Learning of Behavior Trees for Autonomous Agents

Michele Colledanchise, Ramviyas Parasuraman, and Petter Ögren

***Abstract*— Definition of an accurate system model for Automated Planner (AP) is often impractical, especially for real-world problems. Conversely, off-the-shelf planners fail to scale up and are domain dependent. These drawbacks are inherited from conventional transition systems such as Finite State Machines (FSMs) that describes the action-plan execution generated by the AP. On the other hand, Behavior Trees (BTs) represent a valid alternative to FSMs presenting many advantages in terms of modularity, reactiveness, scalability and domain-independence.**

In this paper, we propose a model-free AP framework using Genetic Programming (GP) to derive an optimal BT for an autonomous agent to achieve a given goal in unknown (but fully observable) environments. We illustrate the proposed framework using experiments conducted with an open source benchmark *Mario AI* for automated generation of BTs that can play the game character *Mario* to complete a certain level at various levels of difficulty to include enemies and obstacles.

***Index Terms*— Behavior trees, Evolutionary learning, Genetic Programming, Intelligent agents, Autonomous Robots**

I. Introduction

Automated planning is a branch of Artificial Intelligence (AI) that concerns the realization of strategies or action sequences, typically for execution by intelligent agents, autonomous robots and unmanned vehicles. Unlike classical control and classification problems, the solutions are complex and must be discovered and optimized in multidimensional space. According to [1], there are four common subjects that concern the use of Automated Planners (APs). First, the *knowledge representation*. That is the type of knowledge that an AP will learn must be defined; Second, the *extraction of experience*. That is how learning examples are collected; Third, the *learning algorithm*. That is how to capture patterns from the collected experience; Finally, the *exploitation of collected knowledge*. That is how the AP benefits from the learned knowledge.

Applying AP in a real word scenario is still an open problem [2]. In fully known environments with available models, the planning can be done offline. Solutions can be found and evaluated prior to execution. Unfortunately in most cases the environment is unknown and the strategy needs to be revised online. Recent works are extending the application of APs, from toy examples to real problems such as planning space mission [3], fire extinction, [4] underwater navigation [5]. However, as highlighted in [6], most of these planners are hard to scale up and presents issues when it comes to extend their domain. Despite these successful

The authors are with the Centre for Autonomous Systems, Computer Vision and Active Perception Lab, School of Computer Science and Communication, The Royal Institute of Technology - KTH, Stockholm, Sweden. e-mail: {miccol|ramviyas|petter}@kth.se

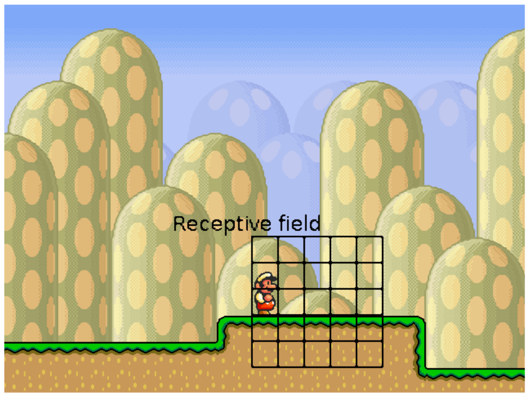


Fig. 1. Benchmark used to validate the framework.

examples, the application of APs to real worlds problem suffers of two main problems: *Planning Task*: Generally, APs require accurate description of the planning task. These descriptions include the model of the action that can be performed in the environment, the specification of the state of the environment and the goal to achieve. Generating exact definition of the planning is often unfeasible for real-world problems; *Extensibility*: Usually, a solution of an AP is a PSpace-complete problem [7], [8]. Recent works tackle this problem through reachability analysis [9], [10], but still search control knowledge is more difficult than the planning task because it requires expertise in the task to solve as well as in the planning algorithm [11].

The task's goal is described using a fitness function defined by the user. The derived action execution is described as a composition of sub-planners using a tree-structured framework inherited from computer game industry [12], namely BT. BTs are a recent modular alternative to Controlled Hybrid Systems (CHSs) to describe reactive fault tolerant executions of robot tasks [13]. BTs were first introduced in artificial intelligence for computer games, to meet their needs of programming the behavior of in-game non player opponents [14]. Their tree structure, which encompasses modularity; flexibility; and ease of human understanding, have made them very popular in industry, and their graph representations have created a growing amount of attention in academia [13], [15]–[18] and robotic industry [19]. The main advantage of BTs as compared to CHSs can be seen by the following programming language analogy. In most CHSs, the state transitions are encoded in the states themselves, and switching from one state to the other leaves no memory of where the transition was made from. This is very general and flexible, but actually very similar to the now obsolete

GOTO statement, that was an important part of many early programming languages, e.g., BASIC.

