Evolving Behavior Trees for Bomberland using Genetic Programming

Florian VENOT

I. Contents

II. Introduction

Nowadays, lots of Artificial Intelligence (AI) competitions, with cash prizes, are organised on Internet by associations and companies for everyone to train or compete on complex tasks and problems that can be more or less close to real problems. A lot of them are taking place in the form of a game where agents will compete against each other or again the environment. These competitions are a way to boost the development and the research around AI and their practical applications in the real world, something that can also benefit to companies in the long term.

For this project, I chose to make an AI for the game Bomberland, subject of the past annual AI tournament Coder One. As a first time doing an AI for a game, I settled on a game not too simple but too complex either, in terms of action possibilities or environment constraints.

III. The Game: Bomberland

The Game, brought by CoderOne [1], is highly inspired by Bomberman, a popular game by originally edited by Hudson Soft, but here two agents/players are facing, each controlling a team of 3 units and trying to be the last one with one or more units alive. It's played on a 15x15 square grid where different types of blocks (wood, ore, and metal) are initially placed. Some of them can be broken by bombs that each unit have in their inventory, in the number of 3, and that they can place and detonate in order also to attack enemy units. Some items can appear during the game and units can pick them up to either regain a bomb or increase the bomb range. Depending on the parameters at some point of the game fire will start to spawn at the edges of the grid and will progressively fill it in spiral to force the units to move and fight until only one agent is remaining, meaning all the unit of the other agent are dead. [1]

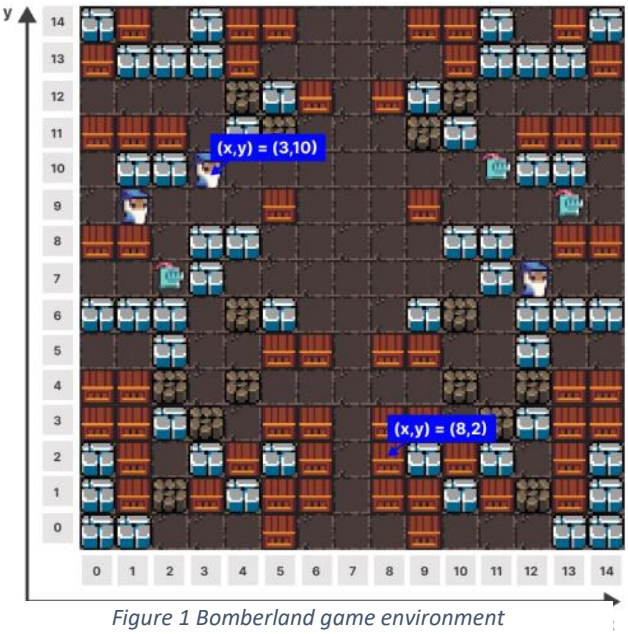

Figure 1 Bomberland game environment

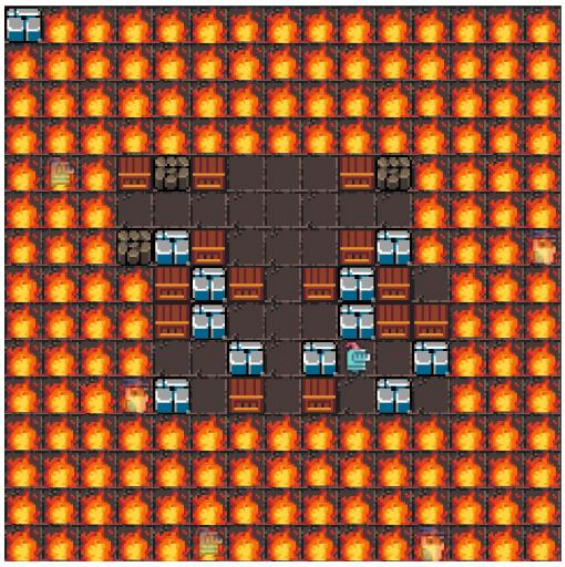

Figure 2 Ring of Fire closing

IV. Behavior Trees

A. What are Behavior Trees?

Behavior trees are a task switching structure in the form of a tree i.e., acyclic oriented graph, used usually to control the comportment of Non-Playable Characters in video games or robots in real life.

