Agents that learn to behave with reinforcement learning and behavior trees

Rafael Marques Marçal de Sousa

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor: Prof. Rodrigo Ventura

Examination Committee

Chairperson: Prof. José Fernando Alves da Silva
Supervisor: Prof. Rodrigo Ventura
Member of the Committee: Francisco António Chaves Saraiva de Melo

December 2021
Acknowledgments

I would like to thank my supervisors for their guidance. First, to professor Rodrigo Ventura for being an excellent teacher and igniting my interest in the field of AI and for providing his expertise in guiding this work. Secondly, to João Avelino, for his immense patience and support. His invaluable recommendations (which always turned out to be correct) made this thesis possible.

I would like to thank my family and friends for helping me throughout this journey. To my friends from the Capiroto Crew and Nic for cheering me up through any means and memes necessary. To my dear friends Joana, Ana and Alice, for helping me soldier through uni.

To the most amazing and awesome people in the world, Margarida and Leonor, for being amazingly awesome friends.

And finally, I thank my dear Raquel, without whom none of this would have been possible.
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
Abstract

Reinforcement Learning (RL) methods allow the development of autonomous agents capable of performing complex tasks. However, the policies that govern these agents are often represented in ways that are hard to interpret for human users. Additionally, it is not straightforward how to include user guidance in the learning process. This work proposes a novel decanonicalization algorithm that converts a policy obtained from RL into a human legible Behavior Tree (BT), an architecture praised for its modularity and reactivity. Configurable parameters yield a multitude of resulting BTs, from which the optimal ones are obtained with a multicriteria optimization problem. Results in the Mario AI framework show this method can improve on the readability of previous designs, considering the number of nodes and complexity of conditions as a metric for legibility. We also investigated ways to incorporate expert knowledge encoded as a BT in the learning task.

As our pilot attempt, we proposed and evaluated Double $\epsilon$-greedy, which did not achieve the desired outcome of improving learning convergence. Thus, this last problem remains an open challenge that we intend to investigate in future works. The proposed methods can thus can assist designers in testing and debugging, since a readable system allows the users to better predict its behavior, thus cutting on development time and costs.

Keywords

Artificial Intelligence; Reinforcement Learning; Behavior Trees;
Resumo

Os métodos de Aprendizagem por Reforço permitem o desenvolvimento de agentes autónomos capazes de desempenhar tarefas complexas. No entanto, as políticas que governam estes agentes são habitualmente representadas com métodos que são difíceis de ler e interpretar por usuários humanos. Adicionalmente, não é trivial como incluir o conhecimento do utilizador no processo de aprendizagem. O presente trabalho propõe um novo algoritmo de descanonização que converte uma política obtida com Aprendizagem por Reforço numa Árvore de Comportamento, uma arquitetura de controlo prezada pela sua modularidade, reatividade e legibilidade. Parâmetros configuráveis permitem obter uma variedade de árvores resultantes, de entre as quais as melhores são elegidas formulando como um problema de otimização multi critério. Resultados obtidos com a Mario AI framework demonstram que este método consegue melhorar a legibilidade de paradigmas anteriores, considerando o número de nós e a complexidade dos nós condição como uma métrica de legibilidade. Também foi investigado como incorporar o conhecimento de um especialista codificado como uma Árvore de Comportamento no processo de aprendizagem. Como uma primeira abordagem, foi proposto e avaliado o método Double \( \epsilon \)-greedy, o qual não atingiu o resultado desejado de melhorar a convergência da aprendizagem. Portanto, este problema mantém-se um desafio em aberto que será investigado em trabalhos futuros. Os métodos propostos podem assim auxiliar os utilizadores nas fases de teste e depuração, visto que um sistema legível permite que os usuários consigam prever mais facilmente o seu comportamento e portanto poupar no tempo e custos de desenvolvimento.

Palavras Chave

Inteligência Artificial; Aprendizagem por Reforço; Árvores de Comportamento;
Contents

1 Introduction 1
   1.1 Motivation .................................................. 3
   1.2 Objectives .................................................. 3
   1.3 Organization of the Document .............................. 4

2 Background 5
   2.1 Reinforcement Learning .................................... 7
   2.2 Behavior Trees .............................................. 10
      2.2.1 Fundamentals of Behavior Trees ....................... 10
      2.2.1.A Execution Nodes ...................................... 10
      2.2.1.B Control Flow Nodes .................................. 11
      2.2.1.C Control Flow nodes with memory ................... 12
      2.2.2 Execution Example of a Behavior Tree ................ 12
      2.2.3 Behavior Trees Properties ............................ 14

3 Related Work 15
   3.1 Autonomous Generation of Behavior Trees ................. 17
      3.1.1 Behavior Trees and Learning from Demonstrations ...... 17
      3.1.2 Behavior Trees and Planning ........................... 17
      3.1.3 Behavior Trees and Genetic Algorithms ................ 18
      3.1.4 Behavior Trees and Reinforcement Learning ............ 18
   3.2 Banerjee’s method ........................................... 19
   3.3 Current Limitations .......................................... 22

4 Methodology 25
   4.1 System Overview ............................................ 27
   4.2 Proposed Decanonicalization Algorithm ..................... 28
      4.2.1 Objectives .............................................. 28
      4.2.2 Assumptions ............................................ 28
      4.2.3 Inputs/Outputs, Parameters and Metrics ................. 28
# List of Figures

2.1 The BT nodes. From left to right: condition, action, sequence, fallback, parallel and decorator. ................................................................. 12
2.2 A memory sequence 2.2(a) and its equivalent BT 2.2(b) .................... 13
2.3 An example BT. An agent has the task to eat some food. .................... 13
3.1 A Canonical Behavior Tree (CBT): a fallback node with atomic units as children. ................................................................. 20
3.2 The subunits proposed in [1]. From left to right, X, Y and Z. ............... 20
3.3 Example of the left-to-right order simplification. .............................. 22
3.4 Two BTs obtained from Banerjee’s [1] method, with different order of inputs. ................................................................. 23
4.1 System overview. ........................................................................ 27
4.2 A Joiner unit with N children. .......................................................... 32
4.3 The resulting BT from the example, ignoring logic minimization steps. .......... 32
4.4 Two BTs for the same task, adapted from [2]. .................................. 33
4.5 Expansion of a composite unit (a) into several singular units (b). .......... 33
4.6 Example of the effect result of limiting how many predicates are checked. .......... 34
4.7 Double $\epsilon$-greedy scheme. ...................................................... 35
5.1 A screenshot of the Mario AI Benchmark. ......................................... 39
5.2 Mario detecting an enemy in the second column. .................................. 41
5.3 $\epsilon_1$ (top) and $\epsilon_2$ (bottom) as a function of the episode $t$. ............. 42
5.4 Accumulated probability for each outcome of Double $\epsilon$-greedy. .......... 43
5.5 The optimal BT. ......................................................................... 44
5.6 Two handcrafted BTs, one capable of finishing the level 5.6(a) and other who is not 5.6(b) ....................................................... 44
6.1 The resulting BTs and the Pareto frontier. ......................................... 48
6.2 The CBT simplified with logic minimization. ....................................... 49
6.3 BT E from the Pareto frontier. ...................................................... 49
6.4 BT C from the Pareto frontier. ................................................................. 50
6.5 BT A from the Pareto frontier. ................................................................. 50
6.6 BT obtained with Banerjee’s method. ......................................................... 51
6.7 Average reward comparison between \( \epsilon \)-greedy and Double \( \epsilon \)-greedy with different input BTs. ................................................................. 53
List of Tables

3.1 Exemplification of the left-to-right guard simplification process. . . . . . . . . . . . . . . . . 21

5.1 The scores of each BT to use as input in the Double $\epsilon$-greedy method. . . . . . . . . . . . . . 44

6.1 Guards obtained for each action. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

6.2 Configuration of the parameters that resulted in the BTs of the Pareto frontier. . . . . . . . . . . . 48
List of Algorithms

2.1 Q-Learning algorithm .......................................................... 9
2.2 Pseudocode of a Sequence node ........................................... 11
2.3 Pseudocode of a Fallback node ............................................ 11
2.4 Pseudocode of a Parallel node ............................................. 12
3.1 Left-to-right guard simplification ......................................... 21
4.1 Proposed Decanonicalization Algorithm ................................. 30
4.2 Function Hierarchize of the proposed Decanonicalization Algorithm ........................................... 31
# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>BT</td>
<td>Behavior Tree</td>
</tr>
<tr>
<td>CBT</td>
<td>Canonical Behavior Tree</td>
</tr>
<tr>
<td>CBTS</td>
<td>Canonical Behavior Tree Simplified</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm</td>
</tr>
<tr>
<td>LfD</td>
<td>Learning from Demonstrations</td>
</tr>
<tr>
<td>MDP</td>
<td>Markov Decision Process</td>
</tr>
<tr>
<td>NPC</td>
<td>Non Player Character</td>
</tr>
<tr>
<td>RL</td>
<td>Reinforcement Learning</td>
</tr>
<tr>
<td>TD</td>
<td>Temporal Difference</td>
</tr>
</tbody>
</table>
1 Introduction

Contents

1.1 Motivation ................................................. 3
1.2 Objectives .................................................. 3
1.3 Organization of the Document ......................... 4
1.1 Motivation

The goal of Artificial Intelligence (AI) is to develop ways to automate tasks that would otherwise be performed by a human. Virtual and real agents that autonomously learn to behave can save development time and cost and be potentially more effective at complex tasks, like Non Player Character (NPC) behaving in video games or social robots interacting with people. This problem is a hot topic, with researchers actively developing solutions with distinct strategies, the most famous being Learning from Demonstrations (LfD), Planning, Genetic Algorithm (GA) and Reinforcement Learning (RL).

LfD requires the generation of experience traces from a human or virtual agent, a task that can be time consuming. Planning approaches are often limited by the amount of previous knowledge of the system. And not only GAs tend to converge to local optima but they also rely on a proper fitness evaluation function, which can be difficult to design. On the other hand, RL agents can learn autonomously by interacting with the environment, independently of previous existing information.

However, the representations of the policies that govern these agents are difficult to interpret and manually modify. Moreover, it is not always straightforward how to integrate previous knowledge from human behavior design experts since they usually encode it with frameworks like Behavior Trees, Petri Nets, or Finite State Machines models.