In BTs the equivalents of state transitions are governed by function calls and return values being passed up and down the tree structure. This is also flexible, but similar to the calls of FUNCTIONS that has replaced GOTO in almost all modern programming languages. Thus, BTs exhibit many of the advantages in terms of readability, modularity and reusability that was gained when going from GOTO to FUNCTION calls in the 1980s. Moreover in a CHSs adding a state turns in evaluating each possible transition from/to the new state and removing a state can require the re-evaluation of all the transitions in the system. BTs reveal to have a natural way to connect/disconnect new states avoiding redundant evaluation of state transitions. In a tree-structured framework as BT, the relation between nodes are defined by parent-child relations. These relations are plausible in Genetic Programming (GP) allowing entire sub-trees to cross-over and mutate through generations to yield an optimized BT that generates a plan leading to the desired goal.

In this paper we propose a model free algorithm based framework that generates a BT for an autonomous agent to achieve a given goal in unknown environments. The advantages of our approach lies on the advantages of BTs over a general CHS. Hence our approach is modular and we can reduce the complexity dividing the goal in sub-goals.

II. Related Work

Evolutionary algorithms has been successfully applied in evolving robot or agent's behaviors [20]–[23]. For instance, in [22], the authors used GP methodology to result in a better wall-follower algorithm for a mobile robot. In another interesting example by [24], the authors applied *Grammatical Evolution* to generate different levels of simulation environment for a game benchmark (MarioAI). Learning the agent's behaviors using evolutionary algorithms has shown to outperform reinforcement learning strategies at least in agents that possess ambiguity in its perception abilities [25].

BTs are originally used in gaming industry where the computer (autonomous) player uses BTs for its decision making. Recently, there has been works to improve a BT using several learning techniques, for example, Q-learning [26] and evolutionary approaches [23], [27].

In a work by Perez et. al. [23], the authors used GE to evolve BTs to create a AI controller for an autonomous agent (game character). Despite being the most relevant work, we depart from their work by using a metaheuristic evolutionary learning algorithms instead of grammatical evolution as the GP algorithm provides a natural way of manipulating BTs and applying genetic operators.

Scheper et. al [28] applied evolutionary learning to BTs for a real-world robotic (Micro Air Vehicle) application. It appears as the first real-world robotic application of evolving BTs. They used a (sub-optimal) manually crafted BT as an initial BT in the evolutionary learning process, and conducted experiments with a flying robot, while the BT that controls the robot is learning itself in every experiment. Finally, they demonstrated significant improvement in the performances of the evolved final BT comparing to the initial user-defined BT. While we take inspirations from this work, the downside is that this work require an initial BT for it to work, which goes against our model-free objective.

Even though the above-mentioned works motivates our present research, we intend to use a model-free framework as against model-based frameworks or frameworks that needs extensive prior information. Hence, we propose a framework that is more robust and require no information about the environment but thrives on the fact that the environment is fully-observable. Although we do not make a direct comparison of our work with other relevant works in this paper, we envisage it in our further works.

III. Background: BT and GP

In this section we briefly describe BTs and GP. A more detailed description of BTs can be found in [14].

A. Behavior Tree

A Behavior Tree is a graphical modeling language and a representation for execution of actions based on conditions and observations in a system. While BTs have become popular for modeling the Artificial Intelligence in computer games, they are similar to a combination of hierarchical finite state machines or hierarchical task network planners.

A BT is a directed rooted tree where each node is either a control flow node or an execution node (or the root). For each connected nodes we define as *parent* the outgoing node and *child* the incoming node. The root has no parents and only one child, the control flow nodes have one parent and one or more child, and the execution nodes have one parent and no children. Graphically, the children of control flow nodes are placed below it. The children nodes are executed in the order from left to right, as shown in Fig.s 2-4.

The execution of a BT begins from the root node. It sends *ticks* [1] with a given frequency to its child. When a parent sends a tick to a child, the execution of this is allowed. The child returns to the parent a status *running* if its execution has not finished yet, *success* if it has achieved its goal, or *failure* otherwise.
There are four types of control flow nodes (selector, sequence, parallel, and decorator) and two execution nodes (action and condition). Their execution is explained as follows.

Selector: The selector node ticks its children from the most left, returning success (running) as soon as it finds a child that returns success (running). It returns failure only if all the children return failure. When a child return running or success, the selector node does not tick the next children (if any). The selector node is graphically represented by a box with a "?", as in Fig. 2.

Sequence: The sequence node ticks its children from the most left, returning failure (running) as soon as it finds a child that returns failure (running). It returns success only if all the children return success. When a child return running

[1] A tick is a signal that allows the execution of a child

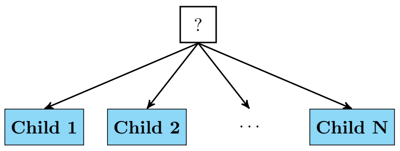


Fig. 2. Graphical representation of a fallback node with N children.

Algorithm 1: Pseudocode of a fallback node with N children

```
1 for $i \leftarrow 1$ to $N$ do
2     childStatus ← Tick(child(i))
3     if childStatus = running then
4         return running
5     else if childStatus = success then
6         return success
7 return failure
```

or failure, the sequence node does not tick the next children (if any). The sequence node is graphically represented by a box with a "→", as in Fig. 3.

