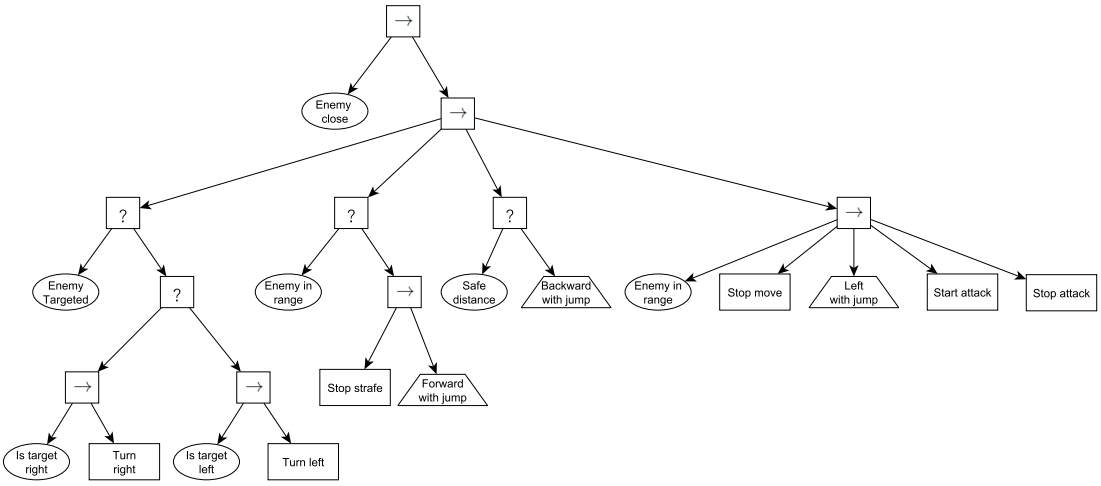

Figure 21: Second, extended version of the manual tree for the control flow node experiment

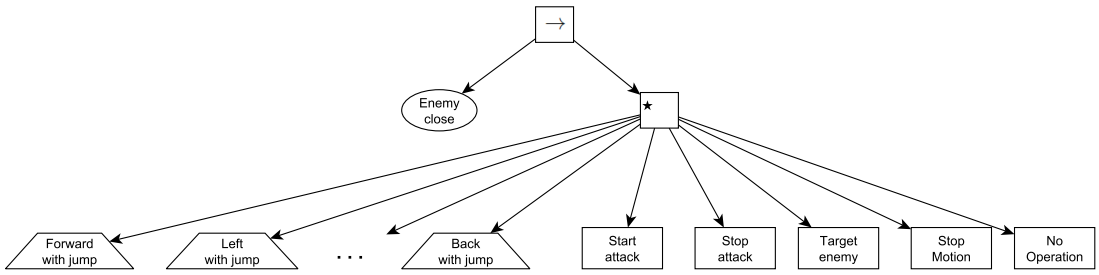

Figure 22: Learning enhanced tree for the control flow node experiment

execute the jump action when movement is obstructed. The content of this repeatedly reused branch structure is seen in Figure 23. It is a simple conditional structure which executes a jump if the direction in which the agent moves is blocked. The list of movement actions contains the following actions, each of which is embedded into the *Move with jump* sub-tree:

- Move forward
- Move left
- Move right
- Move left forward
- Move left back
- Move right forward
- Move right back
- Move back

An additional configuration is created to test the effect of modifying an action's temporal duration. This means that an action will have to be executed multiple

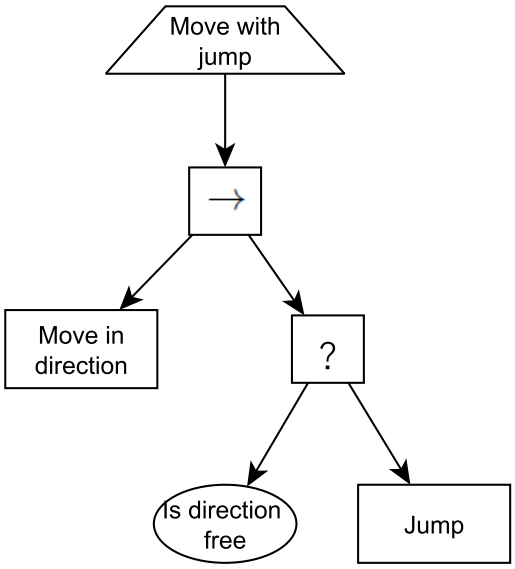

Figure 23: Move with jump sub-tree for the control flow node experiment

times in order to achieve the desired outcome. Specifically, a new turning branch is created to replace the *target enemy* action, changing the targeting functionality to behave closer to that of how a human player would. Instead of targeting the enemy instantly, the agent now has to turn in a continuous manner. The branch can now return Running if it has not yet targeted the enemy. The policy for this extended setup will be learned, as well as evaluated with previously trained policies. The extended tree configuration is shown in Figure 24.

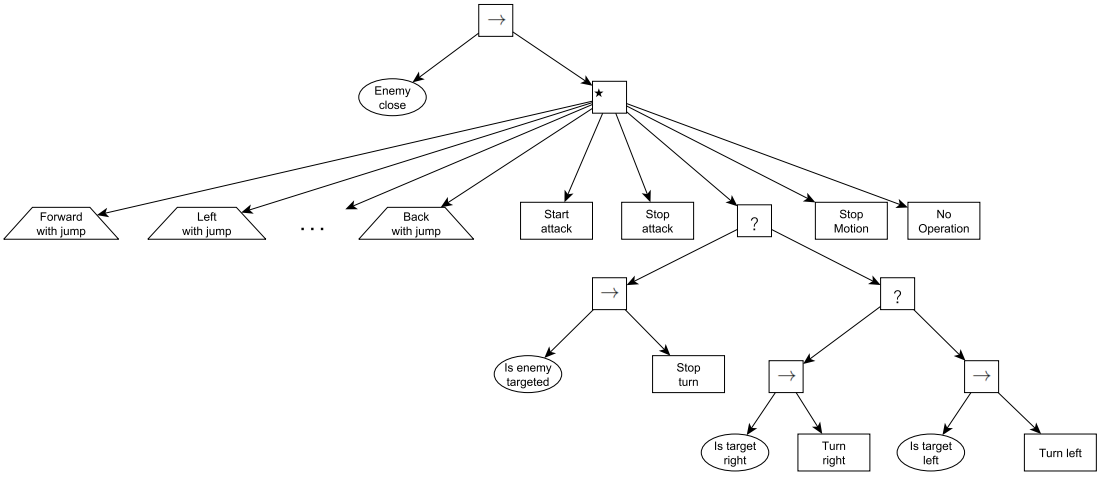

Figure 24: Extended tree for control flow node experiment

As before, all configurations use the default Stable Baselines Multi-Layer perceptron (MLP) network of two layers with 64 neurons for the network.

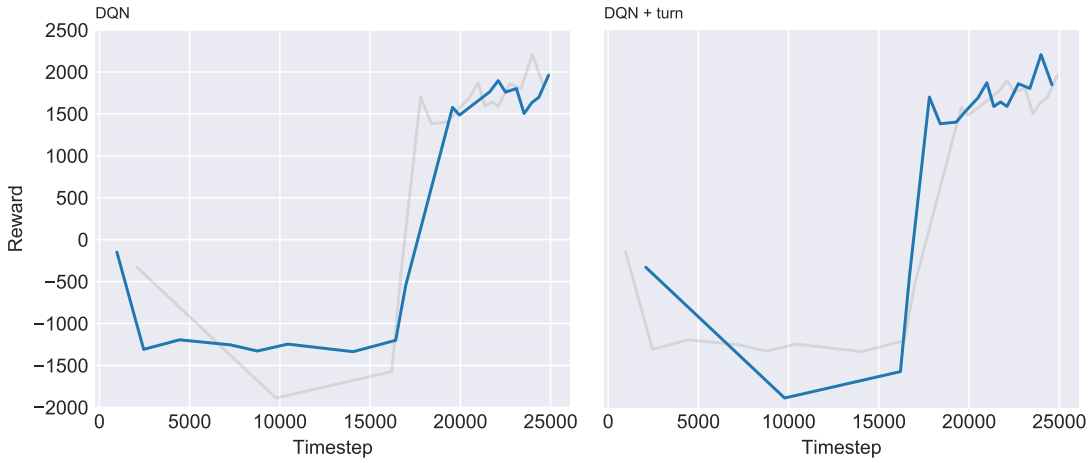

Figure 25: Control flow node training reward evolution

Configuration	Reward			Health remaining			Step count		
	Max	Min	Mean	Max	Min	Mean	Max	Min	Mean
DQN	2406.64	-263.92	1912.35	20.0	18.0	19.96	20317.0	454.0	1001.06
DQN+turn	2644.23	1604.04	2010.85	20.0	10.83	19.17	657.0	349.0	458.46
DQN+reuse	2122.25	1428.83	1834.32	20.0	13.0	19.0	1849.0	346.0	500.38
DQN+reuse+turn	2216.95	1458.86	1851.24	20.0	11.0	18.59	683.0	327.0	485.4
Manual	1984.58	1404.64	1630.4	20.0	12.5	19.72	893.0	353.0	620.18

Table 4: Results for control flow node evaluation over 50 episodes

5.2 Learning control flow node results

This section contains the results of the experiment described in Section 5.1. As in the previous experiment, Tensorboard was set up in order to assess the performance during training. The graphs containing the per episode rewards are displayed in Figure 25.