Human readability is an important aspect of any control architecture, as it helps to build the proper levels of trust in the system by the designer. A trustable system should be easily explainable and allow the user to agree when decisions make sense and disagree when they do not [3]. Legibility is also crucial for collaborative work, allowing designers to better communicate in large projects, thus alleviating time allocated for testing and debugging [2].

Behavior Trees (BTs) are an increasingly popular control architecture. Having originated in the video game industry [4] as a mean to control NPCs, they have since attracted attention in the fields of robotics, AI and academia [5]. BTs are commonly praised for their modularity (the capacity for being deconstructed in their building blocks), reactiveness (ability to efficiently react to changes in the environment), and human readability [2,6]. BTs have also been shown to generalize previous paradigms, such as Finite State Machines (FSMs), teleo-reactive programs, the subsumption architecture and decision trees [7,8].

It then stands that it could prove useful to express a learned control policy as a BT. The advantageous modularity and legibility of BTs can cover the drawback of RL policies being difficult to read and alter. If built with this purpose, a BT can potentially increase legibility and thus, assist human designers.

1.2 Objectives

In our work, we intend to use Reinforcement Learning and Behavior Trees to learn and represent learned policies that are easy to interpret and modify.
In addition, since a handcrafted BT represents an expert-made policy, we believe that finding a way to integrate this human knowledge in the learning process may boost it, as shown in past works [9, 10].

Our contributions are two-fold. First, we propose an algorithm that converts knowledge gathered through Q-Learning into a human-readable BT. Innovating from other approaches, this algorithm can produce BTs with distinct sets of characteristics (depth and condition size), allowing for a search over options. Second, we study an approach to integrate past knowledge encoded as a BT into the learning task, improving human-designed behaviors.

1.3 Organization of the Document

This thesis is organized as follows: in chapter 2 we lay the background needed to understand the tools used in this work, specifically RL and BT. Then in chapter 3 we analyse the various ways in the literature to autonomously build a BT. Namely, we look at the existing approaches of combining BTs with RL and other non-RL methods, such as LfD, Planning and GA. Chapter 4 starts with an overview of the system developed and its pipeline is briefly explained. Then we introduce our proposed decanonicalization algorithm to convert an input policy into a human legible BT. Finally, we propose Double $\epsilon$-greedy, a method to introduce expert knowledge coded as a BT into the learning process. The evaluation domain used, the Mario AI framework, is described in chapter 5. The considerations taken for the RL context are also set. The results of the methods developed are shown and discussed in chapter 6. First we analyze the proposed decanonicalization and then the Double $\epsilon$-greedy method. Finally, the contributions of this work are summarized in chapter 7. Additionally we discuss the limitations of the methods developed and provide possible directions for future work.
2 Background

Contents

2.1 Reinforcement Learning ........................................... 7
2.2 Behavior Trees ..................................................... 10
The present chapter describes the tools used for this work: RL and BTs. First we look at RL in section 2.1, explaining its basics and describing the Q-Learning algorithm. Then the control architecture of BTs is shown and its core functionalities and properties are addressed in section 2.2.

## 2.1 Reinforcement Learning

Reinforcement Learning problems [11] have their basis on the Markov Decision Process (MDP). MDPs are defined by the tuple \( \langle S, A, R, P \rangle \), where \( S \) is a finite set of states that the agent and the environment can be at a given time, \( A \) is a finite set of actions the agent can take, \( R \) is the reward function and \( P \) is the transition model of the system. \( P \) specifies the probability of transitioning from state \( s \) to state \( s' \) by performing action \( a \) and is given by:

\[
P_{s \to s'}^a = P(s'|s, a)
\]  

(2.1)

Similarly, the expected reward from transitioning from state \( s \) to state \( s' \) with action \( a \) is

\[
R_{s \to s'}^a = R(s'|s, a)
\]  

(2.2)

MDPs also have the Markov Property, which specifies that the environment’s response, whether regarding state transition or reward, only depends on the current state and action taken and is independent on the past history of states.

The goal of RL is for the agent to obtain the maximum expected return \( R_t \), which is defined as the sum of discounted rewards:

\[
R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}
\]  

(2.3)

where \( \gamma \in [0, 1] \) is the discount factor. It can be seen as a measure of how important future rewards are when compared to immediate rewards. A higher \( \gamma \) means that future rewards have comparable importance to immediate rewards, while a lower \( \gamma \) signifies that immediate rewards are more important. The inclusion of the discount factor is to ensure that the sum of rewards is finite and the problem is bounded.

The agent’s objective is then to learn the optimal policy \( \pi^* : S \to A \), i.e., a mapping from state to actions, that maximizes the sum of expected discounted rewards for each state \( s \) encountered.

Reinforcement Learning algorithms are often based on value functions. The value (or utility) of a state \( V^\pi(s) \) is the expected value of the sum of discounted rewards when starting from state \( s \) and following policy \( \pi \). In other words, \( V^\pi(s) \) is a measure of how good it is for the agent to be in that state.
It is given by:

\[ V^\pi(s) = E(R_t|s_t = s) = E(\sum_{n=0}^{\infty} \gamma^n r_n|s_0 = s, \pi) \]  

(2.4)

Often times instead of considering the value of a state, it is preferable to consider the value of a state-action pair, known as a Q-value. The Q-value \( Q^\pi(s, a) \) represents the utility of taking action \( a \) in state \( s \) and following policy \( \pi \) thereafter and is given by:

\[ Q^\pi(s, a) = E(\sum_{n=0}^{\infty} \gamma^n r_n|s_0 = s, a_0 = a, \pi) \]  

(2.5)

Equations 2.4 and 2.5 can be rewritten as equations 2.6 and 2.7, respectively, where \( \pi(s, a) \) is the probability of policy \( \pi \) taking action \( a \) while on state \( s \). These are known as the Bellman equations. With them, it is possible to obtain optimal strategies using dynamic programming. Methods such as policy or value iteration can converge to the optimal policy. However, it is necessary to know or estimate the transition model of the system \( P_{ss'} \). Such methods are called model-based.

\[ V^\pi(s) = \sum_{a \in A} \pi(s, a) \sum_{s' \in S} P_{ss'}^a (R_{ss'}^a + \gamma V^\pi(s')) \]  

(2.6)

\[ Q^\pi(s, a) = \sum_{s' \in S} P_{ss'}^a (R_{ss'}^a + \gamma \sum_{a' \in A} \pi(s, a')Q^\pi(s', a')) \]  

(2.7)

Estimating the transition model is possible, by tracking each action and state transition and considering \( P(s'|s, a) \) from the frequency \( s' \) is reached from performing \( a \) in \( s \). Given infinite tries, the estimated model converges to the real model. However, this has several drawbacks. For large state spaces it might be unfeasible to try every state transition enough times to obtain a confident model. The method also heavily relies on the estimated model being correct, which might not hold true.

Another solution that avoids solving the transition model is to use observations to adapt the utilities of visited states in a way that conforms with the Bellman equations. The idea is for the agent to interact with the environment and learn from experiences. By observing states, actions taken, and rewards received, the utility values are updated, all without requiring learning the model of the system. Such methods are model-free. There exists a variety of classical model-free approaches, such as Monte Carlo and Temporal Difference (TD) methods.

The Monte Carlo method requires samples of experiences - sequence of states, actions and rewards observed while interacting from the environment. It is an episodic method, since it needs to observe an entire episode before values can be estimated and policies changed. The utility of a state is obtained by averaging the returns acquired after visiting that state. A basic Monte Carlo method is given by

\[ V(s_t) \leftarrow V(s_t) + \alpha[R_t - V(s_t)] \]  

(2.8)
where \( \alpha \in [0,1] \) is the learning rate. It is a measure of how much new experiences override previous knowledge. Higher \( \alpha \) means new observations have greater impact when updating the utility values.

Unlike Monte Carlo methods, with TD the agent does not need to wait until an episode is complete to know the return. Instead, the agent only needs to take one or multiple steps in consideration. In the simplest case of one-step TD (or TD(0)), the agent updates the state’s utility based on the immediate reward received after one step, following:

\[
V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]
\]  

(2.9)

One important breakthrough in the field of RL was the development of Q-Learning [12]. It is a TD model free method. The Q-Learning method is shown in Algorithm 2.1. Q-values are stored in a table and initialized, typically to zero. Then, at each time step the agent is in current state \( s \in S \), chooses an action \( a \in A \) and transitions to state \( s' \). This transition can depend on both the previous state and the action taken. The agent then receives a reward \( r \) and updates the Q-Value for the current state, following equation 2.10. An episode ends when a terminal state is reached.

\[
Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{b \in A} Q(s',b) - Q(s,a))
\]  

(2.10)

**Algorithm 2.1: Q-Learning algorithm**

1. Initialize all \( Q(s,a) \)
2. for each episode do
3.     Initialize \( s \)
4.     for each step do
5.         Choose \( a \) with, for example, \( \epsilon - greedy \)
6.         Take action \( a \)
7.         Observe \( s' \) and \( r \)
8.         Update \( Q(s,a) \) following equation 2.10
9.         \( s \leftarrow s' \)
10. until \( s \) is terminal

The final deterministic policy is obtained by assigning to each state the action with the highest corresponding utility, following:

\[
\pi^*(s) = \arg\max_a Q(s,a)
\]  

(2.11)

Q-Learning is an off-policy method (in contrast to on-policy approaches) because the learned action-value function converges to optimality independently of the policy being used to generate data and explore the environment.

One known problem in RL is that of the exploration/exploitation trade-off. The question is whether the agent should perform the actions it already knows are good (exploitation) or try a new action to possible
achieve higher rewards in the future (exploration). One classic and simple method to deal with this issue is with \( \epsilon \)-greedy. During the learning process the agent takes a random action with probability \( \epsilon \) and with probability \( 1 - \epsilon \) a greedy action, i.e, the current best action estimated so far.

### 2.2 Behavior Trees

A Behavior Tree (BT) [2] is a control architecture for an autonomous agent. It can be seen as an action switching mechanism, selecting what action should be taken given the current state of the system [7]. Its main advantages are modularity, reactivity and readability, three important aspects for development and maintainability.

BTs were first introduced in the videogame industry as a mean to control Non-Playable Characters [4]. In this industry, modularity and reusability of code are of utmost importance, a fact that has propelled BTs to become increasingly popular. They have since attracted attention in the fields of robotics, AI and academia [5].