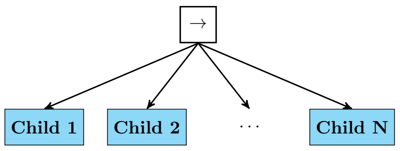


Fig. 3. Graphical representation of a sequence node with N children.

Algorithm 2: Pseudocode of a sequence node with N children

```
1 for $i \leftarrow 1$ to $N$ do
2     childStatus ← Tick(child(i))
3     if childStatus = running then
4         return running
5     else if childStatus = failure then
6         return failure
7 return success
```

Parallel: The parallel node ticks its children in parallel and returns success if $M \leq N$ children return success, it returns failure if $N - M + 1$ children return failure, and it returns running otherwise. The parallel node is graphically represented by a box with two arrows, as in Fig. 4.

Decorator: The decorator node manipulates the return status of its child according to the policy defined by the user (e.g. it inverts the success/failure status of the child). The decorator is graphically represented in Fig. 5(a).

Action: The action node performs an action, returning success if the action is completed and failure if the action cannot be completed. Otherwise it returns running. The action node is represented in Fig. 5(b)

Condition: The condition node check whenever a condition is satisfied or not, returning success or failure accordingly. The condition node never returns running. The condition node is represented in Fig. 5(c)

Root: The root node generates ticks. It is graphically represented as a white box labeled with "∅"

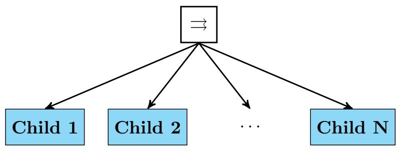
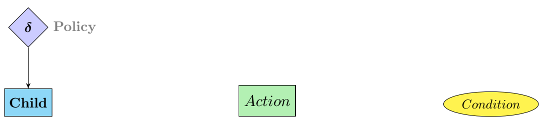


Fig. 4. Graphical representation of a parallel node with N children.



(a) Decorator node. The label describes the user defined policy. (b) Action node. The label describes the action performed (c) Condition node. The label describes the condition verified

Fig. 5. Graphical representation of a decorator, action, and condition node.

B. Genetic Programming

GP is an optimization algorithm, takes inspiration from biological evolution techniques and is a specialization of genetic algorithms where each individual itself is a computer program (in this work, each individual is a BT). We use GP to optimize a *population* of randomly-generated BT according to a user-defined *fitness function* determined by a BT's ability to achieve a given goal.

GP has been used as a powerful tool to solve complex engineering problems through evolution strategies [29]. In GP individuals are BTs that are evolved using genetic operations of reproduction, cross-over, and mutation. In each *population*, individuals are selected according to the *fitness function* and then mated, crossing over parts of their subtrees to form an *offspring*. The offspring is finally mutated generating a new population. This process continues until the GP finds a BT that satisfies the goal (such as minimize the fitness function and satisfy all constraints) is reached.

Often, the size of the final generated BT using GP is large even though there might exist smaller BT with the same fitness value and performance. This phenomenon of generating a BT of larger size than necessary can be termed as bloat. We also apply boat control at the end to optimize the size of the generated BT.

The GP used with BTs allows entire sub-trees to cross-over and mutate through generations. The previous generation are called parents and produces children BTs after applying genetic operators. The best performing children are selected from the child population to act as the parent population for the next generation.

Crossover, mutation and selection are the three major genetic operations that we use in our approach. The crossover performs exchanges of subtrees between two parent BTs. Mutation is an operation that replace a node with a randomly selected parent BT. Selection is the process of choosing BTs for the next population. The probability of being selected for the next population is proportional to a *fitness* function that describes "how close" the agent is from the goal.

1) Two-point crossover in two BTs: The crossover is performed by randomly swapping a sub-tree from a BT with a sub-tree of another BT at any level [30]. Fig. 6 and Fig. 7 show two BTs before and after a cross over operation.

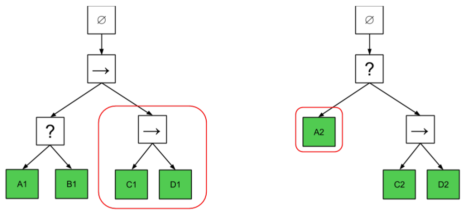


Fig. 6. BTs before the cross over of the highlighted sub-trees.

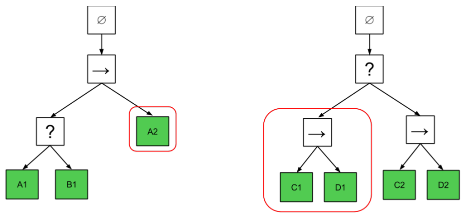


Fig. 7. BTs after the cross over of the highlighted sub-trees.

Remark 1: Note that, using BTs as the knowledge representation framework, avoids the problem of logic violation during cross-over experienced in [31].