There are different types of nodes that compose a behavior tree: the composite nodes that have child nodes and that manipulate the control flow and the leaf nodes, representing actions and conditions, more on the execution side. A behavior tree is run by being tick repeatedly from the root down to its children, each time starting from the leftmost on each depth. The nodes then report a tick to their parents associated with a status being either Success, Running or Failure, related to the execution of the node or its children.

Amongst the composite nodes you can mainly find the following ones:

- The Fallback (Selector): execute its children in order (left to right) and report success to its parent if one of them is executed successfully
- The Sequence: execute its children in order (left to right) and report success only if all are executed successfully

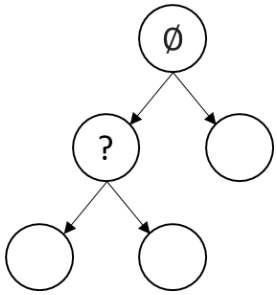

Figure 3 A Simple Behavior Tree

In some framework you can also find other variations such as the Parallel node, with a behavior like the Sequence with the exception that it tries to execute all the children at the same time i.e., the execution order does not matter. It also transmits a success tick if all the children execution are successful. [2]–[5]

B. Why Behavior Trees instead of Neural Networks?

In the world of AI, the Neural Networks are used a lot and it's a legit question to ask, what are the advantages of a BT over Neural networks?
It mainly comes down to the readability of the end model, the understanding of the decision it'll take, and the transparency needed to also know its limitations. Neural networks are black box that are difficult to analyse past a certain number of dimensions of the problem it's applied to. With BT we can analyse the optimal models that have been generated to recognize patterns and rationally understand how it works.

It's also important to note that there is also another type of task-switching structure which is popular in the Finite State Machines and that could be used to control an agent behaviour. The drawbacks of the FSM will come here with the special needs of evolving the structure often, which will require to update a lot of transitions each time when removing or adding states in comparison to the tree structure where it's easy to add, swap or remove whole part of the tree i.e., subtrees. That's why I'll concentrate here on BT, which are convenient from a modularity point of view. FSM can also become quite messy to read and understand with a growing number of states.

V. Evolutionary Computation: Genetic Algorithm and Genetic Programming

A. Introduction

Amongst the different approaches in artificial intelligence, Evolutionary Computation (EC) occupies a special place in the way it tries to mimic what's happening in the nature with the evolution by natural selection, conceived by Darwin in particular, by replicating different key mechanisms such as selection, reproduction, or mutation, to find optimized solutions for a wide variety of problems. More precisely in EC you can find the subfields of Genetic Algorithms (GA) that uses the biological mechanism of cross-over i.e., reproduction to mix the genomes of two individuals alongside the mutation and selection[6]–[8]. Usually, the genome is represented as a string or a bit string in the cases where the problems is searching for an optimal set of values as a solution to a problem, such as

finding a multi-dimensional function extremes or an order of cities for the Travelling Salesman Problem.

In our case we'll use a similar subfield called Genetic Programming (GP) but which, at the difference of GA, manipulates programs to solve a problem. The program is then given some inputs to determine its performance for the said problem, which can range from finding a mathematical function to complex algorithms. In fact, Behavior Trees can be used with GP because they're a kind of program that will be executed to control the behavior of an agent in a game for example.

