# Evolving Behavior Trees for Ms. Pac-Man

Simon Bechmann and Roman Sahel, *Students at IT University of Copenhagen*

## I. INTRODUCTION

In this paper, we will explain and discuss our method to generate and evolve Behavior Tree (BT) controllers for the player in the game *Ms. Pac-Man*. BT is a method to design complex behaviors, often used in the game industry for non-player-characters, yet they are easy to implement, easy to understand and readable by humans. The problem for the human is to actually design the tree, which tasks should the AI do in which order. Additionally comes testing of the designed tree which also can be time consuming.

Our goal is to solve these two issues in one go by the use of an Evolutionary Algorithm (EA) that generates random BTs and evolve them to perform as desired. This might also lead to another benefit since the stochastic process of EAs could result in non-obvious solutions, that a human would not have thought of.

To test our approach, we decided to use the Ms. Pac-Man framework created by Philipp Rohlfshagen, David Robles and Simon Lucas at the University of Essex, UK, who made this framework in order to hold a competition called "*Ms. Pac-Man vs Ghosts Competition*". The framework includes several AI controllers for the ghosts which our evolved BTs will be tested against.

Other researchers have been doing similar projects like C. Lim et al. [6], who evolved several behavior trees for specific game situations in a real-time strategy game, and combined them in the end to a single AI component. However, where we want to stand out is first of all by only evolving a single behavior tree but also by giving most of the control to the evolutionary algorithm, to see if it still can output well performing and meaningful behavior trees.

## II. APPROACH

This section will first describe what Behavior Trees and Evolutionary Algorithms are in general and end with a summary of other researchers' work that relates to this project in order to get an overview of what have been done before.

### *A. Behavior Trees*

Behavior Trees have become a widely used method for AI controllers in the game industry since its popularization by *Halo 2* in 2004 [8] and has since spread to robotics as well [5]. BTs has a hierarchical tree structure and network of nodes, where some nodes control how the tree is going to be traversed, until it gets to a leaf node which can be an action or a condition check that perhaps will lead to an action. The power of BTs is that they are efficient and modular, meaning they can easily be modified even if the tree is complex. This stands in contrast with finite state machines which can quickly seem overwhelming if they get too complex [2].

BTs are made of four different types of nodes: conditions, actions, composites and decorators. Conditions and actions are leaf node where conditions check if a specific game state meet some sort of requirement, where an action node performs a certain output from the AI, for example movement. Composites control the flow of the tree, and the two most common ones are called selectors and sequencers. The children of composite nodes can be either conditions, actions or other composite nodes. All nodes in a tree will return a success or a fail and this will determine how the flow is run through the tree. A selector will run through all its children node until one of them is a success and thereby succeed. It will only fail if all its children fails. Meanwhile, a sequencer will stop and fail if just one of its children fails. Decorators only have a single child and their function is to transform the output of the child by inverting it for example [2].

For our project, we designed the possible actions and condition beforehand and let the algorithm put them together in a meaningful way in a BT. The actions and conditions are shown in section 4.

### *B. Evolutionary Algorithms*

Evolutionary Algorithms are, as the name suggests, inspired by biological evolution including terms like genomes, populations, reproduction and mutations. These algorithms concern optimization and can be used in various domains. The first thing to do when making an EA is to encode the optimizable solution into a genome consisting of chromosomes. In our project, Ms. Pac-Man represents the genome and the BT represents the chromosomes. Following, an initial population of genomes is generated with random chromosomes and are evaluated through a fitness function. The following generations are then based on how it is decided to reproduce, mutate and perhaps let some genomes survive [1].

Our exact setup is described in section 4 meanwhile we will be using the score of Ms. Pac-Man as our fitness function. This means the BTs with the highest scores will be used to generate the new generations, and thus in the end, hopefully create a well performing BT.

### C. Related Work

M. Colledanchise et al. [3] used the Mario AI benchmark to validate their framework, where they aimed to evolve model-free BTs meaning they were self-learning in terms of how and when to perform actions. They started with BTs consisting of a single node and then gradually evolved them to be more complex depending of their fitness. Their genetic algorithm created behavior trees without size restrictions and then trimmed the BTs afterwards, when they could complete the goal.