2) Mutation operation in BT: We use unary mutation operator where the mutation is carried out by replacing a node in a BT with another node of the same type (i.e. we do not replace a execution node with a control flow node or vice versa). This increases diversity, crucial in GP [30]. To improve convergence properties we use the so-called *simulated annealing* [32] performing the mutation on several nodes of the first population of BT and reducing gradually the number of mutated nodes in each new population. In this way we start with a very high diversity to avoid possible local minima of the fitness function and we get a smaller diversity as we get close to the goal.

3) Selection mechanism: From the mutated population, also called *offspring*, individuals are selected for the next population. The selection process is a random process which select a given $\mathcal{T}_i$ with a probability p_i. This probability is proportional to the a *fitness function* f_i which quantitatively measures how many sub-goals are satisfied. There are three most used way to compute p_i given the fitness function [1]:

1) Naive Method: $p_i = \frac{f_i}{\sum_j f_j}$ that is the fitness divided by the sum of all the fitness of the individuals in the population (to ensure $p_i \in [0, 1]$)
2) Rank Space Method: We P_c set as the probability of the highest ranking individual (individual with highest f_i), then we sort the trees in the population in descending order w.r.t. the fitness. (i.e. $\mathcal{T}_i$ has higher or equal fitness of $\mathcal{T}i-1$). Then then the probabilities p_i are defined as follows.

$$p_k = (1-P_c)^{k-1}P_c \ \forall k \in \{1,2,\dots,N-1\} \quad (1)$$
$$p_N = (1-P_c)^{k-1} \quad (2)$$

3) Diversity Rank Method: We measure the *diversity* d_i of an individual $\mathcal{T}_i$ w.r.t. the others in the population. The probability p_i encompasses both diversity and fitness. Let $\bar{d}$ and $\bar{f}$ be the maximal value of d_i and f_i in the population respectively, the probability p_i is given by:

$$p_i = 1 - \frac{||[d_i, p_i] - [\bar{d}, \bar{p}]||}{||[\bar{d}, \bar{p}]||} \quad (3)$$

Individuals with the same survival probability lies on the so-called *iso-goodness* curves.

IV. Problem Formulation

Here we formulate definitions and assumptions, then we state the main problem, and finally illustrate the approach with an example.

Assumption 1: The environment is unknown but fully observable. We consider the problem so called *learning stochastic models in fully observable environment* [2].

Assumption 2: There exists a finite set of actions that, performed, lead from the initial condition to the goal.

Definition 1: $\mathcal{S}$ is state space of the environment.

Remark 2: We only know the initial state and the final state.

Definition 2: Σ is a finite set of actions.

Definition 3: $\gamma : \mathcal{S} \times \Sigma \to [0, 1]$ is the *fitness* function. It takes value 1 if and only if performing the finite set of actions Σ change the state of the environment from an initial state to a final state that satisfies the goal.

Problem 1: Given a goal described by a fitness function γ and an arbitrary initial state $s_0 \in \mathcal{S}$ derive an action sequence Σ such that $\gamma(s_0, \Sigma) = 1$.

V. Proposed Approach

In this section we describe the proposed approach. We begin with defining which actions the agent can perform and which conditions it can observe. We also define the appropriate fitness function that takes input a BT and results in a fitness value proportionate to how closer the BT is in achieving a given goal. An empirically determined moving time widow τ (seconds) is used in the execution process, where the BT is executed continuously but the fitness function is evaluated for the past τ seconds. The progressive change in fitness function are assessed to determine the course of the learning algorithm.

We follow a metaheuristic learning strategy, where we use a greedy algorithm first and when it fails, we use the GP. The GP will also be used when the greedy algorithm cannot provide any results or when the complexity of the solution is increased. This mixed-learning based heuristic approach was to minimize the learning time significantly compared to using pure GP, while still achieving an optimal BT that satisfies a given goal.

At an initial state s_0 we start with a BT that consists of only one node which is an action node. To choose that action node, we use a greedy search process where each action is executed until we find an action that, when executed for the τ seconds, the value of the fitness function keep increasing. if such an action is found, that action is added to the BT. However if no actions are found, the GP process will be initiated with a population of binary trees (two nodes in a BT) with random node assignments consisting of combination of condition and action nodes, and results in an initial BT that increases the fitness value the most.

In the next stages, the resultant BT is executed again, and the changes in the conditions and fitness values are monitored. When the fitness value starts decreasing, the recently changed conditions (within τ seconds) will be composed (randomly) as a subtree (as in Fig. 2 and added to the existing BT. Then, we use the greedy search algorithm as above to find the action node for the previously added subtree by adding each possible actions to that subtree and whole BT is executed. Once again, when an action that increase the fitness value is found, that action is added to the recent subtree of the BT and the whole process continues. If no such action is found then the GP will be used to determine the subtree which increases the fitness value. We iterate these processes until the goal is achieved. Finally, we remove the possible unnecessary nodes in the BT by applying anti-bloat control operation.

We now address the concerns raised by [1] using the AP proposed using BTs.

A. Knowledge representation

The knowledge is represented as a BT. A BT can be seen as a rule-based system where it describes which action to perform when some conditions are satisfied.