GP use the same evolution mechanism as GA through selection, crossover, and mutation, and which can be applied to tree data structures, a popular and practical way of representing programs.[8]

B. Initial Population

To be able to simulate the evolution of a population with GP, you first need to initialize it. The initialization is an important step because if all the individuals are initially the same, crossover won't be effective and only mutation will slowly be able to change the population. Therefore, we want the initial population to be as diverse as possible in order to explore multiple parts of the universe of possible trees given an alphabet of composites and leaf nodes. To generate random trees, 2 solutions mainly exist: grow method, where a maximum depth is specified and all the leaves are not necessarily at the same depth and full method, where all the leaves will be at the previously determined depth. The method ramped half-half combines the two by apply each one to half of the population. [6]–[8]

In our implementation we'll use the grow method: a maximum depth is specified and for each node on higher depth, a defined probability is also used to choose if the node is a composite or a leaf node. As a leaf node can't have any children, that means some branch of the tree will stop at higher depth that the maximum. This will allow a diverse population in terms of tree shapes.

C. Fitness function

The fitness function is the key element of the selection because it helps to evaluate how each individual performs in relation to the problem. For some problems it can be quite straightforward, such as the total distance for the TSP or the function value at some coordinates. The fitness function also needs to be consistent concerning the genome, which implies a repeatability criterion. Because if two individuals have the same genome, they should have the same fitness. In the case of multi-criterion problems, the way each criterion will be considered in the fitness function will influence the direction towards which the population will evolve.[8]

D. Selection step

The selection step comes after the evaluation of offspring and population's fitness. The goal of this step is to mimic the natural selection and form the base population for the new generation. The main idea is that the fittest the individuals are, the bigger their chance to be selected is. They are multiple selection methods that exists, note that here we're trying to maximize the fitness of individuals:

- Fitness proportional selection: This selection first calculates the chances of being selected for an individual by dividing it's fitness by the fitness of the population. Cumulative probabilities are then calculated and as much as needed to recreate the population, random numbers between 0 and 1 are generated. For each number, find the individual for which it is located between the bounds of the cumulative probability, note that the left bound is included and the right one excluded.

- Ranking selection: First ranking the individuals by fitness and then apply a linear or non-linear function to calculate the probabilities of getting selected. Cumulative probabilities are calculated similarly to the Fitness proportional selection and individual are picked in the same way.
- Tournament selection: (Different variations exists) Individuals are paired randomly and the one with the highest of the 2 is selected for example. Some adaptations might be needed to have enough individuals in the population (as set in parameter for the simulation). [5]–[8]

E. Crossover

Crossover allows the mixing of two genomes so as to create two new ones. In a bit string representation, the first step is usually to select one or more points where the two genomes will be mixed up before alternatively selecting part of one and the other to make the new ones. In genetic programming, when working with tree representations, the crossover consists of selecting a random node, usually different than the root node, for each tree and then creating two offspring each on with the main tree of on parent and replaced subtree of the other. [2], [3], [5], [8]

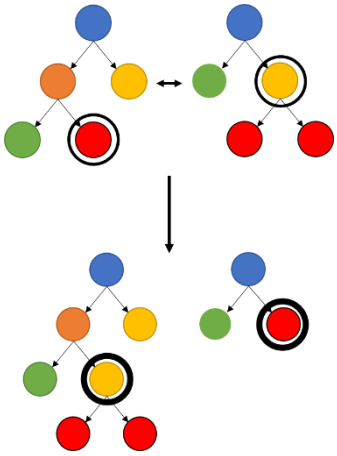

Figure 4 Crossover on Trees

F. Mutation

Mutation is the other evolution mechanism that allows some diversity and novelty in the population and, as it can happen in the nature, maybe help useful patterns or schemas to emerge, making the population even fitter.

With Behavior Tree, each nodes have a probability p of mutating, either set before to a fixed value or to $\frac{1}{nodeCount_{tree}}$. If a node is selected for mutation, it can take multiple forms: swapping the node by one of the same type, e.g., replacing a composite node by another one, a leaf node by another one, removing the node or if the node has children, adding another children. In the last case for the implementation, I only added leaf nodes but another way to do this could be to grow a subtree with probabilities of generating both composite and leaf nodes until only leaf nodes are left.[6]–[8]

VI. Implementation

A. Language and environment

For the implementation, I chose to use Python 3 language firstly because a starter kit was made available, including the basic interface to communicate with the server and the code for a basic agent sending random actions already set up. I also chose to go with python because a library to make Behavior Tree was already existing, py_trees [9]. And finally, as I will mainly do high level programming, it was more convenient to rapidly write code.