Their goal of making their framework model-free is out of scope for this project, but their method of starting with simple trees and slowly making them more complex is an interesting approach, which we also considered, but deemed unnecessary since we have a limited set of actions and conditions. Furthermore, our goal is to see if the EA by itself can output meaningful trees, and this seems like a restriction to its stochastic process.

R. Pereira & P. Engel [4] aimed to implement Reinforcement Learning nodes in BTs in order to add learning capabilities in an otherwise constrained method, but also the same time tried to avoid that it learned bad behaviors. This framework allowed a user to manually design a BT, and to specify where learning can be applied.

This approach is quite different from ours, since they keep the human as the designer which our goal is to avoid, but their goal of making BTs more adaptive, is something this project could look into in the future, so the BTs gets evolved further, if something changes in the game environment for example.

C. Lim et al. [6] used evolutionary algorithms to make a competitive AI that could outperform the original bot in the real-time strategy game *DEFCON*. They evolved randomly generated behavior trees for several specific game situations with different fitness functions and combined them in the end to a single AI component. An issue they might encountered was an over-fitted AI since they only tested against one type of AI.

This approach is similar to what we want to achieve, but we decided not to try to evolve several BTs and combine them, but simply just evolve a single one, since the gameplay of Ms. Pac-Man is quite simple compared to a real-time strategy game. Furthermore, the set of actions and conditions we have is also limited.

D. Perez et al. [7] used the Mario AI benchmark to test their framework, where they used grammatical evolution to create a phenotypic program, syntactically correct for the problem domain.

They made use of grammatical evolution to ensure that the BTs got evolved in a meaningful way, however we decided to let evolution handle the situation as an initial approach, to see if that could lead to meaningful BTs.

K. Scheper et al [5] used a flying micro-robot with limited sensory capabilities and computational power, which therefore generally is difficult to design complex behaviors for, to test if BTs combined with an EA could achieve that. For selection in their EA they preferred smaller BTs if they had the same fitness with larger BTs. They tested their procedure in a simulated environment and the optimized BT proved to have a higher success rate and a smaller tree than the human designed BT, but was slower to succeed with the task. The compared results was similar in a real environment.

Their approach is very similar to ours but their circumstances and goals are quite different since they are working with a micro-robot, but their decision to make the algorithm prefer smaller BTs is quite clever, and perhaps was something we could have implemented as well. However, as mentioned before, we do not want to restrict the EA too much.

## III. GAME MECHANICS

The gameplay of *Ms. Pac-Man* is essentially the same as the original arcade game *Pac-Man*, where the user moves Ms. Pac-Man around in a maze and tries to consume all the pills in the level in order to proceed to the next level. In *Ms. Pac-Man* there are four different mazes/levels compared to only one in *Pac-Man* together with some other changes as well [9]. While consuming pills, the user must avoid the ghosts or consume a special powerpill, which is placed in the four corners of the level, which enables the user to eat the ghosts. The overall goal for the user is to get the highest score as possible and the game is over when the user has no more lives left, where one is lost if the ghosts touches Ms. Pac-Man.

As mentioned earlier, the framework used for the project consists of several AI controllers for the ghosts with different skill levels, and our evolved controllers will be tested against these. These kind of frameworks are very useful when experimenting with AI methods.