In this experiment all the policies use DQN for training purposes. The *DQN* identifier represents the control flow node with jumping functionality, seen in Figure 22. The *continuous turn* identifier indicates the modification which adds a temporal dimension to the targeting functionality, seen in Figure 24. The *reuse* identifier indicates configurations that use a pre-trained model from the previous experiment. The policy for the *reuse* configurations is the one from the previously trained *8 move stop* configuration described in Section 4.1. Section 5.1 describes the configurations in greater detail.

The experiment was evaluated using the same pattern as before. Each model was run for an additional 50 episodes for evaluation after learning. The reward function was used for comparison alongside remaining health and the number of simulation steps taken. The reward function was applied as done during training - at each timestep. The health and final step count were recorded at the end of each mission. The results of this evaluation can be seen in Table 4 and Figures 26, 27 and 28.

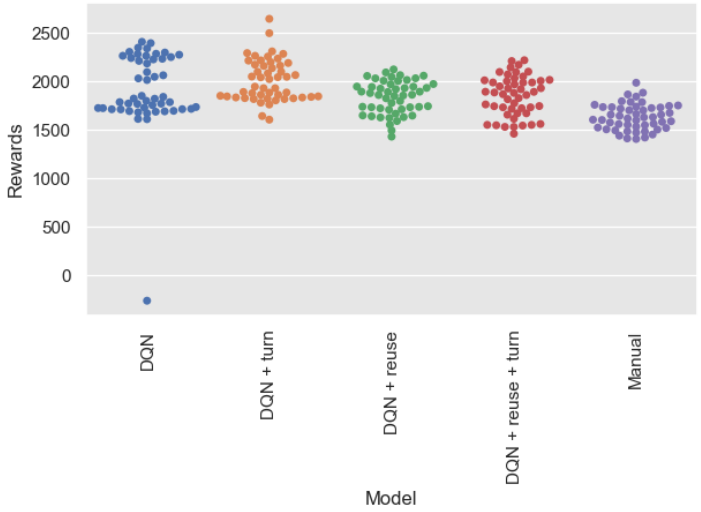

Figure 26: Rewards for control flow node evaluations over 50 episodes

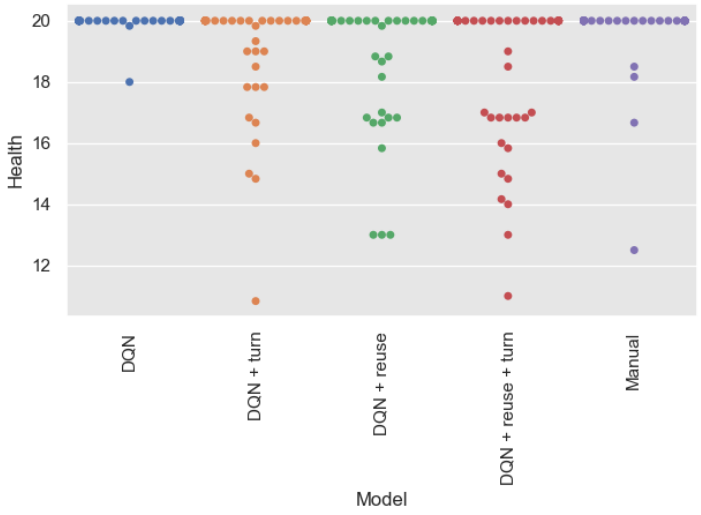

Figure 27: Health for control flow node evaluations over 50 episodes

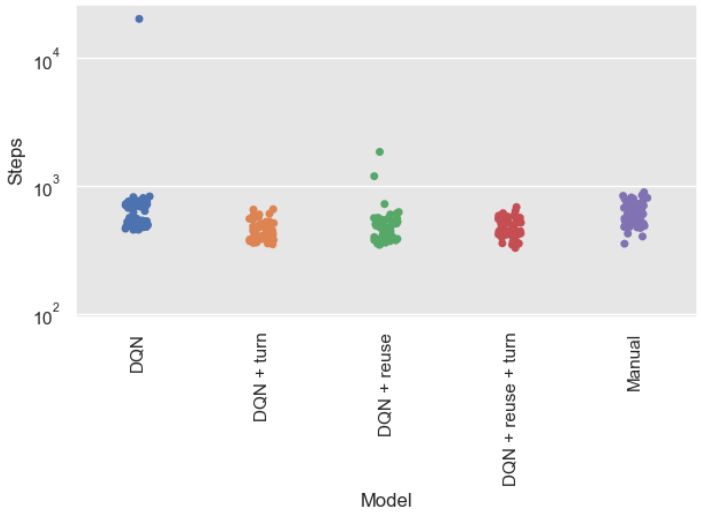

Figure 28: Steps taken for control flow node evaluations over 50 episodes

5.3 Learning control flow node discussion

All configurations were able to learn a meaningful behavior similar to the results with DQN in the previous experiment. This is not surprising from an RL perspective as the formulation is similar to the ones in Section 2.1.2. This illustrates that there is no difference on what can be learned by RL in terms of BT node types, which also serves as an answer to question Q3B, as all the findings for Q3A regarding choice of algorithm and the modularity of RL also apply here. There are some benefits to this formulation from the BT perspective however. It allows reuse of the actions that have been previously defined without code duplication or specific definitions. It also allows for extensions of the individual actions into more complicated BT sub-trees, while still maintaining the flow of execution statuses inside the tree. This means that an action that was originally trained as a simple isolated RL behavior can always be reworked into a control flow node later, allowing for greater modularity if the need arises.

The extended action node configurations make it harder for the agent to learn the new pattern. This is illustrated by the differences in the learning curves in Figures 25 and 16. In the execution node experiment the episodic reward starts to increase and stabilise at around 5000 time-steps, while it takes around 15000 for the extended configuration in this experiment. As the changes in movement do not have any temporal effects on execution, the cause is most likely related to the changes in targeting.

Despite the problem being technically harder to learn, the old model is still able to use the modified actions. There is only a small difference between the newly trained and *reuse* configurations. This indicates that it is possible to reuse models with modifications and still get good results if the goals of the actions do not change fundamentally. Therefore it should also be possible to add safety features such as conditions after training if necessary, without having to discard the learned model.

The new models did get slightly better results after training than the reused models. This suggests that it might be useful to do some additional training on an action after it has been modified, so that the model can adjust to the changes.

6 Type specific learning

This section will focus on answering sub-question Q4 and Q5.

The experiment will test a type specific approach to instances of similar situations, by selecting between a set of RL models based on a type identifier. The idea is to show that it is possible to learn models specific to a type of enemy, that will be at least as effective as a single model that can fight all types of enemies. This scenario will not feature a manual agent and will focus on comparing approaches for handling sub-types within a problem.

6.1 Type specific learning experimental setup

The goal of the agent in this scenario is an extension of the one in the earlier two experiments where the agent was fighting a single type of enemy - a skeleton. In this scenario, the agent must face three different types of enemies which behave differently. The agent must still fight a skeleton, as before, but in addition it will also have to fight zombies and creepers, see Figure 29. Compared to a skeleton, the zombie is quite similar in behavior, but there are a few differences. First of all it is slower, but has more health and cannot be knocked back as far as the skeleton can. The creeper on the other hand is a completely different enemy - essentially a sentient explosive that can move around. When it gets close to the agent, it will detonate, potentially killing the agent if it is too close. This will require a different type of approach to achieve good results.

(a) Skeleton (b) Creeper (c) Zombie

Figure 29: Different types of enemies

The agent will fight each of these enemies in a continuous rotation, each mission featuring a different enemy. The environment is the same as was used previously, an area separated with raised edges and containing random obstacles as seen in Figure 20.

The experiment features two different setups, illustrated in Figures 30 and 31. A graphical representation of the behavior tree with 2 type specific RL models can be seen in Figure 30. In this case the relatively similar Skeleton and Zombie are considered a single type, with the creeper handled in a different policy. Another configuration is created, which has a different RL policy for each enemy. A graphical representation of the behavior tree with 3 type specific models can be seen in Figure 31.

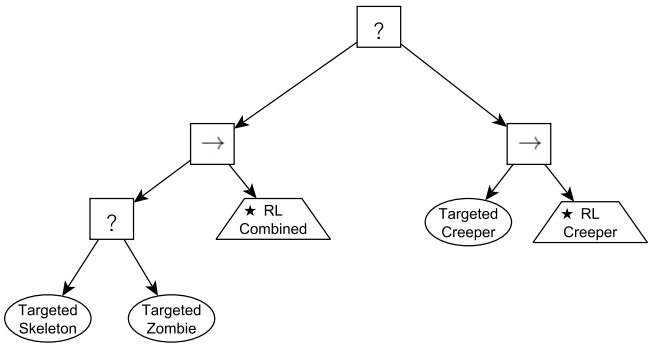

Figure 30: Tree with 2 types of policies for type specific learning experiment

The primary building block will be the extended configuration from the previous experiment, described in Section 5.1. This experiment uses the previously described tree structure, of Figure 24 as a sub-tree. It embeds each sub-tree with a different model and uses a decision structure to choose between them.