The game environment published by CoderOne is under the following form: a server, to which the agents have to connect, is managing the game and can be launched locally with docker with a set of available parameters and rules for the game such as the number of ticks per second or the time, in ticks, at which the fire is appearing in spiral on the game board.

B. Program and training organizations

Concerning the general organization of the project, there is a class called genetic_progamming.py where all the GP-related logic and execution loops to run the evolution are. From this, the evolution loop can be run with different parameters, first Initializing the population, then running the main

loop until a stopping condition is reached, here the maximum number of generations. The steps composing this main loop are the following:

- Evaluate the population and offspring
- Select the new population (and export it)
- Generating crossover offspring
- Generating mutation offspring

C. Behavior Tree and Nodes

With the py_trees library, tree nodes are called behaviours and multiple types of behaviours such as composite or decorators, which we'll not use in our implementation for simplification reasons, but they can be useful to modify the signal transmitted by a child, repeat multiple times part of a tree to simplify it and possibly avoid repetitions. Composites nodes available and that we'll use here are Selector (Fallback) and Sequence. More of them exist such as Parallel but to keep things simple for the exploration, it's better to start with reduce set of nodes.

About the leaf nodes, it's necessary to create them by extending the available behaviour class. In this implementation, I created two main types of leaf nodes with the actions, that send an action to the server, and conditions, that can help as a checks or guard to prevent executing some parts of the tree. Finally, the actions are split in two parts, the Simple Actions, that correspond to the set of move actually sent to the server by the agent (up, down, left, right, place bomb, detonate bomb) and the Complex Actions which are more high-level actions that execute a bigger goal by splitting it into small parts of the basic set of moves. Here is a list of the nodes that can constitute the BT:

- Composite
    - Selector
    - Sequence
- Leaf
    - Actions
        - Simple
            - Move (Up, Down, Left, Right)
            - PlaceBomb
            - DetonateLastBomb
            - Idle: Do nothing
        - Complex
            - AttackEnemy: if enemy in an adjacent case, place a bomb
            - MoveToSafePlace: if unit in danger, move to a safer tile
    - Conditions
        - HealthLow: if unit health is at 1hp (Instead of 3)
        - EnemyClose: if enemy unit is in an adjacent case
        - InDanger: if bomb or fire is in an adjacent case

I will describe as an example the algorithm behind the complex action MoveToSafePlace. There is first a hierarchy of the danger in this function where we are first checking if the danger is present directly on the unit position (useful when placing bombs), then on the directly adjacent cases, accessible in one move and finally on the corner of the 3x3 square with the unit at the centre. It helps the unit naturally move when the fire ring is closing and not directly be killed by it.

D. Initialization

As said in the first part on GA/GP, the implementation is using the grow method to initialize the population. The parameters available for the initialization are the population size, the maximum depth, the probability for each generated node to be a composite node and finally the maximum number of children per composite node (minimum is 2). In order to start with a population potentially better than a totally random one, we check is the BT is valid, in the point of view of the problem e.g., if mandatory leaf nodes are included. Indeed, we can assure that a BT can't be fit with at least the basic set of moves such as moving in the 4 directions. We could also argue that placing a bomb is a mandatory move, but some games are finishing with the fire right in the centre when neither of the unit of each agent have any bomb left, so it's still possible to have a fit BT without using bombs.

E. Fitness calculation

In our case, to evaluate the fitness of each behavior tree, we need to make it control an agent and play at least one game against another one. The enemy agent needs to be the same for every tree, at least of the same generation, or then the results can be biased during the selection. The repeatability also implies that the randomness might be reduced to the bare minimum for the fitness to reflect the real performance of the tree.