#### 2.2.1 Fundamentals of Behavior Trees

Formally speaking, a BT [2] is a directed acyclic rooted graph. A series of nodes starts at the root and is linked to other nodes by connections. A node is a parent when linked to another node below, called its child. The root is the only node without a parent, while all other nodes have exactly one parent.

A BT’s operation is based on signals called ticks. The root emits ticks with a given frequency and propagates them towards the rest of the tree. A node is executed when and only when it receives a tick. When ticked, a node returns one of three possible return status to its parent: Success, Failure or Running. Running means a task is still ongoing, while Success and Failure mean the task has finished successfully or not, respectively. Nodes are classified as control flow or execution nodes, which are explained below.

#### 2.2.1.A Execution Nodes

Execution nodes are leaf nodes, meaning they have no children. There are two types of execution nodes: action and condition.

The Action node executes a command when a tick is received. If the action is ongoing, it returns Running. If it completes successfully/fails it returns Success/Failure. Action nodes can have internal implicit conditions that measure the execution status, which are used to determine the return value. In this work we represent them as a green box, with the name of the action written inside.
When ticked, a Condition node checks for a condition or proposition. If the proposition holds true, the node returns Success. Otherwise it returns Failure. The propositions checked are usually some observable environmental variable or an internal perception of the agent. Notably, a condition node never returns Running. Condition nodes can be seen as a special type of action nodes that immediately return Success or Failure and do not change the environment. For this work, we represent them as a yellow oval/box, with its proposition to be checked written inside.

2.2.1.B Control Flow Nodes

Control flow nodes are non-leaf nodes and govern tree traversal logic by rerouting ticks to other nodes. There are four types of control flow nodes: fallback, sequence, parallel and decorator.

The Sequence node ticks its children in left-to-right order until a child returns either Failure or Running. Then, it returns that status to its own parent. Success is returned if and only if all of the node's children return Success. The sequence node is analogous to the logic gate AND. The execution algorithm for the Sequence node is represented in Algorithm 2.2 and its graphical symbol is "→".

Algorithm 2.2: Pseudocode of a Sequence node

```pseudocode
for i ← 1 to N do
    childStatus ← Tick(child(i))
    if childStatus == Running then
        return Running
    else if childStatus == Failure then
        return Failure
    return Success
```

The Fallback node (also referred to as Selector) operates in contrast to the sequence node. It ticks its children in left-to-right order until a child returns either Success or Running. Then, it returns that status to its own parent. Failure is returned if and only if all of the node's children return Failure. The fallback node is analogous to the OR logic gate. The execution algorithm for the Fallback node is represented in Algorithm 2.3 and its graphical symbol is "?".

Algorithm 2.3: Pseudocode of a Fallback node

```pseudocode
for i ← 1 to N do
    childStatus ← Tick(child(i))
    if childStatus == Running then
        return Running
    else if childStatus == Success then
        return Success
    return Failure
```
The Parallel node ticks all of its children simultaneously. Considering \( N \) the number of children and \( M \) a user defined parameter such that \( M \leq N \), the node returns \textit{Success} if at least \( M \) children return \textit{Success} and \textit{Failure} if at least \( N - M + 1 \) children return \textit{Failure}. Otherwise it returns \textit{Running}. The execution algorithm for the Parallel node is represented in Algorithm 2.4 and its graphical symbol is ".-".

\begin{algorithm}
\caption{Pseudocode of a Parallel node}
\begin{algorithmic}[1]
\For {\( i \leftarrow 1 \) \text{to} \( N \)}
\State \( \text{childStatus} \leftarrow \text{Tick}(\text{child}(i)) \)
\If {\( \sum_i \text{childStatus}(i) = \text{Success} \geq M \)}
\State \text{return Success}
\ElsIf {\( \sum_i \text{childStatus}(i) = \text{Failure} > N-M+1 \)}
\State \text{return Failure}
\Else
\State \text{return Running}
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}

The Decorator node has a single child. It operates by either manipulating the return value of its child, selectively ticking it following a predefined rule or both. For example, a decorator node can invert the \textit{Success}/\textit{Failure} return value of its child, ticking it until it returns \textit{Success} or try a determined amount of ticks until it returns \textit{Failure}. Graphically represented with a rhombus.

Figure 2.1 shows the graphic representation of all the nodes mentioned.

\textbf{Figure 2.1:} The BT nodes. From left to right: condition, action, sequence, fallback, parallel and decorator.

\subsection{Control Flow nodes with memory}
Control flow nodes with memory can avoid repeatedly re-executing some nodes. They keep in memory the last child that has returned \textit{Success}/\textit{Failure}. That child and the children to its left are not ticked again until the control flow nodes finishes in either \textit{Success} or \textit{Failure}. Then, its memory is cleared, enabling the node to again reconsider all of its children and tick the leftmost nodes. The graphical representation of a memory node is the same as its normal control flow node counterpart with the addition of an asterisk.

Adding a control flow node with memory is equivalent to adding conditions checking whether actions have finished successfully or failed. Figure 2.2 shows a memory sequence and its equivalent non-memory representation.

\subsection{Execution Example of a Behavior Tree}
The following example illustrates the execution of a BT. We have a BT, shown in Figure 2.3, that governs the actions of a hungry agent who needs to eat. The fallback node at the top ticks its first child, a
condition node. The agent has not eaten yet, therefore it is not full and the condition returns Failure. The fallback then routes the tick to the sequence node and the signal is propagated until it reaches the second condition node. The agent has no food yet, and thus that node returns Failure, which prompts the action to grab food. When that action succeeds, the sequence node prompts the agent to eat its food. When the agent finishes eating the Eat Food node returns Success, which gets propagated to the sequence node and then the top fallback. The tree has returned success and the task is done.

To illustrate the reactivity of BTs, let us now imagine a malicious agent appears and steals the food while the agent is eating. The action fails and the tree returns Failure. On a subsequent tick, the tree is traversed again and the agent rechecks if it is full. It is not, since it did not finish eating, thus re-executing the action Grab Food and later, Eat Food. Should the malicious agent again grab the food but this time place it in our agent's hand, it would then see that it still has food and be prompted to eat it. It would not need to grab food again.
2.2.3 Behavior Trees Properties

In this subsection we describe some of the main properties and positive aspects of BTs.

BTs are modular [6]. Modularity is the system's capacity to be deconstructed into its most elementary parts and change or reuse them to compose new structures.

The advent of BTs originated from a need of modularity, which was lacking in previous designs such as Finite State Machines [2]. In a FSM, it is each state who decides what to do next and each state needs to encode its transitions. This is a case of a one-way control transfer and emulates the obsolete Goto statement in programming languages. Adding or removing a state requires a possibly large number of transitions to have to be analysed. This hurts scalability and reusability, making the design of a FSM difficult and prone to error. On the other hand, in a BT it is the parent of a node who controls the command flow. A node activates its child, gets a return value and then decides what to do next. This is a form of a two-way control transfer, which, repeating the parallelism to programming languages, is equivalent to the widely used function calls. This makes BTs highly modular, as there are few dependencies between its components. Subtrees can be designed and tested separately and then put together or used in other parts of the tree.

BTs are reactive [2, 7]. Reactivity is the capacity for the system to quickly and efficiently react to changes in the environment, independently of the past history of inputs. The periodic ticking by the root and constant checking of the environment by condition nodes allow the tree to respond to changes. The Running status allows for actions to be interrupted in favor of more important commands. Action nodes signaling their Success and Failure statuses allow control flow nodes to handle these cases by redirecting ticks to other nodes. Therefore, BTs are reactive since they respond to changes in the environment.

BTs are human readable [2]. The modularity of BTs and the fact that they can be built with an hierarchical structure, and thus incorporate several layers of decision making, make them easy to interpret by humans. This is an important factor for their design and development and it also eases collaborative work.
3 Related Work

Contents

3.1 Autonomous Generation of Behavior Trees ........................................... 17
3.2 Banerjee’s method ............................................................................. 19
3.3 Current Limitations ......................................................................... 22
This chapter analyses various approaches in the literature to autonomously generate a BT or to improve upon an initial one. In section 3.1 we look at some existing alternatives to RL, particularly Learning from Demonstrations (LfD), Planning and Genetic Algorithms (GA). Then explore approaches combining RL with BTs. In section 3.2 special detail is given to Banerjee’s [1] method, since this work follows its pipeline. Finally we address some of the drawbacks that the existing paradigms have in section 3.3.

3.1 Autonomous Generation of Behavior Trees

3.1.1 Behavior Trees and Learning from Demonstrations

In [13, 14] LfD is used to generate a BT. A human agent plays a task and the experience traces are recorded. These are used in a decision tree algorithm to generate a set of conditions, which are then simplified with a greedy algorithm [14] or logic minimization [13]. Action nodes are paired with guard nodes containing conditions from the rules and placed under a parallel node, finalizing the BT.

In [15], a BT is built based on traces from a player in a real time strategy game. The initial BT is highly overfit and is reduced algorithmically by finding and merging common patterns of actions, with the process finishing when no more common patterns are present. The final BT contains significantly less nodes than the initial one.

3.1.2 Behavior Trees and Planning

Collendachise et al. [16] propose a method to incorporate automated planning with BTs, based on the idea of backward chaining. Taking as input a Precondition-Postcondition-Action table, the initial BT contains only the goal condition. When that condition does not hold, the BT is expanded by adding the actions that fulfill that condition and their corresponding preconditions. If any of the added conditions fail, a similar BT with actions and preconditions is incorporated. This process repeats and the BT is incremented until the original goal is met. The method is reactive, in the sense that the agent will skip or repeat an action should an external force help or hinder the agent.

In a followup work, Ögren [17] studies the convergence guarantees of backward chained BTs. It is shown that if the BT is non-conflicting (meaning it locally satisfies its postcondition in finite time and does not violate previously achieved sub-goals) and its equivalent AND-OR tree is satisfied, convergence is guaranteed in finite time.
3.1.3 Behavior Trees and Genetic Algorithms

In [18], grammatical evolution was used to develop a BT to navigate a platform game. The structure of the BTs was subject to some constraints: the root is a selector node containing various subtrees; each subtree consists of a sequence with one or two conditions followed by a sequence of one or more actions or subtrees; the rightmost child of the root is left unconditioned. The resulting controller was sent to the 2010 Mario AI competition and reached fourth place [19]. Dead ends proved to be difficult for the agent to traverse. To counteract this, an A* planner was introduced to add path planning behaviors to the action pool [20].