B. Extraction of the experience

The experiences (knowledge) are extracted in terms of conditions observed from the environment using sensory perceptions in the autonomous agents. Examples of conditions for a robot could be obstacle position, position information, energy level, etc. Similarly, example conditions for a game character could be "enemy in front", "obstacle close-by", "level reached", "points collected", "number of bullets remaining", etc.

C. Learning algorithm

Algorithm 3 presents the pseudo-code for the learning algorithm. The learning algorithm has 2 steps. The first step aims to identify which condition have to be verified in order to perform some actions. The second step aims to learn the actions to perform.

As mentioned earlier, the framework starts at the initial state s_0. If the value of the fitness function does not increase, a greedy algorithm is used to try each action until it finds the one that leads to an increase of the fitness value. If no actions are found, it start the GP to learn a BT composition of actions as explained before. We call the learned BT $\mathcal{T}_0$.

Remark 3: In case the framework learns a single action, $\mathcal{T}_0$ is a degenerate BT composed by a single action.

Let $C_F \subseteq \mathcal{C}$ be the set of conditions that have changed from true to false and let $C_T \subseteq \mathcal{C}$ be the set of conditions that have changed from false to true during τ. The BT composition of those conditions, $\mathcal{T}_{cond}$, is depicted in Fig. 8.

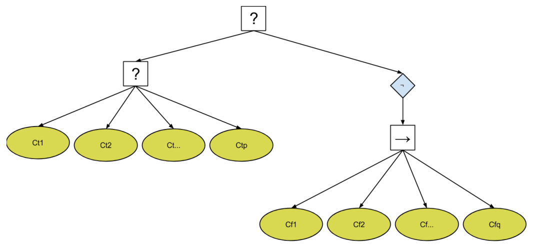


Fig. 8. Graphical representation of $\mathcal{T}_{cond}$.

The conditions encoded in $\mathcal{T}_{cond1}$ make the fitness value decrease (i.e. when $\mathcal{T}_{cond1}$ return true, the fitness value decreases). Thus we need to learn the BT to be performed whenever $\mathcal{T}_{cond1}$ returns success to enable increase in fitness value. The learning procedure continues. Let $\mathcal{T}_{acts1}$ be the learned BT to be performed when $\mathcal{T}_{cond1}$ returns success, the BT that the agent runs is now

$$\mathcal{T}_1 \triangleq \text{selector}(\tilde{\mathcal{T}}_1, \mathcal{T}_0) \tag{4}$$

where

$$\tilde{\mathcal{T}}_1 \triangleq \text{selector}(\mathcal{T}_{cond1}, \mathcal{T}_{acts1}). \tag{5}$$

The agent runs $\mathcal{T}_1$ as long as the value fitness function increases. When the fitness stops to increase a new BT is learned following the previous procedure $\mathcal{T}_1$. Generally as long as the goal is not reached, the learned BT is:

$$\mathcal{T}_i \triangleq \text{selector}(\tilde{\mathcal{T}}_i, \mathcal{T}_{i-1}) \tag{6}$$

When the final BT is learned (i.e. that BT that leads the agent to the goal), we run the anti-bloat algorithm to remove possibly inefficient nodes introduced due to a large time window τ or due to the randomness needed for the GP.

Algorithm 3: Pseudocode of the learning algorithm

```
$\gamma_{old} \leftarrow$ GetFitness($nil$)
$t_0 \leftarrow$ GetFirstBT()
$t \leftarrow t_0$
while $\gamma < 1$ do
    $\gamma \leftarrow$ GetFitness($t$)
    if $\gamma_{old} \geq \gamma$ then
        $t_{cond} \leftarrow$ GetChangedConditions()
        $t_{acts} \leftarrow$ LearnSingleAction($t$)
        if $t_{acts} = nil$ then
            $t_{act} \leftarrow$ LearnBT($t$)
        $\tilde{t} \leftarrow$ Sequence($t_{cond}$, $t_{acts}$)
    $t \leftarrow$ Selector($\tilde{t}$, $t$)
    $\gamma_{old} \leftarrow \gamma$
return $t$
```

D. Exploitation of the collected knowledge

At each stage, the resulting BTs are executed in a simulated (or real) environment to evaluate against a fitness function. Based on the value of the fitness function, the learning algorithm decides the future course. The fitness function is defined in accordance to a given goal. For instance, if the goal is to complete a level in a game, then the fitness function is a function of the following: the game points acquired by the agent (game character), how far (distance) the agent has traversed, how much time the agent has spent in the game level, how many enemies are shot by the agent, etc.

E. Anti-bloat control

Once we obtained the BT that satisfies the goal, we search for ineffective sub-trees, i.e. those action compositions that are superfluous for the goal reaching. This process is called anti-bloat control in GP. Most often, the genetic operators (such as size fair crossover and size fair mutation) or the selection mechanism in GP applies the bloat control. However, in this work, we first generate the BT using the GP without size/depth restrictions in order to achieve a complex yet practical BT. Then we apply bloat control using a separate breadth-first algorithm that reduces the size and depth of the generated BT while keeping the properties of the BT and its performance at the same time.