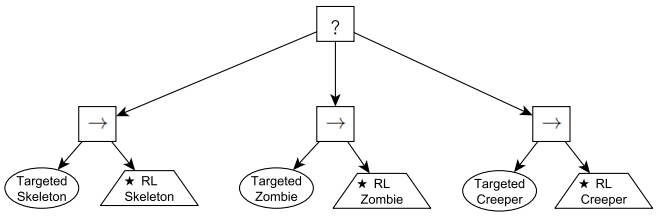

Figure 31: Tree with 3 types of policies for type specific learning experiment

One the configurations will use a single policy, but will instead have an additional observation corresponding to the type of opponent. A policy with no information about the type of enemy is also trained for comparison with all the previously described configurations. In the end, the best individual models will be hand-picked and combined into an agent, to test the modularity of the approach.

As before, all configurations use the default Stable Baselines Multi-Layer perceptron (MLP) network of two layers with 64 neurons for the models.

6.2 Type specific learning results

Tensorboard was set up in order to assess the performance during training. In this scenario, the agent was switching between enemy types at the end of every mission,

Configuration	Reward			Health remaining			Step count		
	Max	Min	Mean	Max	Min	Mean	Max	Min	Mean
2 types	3064.18	-1223.9	**1375.72**	20.0	0.0	**14.06**	623.0	58.0	327.2
3 types	2351.17	801.5	**1539.1**	20.0	1.0	**10.1**	653.0	54.0	276.16
No type info	2083.36	-1068.69	730.34	20.0	0.0	5.1	649.0	50.0	294.12
Observation	2410.44	845.24	**1531.68**	20.0	2.0	**12.37**	658.0	52.0	282.88
Combined	2590.14	-1472.9	**1176.59**	20.0	10.0	**17.38**	**7982.0**	48.0	**878.16**

Table 5: Results for type specific model evaluation over 50 episodes

which corresponds to an episode in the training rewards. While the total number of timesteps was fixed, the amount of time spent in each mission varies. Therefore the amount of learning done with each model differs from model to model. As the number of different enemies was increased, the total number of timesteps for learning was also increased to 75000. The graphs containing the per episode rewards are displayed in Figure 32.

The identifiers *2 types* and *3 types* represent respective configurations which use either 2 or 3 models for the enemies. *No type info* is a model which has no information about the type enemy it is fighting. *Observation* represents the configuration which only has a single model, but is given an additional discrete observation corresponding to the type of enemy. Section 6.1 describes the configurations in greater detail. An additional policy for the Creeper enemy was trained for the purpose of the *Combined* agent, which bears the identifier *Additional creeper policy*.

As with previous experiments, each model was run for an additional 50 episodes for evaluation after learning. The reward function was used for comparison alongside remaining health and the number of simulation steps taken during each mission. The reward function was applied, as if during training, at each timestep. The health and final step count were recorded at the end of each mission. The results of this evaluation can be seen in Table 5 and visualised in Figures 33, 34 and 35.

6.3 Learning with type specific models discussion

All policies, except the one in the *No types* configuration, were trained successfully on zombies and skeletons, but the results were more mixed for the creepers. Creepers are harder to train on due to less exposure. The amount of experience gained from a single creeper is more limited due to their explosive behavior. As their primary attack also kills themselves, the episodes are a lot shorter than that of the other enemies. The baselines package only allows to define the number of timesteps the agent trains for. As the training environment was set up to iterate through enemies for the duration of the training session, this meant that the agent would see each opponent type for an equal number of episodes. Subsequently the total amount of training time on creepers constitutes a smaller portion of the session, as can be seen by comparing the creeper specific rewards to the other types in Figure 33. It is also to be noted that the creeper problem is harder to learn as there is less room for error, as can be noted from the erratic changes in reward of the creeper specific policies.