In [21], a mixture of genetic programming and a greedy algorithm is used to evolve a BT. A one step greedy search is used to try each action until it finds one that leads to an increase in fitness value. If no such actions are found, it resorts to genetic programming. Finally, an anti-bloat algorithm is used to remove potentially ineffective subtrees, by trying to run the BT without a subtree. If the fitness decreases the subtree is kept, otherwise it is removed.

Styrud et al. [22] combine the planning approach from [16] with genetic algorithms. A backward chained BT originated from planning is introduced in the initial population of the genetic approach and forcefully kept throughout its generations. The evaluation function includes a penalty for each node in the tree, in order to encourage the creation of more compact trees. The final BTs obtained were shown to outperform the baseline genetic programming method and the input backward chained BT. Notably, the behaviors had an improved ordering of behaviors and removed redundant nodes. However, the removal of condition nodes might arguably make the final tree less reactive.

3.1.4 Behavior Trees and Reinforcement Learning

Dey & Child [23] propose a method to integrate Q-Learning in BT design, with the objective of assisting developers by reducing the amount of condition nodes needed. The lowest level sequences of an input BT are used as the actions in a Q-Learning phase, resulting in a Q-Value table. Each sequence is given a Q-Condition node, a proposed type of node containing the highest utility states from the table associated to that sequence. In essence, this replaces the condition nodes usually found in the beginning of a sequence with a single, more compact node. Finally, the tree topology is reorganized by sorting each node's children by their maximum Q-Value.

Pereira & Engel [24] propose a framework that uses RL to add learning capabilities to BTs. Two nodes are proposed: learning action node and learning fallback node. The former encapsulates a Q-Learning algorithm, internally containing user-defined states, actions and reward structure. The objective is to learn which action to apply to each state. The learning fallback node differs by considering as actions the children of the node, with the objective of learning which child to tick given the current state.
Zhang et al. [25, 26] extend on the previously mentioned work by allowing nodes to store accumulated discounted rewards and the number of time steps elapsed since the node has been activated. When the node finishes execution, both values are propagated upwards, alongside with the Success or Failure return status. After the learning process is done, the tree’s topology is reorganized based on the expected utility of each learning node’s children. A similar line of ideas of propagating rewards from bottom up is explored in [27].

Kartasev [28] uses the condition and action nodes of an input BT as the state and action space in an RL process, essentially condensing the problem into a single node. Additionally, they consider learning nodes, similar to the principles in [24], and show that they can be used simultaneously, by considering uninterrupted ticks to a node as an RL episode. Nonetheless, nesting a learning node inside the subtree of another learning node proved to be inefficient.

### 3.2 Banerjee’s method

Banerjee’s [1] approach is explained in greater detail in this section, since our work follows its pipeline. They use RL to translate a policy obtained from Q-Learning as a human legible BT. Differing from other works employing RL, their method does not require an input BT. Instead of refining or expanding an already existent tree, a BT is directly created from scratch. The idea is that the designer only needs to program the actions and the agent will autonomously acquire its BT.

In the RL context, states are composed by a set of Boolean variables (predicates) in disjunctive normal form (a sum of products). Visited states are the minterms, expressions containing each variable or its negation (called a literal) exactly once. $Minterms(A_i)$ represents the set of states that are mapped to action $A_i$ according to the policy obtained with equation 2.11. States not seen in the learning process (either because the agent never visited them or because they are impossible given the problem formulation) are don’tcares and can have any action associated to them.

Performing logic minimization over minterms and don’tcares yields the guards for each action, following:

$$g_i = \text{LogicMinimization}(\text{Minterms}(A_i), \text{dontcares})$$

The guard $g_i$ is seen as a rule or a condition that must hold true in order for action $A_i$ to execute. Notably, due to the logic minimization, the guards are more compact than their corresponding minterms.

Banerjee also proposes the notion of a Canonical Behavior Tree (CBT). The CBT has a fixed structure: a fallback node at the root, having as children a set of what we will call atomic units. An atomic unit consists in a sequence node with two children: on the left, a condition node with argument $g_i$; on the right, an action node with argument $A_i$. The CBT is quite similar to the results obtained in [13, 14],
differing from those works by using a fallback node at the root instead of a parallel node. Figure 3.1 shows a CBT with atomic units.

![Figure 3.1: A Canonical Behavior Tree (CBT): a fallback node with atomic units as children.](image)

In order to convert the CBT into a BT closer to what a human might design, and thus making it more legible, a decanonicalization algorithm is proposed. The method operates on two basic principles: combining guard-action pairs into more compact structures and using logic minimization steps to shorten the guards in condition nodes. Figure 3.2 shows the subunits used in that work.

To implement sequential behaviors in the resulting BT, Banerjee adds to the learning process a predicate $A_i$-done for each action. This enables the use of memory sequence nodes, seen in their subunits X and Y in Figures 3.2(a) and 3.2(b), respectively.

![Figure 3.2: The subunits proposed in [1]. From left to right, X, Y and Z.](image)

One of the steps they use in their decanonicalization algorithm is Algorithm 3.1. It makes use of the left-to-right ordering of ticking in order to further simplify condition nodes. The minterms of each subunit are sequentially added to the don'tcares, allowing for more compact guards. This causes the right-most action to invariable have its guard as True, hence why units are ordered by guard length: the longest guard is shortened to True. An important aspect to note is that this method relies on the assumption that actions do not fail.

The following example illustrates this process. Let us suppose we have four actions ($X$, $Y$, $Z$, $W$) and our states are expressed with three Boolean variables $ABC$. Q-Learning maps visited states (minterms)
**Algorithm 3.1**: Left-to-right guard simplification

**Input**: The sets of units and dontcares $dc$

1. **Function** Left-to-right simplification($units$, $dc$):
   2. Order units by increasing length of guard $g$
   3. for unit $X_i$ in the defined order do
   4. \[ g_i \leftarrow \text{LogicMin}(\text{minterms}(X_i), dc) \]
   5. \[ dc \leftarrow \text{dontcares} \cup \text{minterms}(X_i) \]

Performing Algorithm 3.1 transforms the CBT into the BT shown in Figure 3.3(b). At each step of the loop the minterms of each unit are added to the dontcares. The row **Dontcares** in Table 3.1 shows the dontcares after each passage of the loop. Performing logic minimization over the modified dontcares yields the final guards. This process essentially makes use of the left-to-right order of ticking to pass information to the right. If the guard $A$ of the first unit does not hold true, this can be used to simplify the guards of the following units. The second unit’s guard, $\bar{A}\bar{B}$, can be shortened to $\bar{B}$, since it is already known that $A$ is false. The last unit’s guard is True, hence why the condition and sequence nodes can be omitted.

---

**Table 3.1**: Exemplification of the left-to-right guard simplification process.

<table>
<thead>
<tr>
<th>Action</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Minterms</strong></td>
<td>$\overline{ABC}$</td>
<td>$\overline{ABC}$</td>
<td>$\overline{ABC}$</td>
<td>$\overline{ABC}$</td>
</tr>
<tr>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
</tr>
<tr>
<td>$AB\bar{C}$</td>
<td>$AB\bar{C}$</td>
<td>$AB\bar{C}$</td>
<td>$AB\bar{C}$</td>
<td>$AB\bar{C}$</td>
</tr>
<tr>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
</tr>
<tr>
<td>$\bar{ABC}$</td>
<td>$\bar{ABC}$</td>
<td>$\bar{ABC}$</td>
<td>$\bar{ABC}$</td>
<td>$\bar{ABC}$</td>
</tr>
<tr>
<td><strong>Dontcares</strong></td>
<td>$-$</td>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
</tr>
<tr>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
</tr>
<tr>
<td>$AB\bar{C}$</td>
<td>$AB\bar{C}$</td>
<td>$AB\bar{C}$</td>
<td>$AB\bar{C}$</td>
<td>$AB\bar{C}$</td>
</tr>
<tr>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
<td>$ABC$</td>
</tr>
<tr>
<td>$\bar{ABC}$</td>
<td>$\bar{ABC}$</td>
<td>$\bar{ABC}$</td>
<td>$\bar{ABC}$</td>
<td>$\bar{ABC}$</td>
</tr>
<tr>
<td><strong>Initial guard</strong></td>
<td>$A$</td>
<td>$\overline{AB}$</td>
<td>$\overline{ABC}$</td>
<td>$\overline{ABC}$</td>
</tr>
<tr>
<td><strong>Guard after Logic Min</strong></td>
<td>$A$</td>
<td>$B$</td>
<td>$C$</td>
<td>True</td>
</tr>
</tbody>
</table>
3.3 Current Limitations

Learning from demonstration techniques have the disadvantage of requiring to generate a set of sampled data to be imitated from. This could mean requiring a human agent to play a level in a game or performing a task, something that can be tedious and time consuming. Additionally, the human agent might perform the task in a suboptimal manner, likely leading to the learned behavior to also be suboptimal.

Planning approaches are often limited by the amount of previously existing information about the environment, which might not be available without interacting with it first.

Genetic algorithms make good use of the modularity of BTs and do not require an initial BT. However, they heavily rely on the proper design of a fitness function to evaluate the tree as a whole, a fact that can prove difficult. Additionally, these methods have a tendency to converge to local optima.

RL methods, just like genetic algorithms, operate on the principle of interacting with the environment to generate knowledge. But the reward structure is tied to the task level and is often easier to construct than the fitness function of genetic approaches [1]. When applying RL to BTs, most of the mentioned methods attempt to optimize a set of designated nodes. This has the advantage of working with a smaller state space, effectively dividing the issue into smaller subproblems [2]. These methods, however, require an input BT. Also, the non-learning portion of the tree might be suboptimal. Additionally, by having nodes encapsulating an RL policy, states and associated actions are internally hidden. This black box approach may be positive if abstraction is desired but it can negatively impact legibility.