To identify the redundant or unnecessary sub-trees, we enumerate the sub-trees with a Breadth-first enumeration. We run the BT without the first sub tree and we check if the fitness function has a lower value or not. In the former case the sub-tree is kept, in the latter case the sub-tree is removed creating a new BT without the sub-tree mentioned. Then we run the same procedure on the new BT. The procedure stops when there are no ineffective sub-tree found. Algorithm 4 presents a preudo-code of the procedure.

Algorithm 4: Pseudocode of a anti-bloat control for inefficient subtree(s) removal.

```
1 $t_{new} \leftarrow t$
2 $i \leftarrow 0$
3 while $i \le$ GetNodesNumber($t_{new}$) do
4     $i \leftarrow i + 1$
5     $t_{rem} \leftarrow$ RemoveSubtree($t_{new}$, $i$)
6     if GetFintess($t_{rem}$) $\ge$ GetFintess($t_{new}$) then
7         $t_{new} \leftarrow t_{rem}$
8         $i \leftarrow 0$
9 return $t_{new}$
```

Remark 4: The procedure is trivial using a BT due to its tree structure.

VI. Preliminary experiments

To experimentally verify the proposed approach, we used the Mario AI [33] open-source benchmark for the Super Mario Bros game developed initially by Nintendo. The gameplay in Mario AI, as in the original Nintendo's version, consists in moving the controlled character, namely Mario, through two-dimensional levels, which are viewed sideways. Mario can walk and run to the right and left, jump, and (depending on which state he is in) shoot fireballs. Gravity acts on Mario, making it necessary to jump over cliffs to get past them. Mario can be in one of three states: *Small*, *Big* (can kill enemies by jumping onto them), and *Fire* (can shoot fireballs).

The main goal of each level is to get to the end of the level, which means traversing it from left to right. Auxiliary goals include collecting as many coins as possible, finishing the level as fast as possible, and collecting the highest score, which in part depends on number of collected coins and killed enemies.

Complicating matters is the presence of cliffs and moving enemies. If Mario falls down a hole, he loses a life. If he touches an enemy, he gets hurt; this means losing a life if he is currently in the Small state. Otherwise, his state degrades from Fire to Big or from Big to Small.

a) Actions: In the benchmark there are five action available: Walk right, walk left, crouch, shoot, and jump.

b) Conditions: In the benchmark there is a receptive field of observations. We chose a 5×5 grid for such receptive field as shown in Fig. 9. For each box of the grid there are 2 conditions available: if the box is occupied by an enemy and if the box is occupied by an obstacle. For a total of 50 conditions.

c) Fitness Functions: The fitness function is given by a non linear combination of the distance passed, enemy killed, number of hurts, and time left when the end of the level is reached. Fig.9 illustrates the receptive field around Mario, used our experiments.

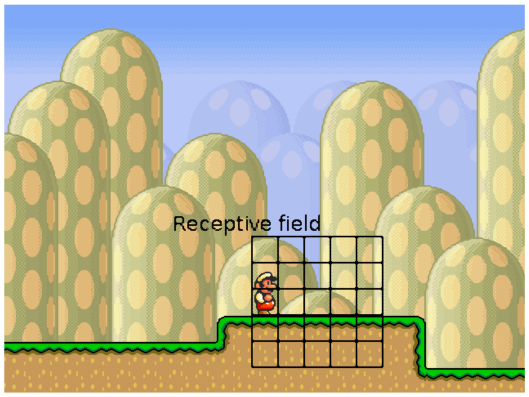


Fig. 9. Receptive field around Mario. In this case Mario is in state Fire, hence he occupies 2 blocks.

A. Testbed 1: No enemies and no cliffs

This is a simple case. The agent has to learn how to move towards the end of the level and how to jump obstacles. The selection method in the GP is the rank-space method. A youtube video shows the learning phase in real time (https://youtu.be/uaqHbzRbqrk). Fig. 10 illustrates a resulting BT learned for the Testbed 1.

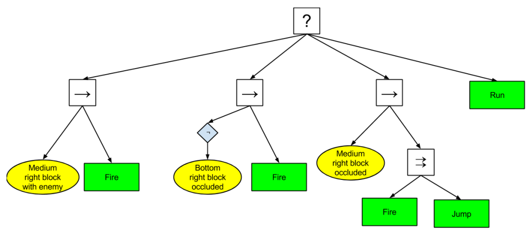

?
→
→
→
Run
Medium right block with enemy
Fire
¬
Bottom right block occluded
Fire
Medium right block occluded
⇉
Fire
Jump

Fig. 10. BT learned for Testbed 1.

B. Testbed 2: Walking Enemies and No Cliffs

This is slightly more complex than Tesbed 1. The agent has to learn how to move towards the end of the level, how to jump obstacles, and how to kill the enemies. The selection method in the GP is the rank-space method. A youtube video shows the learning phase in real time (https://youtu.be/phy98jbdgQc).