To achieve this there are different possibilities:

- Fix the world seed and player's position seed to be the same for every game
- Play multiple games and take the average fitness of all the game

However, we also need to consider a common issue present in AI which is called overfitting. It means that the individual will adapt too much to the dataset it is train with and will perform well on it, but on the other hand, it will perform poorly on data he never went upon. The first possibility is subject to overfitting if the environment never changes.

Another aspect of the fitness function is the time taken to calculate the fitness of every individual. Indeed, the fitness calculation quickly becomes the main source of execution time with problems requiring simulations, as it is the case here.

As a result, there's a compromise to do between the consistency of the fitness function, the correct fitting of the solutions to the problem and the time taken to calculate the fitness.

The choice here for the implementation will be the second method of calculation because it could help select BT that performs well in general and not just on a map in particular. The drawbacks are that for it to be consistent enough through the population, multiple games need to be played and therefore, the execution time will be longer.

It's worth noting that another alternative could have been to change the seed every n generation so that the execution time is kept low while being less subject to overfitting. But we can point out that some seeds will favour some trees over others which, in the end affect the consistency.

Finally, the enemy playing against the agent control by a BT also needs to be consistent and correctly challenging. If it is too good, the agent will all have a similar and low fitness and it'll not be effective to differentiate the good ones from the average or bad ones, even if this can be technically modified rectify in the selection step.

To start, in the implementation here, the enemy will be an idling agent. No randomness is then involved by the enemy and as it can be easy to beat, slight advantages will be selected. But as the

population become fitter and fitter, the problem will be the same as an enemy that's too good, all the population will have a similar fitness. To keep it improving continuously, I thought about changing the enemy every n generation to the best of the last one. And to keep the comparison between the generation possible and quite consistent, the formula takes the fitness of the enemy into account.

The fitness function used here is more precisely the following:

$$fitness = fitness_{enemy} + 200 * e^{-\frac{1}{400} * total_{time}} * \begin{cases} unitLeft, & if\ win \\ -enemyUnitLeft, & if\ lose \end{cases}$$

It takes into account the total time of the game in ticks, which can, depending on the parameters goes up to 600-700 ticks, the number of agent units left in case of a win and the number of enemy units left in case of a lose, which both ranges from 1 to 3. The coefficient 200 and -1/400 are there to balance the importance of finishing a game quickly or with a lot of personal units left. We can see on the attached graph of the fitness function that the fitness will be higher if it finished quickly but with one unit left instead of a slow win with all the agent unit left. The analogue reasoning can be done with the fitness function representing a loss. The worst case being if it loses quickly and with a lot of enemy units left.

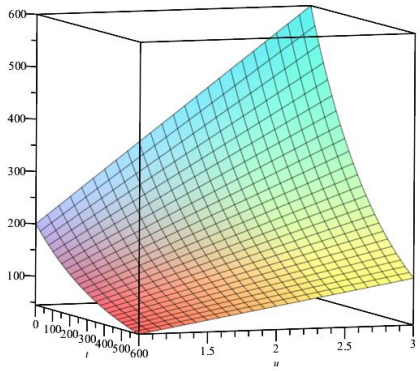

Figure 6 Fitness function if agent wins

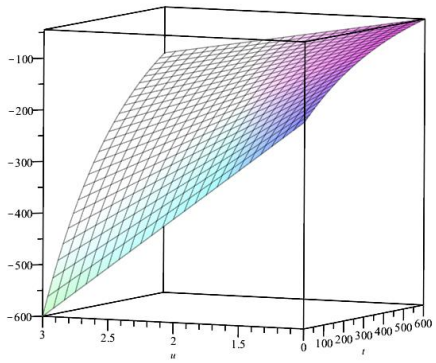

Figure 6 Fitness function if agent loses

These parameters could technically be changed to see how it influences the final behaviour and fitness of the population.