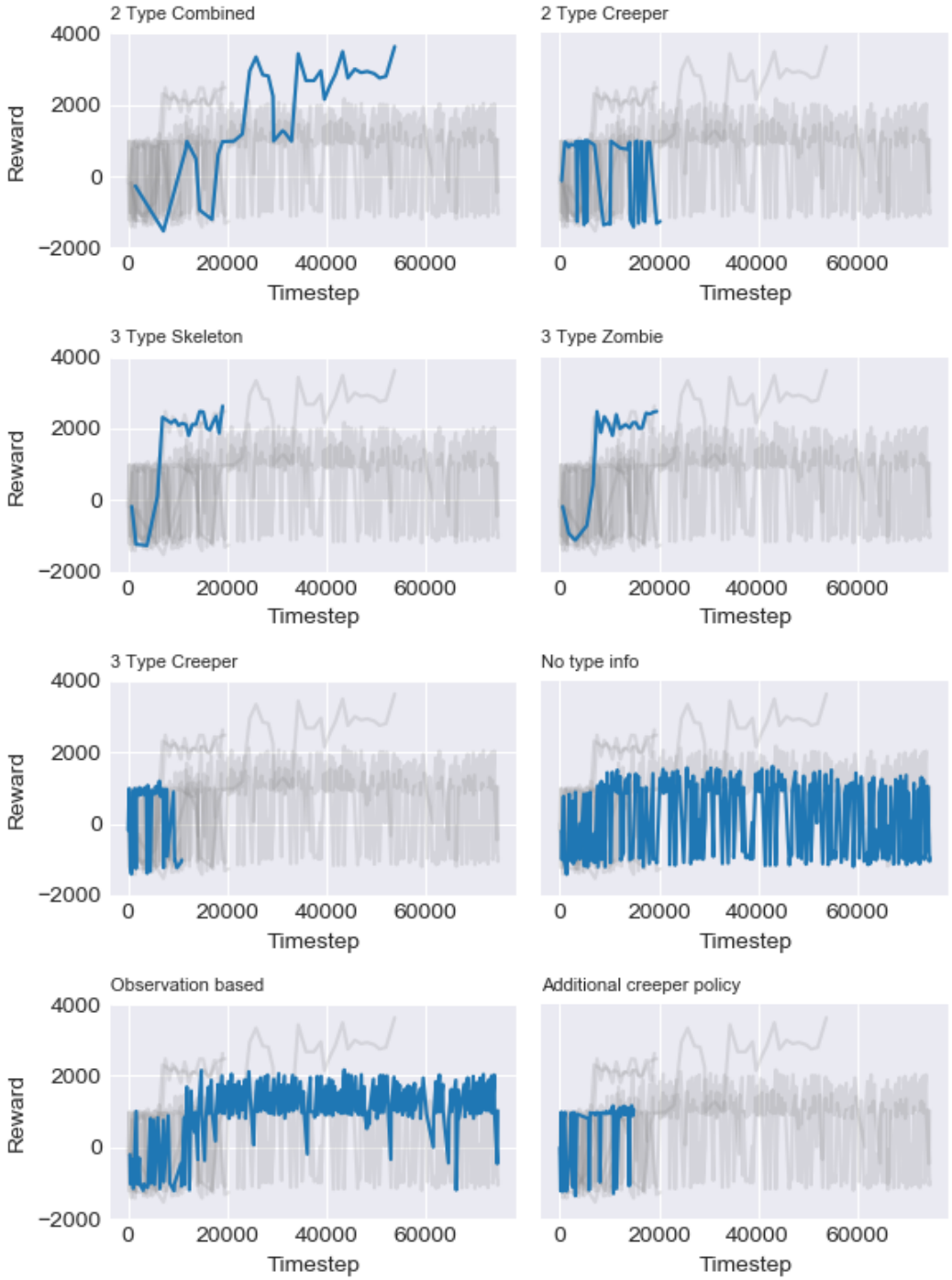

Figure 32: Type specific policy training reward evolution

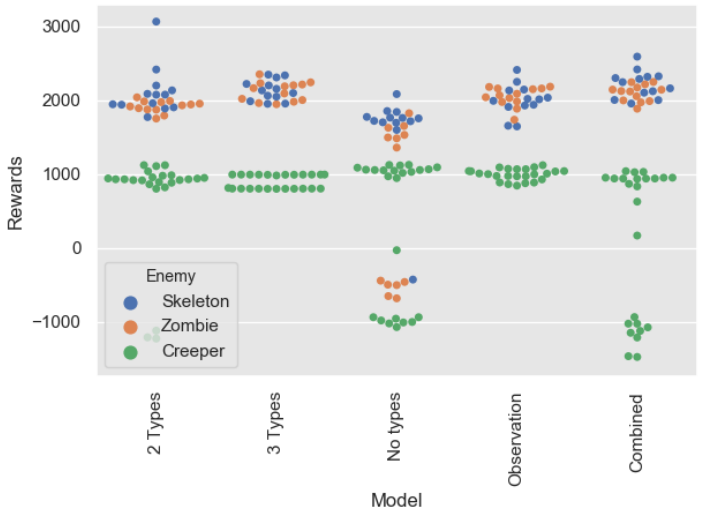

Figure 33: Rewards for type specific model evaluation over 50 episodes

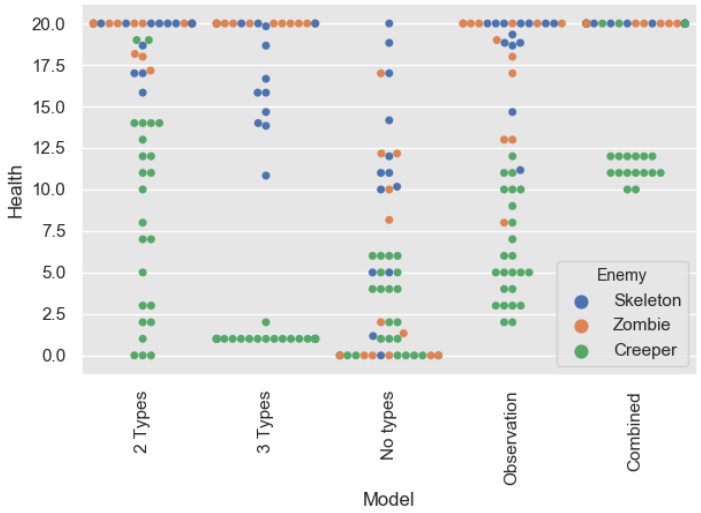

Figure 34: Health for type specific model evaluation over 50 episodes

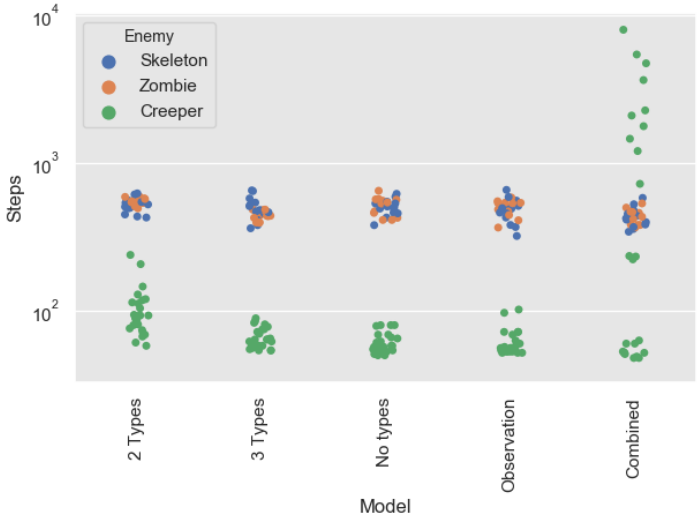

Figure 35: Steps taken for type specific model evaluation over 50 episodes

Looking at Figures 33 and 34 it becomes apparent that even though the results are similar in terms of reward, the agents behave differently in terms of health. No type info does well on average, but dies often with creepers. The observation based policy does reasonably well with creepers. This is probably because it is able to use the knowledge it has gained from fighting zombies and skeletons on the creeper as well. *3-Type* does badly on creepers, probably because it didn't manage to find a good way to deal with them in the limited time-frame. The 2-type policy is has the best overall results, but still has a few unlucky encounters creepers. As the results are relatively similar to the observation based model, despite the reduced learning time for each individual policy due to the separation of models, it can be concluded there is merit to the approach proposed in sub-question Q5.

Having this type of modularity to the RL problem made it possible to manually hand-pick the best trained policies and combine them into a new configuration as suggested by the formulation of Q4. The manually combined configuration, which has 3 types, gives the best results with respect to health. This is achieved by picking the best performing policy for skeletons and zombies, the *2-Type* policy, and combining it with a newly trained creeper model. The newly trained policy is a lot more cautious, staying away from the enemy for longer. This leads it to lose out on average reward compared to other policies due to taking too long to get rid of the enemy, despite having the best results with regards to health. This is a good example of how separating tasks by type can be useful for debugging, as a bad behavior can be re-trained or substituted without throwing away the parts which were performing well. This could alternatively be done by exposing the single shared policy to more of the things that are not performing well, however that could lead to unwanted changes for other types which were not intended to be changed. This shows how it can be more beneficial to identify badly performing sub-policies to retrain individual behaviors. This also serves as a successful example of what is described in Q4.

Hypothetically it should even be possible to create an automated policy manager for types within a problem. If no policy exists for a type, it would attempt to match

an existing policy to it. If no existing policy gives good results after a testing, it could learn a new policy instead. Theoretically an existing policy could be used as a prior, so that existing learning experience can be used for the next task. Reusing an existing policy is supported by the previous experiment where we saw that it is possible to reuse policies and get good results if actions do similar things. In this case the RL definition does not change, only the behavior needs to be adjusted. The best policy amongst the existing could be chosen for this purpose and trained to perform better.

7 Multiple node decomposition learning

This section is dedicated to exploring sub-question Q6.

All the previously described scenarios use only a single learning node or nodes that are assumed to be executing for the entirety of a mission. This scenario will experiment with more complex tree structures in which such assumptions do not hold. The problem will be further decomposed into sub-parts with individual learning capabilities for each part. The aim is to investigate if it is possible to learn a composite tree of multiple learned behaviors for a single task. An alternative way to formulate control flow nodes, with reward signals based on the return condition, was also experimented with as described in Appendix A. This part was moved to the Appendix as the experiment it was unsuccessful.

During the learning process of the previous experiments, we saw that lacking a reward signal for intermediate steps makes it a lot harder for the agent to learn a good behavior. For example having a reward for defeating the enemy, but not for damaging the enemy, makes it a lot harder for the agent to figure out how to defeat the target. Intermediate rewards could be viewed as a solution to this problem and were used in previous experiments. In RL this can lead to a problem known as reward shaping [1]. In the context of BTs it could rather be looked at as a matter of proper decomposition. If the RL task requires complex reward signals, it might be possible to break the problem down into sub-tasks which can be reused. This experiment will decompose a larger problem and use RL simultaneously in different nodes in the tree. The principle is to use actions and conditions which do not require intermediate rewards and to create generic behaviors that can potentially be reused in different contexts. This is done by treating a sequence of timesteps when a node is ticked consecutively as an episode for the model in that node. Therefore, the ticking of a node can stop even if the entire task is not completed. This means that the tree can potentially have multiple RL episodes running in different nodes at the same time.

The environment is no longer a flat world for this experiment. A normal Minecraft world with a so-called temperate biome is used instead. The world generation is done with a fixed seed and the mission is carried out in the same location of the generated world. The agent's goal is to build a simple house, of which an example is given in Figure 36. This requires behaviors that are more extensive than the ones in the earlier experiments. The agent has to navigate, gather, craft and place blocks

(a) Foundation (b) In progress (c) Completed

Figure 36: House building environment

in order to achieve its goals. Many of these sub-tasks are non-trivial. For example, navigation requires avoiding obstacles and dealing with elevation. It can also be difficult to find the right position and viewing angles for the agent when targeting cells for gathering or block placement.

7.1 Manual agent definition

The basis for the agent's actions shall be a manual BT, parts of which are to be replaced with learning capabilities. The definition has been split up into several figures, which will be discussed in turn. The tree is designed to place blocks given a predefined list of coordinates, in order to form a house. To simplify the problem, the locations are predetermined in a way that makes the placement of each block legal and the agent does not have to figure out the order. If the agent does not have the blocks it needs to place, it is designed to gather them and craft them out of the resources available in the environment e.g. planks out of tree logs.

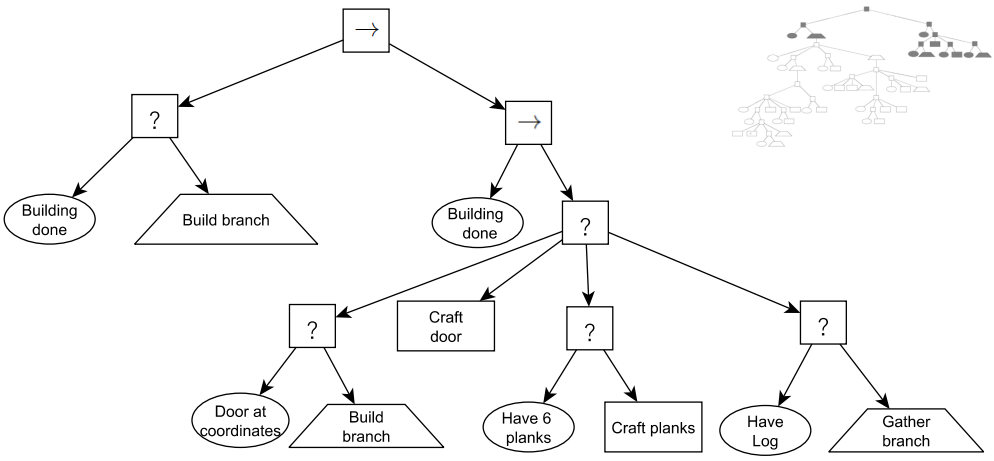

Figure 37: High-level tree definition. In the upper right, the a map of the complete BT is shown with the dark parts indicating what is shown in the figure.

The high-level behavior can be seen in Figure 37. The goal of this structure is to gather a small supply of wooden logs from trees, craft them into planks and use those for the walls of the house. There is also a manual sub-tree for crafting a door

which requires two crafting actions and a larger number of planks in order to be carried out successfully. This differentiates it from the crafting used in the *Build branch* seen in Figure 38.

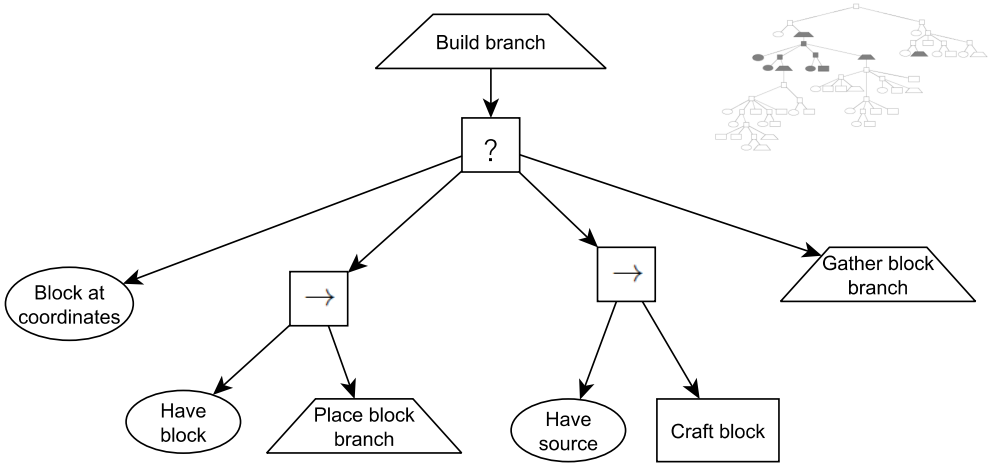

Figure 38: Build sub-tree definition.