Remark 5: The youtube video does not show the initial BT learned $\mathcal{T}_0$, which was a simple action "Right".
Fig. 11 illustrates a resulting BT learned for the Testbed 2.

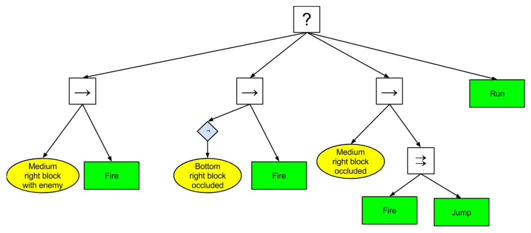

?
→
→
→
Run
Medium right block with enemy
Fire
¬
Bottom right block occluded
Fire
Medium right block occluded
⇉
Fire
Jump

Fig. 11. BT learned for Testbed 2.

C. Testbed 3: Flying Enemies and Cliffs

For this testbed we used a newer version of the Benchmark. The agent has to learn how to move towards the end of the level, how to jump obstacles, and how to kill the enemies and how to avoid cliffs. The enemies in this testbed can fly. The selection method in the GP is the rank-space method. A youtube video shows the final BT (https://www.youtube.com/watch?v=YfvdHY-DXwM).

We avoid to depict the final generated BT because of size restrictions (about 20 nodes).

VII. Conclusions

BT are used to represent the knowledge as it provides a valid alternative to conventional planners such as FSM, in terms of modularity, reactiveness, scalability and domain-independence. In this paper we presented a model-free AP for an autonomous agent using metaheuristic optimization approach involving a combination of GP and greedy-based algorithms to generate an optimal BT to achieve a desired goal. To our best knowledge, this is the first work following a fully model-free framework whereas other relevant works either use model based frameworks or use apriori information for the behavior trees. We have detailed how we addressed the following subjects in AP: knowledge representation; learning algorithm; extraction of experience and exploitation of the collected knowledge. Further, the proposed approach was tested in the open-source "Mario AI" benchmark to simulated autonomous behavior of the game character "Mario" in the benchmark simulator. Some samples of the results are illustrated in this paper. A video of an working example and illustration is available at https://youtu.be/phy98jbdgQc. Even though the results are encouraging and comparable to the state-of-the-art, more vigorous analysis and validation will be needed before extending the proposed approach to real-world robots.

VIII. Future Work

The first future work is to examine our approach in the Mario AI benchmark with extensive experiments and compare our results with other state-of-the-art approaches such as [23]. We further plan to explore dynamic environments and adapt our algorithm accordingly. Inspired by the work in [34], we also plan to look at the possibility of using supervised learning to generate an optimal BT.

Regarding the supervised learning, we are developing a model-free framework to generate BT by learning from training examples. The strength of the approach lies on the possibility of separating the tasks to learn. A youtube video (http://youtu.be/ZositEzjidE) shows a preliminary result of the supervised learning approach implemented in the MarioAI benchmark. In this example, the agent learns separately the task *shoot* and the task *jump* from examples of a game played by an user.

References

[1] T. M. Mitchell, *Machine learning. WCB*. McGraw-Hill Boston, MA:, 1997, vol. 8.

[2] S. Jiménez, T. De La Rosa, S. Fernández, F. Fernández, and D. Borrajo, "A review of machine learning for automated planning," *The Knowledge Engineering Review*, vol. 27, no. 04, pp. 433–467, 2012.

[3] P. Nayak, J. Kurien, G. Dorais, W. Millar, K. Rajan, R. Kanefsky, E. Bernard, B. Gamble Jr, N. Rouquette, D. Smith *et al.*, "Validating the ds-1 remote agent experiment," in *Artificial intelligence, robotics and automation in space*, vol. 440, 1999, p. 349.

[4] J. Fdez-Olivares, L. Castillo, O. Garcıa-Pérez, and F. Palao, "Bringing users and planning technology together. experiences in siadex," in *Proc ICAPS*, 2006, pp. 11–20.

[5] J. G. Bellingham and K. Rajan, "Robotics in remote and hostile environments," *Science*, vol. 318, no. 5853, pp. 1098–1102, 2007.

[6] S. Jiménez, T. De la Rosa, S. Fernández, F. Fernández, and D. Borrajo, "A review of machine learning for automated planning," *The Knowledge Engineering Review*, vol. 27, no. 04, pp. 433–467, 2012.

[7] T. Bylander, "The computational complexity of propositional strips planning," *Artificial Intelligence*, vol. 69, no. 1, pp. 165–204, 1994.

[8] B. Tom, "Complexity results for planning." in *IJCAI*, vol. 10, 1991, pp. 274–279.

[9] F. Bacchus and F. Kabanza, "Using temporal logics to express search control knowledge for planning," *Artificial Intelligence*, vol. 116, no. 1, pp. 123–191, 2000.

[10] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "Shop2: An htn planning system," *J. Artif. Intell. Res.(JAIR)*, vol. 20, pp. 379–404, 2003.