Banerjee's [1] approach does not require an initial BT. Instead it converts a learned RL policy into...
a BT via a decanonicalization algorithm. However, their method only groups subunits directly below the root, essentially only having them at the top portion of the tree. It could prove useful to expand subunits into greater depths of the tree as well, as creating a vertical hierarchic structure might improve legibility [2]. Also, two structures are paired into a subunit as soon as both match the criteria of having a predicate in opposition. Not only this can lead to a unit being paired with another unit with few guard commonality, but the a-priori ordering of inputs can affect the final BT. Figure 3.4 shows this happening: if behaviors GC and DC are checked first, the result is the BT in Figure 3.4(a). This is the result shown in that work. If instead, for example, GA and GC are seen first, the result is the BT in Figure 3.4(b). Additionally, through some tests we noted their subunit Z (Figure 3.2(c)) can introduce some unwanted sequentiality: when behavior Bi finishes successfully, if condition c holds true, it will prompt the agent to start behavior Bj, even if its guard c is violated, thus not fully respecting the RL policy.

![Diagram](image_url)

**Figure 3.4:** Two BTs obtained from Banerjee's [1] method, with different order of inputs.
4 Methodology

Contents

4.1 System Overview .................................................. 27
4.2 Proposed Decanonicalization Algorithm .......................... 28
4.3 Using past Expert Knowledge ................................. 34
This chapter explains the methods developed for this work. First, a brief overview of the system and its pipeline is shown in section 4.1. Then we detail our proposed decanonicalization algorithm, whose purpose is to convert a policy into a readable BT in section 4.2. Finally, we address the challenge of introducing expert knowledge coded as a BT to assist in the learning process via our proposed Double $\epsilon$-greedy method in section 4.3.

4.1 System Overview

To generate a human readable BT with RL and expert knowledge, we propose the system depicted in Figure 4.1. Its input can either be empty or optionally, a previously designed BT. In the latter case, the BT handcrafted by an expert will guide the agent in a proposed exploration-exploitation scheme (section 4.3), with the objective of using the designer’s knowledge to improve learning efficiency.

If no initial input is given, regular Q-Learning with $\epsilon$-greedy is used to generate a Q-Value table. Then, we follow Banerjee’s [1] pipeline, which we briefly summarize: visited states are considered the minterms with the rest being dontcares. Logic minimization over both yields a set of guard-action pairs, which are translated as atomic units and form a CBT.

Then our proposed decanonicalization algorithm (section 4.2) transforms the CBT into a readable BT. Innovating from their approach, our method contains configurable parameters that allow to to obtain a plethora of BTs, differing in characteristics such as depth and number of nodes. Formulating it as a multicriteria optimization problem allows us to identify the best candidates: the ones belonging in the Pareto frontier (subsection 4.2.3). This system essentially represents the tabular Q-Learning knowledge into a BT that is easier to understand by human users.

Figure 4.1: System overview.
4.2 Proposed Decanonicalization Algorithm

4.2.1 Objectives

The main objective of this algorithm is to convert a learned control policy into a human legible BT. For this, we propose a method in the vein of Banerjee's. Namely, our method operates on the same principles of merging units into more compact structures and employing logic minimization steps. At the same time we attempt to solve the issues mentioned previously in section 3.3.

Thus, the objectives of this method are the following:

1. Units can be merged together into more compact structures;
2. Employing logic minimization steps to simplify condition nodes;
3. Units should be paired with other units with the most similar guards;
4. The BT should be able to be expanded vertically, creating a notion of hierarchy;
5. The output BT should be independent of the a-priori order of the inputs;
6. The resulting BT must faithfully express the input policy.

Additionally, we propose configurable parameters, which are set prior to the use of the algorithm. These affect the resulting BT and allow to obtain a multitude of BTs with different characteristics, enabling a search over options for the best solution. They are explained in greater detail afterwards in subsection 4.2.5.

Besides the above objectives, we also aim to improve upon the legibility of Banerjee's method. In order to compare both approaches, an objective metric for legibility is proposed in subsection 4.2.3.

4.2.2 Assumptions

Similar to [1], this method assumes that actions do not fail. This allows the use of the left-to-right guard simplification step explained in section 3.2.

Another consideration is that the processing of nodes takes negligible time when compared to the action and real world scale [7]. That is, the ticking frequency is such that it allows the root to send a signal and receive a return value before a new tick is sent. This implies that the computational time of nodes is minimal and therefore, adding more nodes to a BT has no visible impact on its performance.

4.2.3 Inputs/Outputs, Parameters and Metrics

The input of the algorithm is a set of guard-action pairs, expressed as atomic units, resulting from Q-Learning. Its output is a decanonicalized BT, a BT that might be easier to understand by users.
One guarantee that must be kept is that the resulting BT faithfully represents the learned policy. For this, we propose a notion of equivalency between a BT and a policy:

**Definition 4.2.1 (Data-equivalent Behavior Trees).** A BT is data-equivalent to a policy \( \pi \) if for every minterm state the BT executes the same action as the policy.

Notice how this definition leaves out the don'tcares. Two BTs that are data-equivalent with respect to the same policy may not necessarily map to the same action when faced with a don'tcare state. Since don'tcares can have any action associated to them, it is undetermined what action will be related to it. The action depends on the ordering of behaviors along the tree.

Computational time is also left out of this definition. While having more nodes can imply more computational time, as per the assumption made previously in subsection 4.2.2, its impact is minimal.

In order to obtain a legible BT a metric for legibility must be defined. To measure the readability of a BT we will consider the following hypothesis: a BT is more legible the less nodes it contains and the less complex its condition nodes are. For this reason we aim to minimize two criteria:

- The total number of nodes in the BT;
- The total amount of literals in the arguments of all condition nodes.

Each separate occurrence of a variable, either in positive or negated form, counts as a literal. For example, a condition node with guard \( g = AB\overline{C} \lor \overline{A}C \lor BC \) would amount for 7 literals in the second criteria.

One way to analyze multi objective problems is with the Pareto frontier [29]. It allows to distinguish between good and bad solutions through a dominance structure. Decision vectors can dominate other possible solutions strongly or weakly:

**Definition 4.2.2 (Strong domination).** Given a set of criteria \( f \) to be minimized, a decision vector \( x_1 \) strongly dominates another decision vector \( x_2 \) if \( f_i(x_1) \leq f_i(x_2) \) for all \( i = 1, \ldots, k \) and \( f_j(x_1) < f_j(x_2) \) for at least one index \( j \).

**Definition 4.2.3 (Weak domination).** Given a set of criteria \( f \) to be minimized, a decision vector \( x_1 \) weakly dominates another decision vector \( x_2 \) if \( f_j(x_1) < f_j(x_2) \) for at least one index \( j \).

In other words, strong domination means \( x_1 \) is equal to or better than \( x_2 \) for all criteria and better in at least one. Weak domination means that \( x_1 \) is better in at least one objective.

The Pareto frontier is a set of decision vectors such that each element weakly dominates each other and strongly dominates all other vectors. BTs belonging in the Pareto frontier are Pareto optimal and can be considered the best solutions. Firstly, because amongst the frontier we cannot improve on one criteria without worsening the other and secondly because the decision vectors in the set are strictly better than all other options outside it.
Another note to remark is that while one of the objectives of our method is to enable vertical expansion, this cannot be measured explicitly with the suggested metric. However, it is widely considered that an hierarchical structure is beneficial for readability [2], therefore we intend to implement it, even though it is a somewhat qualitative aspect.

4.2.4 The Algorithm in detail

Now, the method is explained in detail. Algorithms 4.1 and 4.2 show the proposed decanonicalization. First, a hierarchy order is defined for the predicates (line 2, Algorithm 4.1). This is achieved by counting the number of occurrences of each variable amongst units’ guards and ordering them by decreasing order of prevalence. This aims to improve legibility by placing predicates with higher prevalence in the upper parts of the final BT.

**Algorithm 4.1: Proposed Decanonicalization Algorithm**

```
Function Decanonicalize(preds, atomic units, dc, max depth):
  Order preds by decreasing order of prevalence amongst guards
  Classify atomic units as singular or composite
  children ← Hierarchize(preds[1:max depth], singulars, dc)
  children ← children \cup composites
  Left-to-right simplification(children, dc)
  Place children under root, a fallback node
  cleanDuplicateGuards(root)
  return root
```

Then, atomic units are categorized as singular or composite. A singular unit has a single term in its guard (i.e., a single product of literals) while a composite unit has multiple terms (i.e., a sum of multiple products). For example, \( g_1 = ABC \) is singular, while \( g_2 = AB \lor C \) is composite. Composite subunits are left out of the following merging step and are later rejoined for guard simplification.

Then, the merging process takes place with function `Hierarchize` (Algorithm 4.2). The first predicate in the previously established order is checked. Then atomic units are forked into one of three paths: a positive path if the units’ guards contain the checked predicate and in positive form, a negative path if they contain it but in negative form and a do not have path if the guards do not contain it. The function is then invoked recursively, checking the next predicates in the defined order. Units are forked progressively in a downwards direction, until the stopping condition of not having more predicates to check is met. Then, units are returned upwards, potentially joining them together.

At each stage of the upwards direction, if more than one unit is received in a positive or negative path, they are merged together into a joiner unit (Figure 4.2). It is composed by a sequence node with two children: on the right a fallback node with the units to be joined and on the left a condition node.
containing the conditions in common amongst those units. Left-to-right logic minimization is performed on the units in the fallback node and the joiner unit inherits the union of their minterms. The children of the fallback node can be atomic units or other joined units. The fact that joined units can contain other joined units makes it so that the tree is expanded vertically.

Logic minimization is a known NP-Complete problem, with exponential complexity. For this reason, the heuristic based Espresso algorithm [30] is used for minimization steps. French et al. [13] compare this method with the exact logic minimizer algorithm Quine-McCluskey [31]. With the Espresso method computational time took less than one second, while with Quine-McCluskey it took over 7 minutes. Given the better efficiency of Espresso, it is the chosen method in this work.