The *Build branch* in Figure 38 is a small intermediary for the place block and gather block branches. It checks if the desired block is in the location with the *Block at* condition. This condition extracts the type of block from a grid relative to the agent's position and checks its type against the type to be placed. If the block at that location is not of the right type it will be placed if one is available in the inventory. Otherwise, if a source for that block is available, the block will be crafted. If neither are available, the material will be gathered using the respective sub-tree.

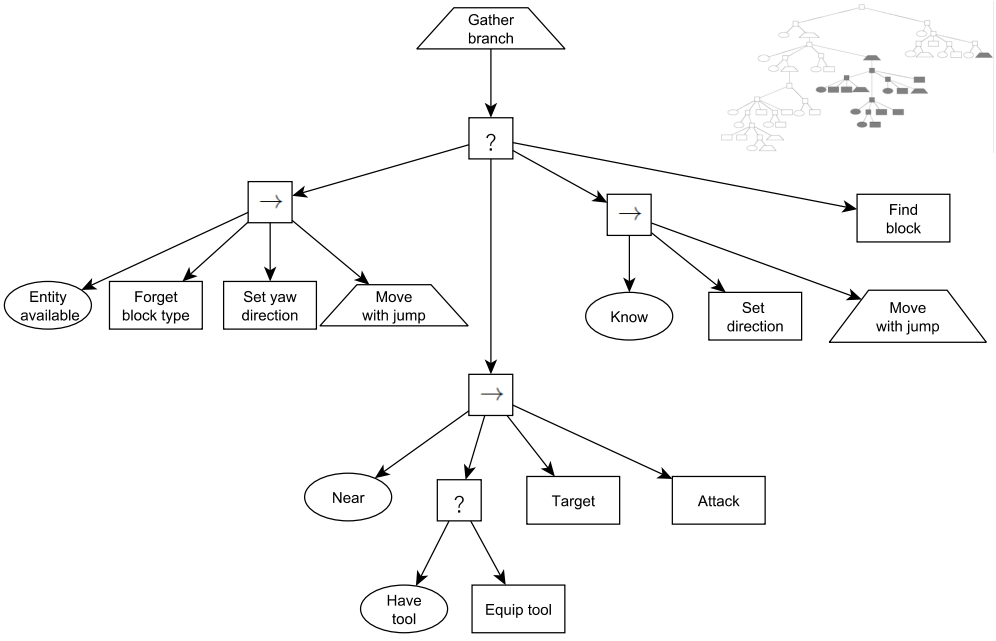

Figure 39: Gather branch sub-tree definition.

The *Gather branch* is an implicit sequence made up of 4 primary steps. First of all

it checks for the material in the form of an entity (described in Section 3.1). This is necessary, as after removing a block it is normally dropped as an entity. If the agent stands close it will usually be picked up automatically. However, in some cases it is possible that it will drop further away and the agent will therefore have to be move to it in order to be pick it up. An entity is automatically picked up by an agent in Minecraft if it is close enough.

If no entity is available, the agent will check if it is close to a block of the required type. A block needs to be within 3.5 distance units in order to be interacted with. If so, a tool will be selected if an appropriate type is available for the target block and that block will be removed and turned into an entity. This can then be picked up in the previous step of the sequence. If no block of the correct type is close enough the agent will move towards the closest one. If there are no known sources of that block, the agent will attempt to find one with the *Find block* action.

It is necessary to introduce memory to the agent, in the form of the *Know*, *Forget* and *Find* nodes. Repeatedly searching through the observation grid of blocks around the agent is computationally expensive, therefore it is better to remember the result of a search and update the information at certain steps. In order to prevent the agent from acting with old knowledge the memory is cleared after a fixed amount of steps. Entities are typically present in lesser quantities than blocks and are of a more transient nature. Therefore it is not necessary to remember information about entities.

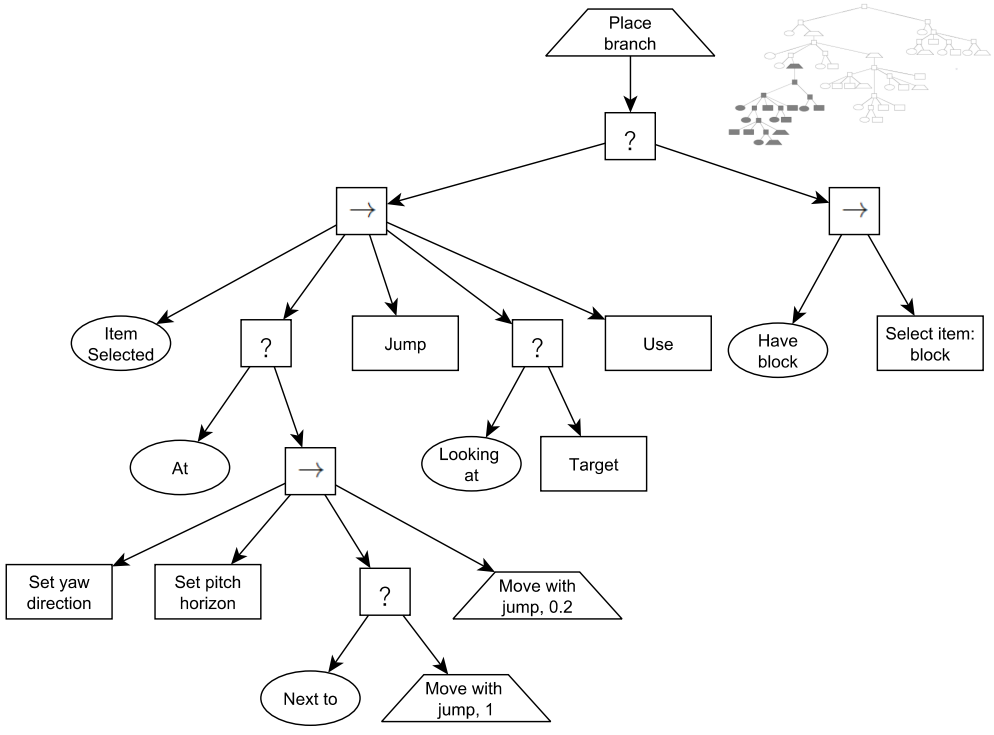

Figure 40: Place block sub-tree definition for the simultaneous nodes experiment.

The *place branch* in Figure 40 takes care of the actual placement of blocks. It first checks if a block of the correct type is selected and if not, selects that block if it

has one. On further iteration it will move towards the position if it is not already at the location where the block is to be placed. If it is closer to the target it moves slower in order to avoid overshooting the target during movement. When it is At the target, meaning withing 0.5 distance units, the agent will jump up and look down at the target in order to place the block under itself. This is the simplest way to assure that the block ends up in the correct position as the view could otherwise get obstructed in many different ways. The ordering of block placement is handcrafted with this nuance in mind.

Despite its scope, the manual design in Figures 37 - 40 is lacking certain features in terms of the mission objective. Depending on the environment it can get stuck behind obstacles like trees or the walls the agent itself has built. It is also unable to reliably place roof blocks as it is programmed to look at the center of a block during targeting. This can be a problem, as in Minecraft a block is always placed next to the face of the neighbouring block that is currently being targeted. Both the targeting and movement problems could be overcome by expanding the tree structure further, but in this scenario it will be attempted with learning.

7.2 Learning agent definition

The manual BT design was modified to use learning for movement and targeting in order to overcome the primary shortcomings of it. Both targeting and movement nodes use a similar action space, but the targeting node uses slower speeds for its move actions. The action space for the nodes consists of the following actions.

- Move forward
- Move backward
- Move left
- Move left & forward
- Move left & back
- Move right & forward
- Move right & back
- Move right
- Turn right
- Turn left
- Pitch up (Only for the targeting node)
- Pitch down (Only for the targeting node)
- Jump
- No operation

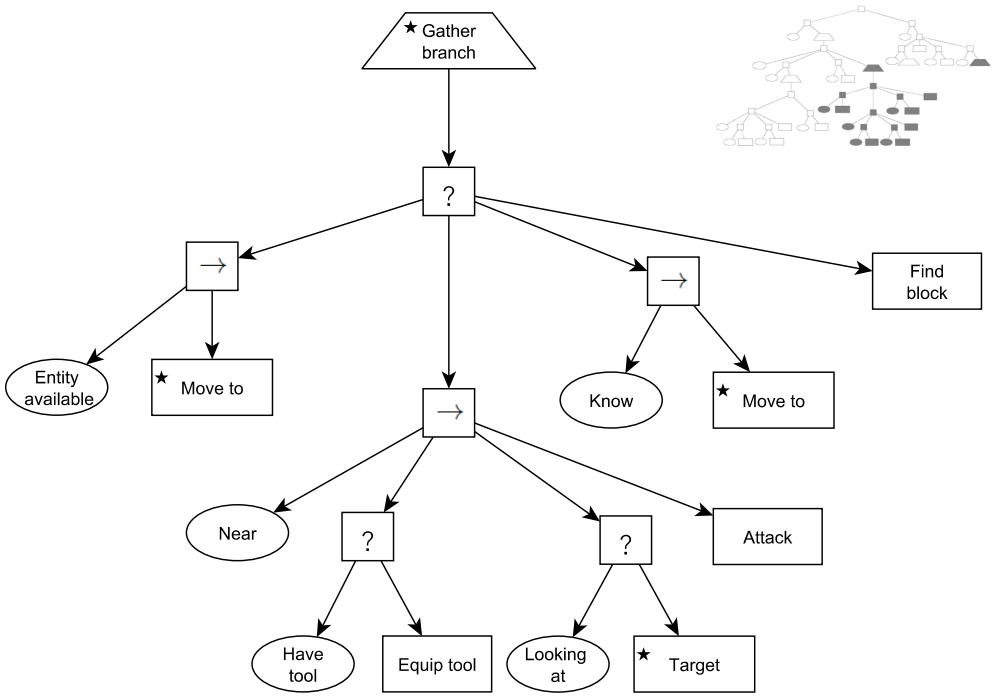

Figure 41: Gather block definition with learning, the learning version of Figure 39

The learning nodes are embedded into the tree as seen in Figures 41 and 42.

