Multi Agent Systems Project

Team - 8

November 2023

1 Base Paper

Title

Learning Behavior Trees for Autonomous Agents with Hybrid Constraints Evolution

Authors

Qi Zhang, Jian Yao, Quanjun Yin and Yabing Zha

Link

https://doi.org/10.3390/app8071077

Abstract

In modern training, entertainment and education applications, behavior trees (BTs) have already become a fantastic alternative to finite state machines (FSMs) in modeling and controlling autonomous agents. However, it is expensive and inefficient to create BTs for various task scenarios manually. Thus, the genetic programming (GP) approach has been devised to evolve BTs automatically but only received limited success. The standard GP approaches to evolve BTs fail to scale up and to provide good solutions, while GP approaches with domain-specific constraints can accelerate learning but need significant knowledge engineering effort. In this paper, we propose a modified approach, named evolving BTs with hybrid constraints (EBT-HC), to improve the evolution of BTs for autonomous agents. We first propose a novel idea of dynamic constraint based on frequent sub-trees mining, which can accelerate evolution by protecting preponderant behavior sub-trees from undesired crossover. Then we introduce the existing 'static' structural constraint into our dynamic constraint to form the evolving BTs with hybrid constraints. The static structure can constrain expected BT form to reduce the size of the search space, thus the hybrid

constraints would lead more efficient learning and find better solutions without the loss of the domain-independence. Preliminary experiments, carried out on the Pac-Man game environment, show that the hybrid EBT-HC outperforms other approaches in facilitating the BT design by achieving better behavior performance within fewer generations. Moreover, the generated behavior models by EBT-HC are human readable and easy to be fine-tuned by domain experts.

2 Key Contributions of the paper

- Use of Evolving Behaviour Trees (EBT) approach to train behavior trees using Genetic Programming.
- Introducing constraints to optimize the training of the trees.
- Constraints proposed - Hybrid Constraints
- Two Components of Hybrid Constraints - Static Structural Constraints and Dynamic Constraints
- **Static Structural Constraints** : These are the syntax rules of tree generation to reduce redundancy i.e. reduce the number of trees possible for the exact same behavior.
- **Static Structural Constraints rules** :
  - Selector node may only be placed at depth levels that are even.
  - Sequence node may only be placed at depth levels that are odd.
  - All terminal child nodes of a node must be adjacent, and those child nodes must be one or more condition nodes followed by on or more action nodes. If there is only one terminal child node, it must be an action node.
- **Dynamic Constraints** : The idea behind dynamic constraints is that there may be an optimal subtree we find in a non-final generation executing the ideal behavior in a certain situation, hence we assume if there exists such a tree it would occur frequently in the higher fitness trees in a generation. We find such subtree and adjust node cross over probabilities to preserve the subtree structure.
- **Dynamic Constraints Steps** :
  - **Frequent Sub-Tree Mining** : An adaptation of FREQT, a classic pattern mining algorithm, was used to mine frequent sub-tree structures in population.
  - **Node Cross-Over Probability Adjustment** : In each tree nodes are classified into protected and unprotected set according to the frequent subtrees found and crossover probabilities are adjusted to preserve the sructure of frequent subtrees

- **Game Environment** : The trees were evaluated in the java based 'Ms.PacMan vs Ghost' game competition environment and the condition and action nodes were defined accordingly.

3 Our Implementation

3.1 Behavior Tree

3.1.1 Node

The node class is the backbone of the execution system in the behavior tree. The node class consists of a the following important attributes `parent`, `siblingOrder`, `children`.

3.1.2 Action Node

The Action Node inherits from the `Node`, and builds upon it by including `actionFunctions` and `actionDescription`. These point to function object that is the actual function that implements the logic behind it.

3.1.3 Condition Node

The `ConditionNode` in similarity to `ActionNode`, implements a `conditionFunction` and `conditionDescription`.

3.1.4 Selector Node

The `SelectorNode` is a parent node that iteratively runs all the children node from left to right, whenever a node returns `Failure`, the Selector node moves to the next child, whereas if a node returns `Success`, the Selector node returns the value back.

3.1.5 Sequence Node

The `SequenceNode` is similar to the `SelectorNode` but it returns back if any of the children returns `Failure`, and continues on `Success`.