The last part of the fitness function is applying the following coefficient if the number of nodes in the BT is greater than the maximum defined : $e^{-(nodeNumber_{tree} - nodeNumber_{maximumBeforePenalty})}$, which helps to prevent trees growing too much or else they'll be penalized. Another way to prevent bloating, i.e., trees that grows too much with a lot of useless parts, a technique called pruning and can be applied to remove the useless nodes.[4], [5]

There could be a lot of other parameters that could be useful such as the number of bombs placed or left in the inventory, or maybe the health of each unit to precise if the winner was a lot better than the loser, but also the number of powerups picked up by the units. The last parameter could probably influence the agents to pick up more powerups throughout the evolution of the population.

Depending on how the game ends each time for very fit individuals, considering the number of times one beat the other can maybe induce bigger fitness change in the case every game finish with a 1v1 unit and with the spiral of fire totally filling the grid.

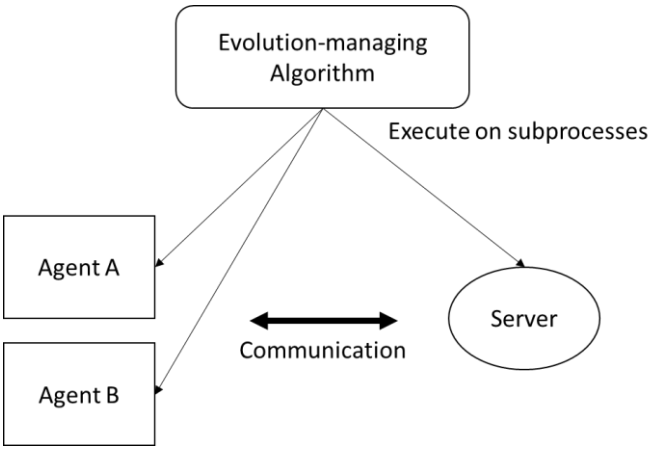

Figure 7 Processes present for the fitness calculation

To finish with the practical side of the implementation, each time a fitness needs to be calculated, the program will start the server with a random world and generation seed and run the two agents that'll face each other. The server and the agents are all ran in subprocesses. The main loop then waits for the game to finish before continuing. To transmit the BT object to the subprocesses, they are serialized and deserialized using the utility functions in the class BTree.py.

F. Selection

The solution that was chosen for the implementation is the Fitness proportional selection, for its simplicity yet still allowing for less fit individual to be selected and participate to the diversity of the population.

In addition to this, some elite selection can be applied (usually before), to be sure that some percentage of the population, among the fittest, is assured to be selected for the next generation. Because we the methods such as Fitness proportional and Ranking selection, the fittest individual might not be selected, this is not the case with the Tournament selection.

An alternative approach for selection could have been a variant of the tournament selection but where individuals are playing against each other and the winner of a best of 3 or 5 games goes on to the next round. The selection finishes when the number of individuals left is equal to the population size.

When the selection is finished, the generation is serialized and saved to keep track of the population evolutions when the program has terminated.

G. Crossover and Mutation

The method used for the implementation of crossover is the one described in the previous part. A part of the population is randomly selected for the crossover with a defined probability, usually quite high to boost the exploration of the universe of possible trees. The parents aren't modified, a pair of offspring is created through the crossover for each pair. The list of offspring is then returned to be later evaluated.

The mutation method used has been described previously. A precision about the implementation is about removing a node. If the node is a composite node, we remove the whole subtree that has this node as a root. Also, when removing the node, if the node is the last children of its parent composite node, we also remove the parent because a composite node can't be a leaf node in the tree. The only exception is if the parent node is root, we cancel the mutation because we don't want to delete the entire tree.