The nodes use the pitch and yaw to the target as observations, as well as a small grid observation of blocks immediately next to the agent in order to understand when the movement or vision is obstructed. The movement node also has an observation of distance to the target. The following listing describes the observation space for these agents.

- ΔDistance to target, [0, 50] ([0, 5] for targeting)
- ΔYaw to target, [-180, 180]
- ΔPitch to target, [-180, 180] (Only for targeting)
- Discrete grid of size 3x4x3, representing blocks surrounding the agent

Observations	Time step	Target	Distance
Targeting	-0.50	1000	-3000
Movement	-0.50	0	Δ·25

Table 6: Rewards for simultaneous learning nodes experiment

The reward definitions are based on the underlying measurements of the conditions that each node is trying to satisfy. The reward for targeting is a constant value for successfully targeting the correct block as well as a small negative reward for each timestep. For movement there is also a small negative reward for each timestep and a reward corresponding to the change in distance to the target between the previous and current timesteps. A summary of the rewards can be found in Table 7. The

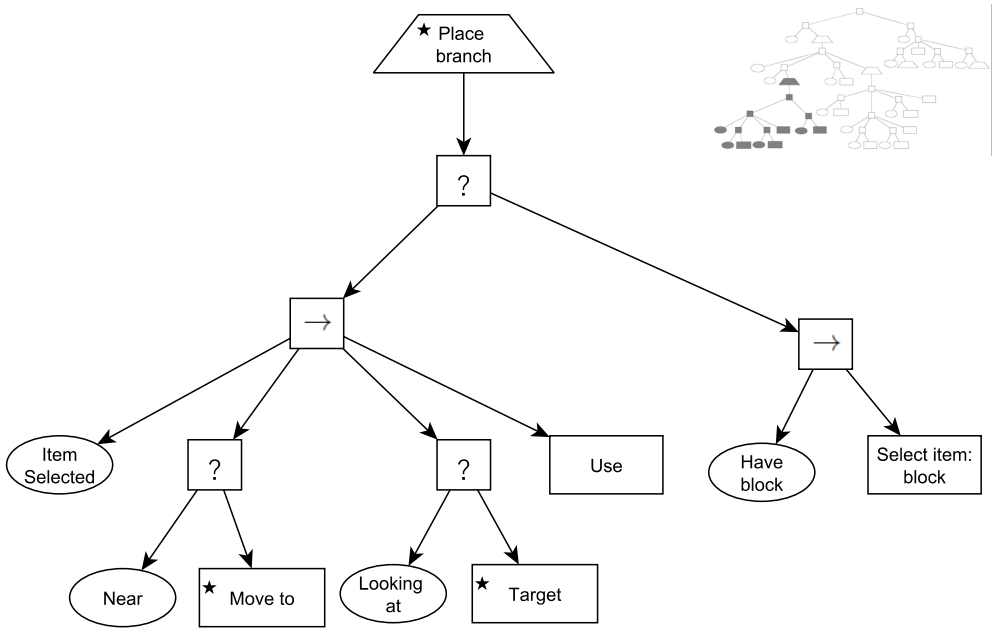

Figure 42: Place block definition with learning, the learning version of Figure 40

targeting node gets a large negative reward for moving too far from the target and causing the movement node to take over, ending the episode.

Reward	Time step	Target	Distance
Targeting	-0.50	1000	-3000
Movement	-0.50	0	$\Delta \cdot$ 25

Table 7: Rewards for simultaneous learning nodes experiment

The learning nodes for movement and targeting were custom MLP networks of three layers, with 128, 64 and 32 nodes respectively. Both nodes use the DQN algorithm from Stable-Baselines for training.

An additional setup was created with the purpose of allowing the agent to optimise the use of the gathering and building sub-trees. However, as the configuration proved to be too unstable. It is instead made available in Appendix A.

7.3 Multiple node decomposition learning results

As with previous experiments, the RL progress was monitored with tensorboard. In this scenario, the agent was treating sub-goals as episodes. Every time the control flow stopped ticking a node the current episode ended and a new one started when it was ticked again. This is represented as episodes in the rewards graph displayed in Figure 43. While the total number of timesteps was fixed, the amount of time spent training with each node varies, as each node would be called when needed. Therefore the amount of learning done in timesteps differs from model to model. As

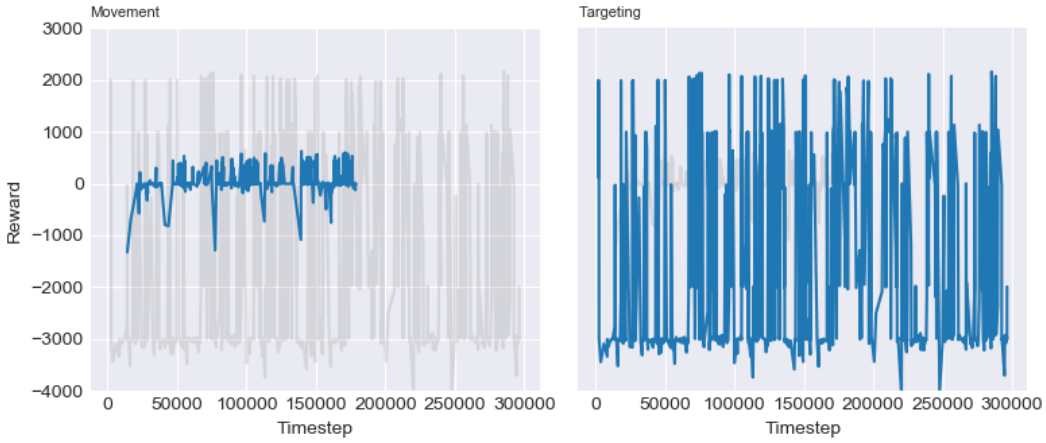

Figure 43: Multiple node decomposition episodic reward

Configuration	Step count		
	Max	Min	Mean
Learned	19385.0	5503.0	9758.48
Manual	3835.0	1094.0	1645.8

Table 8: Multiple node decomposition evaluation over 25 missions

the problems used for training were more complex this time, the total number of timesteps for learning was also higher. The training was run for a total of 500 000 timesteps.

The agent configuration with the high-level gathering optimisation node, presented in the appendix, proved to be too unstable for use and will not be included in the results section. The implications of this will be examined in the discussion section.

The primary metric for evaluation was the amount of time spent in order to finish building. In this scenario, the capability of the manual agent is surpassed by the learning agent to the point that it is able to finish things that the manual agent is not. In order to compare the two quantitatively as well, the evaluation scenario was simplified such that the agents only have to build walls, but not the roof of the house. As both agents are able to complete this task successfully, it should be more useful for comparison. Because the process takes longer to complete the number of missions was changed to 25. The results of this evaluation can be seen in Figure 44 and Table 8.

7.4 Multiple node decomposition learning discussion

It can be noted, by observing Figure 43, that the movement node reward stabilises relatively quickly in comparison to the targeting behavior. Despite having some downward spikes, the movement reward stays close to zero. As the time travelled plays a part in the amount of reward gained by the movement node, the differences in the size of positive rewards is to be expected. The negative spikes are likely due to

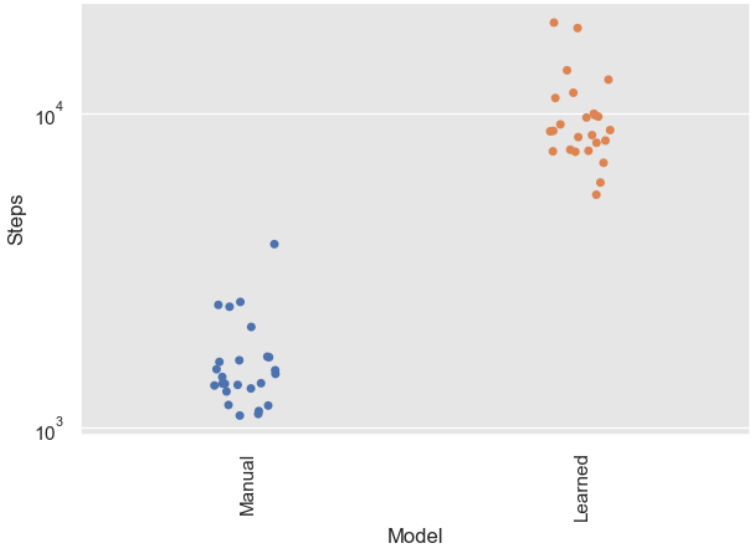

Figure 44: Multiple node decomposition time spent

corners or holes where the agent can sometimes gets stuck in for an extended period of time before working its way out. During evaluation the movement behavior works well on observation.