The Behavior Tree is implemented in the `Tree` class located in `classes` directory. The Behaviour Tree is a `Node` and its subsequent which recursively have children of their own.

A key function that we use to parse the tree is `self.getExecutionOrder` which returns the node objects as a list in the order they would be normally executed.

We have functions to `performMutation` of various kinds, (add, remove, replace) that all are called with some fixed probability to ensure diversity in the population which is important as the Static Structural constraints limit the Crossover operation.

3.2 Dynamic Constraints

3.2.1 Frequent Tree Mining

FREQT is a method to find common tree patterns in a set of labeled ordered trees (LOT), which are trees with labels and orders for their nodes. FREQT uses a technique called rightmost expansion to generate candidate patterns step by step. It also calculates the frequencies of the candidates by only keeping track of the rightmost leaf occurrences3. FREQT has shown to be almost linear in its scalability with respect to the total size of the maximal patterns, depending on the length of the longest pattern. In this paper, FREQT is used to mine frequent sub-tree structures in the population of behavior trees (BTs) that are evolved by genetic programming (GP). The frequent sub-trees represent preponderant behaviors that can deal with certain situations effectively. By adjusting the crossover probabilities based on the frequent sub-trees, the proposed approach can protect and exploit these behaviors and speed up the evolution of BTs.

FREQT Algorithm. Given a set of labels and a data tree, along with a minimum support σ, the task is to find all patterns that appear as subtrees σ number of times in the data tree. This can be easily extended to the case of multiple data trees by solving individually for each tree and taking aggregate measurements for determining overall support.

First, a set of all σ-frequent patterns of size 1 are calculated, along with the set of their right-most occurrences in the data tree, referred to as RMOs. This is done simply by iterating over the nodes in the data tree.

Then, for each $k > 1$, σ-frequent patterns of size k are calculated by performing right-most expansions over patterns of size $k - 1$. This is done using *Expand-Trees* algorithms.

***Expand-Trees* Algorithm.** Given the set of right-most expansions of the $k - 1$ sized patterns, we iterate over all possible expansions on the patterns by trying all possible siblings, and siblings of the paterns, of existing RMOs. This is done using *Update-RMOs*.

***Update-RMOs* Algorithm.** This iterates over all possible candidates of the rightmost branch of a given pattern in the data tree. The rightmost branch is defined as the path from the root of the data tree to its rightmost leaf. Then, it tries to perform an extension by picking a node on this branch and finding new patterns of one greater size.

3.2.2 Node Cross-Over Probability Adjustment

Let $V(T)$ be the set of all nodes in tree T except it's root node. Let T_i be the frequent subtrees found in T where $i = 1, 2, 3, ..., N$. We define $V^r(T_i)$ as the root node of T_i and $V^{in}(T_i) = V(T_i) \setminus V^r(T_i)$.

Here we define the protected set of nodes as $V^{pro}(T) = \bigcup_{i=1}^{N} V^{in}(T_i)$ and unprotected set as $V^{unpro}(T) = V(T) \setminus V^{pro}(T)$. This ensures that the protected set contains all nodes except the root nodes of the frequent subtrees (unless a frequent subtree is a subtree of another frequent subtree, in which case the root is in protected set).

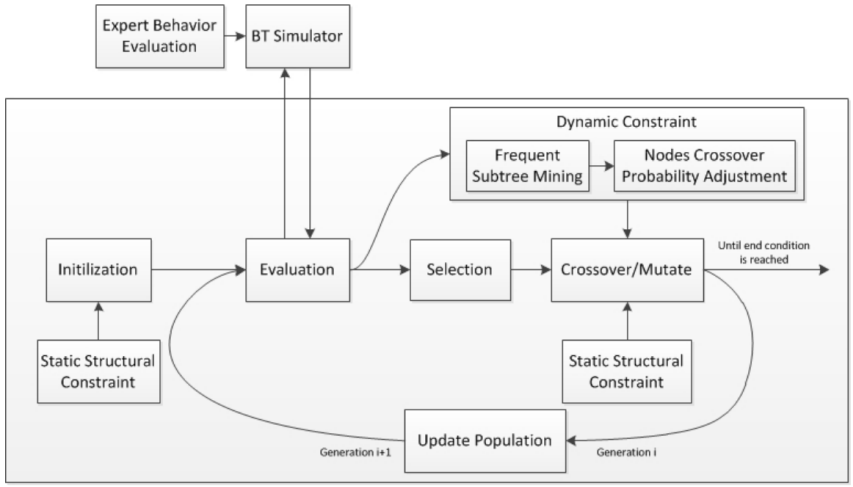
Then we adjust a probabilities of crossover happening at these nodes as follows :