Algorithm 4.2: Function Hierarchize of the proposed Decanonicalization Algorithm

1  Function Hierarchize(preds, subset of units, dc):
2       if preds is empty then
3           return units
4       pred = next on preds
5       Separate units into 3 groups:
6           positive units: units whose guard contain pred in positive form
7           negative units: contain pred in negative form
8           dont have units: do not contain pred
9       if at least one positive unit then
10          children = Hierarchize(preds[1:], positive units, dc)
11          if only one child then
12              ret.append(child)
13          else
14              Left-to-right simplification(children, dc)
15              Create a joiner unit, with conditions in common as guard and children under the fallback node
16              minterms(joiner unit) ← union of children’s minterms
17              ret.append(joiner unit)
18       Do the same for negative units
19       if at least one dont have unit then
20          children = Hierarchize(preds[1:], dont have unit, dc)
21          Left-to-right simplification(children, dc)
22          ret.append(children)
23       return ret

The following example better illustrates this step. Let us suppose we have three units $u_1$, $u_2$ and $u_3$ whose guard-action pairs are $(ABC, X)$, $(ABC, Y)$ and $(AB, Z)$, respectively. Let us also assume the hierarchy for the predicates is in alphabetical order and let us ignore logic minimization steps. The first predicate checked is $A$. All units contain it in positive form, therefore they advance together in a positive path. Then we check for $B$. Both $u_1$ and $u_2$ proceed through the positive path while $u_3$ goes through the negative, as it contains $B$ negated. $u_1$ and $u_2$ then branch apart when checking for $C$. With all predicates
checked, units are returned upwards. A joiner unit is created from $u_1$ and $u_2$, with $AB$ as guard. Then this unit meets with $u_3$, creating a new joiner unit with guard $A$. Figure 4.3 shows the resulting structure.

Composite units rejoin the result of function $Hierarchize$ (line 5) and they all undergo a final pass of logic minimization.

Line 8 of Algorithm 4.2 has the effect of further simplifying condition nodes. The tree is traversed in a depth-first manner and when a condition node is encountered its argument is stored. When a lower condition node is met the stored conditions are removed from it.

Placing all resulting units under a fallback node yields the final, decanonicalized BT.

### 4.2.5 Configurable Parameters

The configurable parameters are explained next. First, we motivate their need by noting that different BTs, with different structure and amount of nodes, can perform the same task equivalently. See the example in Figure 4.4 (adapted from [2]): an agent has a task to do and a battery that should be recharged when low, with the threshold for recharging depending on the urgency of the task. Figure 4.4(a) contains less nodes but has a more extensive guard, while 4.4(b) expands its actions throughout the tree, creating more nodes but having simpler conditions. Although both BTs perform equally, they
differ in number of nodes and complexity of condition nodes. Thus, it could be interesting to explore a multitude of options to find the most legible tree. For this, we propose two configurable parameters for the algorithm.

![Figure 4.4: Two BTs for the same task, adapted from [2].](image)

The first parameter is choosing which composite units to expand into several singular units, as shown in Figure 4.5. Recall that atomic units are categorized as singular or composite based on whether they contain a single or multiple terms in their guards. This way we can explore whether it is better to have actions appearing multiple times at greater depths or at the right-most part of the tree, since composite units are left out of Algorithm 4.2.

The second parameter controls how many predicates are checked with the algorithm. This has the effect of limiting the depth of the resulting tree. For example, by choosing \( \text{max depth} = 3 \), the algorithm looks only at the first 3 predicates in the chosen order (line 4 of Algorithm 4.1). It is an optional input; if

![Figure 4.5: Expansion of a composite unit (a) into several singular units (b).](image)
The following example showcases the effect of this parameter. Let us consider the same case used previously: three atomic units $u_1, u_2$ and $u_3$ with guard-action pairs are $(ABC, X)$, $(AB\bar{C}, Y)$ and $(A\bar{B}, Z)$, respectively. We will consider checking only the first predicate $A$. All units contain $A$ in positive form, therefore they advance together through a positive path. Then, since all predicates are checked, the stopping condition (line 3 of Algorithm 4.2) is reached. The units are returned upwards and are merged together as a joiner unit. Since $A$ is the only condition in common amongst the units’ guards, the guard of the joiner unit will be $A$. The resulting BT is shown in Figure 4.6. Again, logic minimization steps are ignored.

![Figure 4.6: Example of the effect result of limiting how many predicates are checked.](image)

Both parameters and their possible combinations are explored before decanonicalization. For $m$ actions we have up to $2^m$ possibilities of expansion of composite units and for $n$ predicates we have $n$ possible limits of predicate checking. This results in a total of up to $2^m \cdot n$ combinations to explore.

### 4.3 Using past Expert Knowledge

We want to investigate how to include expert’s knowledge, coded as a BT, into the learning process. The user provides an input BT, which will guide the agent during the learning process in an exploration/exploitation scheme, inspired in Guided Reinforcement Learning [9].

We propose an extension to the $\epsilon$-greedy action selection mechanism, which we will call Double $\epsilon$-greedy. This method is intended to be simple and easy to implement. Two epsilons are considered, $\epsilon_1$ and $\epsilon_2$. $\epsilon_1$ functions the same as $\epsilon$ in the classic $\epsilon$-greedy formulation: it selects between a random action and a greedy action, i.e., the current best action learned for the given state. $\epsilon_2$ selects between the action pointed by the input BT for the current state and the outcome of $\epsilon_1$. This results in the agent choosing the action pointed by the BT with probability $\epsilon_2$, a random action with probability $\epsilon_1(1 - \epsilon_2)$ and a greedy action with probability $(1 - \epsilon_1)(1 - \epsilon_2)$. Figure 4.7 shows a scheme for the Double $\epsilon$-greedy action selection.
Figure 4.7: Double epsilon-greedy scheme.
Experimental Setup

Contents

5.1 The Mario AI Framework ..................................................... 39
5.2 Q-Learning considerations ................................................... 40
In this chapter the evaluation domain used, the Mario AI framework, is described. The basic functioning of the game is explained in 5.1. Then the considerations taken for the RL context are enunciated in 5.2.

## 5.1 The Mario AI Framework

To validate results, the Mario AI Framework was used. Originally created by Karakovskiy & Togelius [19] based on the popular platform game Super Mario Bros, it has since been widely used in academia. The 10th year edition was used\(^1\). Figure 5.1 shows a screenshot of the game.

In the game, the agent Mario has to navigate a two-dimensional level, while facing enemies, obstacles and pitfalls. The objective is to reach the end goal and maximize the score. This involves secondary objectives, such as collecting coins, killing enemies and clearing the level in the least amount of time possible.

![Figure 5.1: A screenshot of the Mario AI Benchmark.](image)

At each time step the framework provides the agent with knowledge about the game. Mario has access to information about the observable screen, represented by a grid centered around himself. This way he can know the position of obstacles and enemies. Mario also has information about himself, such as whether he is on the ground, his current speed or his size.

Mario can be in one of three sizes: small, big and fire. While in big mode he can can break bricks by headbutting them. In fire mode he can additionally shoot fireballs. Mushrooms and flowers act as power ups, increasing Mario's size. Getting hit by enemies does the opposite, bringing the agent down to the previous state. Being hit when in size small, falling through a pitfall or running out of time issues a game loss.

After receiving environmental information, Mario is queried for an action. Actions in the framework

\(^1\text{github.com/amidos2006/Mario-Al-Framework}\)
correspond to the pressing of the five buttons in the original game: Left, Right, Down, Speed and Jump. Left and Right move Mario in the chosen direction. Down makes him crouch, but has minimal usage. Holding Speed makes Mario run faster and also shoot fireballs when in fire mode. Jump issues a jump with altitude depending on how long the button is pressed. Actions are expressed as a Boolean array of five bits. This results in a total of 32 possible combinations, although some are nonsensical (such as pressing Left and Right at the same time).

5.2 Q-Learning considerations

5.2.1 Actions

In the RL context, some considerations were taken for the actions in order to simplify the problem. It is assumed Mario always walks right. His action choices consist of pressing the Speed button or not (Walk or Run) and issuing one of three types of jump: a short ($J_S$), a medium ($J_M$) and a high jump ($J_H$). The altitude reached for each jump is 1, 2 and 4 tiles, respectively. This gives a total of 8 possible actions: $Walk$, $Walk + J_S$, $Walk + J_M$, $Walk + J_H$, $Run$, $Run + J_S$, $Run + J_M$ and $Run + J_H$.

5.2.2 States

For Mario's states, 10 boolean predicates were considered, for a total of $2^{10} = 1024$ possible states:

- Ground ($G$): True if Mario is on the ground;
- Speeding ($S$): True if Mario’s speed is greater than 6 units per frame;
- Enemy in $i$ ($E_i$): True if there is an enemy $i$ column(s) ahead of Mario, $i = 1, 2, 3$;
- Hole in $i$ ($H_i$): True if there is a pitfall $i$ column(s) ahead of Mario, $i = 1, 2, 3$;
- Small/large obstacle ($O_L/O_S$): True if there is an obstacle with height $\leq 2/\geq 3$ in front of Mario.

Mario faces inertia when running. His top speed is about 4 units per frame when walking and around 9 units per frame when running. If the speed has reached 6 it means the button has been pressed for a few time frames. The columns are checked by looking at the first, second or third tile in front of Mario, at eye level, and then checking every tile below. If an enemy is encountered, $Enemy\ in\ i$ holds true. This means that Mario can detect enemies in front and below, but not above. If there are no blocks until the bottom of the level, it means a pitfall. Figure 5.2 shows Mario detecting an enemy in the second column (in yellow), thus activating the $H_2$ state bit.
Obstacles are identified in a similar manner. The front tile is checked for an object and its height is counted by checking how many tiles on top also contain an obstacle. Given the altitude of the jumps this configuration makes it so that only a high jump can traverse a large obstacle.

![Figure 5.2: Mario detecting an enemy in the second column.](image)

Even though Mario's perception is fairly limited with this configuration, it is important to keep the total number of states low. Each state bit added doubles the size of the Q-Value table. Not only this increases computational load but also lessens the likelihood of a given state being visited, which can worsen convergence. Additionally, the proposed decanonicalization algorithm employs logic minimization, which, as mentioned, is an NP-Complete problem, meaning that complexity increases exponentially with the number of states. Other configurations tried with more state bits took a long amount time to process. 10 state bits was considered a good compromise between having enough detail about the environment and reasonable computational time.

### 5.2.3 Rewards

The reward structure is as follows:

- Reaching the end of the level: +100 plus a bonus equal to the percentage of initial time left;
- Death for timeout: -200;
- Dying to an enemy or pitfall: -200 plus a bonus equal to the percentage of the level progressed;
- Collecting a coin: +10 for collecting a coin;
- Stomping an enemy: +2.

The bonus for finishing the level is to incentive completing the level faster while the bonus for dying to an enemy or pitfall is to reward advancing through the map. Without the latter, the agent would often learn the behavior of dying to the first threat possible.

This structure aims to emulate the original scoring system, although it somewhat deviates from it. The numbers were tried and tweaked until a satisfactory behavior was achieved.
5.2.4 Level used

A custom level based on the first level of the original game was used. Since Mario’s perception with this configuration is fairly limited some considerations were taken.