[11] S. Minton, *Learning search control knowledge: An explanation-based approach*. Springer, 1988, vol. 61.

[12] D. Isla, "Halo 3-building a Better Battle," in *Game Developers Conference*, 2008.

[13] M. Colledanchise, A. Marzinotto, and P. Ögren, "Performance Analysis of Stochastic Behavior Trees," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, June 2014.

[14] P. Ögren, "Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees," in *AIAA Guidance, Navigation and Control Conference, Minneapolis, MN*, 2012.

[15] M. Colledanchise and P. Ogren, “How Behavior Trees Modularize Robustness and Safety in Hybrid Systems,” in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, Sept 2014, pp. 1482–1488.

[16] J. A. D. Bagnell, F. Cavalcanti, L. Cui, T. Galluzzo, M. Hebert, M. Kazemi, M. Klingensmith, J. Libby, T. Y. Liu, N. Pollard, M. Pivtoraiko, J.-S. Valois, and R. Zhu, “An Integrated System for Autonomous Robotics Manipulation,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, October 2012, pp. 2955–2962.

[17] A. Klökner, “Interfacing Behavior Trees with the World Using Description Logic,” in *AIAA conference on Guidance, Navigation and Control, Boston*, 2013.

[18] M. Colledanchise, A. Marzinotto, D. V. Dimarogonas, and P. Ögren, “Adaptive fault tolerant execution of multi-robot missions using behavior trees,” *CoRR*, vol. abs/1502.02960, 2015. [Online]. Available: http://arxiv.org/abs/1502.02960

[19] K. R. Guerin, C. Lea, C. Paxton, and G. D. Hager, “A framework for end-user instruction of a robot assistant for manufacturing.”

[20] G. B. Parker and M. H. Probst, “Using evolution strategies for the real-time learning of controllers for autonomous agents in xpilot-ai.” in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–7.

[21] G. de Croon, L. O’Connor, C. Nicol, and D. Izzo, “Evolutionary robotics approach to odor source localization,” *Neurocomputing*, vol. 121, no. 0, pp. 481 – 497, 2013, advances in Artificial Neural Networks and Machine Learning Selected papers from the 2011 International Work Conference on Artificial Neural Networks (IWANN 2011).

[22] C. Lazarus and H. Hu, “Using genetic programming to evolve robot behaviours,” in *Proceedings of the 3rd British Conference on Autonomous Mobile Robotics and Autonomous Systems. Manchester*, 2001.

[23] D. Perez, M. Nicolau, M. O’Neill, and A. Brabazon, “Evolving behaviour trees for the mario ai competition using grammatical evolution,” in *Proceedings of the 2011 International Conference on Applications of Evolutionary Computation - Volume Part I*, ser. EvoApplications’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 123–132.

[24] N. Shaker, M. Nicolau, G. Yannakakis, J. Togelius, and M. O’Neill, “Evolving levels for super mario bros using grammatical evolution,” in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, Sept 2012, pp. 304–311.

[25] G. D. Croon, M. F. V. Dartel, and E. O. Postma, “Evolutionary learning outperforms reinforcement learning on non-markovian tasks,” in *Workshop on Memory and Learning Mechanisms in Autonomous Robots, 8th European Conference on Artificial Life*, 2005.

[26] R. Dey and C. Child, “Ql-bt: Enhancing behaviour tree design and implementation with q-learning,” in *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, Aug 2013, pp. 1–8.

[27] C.-U. Lim, R. Baumgarten, and S. Colton, “Evolving behaviour trees for the commercial game defcon,” in *Proceedings of the 2010 International Conference on Applications of Evolutionary Computation - Volume Part I*, ser. EvoApplicatons’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 100–110.

[28] K. Y. W. Scheper, S. Tijmons, C. C. de Visser, and G. C. H. E. de Croon, “Behaviour trees for evolutionary robotics,” *CoRR*, vol. abs/1411.7267, 2014.

[29] I. Rechenberg, “Evolution strategy,” *Computational Intelligence: Imitating Life*, vol. 1, 1994.

[30] J. W. Tweedale and L. C. Jain, “Innovation in modern artificial intelligence,” in *Embedded Automation in Human-Agent Environment*. Springer, 2012, pp. 15–31.

[31] Z. Fu, B. L. Golden, S. Lele, S. Raghavan, and E. A. Wasil, “A genetic algorithm-based approach for building accurate decision trees,” *INFORMS Journal on Computing*, vol. 15, no. 1, pp. 3–22, 2003.

[32] L. Davis, *Genetic algorithms and simulated annealing*. Morgan Kaufman Publishers, Inc.,Los Altos, CA, Jan 1987.

[33] S. Karakovskiy and J. Togelius, “The mario ai benchmark and competitions,” *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 55–67, 2012.

[34] E. Tomai and R. Flores, “Adapting in-game agent behavior by observation of players using learning behavior trees,” in *Proceedings of the 9th International Conference on the Foundations of Digital Games (FDG 2014)*, 2014.