$$p_{pro}(v) = \frac{\gamma}{|V_{pro}(T)| + ||V_{unpro}(T)|} \quad v \in V_{pro}(T)$$
$$p_{unpro}(v) = \frac{(1 - |V_{pro}(T)| * p_{pro}(v))}{|V_{unpro}(T)|} \quad v \in V_{unpro}(T)$$

where $\gamma = 0.9$.

This would help preserve the frequent subtree as the probability of crossover happening at a node inside it is lower and even if the crossover happens at it's root, the whole subtree would move.

3.3 Evolving Behavior Trees and Genetic Programming



We initialize random trees which follow static structural constraints. Then

these trees are evaluated and high fitness trees are selected for crossover and mutation. Crossover happens according to the adjusted probabilities due to Dynamic Constraints. Mutation is independent of of Dynamic Constraints. Both these operations happen satisfying the Static Structural Constraints. The population is updated and this process is repeated till 100 generations.

3.4 Game Environment

The game environment used in the paper is java based. Since we decided to work in python we found an implementation of PacMan in python (inspired by an assignment of University of California, Berkeley). We modified it according to our needs including things like adding option to toggle display mode, coding up our own ghost behavior, etc. We used this version of the PacMan game made in python as our testing environment.

3.4.1 Condition Nodes

- **isInedibleGhostCloseVLow/Low/Med/High/VHigh/Long** : six condition nodes which return *Success* if there is a ghost in the *Inedible* state within a certain fixed distance range, as well as targeting that ghost. The distance/path length ranges are as follows :
  - *VLow* : [0, 5]
  - *Low* : [6, 10]
  - *Med* : [11, 15]
  - *High* : [16, 20]
  - *VHigh* : [21, 25]
  - *Long* : [26, 30]
- **isEdibleGhostCloseVLow, Low/Med/High/VHigh/Long** : six condition nodes which return *Success* if there is a ghost in the *Edible* state within a certain fixed distance range, as well as targeting that ghost. The distance/path length ranges are as follows :
  - *VLow* : [0, 5]
  - *Low* : [6, 10]
  - *Med* : [11, 15]
  - *High* : [16, 20]
  - *VHigh* : [21, 25]
  - *Long* : [26, 30]
- **isTargetGhostEdibleTimeLow/Med/High** : three condition nodes which return *Success* if a previous condition node has targeted a ghost, which is edible and whose remaining time in the *Edible* state is within a certain fixed range. The time ranges are as follows :

- *Low* : [0, 120)
- *Med* : [120, 240)
- *High* : [240, 361)

- **isGhostScoreHigh/VHigh/Max** : three condition nodes which return *Success* if the current point value for eating a ghost is equal to a certain fixed value. The ghost score values are as follows
  - *High* : 400
  - *VHigh* : 800
  - *Max* : 1600

3.4.2 Action Nodes

- **moveToEatAnyPill** : an action node which set Pac-Man's direction to the nearest pill or power pill, returning *Success* if any such pill exists in the level or *Failure* otherwise.
- **moveToEatNormalPill** : an action node which set Pac-Man's direction to the nearest normal pill, returning *Success* if any such pill exists in the level or *Failure* otherwise.
- **moveToEatPowerPill** : an action node which set Pac-Man's direction to the nearest Power pill, returning *Success* if any such pill exists in the level or *Failure* otherwise.
- **moveAwayFromGhost** : an action node which set Pac-Man's direction away from the nearest ghost that was targeted in the last condition node executed, returning *Success* if a ghost has been targeted or *Failure* otherwise.
- **moveTowardsGhost** : an action node which set Pac-Man's direction towards the ghost that was targeted in the last condition node executed, returning *Success* if a ghost has been targeted or *Failure* otherwise.