Mario starts at size small and the level does not contain mushrooms or flowers. Since the agent does not distinguish between enemy types, only one kind of enemy was used: the Goompa (seen in Figure 5.1), which is the simplest type of enemy. They can be stomped and do not jump.

Several coins were distributed along the level, in relatively equal intervals. This proved to be crucial for the learning process. Without them, rewards would mostly be given only at the end of the episode. This made them too sparse and significantly slowed learning.

5.2.5 Other parameters

The Q-Learning parameters used, discount and learning rates, were respectively $\gamma = 0.9$ and $\alpha = 0.15$.

For action selection, $\epsilon$-greedy was used. Recall that this makes the agent take a random action with probability $\epsilon$, and a greedy action, i.e, the best action estimated so far, with probability $1 - \epsilon$. A decaying exponential was defined for $\epsilon$:

$$\epsilon(t) = e^{-\text{decay} \cdot t}$$  \hspace{1cm} (5.1)

$$\text{decay} = \frac{\ln \frac{1}{T}}{T}$$  \hspace{1cm} (5.2)

Where $t$ is the current episode and $T$ is the total number of episodes, for which 7000 were used. This ensures a minimum of $\epsilon = 0.1$ at the last episode. Figure 5.3 (top) shows $\epsilon$ as a function $t$.

![Figure 5.3: $\epsilon_1$ (top) and $\epsilon_2$ (bottom) as a function of the episode $t$.](image-url)
5.2.6 *Double $\epsilon$-greedy* parameters

For the proposed *Double $\epsilon$-greedy* method (explained in subsection 4.3), two $\epsilon$ are considered: $\epsilon_1$ and $\epsilon_2$. Recall that the agent chooses an action pointed by an input BT with probability $\epsilon_2$, a random action with probability $\epsilon_1(1 - \epsilon_2)$ and a greedy action with probability $(1 - \epsilon_1)(1 - \epsilon_2)$.

$\epsilon_1$ is defined the same way as in the classical $\epsilon$-greedy, mentioned above. $\epsilon_2$ is defined as follows:

$$
\epsilon_2(t) = 1 - \left( \frac{1}{\cosh(T)} + \frac{\text{steepness}}{T} \right) 
$$

$$
cosh = \cosh(\exp(-\text{standardized time}))
$$

$$
\text{standardized time} = \frac{t - \text{left tail} \cdot T}{\text{slope} \cdot T}
$$

where steepness = 0.25, left tail = 0.40 and slope = 0.15. Again, $t$ is the current episode and $T$ the max number of episodes. Figure 5.3 (bottom) shows the evolution of $\epsilon_2$ with $t$.

Figure 5.4 shows the accumulated probabilities for each outcome (BT, random or greedy action). The idea is for the agent to initially mostly apply the action indicated by the BT, then a phase where it explores with random actions and a final phase where it mostly uses greedy actions.

![Figure 5.4: Accumulated probability for each outcome of *Double $\epsilon$-greedy*.](image)

In order to test whether or not this method is able to improve upon the user provided knowledge three input BTs were tested for comparison. The first is a CBT obtained with a run of Q-Learning with $\epsilon$-greedy that achieved a high score. This BT will be deemed as optimal. It is represented in Figure 5.5.

The second is a handcrafted BT, shown in Figure 5.6(a). It can evade obstacles and enemies to finish the level but its score is lower than optimal. The last is also a handcrafted BT, but it causes the agent to die, preventing it from finishing the level. It will be known as the poor handcrafted BT and can be seen in Figure 5.6(b). Table 5.1 shows the scores achieved by each of the mentioned BTs.
Table 5.1: The scores of each BT to use as input in the Double $\epsilon$-greedy method.

<table>
<thead>
<tr>
<th>BT</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>285.82</td>
</tr>
<tr>
<td>Handcrafted</td>
<td>269.08</td>
</tr>
<tr>
<td>Bad</td>
<td>-170.89</td>
</tr>
</tbody>
</table>

The desired outcome is that the agent can make use of the designer's knowledge of the system, which often times is good but not optimal, to improve the efficiency of the learning process. Ideally, convergence is achieved sooner when using both the optimal BT and the better handcrafted one. We also aim to test how the use of the poor BT influences performance. Classical $\epsilon$-greedy will be used as a baseline for comparison.

Figure 5.5: The optimal BT.

Figure 5.6: Two handcrafted BTs, one capable of finishing the level 5.6(a) and other who is not 5.6(b)
6 Results and Discussion

Contents

6.1 Proposed Decanonicalization Algorithm ................................................. 47
6.2 Using Past Expert Knowledge .............................................................. 53
In this chapter we exhibit and discuss the results obtained from the methods developed. In section 6.1 the results from the decanonicalization algorithm proposed in 4.2 are shown. Section 6.2 deals with the Double $\epsilon$-greedy method developed in section 4.3.

### 6.1 Proposed Decanonicalization Algorithm

#### 6.1.1 Results

The result of Q-Learning is a Q-value table, from which a policy is obtained following equation 2.11. 42 states were visited in the learning process (minterms), leaving the remaining $2^{10} - 42$ as the dontcares.

Logic minimization is performed over minterms and dontcares, following equation 3.1, resulting in a set of guards for each action. The obtained guards and their corresponding actions are shown in table 6.1. These form the atomic units and the CBT.

**Table 6.1: Guards obtained for each action.**

<table>
<thead>
<tr>
<th>Action</th>
<th>guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walk</td>
<td>$GSH_1\bar{H}_3E_1E_2E_3\bar{O}_5\bar{O}_L\bar{H}_3 \lor SE_1 \lor GE_1 \lor H_1\bar{H}_2 \lor GSO_S \lor$ $GSH_1H_2 \lor GSH_1 \lor GE_3 \lor GSE_2$</td>
</tr>
<tr>
<td>Walk + J$_S$</td>
<td>$GSH_1\bar{H}_3 \lor GE_3 \lor GSE_2 \lor GSH_1H_2$</td>
</tr>
<tr>
<td>Walk + J$_M$</td>
<td>$G\bar{SE}_1$</td>
</tr>
<tr>
<td>Walk + J$_H$</td>
<td>$G\bar{SO}_L$</td>
</tr>
<tr>
<td>Run</td>
<td>$GSH_1E_1E_2E_3 \lor GSH_1H_2 \lor SH_1H_3E_1E_2E_3\bar{O}_5\bar{O}_L \lor SH_1H_3 \lor$ $GSH_1H_2E_1E_3\bar{O}_S$</td>
</tr>
<tr>
<td>Run + J$_S$</td>
<td>$GSH_2H_3 \lor GSH_1H_2 \lor GSH_1H_3$</td>
</tr>
<tr>
<td>Run + J$_M$</td>
<td>$GSE_2$</td>
</tr>
<tr>
<td>Run + J$_H$</td>
<td>$GOS \lor G\bar{SH}_1H_3 \lor G\bar{SO}_L \lor G\bar{SH}_2H_3$</td>
</tr>
</tbody>
</table>

The resulting hierarchy for the predicates, in descending order, is: $G, S, H_1, H_2, H_3, E_1, E_2, E_3, O_S, O_L$. Recall that this is obtained by counting the prevalence of each predicate amongst the guards and sorting them by descending order.

All combinations of expansion of composite units and max depth were tested. For 8 actions, of which 5 of them form composite units, and 10 predicates this yields a total of $2^5 \cdot 10 = 320$ trees. All underwent the proposed decanonicalization algorithm and were verified to be data-equivalent with respect to the learned policy. This was achieved by ticking the BTs while on each minterm state and confirming that the resulting action is the same as the one indicated by the policy.

Figure 6.1 shows the resulting BTs, comparing them by number of nodes and literals in condition
nodes. A total of 53 BTs with unique combinations were obtained. Many BTs had the same combination.
For example, limiting the depth of the algorithm beyond checking 3 predicates provided no change.
Trees A-E form the Pareto frontier. According to the metric considered, these are the most legible BTs

![Figure 6.1: The resulting BTs and the Pareto frontier.](image)

obtained, as they strongly dominate all other options. Table 6.2 shows how each BT of the frontier was
obtained (actions Walk + J, Walk + J and Run + J are not considered since they represent singular
units). Notably, all of them expand action Run + J and none expand actions Walk or Run, who have
the longest guards. On the other hand, the latter are expanded in all BTs from the top right region
of Figure 6.1. They can also be seen as default or non-reactive behaviors, whereas the jump actions
always react to an appearance of an enemy, pitfall or obstacle. The fact that they are never expanded
in the frontier suggests it is preferable to leave these longer guard actions out of the merging process
of Algorithm 4.2, and instead leave them at the right-most part of the BT, directly under the root. This
makes use of the fact that the left-to-right guard simplification step causes the action at the right end to
have its guard as True, effectively eliminating the longest guard.

<table>
<thead>
<tr>
<th>BT</th>
<th>Action Expanded</th>
<th>Walk</th>
<th>Walk $+_J_S$</th>
<th>Walk $+_J_M$</th>
<th>Walk $+_J_H$</th>
<th>Run</th>
<th>Run $+_J_S$</th>
<th>Run $+_J_M$</th>
<th>Run $+_J_H$</th>
<th>Predicates checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>X</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>X</td>
<td>X</td>
<td>–</td>
<td>–</td>
<td>X</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>X</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>X</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>X</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>X</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>≥ 3</td>
</tr>
<tr>
<td>E</td>
<td>X</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>X</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>≥ 3</td>
</tr>
</tbody>
</table>

Also in the frontier, for the sake of comparison, is the CBT subject to the left-to-right logic simplifica-
tion. That is, it did not undergo decanonicalization, only this minimization step. It is necessarily the BT with the least amount of nodes, since it is only composed of atomic units for all but the right-most action, whose guard can be omitted as it is True. It was also verified to be data-equivalent to the input policy.

Figure 6.2: The CBT simplified with logic minimization.

Now, let us look at the extremes of the frontier and one middle case. With the least amount of literals in condition nodes is BT E, shown in Figure 6.3. It is the only BT in the frontier that defines a maximum depth greater than or equal to 3. We can see the three levels of hierarchies on the guards of the joiner units. At the top left of the tree there is a joiner unit with guard $G$, then, below, we see two joiner units with guards $S$ and $\bar{S}$. Following the path $G\bar{S}$ reaches the lowest level joiner unit, with guard $H_1$. To be noted here is how the left-to-right guard simplification step can also leave an action unconditioned at lower levels of the tree, namely action $Run + JS$ (by following path $G\bar{S}H_1H_3$).