VII. Results

Here are the results of one execution of the genetic algorithm on the Bomberland game with the following main parameters:

- Population size: 20
- Number of runs per fitness: 2
- Base fitness: 500
- Max Generation: 20
- Switch enemy: True
- Switch enemy every n gen: 5

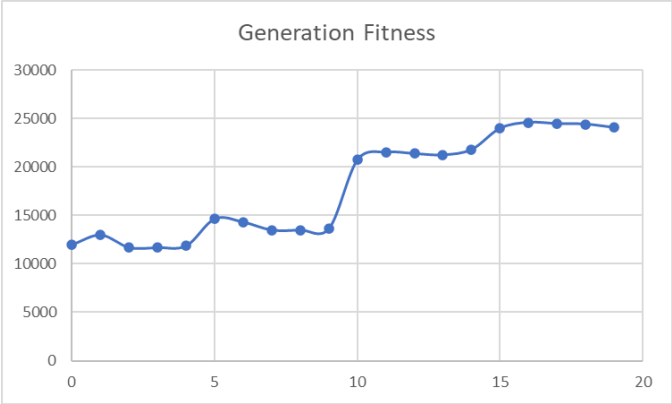

Figure 8 – Generation fitness evolution over the generations

We can first clearly see on the first 10 generations that the generation fitness, sum of all the fitness of the individuals in the generation, is varying a lot, with a jump in fitness every 5 generations, each time we switch of enemy for a more difficult one. The fitness jump at the 10th generation is much bigger than the others and each time we can see than the fitness after the jump quickly stabilizes. It can be due to the fact that the fitness calculation is inconsistent, which is quite probable considering the fact that we only calculate the fitness based on 2 runs where 4 or 5 could assure more consistent results. We can see that the variation between generation if more random for the first generation, because the enemy is doing nothing, and the fitness consist in being lucky but then it starts to be more stable over the generation because the randomness of the map is reduced factor because of fitter individuals. The only thing is that the fitness should slightly improve between generations and not only when switching enemy, so maybe the other parameters related to mutation and crossover are not that good or the used fitness function is not appropriate.

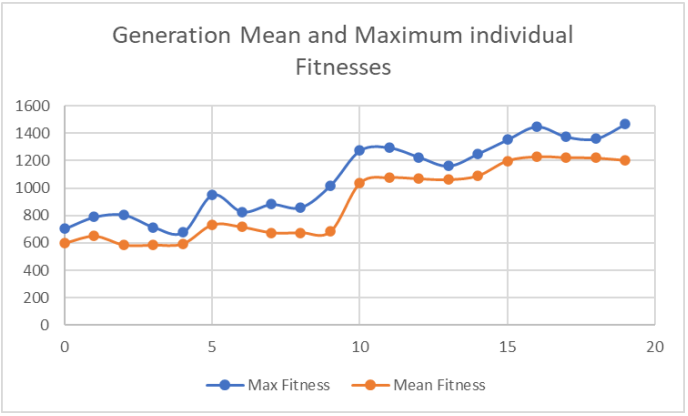

Figure 10 Maximum and mean fitness over the generations

The maximum fitness for each generation seems to vary a lot more than the mean fitness, which can either also point out that the fitness function is inconsistent and not repeatable or that the selection method used is not always keeping the best BT, which seems accurate with what is set up for this execution. An interesting comparison would be with the elite selection being active.

```
BTree(1466.5)
{
    "Selector0": {
        "MoveToSafePlace0": [],
        "EnemyClose1": [],
        "MoveToSafePlace2": []
    }
}
BTree(1362)
{
    "Selector0": {
        "MoveToSafePlace0": [],
        "MoveToSafePlace1": [],
        "MoveToSafePlace2": []
    }
```

Figure 9 Fittest trees of Generation 19

But we can still say that the trees of the last generation are fitter than the one of the first generation, because they almost all contain the node *MoveToSafePlace*, which is the essentially the most useful leaf node in the give alphabet. If we wanted to submit our agent with the fittest BT of the last generation, we would just have to attach it to the gen_agent.py, by importing it and deserializing it.