The targeting node was never able to truly stabilise and, while it achieves its function during evaluation, it does not perform very well by qualitative observation. This could probably be improved by changing the reward formulation as well as the observation space. That indicates that the RL problem definition was not a good one. The targeting functionality taking longer is also evident in the difference in evaluation times. As the manual node is targeting things instantly, that time dimension is completely removed, causing it to perform better in the evaluation. However it is to be emphasised that even though the learned functionality is slower, it is able to complete the whole house building task, which the manual behavior is not.

From the perspective of RL-BT the approach itself works well. Both nodes are able to train meaningful behaviors and although the targeting node is not optimal, it could be improved by changing the RL formulation without altering the RL-BT approach. It can be concluded that the principle of using uninterrupted ticking to a node as an episode for training simultaneous nodes is a sound approach. This is relevant for both Q4 and Q6, as it possible to pinpoint and isolate the problematic areas of the agent as well as train agents with multiple nodes simultaneously.

Regarding Appendix A - It is important to note that the approach does not work with simultaneous learning when the learning nodes are nested into eachother. That is to say, that a high level learning node contains sub-trees with other learning nodes. As the high level node switches branches often, the episodes seen by lower level nodes are short and therefore not very useful. Switching too often without actually reaching the intended episode goals are not good examples of how the agent needs to behave. Because of this, the higher level node is also never able to see any meaningful episodes, as the lower level behaviors are not achieving their goals and the episodes

as a whole do not really represent anything. This approach can still theoretically function reasonably well if the underlying functions are pre-trained. However this is not ideal as it can significantly increase the amount of work, as some type of manual solution is needed in order to manage the training of the sub-behaviors. A possible way to overcome this could be to utilise the Option Framework as proposed in [7] or by propagating rewards up the tree structure as proposed in [9]. This means that in order to provide the full functionality of what is proposed by Q6, more work needs to be done.

8 Summary and conclusions

This section will give an overview of the completed experiments and summarise the conclusions from previous chapters.

8.1 Summary of experiments

We have investigated four different scenarios regarding the use of RL in BTs. These include experimentation with execution and control flow nodes, as well as type specific models and multiple simultaneous learning nodes.

The RL algorithms used in the experiments are very similar to their standard implementation. The only difference is the assurance that all the observations and rewards for learning nodes are seen simultaneously, after the tree has finished a BT tick. As multiple nodes can take an action during a single time-step, this guarantees that all of them also see the observation for the next timestep and get a reward accordingly. This allows each learning node to represent a small isolated RL problem.

The execution node experiment in Section 4 created different types of RL action nodes, based on an existing manually defined BT. The conditions and actions used in a manual implementation were used for defining the action and observation spaces in RL. DQN and PPO were used for training in various different configurations for checking the viability of different RL algorithms in the context of Malmö.

The next experiment focused on the differences between using RL in control flow and action nodes. The experiment in Section 5 was run in a more complex environment to necessitate an expansion of the behavior tree. The original behaviors in the form of action nodes were changed to control sub-trees with additional functionality and modelled as a control flow nodes. The trained models from the previous experiment were reused for evaluation, but new models were also trained for comparison.

The type specific node experiment in Section 6 featured a set of selected problems, which require different behaviors even though they can be solved with the same model definition. This is a problem that is often faced in real world applications, and can often lead to a lot of replication in order to introduce slight modifications for dealing with nuances of different sub-types within the context of one general

problem. This experiment demonstrated one way to use a set of RL models in order to manage this complexity with the help of BTs.

The last experiment in Section 7 was about using learning in a configuration where a learning node has no guarantee of executing for the entirety of a mission. As the problems were no longer isolated, this required a small modification to the RL approach in order to allow the nodes to learn simultaneously. It was possible to facilitate this training by considering consecutive ticks to a node as an episode, meaning that an episode ends when the node is not ticked and a new one starts when it is ticked again.

8.2 Conclusions

Q1 What types of RL algorithms are suited for use in BTs?

The successful execution of the experiments serves as an indication that with a proper approach, the RL problem can be considered separately from the BT. It is shown that RL can be used in BTs and that it should be possible to use most RL algorithms in BTs. Even though most experiments were only run with DQN, the first experiment in Section 2.1.2 additionally featured PPO. As RL can be embedded into BTs as a node, this isolates the RL problem and allows it to be used without large modifications. It is necessary to ensure that the episodes seen by the RL policy remain intact inside the ticking structure of BTs, but this can be assured with only a few assumptions discussed further in the context of Q3. When RL is properly isolated, the choice of algorithm is more or less up to the developer. This also matches the modular design principles of BTs.

Q2 Is it possible to use the elements of normal BT design for learning purposes?

The execution node experiment results show that it is reasonable to use existing conditions and actions from a manually designed tree as input to a RL problem, as proposed in Q2. The nodes can be used without modification as a discrete space. It is also possible to use the observations underlying the conditions for a continuous observation space. This emphasises the similarity between the actions and conditions of BTs to that of the action and observation space of RL. As long as the original set of BT conditions and actions is well defined, the choice of RL algorithm does not seem to be a limiting factor and can be done based on what seems most fitting to solve the task at hand.

Q3 Are the approaches suggested in the existing literature feasible in more complex scenarios?

Q3A Is it feasible to train action nodes that execute an RL behavior directly?

Q3B Is it feasible to create control flow nodes that execute sub-trees based on a learned behavior?

It is confirmed that it is feasible to use RL for both execution (Q3A) and control flow nodes (Q3B), while the approach differs somewhat from earlier works. The experiments were run with a few simple assumptions. First of all, ensuring that all

actions should be taken before observations and rewards are granted, in order to guarantee that a node can take an action regardless of any other nodes. Considering consecutive ticks to a node as an episode is an simple way of extending this principle to multiple nodes, allowing nodes to learn simultaneously. It also allows the same RL policy to be used in different places around the tree, as the principle is able to separate these use cases into different episodes.

It is also worth noting that even though the learning nodes proposed in this thesis only query a single action or node in their action spaces, it would be trivial to extend them for use as fallback or sequence nodes as proposed in previous literature [7][9]. This could be done according to the probabilities for each action according to state, having a set number of highest valued actions to execute or tying the execution of each action to the success and failure conditions according to the fallback or sequence node principle. Even though the approach ends up identical in functionality to the results in other works from a practical point of view, there was no special algorithm required for training. Each node is simply behaving like a small independent RL problem.

Q6 Is it feasible to combine multiple basic RL nodes in a single BT at different levels in the hierarchy?

Simultaneous learning nodes can be used, as long as they are not nested inside sub-trees of other learning nodes. This can be a significant hinderance with the current approach. More work needs to be done in order to learn with nested nodes inside a high-level behavior, which can become unstable. It is still possible to train nested nodes this way, if the lower level nodes are trained first, before the higher level nodes which control them. The Options Framework [7][2] or reward by propagation [9], can be of use to deal with this problem in a more structured fashion. Therefore, more work needs to be done in order to find a reliable solution that always works for the scenario proposed with Q6.

Q4 Does such a decomposition into sub-problems also simplify design and code management? Can it be used for identifying individual problematic parts of the composite behavior during debugging? Is it possible to retrain unsuccessful behavior on the level of single sub-behaviors?

It is hard to say whether using a hierarchical approach gives performance benefits. It is reasonable to conclude that smaller problems are faster to train and set up, however it is hard to say that the decomposition makes the solution better or worse in terms of task performance. It does make things easier from a development perspective however. The modularity of the trained behaviors allows them to be reused in different parts of the tree. This means that it is possible to add new behaviors to the agent, without starting an entirely new training session or modifying the reward function. It allows changes to the agent even if learning is present. This can be done in order to provide safety or for fine tuning and extending behaviors without losing the trained policy. Having isolated policies for specific behaviors also allows the developer to find problematic parts during debugging and add safety guarantees or simply retrain them without affecting other parts of the agent, positively confirming the hypothesis underlying Q4.

Q5 Can a learning based sub-tree in a BT that was initially trained for dealing with a specific type of object in a given task, be duplicated and re-specialised for another type of similar object in the same problem? Is it useful to train a set of similar trees specialised for different types of objects or situations?

This also applies to different sub-types within a problem. It was shown that a BT can manage a set of RL behaviors based on different types within a problem definition, which supports the underlying idea of Q5. By using the same RL model definition and applying a different copy of it for a each type, it is possible to adjust to nuances in a problem that might otherwise start conflicting with each other. While not part of this thesis, it should also be possible to manage this automatically, either with a dedicated custom node or by back-chaining additional actions or entire sub-trees with PPA.

PQ How can existing behavior trees based agents be enhanced with reinforcement learning?

In answer to the primary question PQ of the thesis, it can be concluded from the results that RL can be used reliably in BTs without having to apply special principles, outside of small modifications. The ticking of the tree should be finished before observing the world and reward simultaneously for all learning nodes that were ticked. This matters primarily from the perspective of the BT as the sequences of states, actions and rewards look the same from the perspective of RL. When using multiple RL nodes in a single BT, it is reasonable to consider consecutive ticks to a learning node an episode. There are some problems when dealing with learning within the sub-trees of high-level learning nodes, but these can be amended by training the lower level nodes separately from the higher level nodes.