3.4.3 Fitness Function

The fitness function is the sum of averaged game score and a parsimony pressure value as formula :

$$f_p(x) = f(x) - c\, l(x)$$

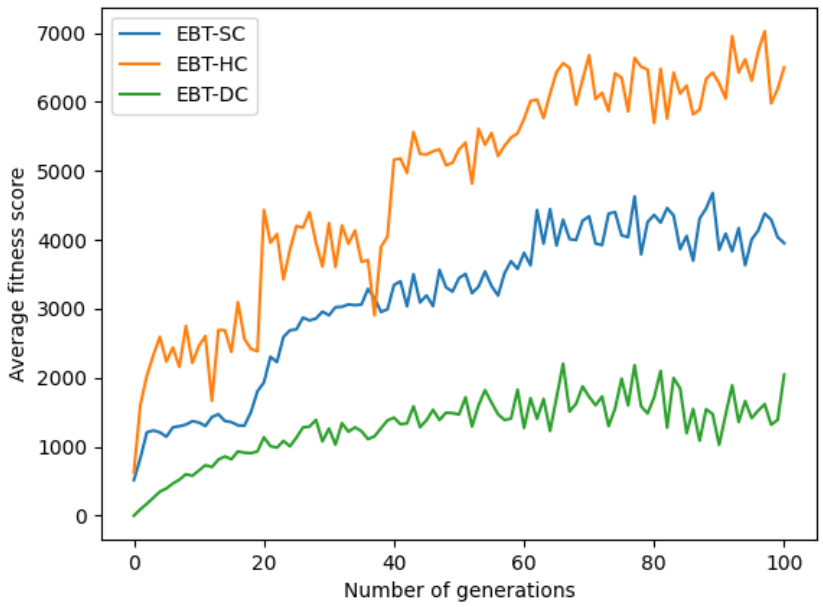
where :

- $f_p(x)$ - Fitness Value
- $f(x)$ - Averaged game score for a few game runs
- c - Parsimony coefficient
- $l(x)$ - Node size of x

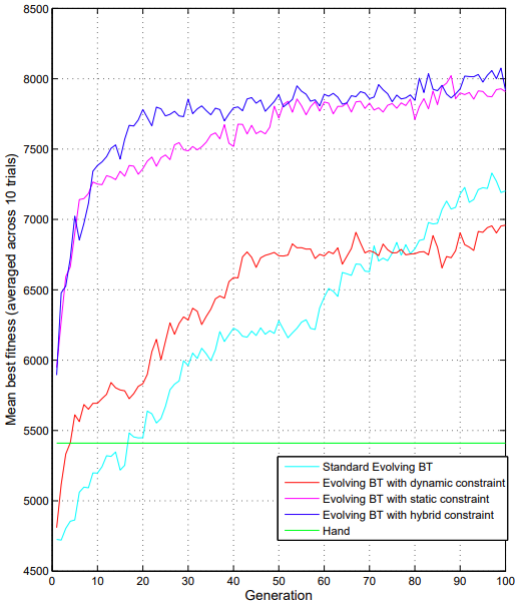
3.5 Parameters

- **Population Size** : 100
- **Generations** : 100
- **Initial Min Depth** : 2
- **Initial Max Depth** : 3
- **Tournament Size** : 5
- **Parsimony Coefficient** : 0.9
- **Crossover Probability** :
- **Mutation Probability** :
-

4 Results



5 Comparison with the paper



As we can see in both our graphs scores with HC rises faster than SC and scores with SC rises faster than DC. There are some noticeable differences in shapes of the graphs, but that is to be expected because of the randomization in GP and because our test environment is different than in the paper.

6 Learnings

The learnings from this paper have been about the use and power of genetic programming in granularly increasing the skill of an agent through variations such as the proposed mutations with effective crossover algorithms.

Hybrid constraints bring adaptability, enhancing performance in dynamic environments. They optimize limited resources that are present for them, and their versatility aids modeling real-world complexities. Moreover, they mitigate overfitting and underfitting risks. This project highlights the potential of hybrid constraints for autonomous systems, with the promise of enhancing decision-making across various domains. Allowing for transparent decision making.