Finally, if we analyse the two fittest trees more in detail, we can see on the side that there's significant difference in

terms of fitness whereas the trees are not really different because only composed of *MoveToSafePlace* nodes. So, there's definitely a problem of consistency and repeatability concerning the fitness calculation.

VIII. Problems and Limitations

Unfortunately, due to time, program execution optimization and computer resource limitations, I only managed to have few results, but not fit enough to be satisfying. In fact, the time taken each time we want to calculate a fitness is really long due to a needed restart of the server to change the different seeds and a tick rate that need to be kept low unless it will be tick too fast for the behavior tree.

To reach satisfying results, it would be necessary to try different combinations of parameters, selection methods, mutation methods, as suggested in the more theorical parts, until we start to see some tendency of constant progression. The alphabet of nodes would also need some additional leaf nodes such as other conditions, maybe more atomic ones, or other complex actions concerning the powerups such as maybe using A* algorithm to calculate the path needed to reach a certain point of the map. This also raises the question of whether it is better to have only leaf atomic leaf nodes with some parameters such as checking if an enemy is present on the tile below, only higher-level nodes with some more complex algorithmic parts or a mix of both. For example, using only low-level leaf nodes could lead to the emergence of bigger and unthought of comportments, but it will surely take more time.

IX. Conclusion

In this report is described the approach used to find an optimal Behavior Tree for the game Bomberland using Genetic Programming. Different choices and possibilities are explored for each step of the execution of the algorithm: initialization, fitness, selection, crossover, and mutation. The key principle to good results would be the consistency of the fitness function and the ability to give an accurate estimation of a tree's fitness, not varying with randomness. This is the problem encountered in the few results we have: the fitness of similar individuals could be quite different, which biased the evolution of the population. Some axes of progression could also be through the available set of leaf nodes, between conditions and actions, with a lot more diverse and atomic ones. Some intermediate checks could also improve help the population evolve in the correct direction quicker by removing the solutions that we know can't work, for example because they lack some necessary nodes, or with some deeper analysis of the behavior achieved by the tree. The main issue still stays here in the execution time needed to evaluate the fitness of a tree, which is really long. For the few results here we had to sacrifice the fitness consistency for a decent execution time, but for example decent execution parameters could be a population size of 20 to 50 individuals, at least 5 games to evaluate the fitness and around 50 to 100 generations. Given that the execution that has led to the presented results took 7h, it would take around 50h to have a result in the current setup and with the said parameters.

X. References

[1] "Documentation | Coder One." https://www.gocoder.one/docs (accessed Jun. 17, 2022).

[2] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon, "Evolving behaviour trees for the Mario AI competition using grammatical evolution," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6624 LNCS, no. PART 1, pp. 123–132, 2011, doi: 10.1007/978-3-642-20525-5_13.

[3] “Evolutionary Algorithms for Controllers in Games,” 2019.

[4] T. Johansson, T. Johansson, T. Olsson, and J. Källström, “Tactical Simulation in Air-To-Air Combat,” *Master Thesis*, 2018.

[5] M. Colledanchise, R. Parasuraman, and P. Ögren, “Learning of behavior trees for autonomous agents,” *IEEE Trans. Games*, vol. 11, no. 2, pp. 183–189, 2019, doi: 10.1109/TG.2018.2816806.

[6] Z. Michalewicz, “Genetic Algorithms + Data Structures = Evolution Programs,” *Genet. Algorithms + Data Struct. = Evol. Programs*, 1996, doi: 10.1007/978-3-662-03315-9.

[7] A. E. Eiben and J. E. Smith, “Introduction to Evolutionary Computing,” 2015, doi: 10.1007/978-3-662-44874-8.

[8] M. Mernik, “Evolutionary Computations Lectures at FERI.” 2022.

[9] “Py Trees — py_trees 2.1.6 documentation.” https://py-trees.readthedocs.io/en/devel/ (accessed Jun. 17, 2022).