References

[1] Andrew Y. Ng et al. "Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping". In: *Proceedings of the Sixteenth International Conference on Machine Learning*. ICML '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 278–287. ISBN: 1-55860-612-2. URL: http://dl.acm.org/citation.cfm?id=645528.657613.

[2] Richard S. Sutton et al. "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial Intelligence* 112.1 (1999), pp. 181–211. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(99)00052-1. URL: http://www.sciencedirect.com/science/article/pii/S0004370299000521.

[3] Michael Mateas et al. "A Behavior Language for Story-Based Believable Agents." In: *Intelligent Systems, IEEE* 17 (Aug. 2002), pp. 39–47. DOI: 10.1109/MIS.2002.1024751.

[4] D Isla. "Handling Complexity in the Halo 2 AI". In: (Jan. 2005).

[5] R. Dey et al. "QL-BT: Enhancing behaviour tree design and implementation with Q-learning". In: *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. Aug. 2013, pp. 1–8. DOI: 10.1109/CIG.2013.6633623.

[6] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[7] Renato de Pontes Pereira et al. “A Framework for Constrained and Adaptive Behavior-Based Agents”. In: *CoRR* abs/1506.02312 (2015). arXiv: 1506.02312. URL: http://arxiv.org/abs/1506.02312.

[8] John Schulman et al. “Trust Region Policy Optimization”. In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[9] Yanchang Fu et al. “A Reinforcement Learning Behavior Tree Framework for Game AI”. In: *2016 International Conference on Economics, Social Science, Arts, Education and Management Engineering*. Atlantis Press, 2016. ISBN: 978-94-6252-220-6. DOI: 10.2991/essaeme-16.2016.120. URL: http://dx.doi.org/10.2991/essaeme-16.2016.120.

[10] Matthew Johnson et al. “The Malmo Platform for Artificial Intelligence Experimentation”. In: AAAI - Association for the Advancement of Artificial Intelligence, July 2016. URL: https://www.microsoft.com/en-us/research/publication/malmo-platform-artificial-intelligence-experimentation/.

[11] Tejas D. Kulkarni et al. “Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation”. In: *CoRR* abs/1604.06057 (2016). arXiv: 1604.06057. URL: http://arxiv.org/abs/1604.06057.

[12] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[13] Ziyu Wang et al. “Sample Efficient Actor-Critic with Experience Replay”. In: *CoRR* abs/1611.01224 (2016). arXiv: 1611.01224. URL: http://arxiv.org/abs/1611.01224.

[14] Fraser Allison et al. “Spontaneous Interactions with a Virtually Embodied Intelligent Assistant in Minecraft”. In: ACM - Association for Computing Machinery, May 2017, pp. 2337–2344. URL: https://www.microsoft.com/en-us/research/publication/spontaneous-interactions-virtually-embodied-intelligent-assistant-minecraft/.

[15] Michele Colledanchise et al. “Behavior Trees in Robotics and AI: An Introduction”. In: *CoRR* abs/1709.00084 (2017). arXiv: 1709.00084. URL: http://arxiv.org/abs/1709.00084.

[16] M. Colledanchise et al. “How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees”. In: *IEEE Transactions on Robotics* 33.2 (Apr. 2017), pp. 372–389. ISSN: 1552-3098. DOI: 10.1109/TRO.2016.2633567.

[17] Prafulla Dhariwal et al. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[18] José Hérnandez-Orallo et al. “A New AI Evaluation Cosmos: Ready to Play the Game?” In: *AI Magazine* 38 (Sept. 2017). URL: https://www.microsoft.com/en-us/research/publication/new-ai-evaluation-cosmos-ready-play-game/.

[19] Mathew Monfort et al. “Asynchronous Data Aggregation for Training End to End Visual Control Networks”. In: International Foundation for Autonomous

Agents and Multiagent Systems, May 2017, pp. 530–537. URL: https://www.microsoft.com/en-us/research/publication/asynchronous-data-aggregation-training-end-end-visual-control-networks/.

[20] OpenAI. *OpenAI Baselines: ACKTR A2C*. https://blog.openai.com/baselines-acktr-a2c/. Blog. 2017.

[21] OpenAI. *Proximal Policy Optimization*. https://blog.openai.com/openai-baselines-ppo/. Blog. 2017.

[22] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[23] Yuhuai Wu et al. "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation". In: *CoRR* abs/1708.05144 (2017). arXiv: 1708.05144. URL: http://arxiv.org/abs/1708.05144.

[24] Ashley Hill et al. *Stable Baselines*. https://github.com/hill-a/stable-baselines. 2018.

[25] Christopher Iliffe Sprague et al. "Adding Neural Network Controllers to Behavior Trees without Destroying Performance Guarantees". In: *CoRR* abs/1809.10283 (2018). arXiv: 1809.10283. URL: http://arxiv.org/abs/1809.10283.

[26] Christopher Iliffe Sprague et al. "Improving the Modularity of AUV Control Systems using Behaviour Trees". In: *CoRR* abs/1811.00426 (2018). arXiv: 1811.00426. URL: http://arxiv.org/abs/1811.00426.

A Gathering optimisation node

An additional configuration was created with the purpose of allowing the agent to optimise the use of the gathering and building sub-trees. The definition is changed so that the build and gather actions are called from a learning node. This configuration uses the already described sub-trees, which also form a discrete action space for this new node. The observations space consists of available planks and logs in the inventory, as well as the number of blocks left to place. The changed high-level tree is depicted in Figure 45.

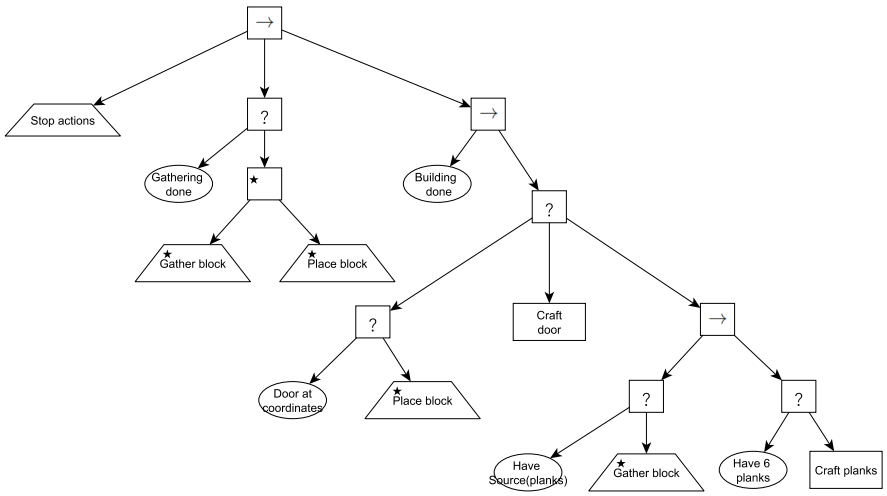

Figure 45: High-level tree definition in learning configuration.

In order to facilitate control over gathering and block placement, the build tree had to be modified as well. The definition is similar, except the gathering is removed and now managed by the learning node seen in Figure 45.

The primary difference of the formulation of a control flow node in this section comes from how the reward signal is defined. While the previous instances of a control flow node define a reward signal manually, this instance assigns a reward to the feedback status returned from a child. An episode is ended when a success is reached. The reward definitions are described in Table 9.

The gathering control flow node uses a smaller two-layer MLP network of 32 and 16 nodes and the DQN algorithm from Stable-Baselines for training.

Status	Success	Running	Failure
Reward	100	-0.5	-10

Table 9: Rewards for alternate control flow node definition

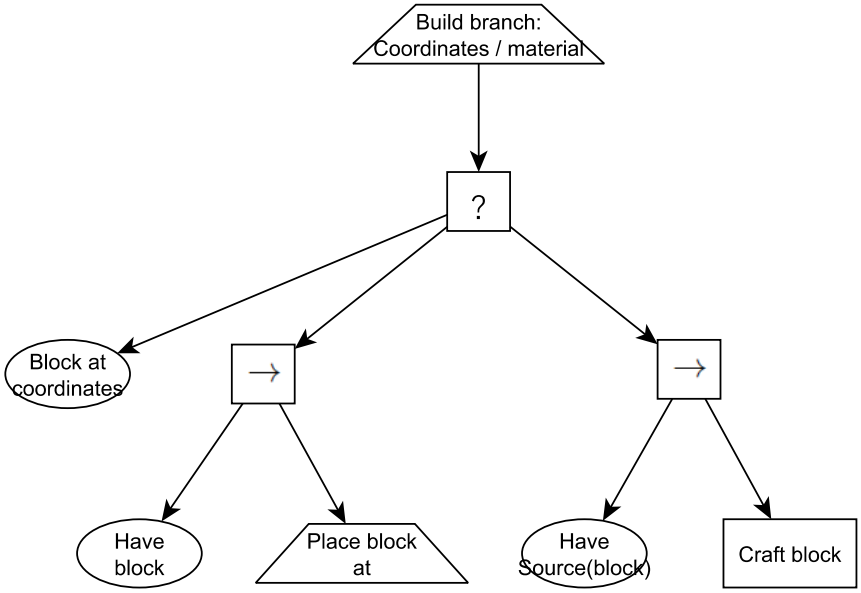

Figure 46: Build sub-tree definition with learning.

TRITA -EECS-EX-2019:577