Figure 6.3: BT E from the Pareto frontier.

Next we look at a middle point of the frontier, BT C, shown in Figure 6.4. It is quite similar in structure
to the previously BT addressed. It differs by expanding one less action \((Run + J_S, \text{seen below the root})\) and limiting the depth to 2 predicates checked. This causes to only have two levels of joiner units.

\[
\begin{align*}
\text{E}_2 & \rightarrow \text{Run} + J_M, \\
\text{O}_1 & \rightarrow \text{Run} + J_M, \\
\text{H}_1 & \rightarrow \text{Run} + J_M, \\
\text{E}_1 & \rightarrow \text{Walk} + J_H, \\
\text{O}_2 & \rightarrow \text{Walk} + J_H, \\
\text{E}_3 & \rightarrow \text{Walk} + J_H, \\
\text{H}_2 & \rightarrow \text{Run} + J_M
\end{align*}
\]

Figure 6.4: BT C from the Pareto frontier.

Finally, we analyse BT A (Figure 6.5), the BT with the least amount of nodes from amongst the Pareto frontier, if we exclude the Canonical Behavior Tree Simplified (CBTS). Its maximum depth is one, meaning it only creates one level of joiner units. In this case, there is a single joiner unit with guard \(G\) and several atomic units as children. If equal weights were given to each criteria then the optimal BT would BT A, since it is the one that minimizes the distance to the origin.

\[
\begin{align*}
\text{E}_1 & \rightarrow \text{Walk} + J_M, \\
\text{O}_1 & \rightarrow \text{Run} + J_M, \\
\text{H}_1 & \rightarrow \text{Run} + J_M, \\
\text{E}_2 & \rightarrow \text{Walk} + J_H, \\
\text{O}_2 & \rightarrow \text{Walk} + J_H, \\
\text{E}_3 & \rightarrow \text{Walk} + J_H, \\
\text{H}_2 & \rightarrow \text{Run} + J_M
\end{align*}
\]

Figure 6.5: BT A from the Pareto frontier.
To be noted is how the farther to the left we traverse the points in the Pareto frontier, that is, in the direction of minimizing nodes, the closer the BT resembles the CBT in structure.

A BT obtained with Banerjee’s method (Figure 6.6) is also represented in Figure 6.1. All BTs in the Pareto frontier improve on both criteria, suggesting that the proposed algorithm and searching over its combinations can result in more legible BTs.

Figure 6.6: BT obtained with Banerjee’s method.

6.1.2 Discussion

In this section we analyze if the decanonicalization algorithm achieves the main objective of this work: translating an RL policy as a human readable BT. Additionally, we consider the design objectives mentioned in subsection 4.2.1.

The proposed method and searching over the options generated by the configurable parameters allowed to obtain BTs more legible than previous approaches, considering the total number of nodes and complexity of condition nodes as objective metrics for readability. Following [3], the control architecture for an autonomous system should be consistent with how humans think about problems and solutions. It should also be transparent and traceable, meaning that the output of the system should be easily identified. Making a more subjective argument, we believe that the proposed algorithm fulfills these requirements. The method represents a policy with BTs, which are inherently modular and readable, and the fact the resulting BTs are created with an hierarchical structure makes for a legible and easier to understand option.

Next, we address the objectives mentioned in subsection 4.2.1. The notion of data-equivalent BTs was proposed to assess whether or not a BT behaves like the input policy. While it is not claimed that the method creates data-equivalent BTs for the general case, all BTs obtained were verified to be so, thus addressing objective 6.

The proposed algorithm forks units in a downwards path based on whether they contain (or not)
each predicate in positive or negative form. This means that the longer two units stay together, the more similar their guards are. By pairing them in the upwards direction it is ensured that the grouped units have commonality amongst their guards. This addresses the objective 3 of pairing together the most related units.

One of the issues mentioned with Banerjee’s [1] method that this work attempted to correct (objective 5) was that the a-priori ordering of the inputs affected the resulting BT. This was attempted by defining a hierarchy for the predicates in the first step of the algorithm, by counting the total amounts of occurrences of each variable amongst the units’ guards. Predicates are then checked in this fixed order. Since this counting is independent of the how the units are ordered beforehand, this contributes for the algorithm producing the same BT independently of the a-priori ordering of the inputs. Tie cases are the exception, however. When counting the predicates, if there is a tie case, the default is to leave the order as it is input. A tie case can also appear in the left-to-right guard simplification method, where units are ordered based on their guard length.

The proposed joiner units achieve objective 1. Their guards contain the predicates in common amongst their children, thus enabling the simplification of the children’s guards and reducing the overall complexity of condition nodes.

The fact that joiner units can have other joiner units as children enables the tree to be expanded vertically and it creates a hierarchical structure (objective 4). The guards for joiner units make for a clear entry point to its children, effectively creating sub-environments where those guards are known to hold true.

The proposed method also makes use of Algorithm 3.1, originally proposed by Banerjee. This way it is possible to employ logic minimization steps and make use of the left-to right ordering of ticks in order to simplify units’ guards, thus addressing objective 2. This, however, assumes a fixed order for the units. Altering this order or adding new behaviors will most likely violate the policy. This limits the scope on how much the BTs can be altered by an user. Additionally, it entails the restriction of not allowing actions to fail.

Two criteria that measure the legibility of BTs were proposed, however, since their weights are not known, i.e, it is not known which one has a greater impact on the readability of the BT, it cannot be determined with certainty which BT from the Pareto frontier is the optimal one. One question that might arise is if considering equal weights is a reasonable approximation. We argue that perhaps not, and shortening condition nodes is possibly preferable, even if at the expense of adding more nodes. Looking at BT E from the Pareto frontier (Figure 6.3), the tree with shorter condition nodes, the majority of nodes added are control flow nodes. These have clear transitions and connections, making their interpretation trivial after some familiarity with BTs. The condition nodes included are also the result of splitting of other conditions, which while it adds more nodes, individually they are easier to understand. Although
this is a somewhat subjective reasoning, the very fact there is interest in the literature to depart from the CBT [1] suggests that a having a minimal amount of nodes does not make for optimal legibility. This is not to say that minimizing the complexity of condition nodes is the best approach either. Just that the weights for the criteria used probably tend towards favoring that direction.

6.2 Using Past Expert Knowledge

Figure 6.7 shows a comparison between Q-Learning with the classic $\epsilon$-greedy and the proposed Double $\epsilon$-greedy approach. Three BTs with different performances were used as input, as explained in 5.2.6.

After each learning episode a greedy agent plays the level and its rewards were saved. They were then averaged over 50 runs, and then averaged for each 100 episodes. Figure 6.7 shows the resulting averaged scores and its evolution with episodes.

As evidenced, the Double $\epsilon$-greedy method does not improve convergence for either input BT. At around the 2000 episode mark all cases of the proposed method equalize the baseline $\epsilon$-greedy. One possible explanation is that this configuration does not allow for sufficient exploration in the earlier stages.
7 Conclusion

Contents

7.1 Conclusions .................................................. 57
7.2 System Limitations and Future Work .................... 57
This chapter concludes the present work. An overview of its main contributions and limitations is provided. Possible directions for future work are also mentioned.

7.1 Conclusions

This work presents an algorithm to convert a policy obtained from tabular Q-Learning into a human readable BT. The method merges units into more compact structures and employs logic minimization steps to shorten the arguments in condition nodes. The resulting BT is built with an hierarchical form that improves on transparency and traceability. Innovating from previous paradigms, this approach contains configurable parameters that can be set in a preliminary step, allowing for the generation of a plethora of trees with different characteristics, such as the number of nodes and depth. This enables a search over the different outcomes, by formulating it as a multi-objective problem. To measure the legibility of a BT, the total amount of nodes and the complexity of condition nodes are used as metrics to be minimized. The result is a set of optimal solutions, which are strictly better than all other options. Results in the Mario AI benchmark show that the optimal BTs obtained improve on the legibility of previous approaches. It was also verified that the resulting trees faithfully represent the input policy.

Additionally, **Double $\epsilon$-greedy**, an extension to the classic $\epsilon$-greedy method for action selection is proposed. Its purpose is integrating user knowledge coded as an input BT to assist in the learning process. It operates by guiding the agent in an exploration/exploitation scheme in which the agent initially follows the actions indicated by the provided BT. Results comparing input BTs with various performances show that this method with the proposed configuration did not achieve the desired outcome of improving convergence. Thus this problem remains an open challenge worth exploring.

The proposed methods thus enable an agent to autonomously learn a control policy via RL, and then represent it as a human legible BT. This can assist designers in testing and debugging, since a readable system allows the users to better predict its behavior, thus cutting on development time and costs. The possibility of integrating the user's knowledge to improve the learning process is also left as a potential future direction.

7.2 System Limitations and Future Work

The proposed decanonicalization algorithm employs logic minimization, which is a known NP-Complete problem. This can limit the scope of uses of the method, as increasing the state space can exponentially increase the complexity of the problem. The simplification steps used also entail a fixed ordering for the behaviors in the tree, restricting the degree to which a user can alter the resulting BT.

Moreover, it is assumed actions do not fail. This can prove too hard of a restriction to many applica-
tions. Therefore, handling action failure is a reasonable extension. Incorporating sequences of actions or actions that trigger when an action fails into the action pool of the learning algorithm could be a potential option. Another alternative is adding variables that check if actions have failed, but this has the downside of increasing the state space, which ties with the aforementioned logic minimization problem.

We considered two criteria that impact the legibility of BTs. However, it is possible that they alone are insufficient to fully measure readability. Other factors, such as the depth of the tree or its overall shape (whether it expands horizontally or vertically) could also have an impact. Additionally, since the relative weights of each metric are unknown, we cannot discern the best BT from amongst the Pareto frontier set. Performing a survey with humans and considering more parameters could better quantify the impact of each criteria used to measure readability.

Tabular Q-Learning has its own set of disadvantages. If the state space is too small, the system may not be discerning enough. If it is too large, it decreases the likelihood of visiting each state-action pair enough times to achieve convergence. An interesting direction could then be to adapt the proposed method do other RL methods, such as neural networks.
Bibliography