## IV. METHODS

### A. Implementing the Behavior Tree Framework

In order to be able to generate and evolve Behavior Trees, we needed to implement a clean and complete structure of the tree and the different node types. The final structure contains only what was

deemed needed within the range of the Pac-Man AI so that it remains very simple, as seen below:

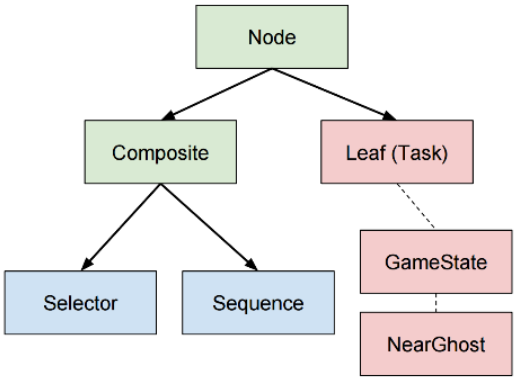



Fig. 1. Class diagram of the behavior tree framework

The Behavior Tree is thus composed of Node objects, which can either be a Leaf or a Composite. The only requirement is for the nodes to implement a DoAction method, which returns a boolean. A Leaf is a node that executes a Task, which has a direct action: it can either test a condition or modify variables (including the move to return, for example). A Composite is a node that does not do anything by itself but that executes its children under some conditions and in a certain order. Here, the only composite node that were implemented were the Selector and the Sequence.

A GameState class is also used analyze the state of the game at each step and keep the important information such as the nearest pill, the nearest power pill, and a list of the ghosts, sorted by distance. All these characteristics are used by the tasks of the Behavior Tree.

*B. Generating Random Trees*

The method we chose to generate trees is to randomly organize nodes to create a complex, and hopefully smart structure after being evolved. The first step is thus to design and create the palette of different nodes the generator will use to build a tree. Compared to M. Colledanchise et al. [3], this means that we generate complete trees, meanwhile they, as an example, start with minimalistic trees and make them more complex over time. The reason why we want to generate complete random trees to begin with is to see if the evolutionary algorithm can output meaningful and well performing trees in the end, without us restricting it in any sense.

Those nodes are the composite nodes, Sequence and Selector, and various tasks. The tasks can be divided into two categories: conditions, which check if the game is in a certain situation, and actions that decides the move the player will do. Following is a description of the actions and conditions implemented.

Actions:

- *Avoiding powerpill*. Ideally, it should be used when ghosts are far away.
- *Chase powerpill*. Ideally, it should be used when ghosts are close so they can be eaten.
- *Chase ghost*. This should be used when a powerpill has been consumed.
- *Eat pill*. This action moves Ms. Pac-Man to the nearest pill.
- *Go to junction*. This action moves Ms. Pac-Man to a specific junction in the maze.
- *Run away from multiple ghosts*. This action moves Ms. Pac-Man in that direction, that will move away from most ghosts.
- *Run away from ghost*. This action moves Ms. Pac-Man away from the nearest ghost.

Conditions:

- *Are ghosts far away*. Check if the distance between Ms. Pac-Man and each ghost is more than a threshold.
- *Is ghost closer*. Returns true if a ghost is closer than the nearest pill and powerpill.
- *Is ghost edible*. Check if there is an edible ghost.
- *Is ghost near*. Check if the distance between Ms. Pac-Man and each ghost is less than a threshold.
- *Is Ms. Pac-Man at junction*. Check if Ms. Pac-Man current position is at a junction.
- *Is path to junction safe*. It checks if one of the three nearest junctions is safe to go to.
- *Is path to powerpill safe*. Check if the path to the nearest powerpill is safe.

As can be seen from the actions and conditions, they are actually designed to be put together in a certain way to make sense, so our evolutionary algorithm will probably create many un-meaningful trees in the beginning, as they are put together randomly. However, it is also interesting to see if the evolutionary algorithm put together some well performing behavior trees, we as human designer would not have thought of. Once all the tasks were implemented, we had all the pieces of the puzzle to generate a tree and we just needed an algorithm to put it all together in a way that make sense.
The algorithm goes as follow:

- It randomly selects whether to create a leaf or a composite node.
- If it is a leaf, there is one rule: if the new node is the last child of the parent node, then it should be an action. Otherwise, it should be a condition. In our situation, the action nodes we designed cannot fail and will always return a move direction. Therefore is the logic of our selectors and sequencers that only the last child

should be an action, as other tasks after an action would not be reached.

- If it is a composite node, we also randomly select whether to create a sequence or a selector. Then, we randomly initialize its parameters: the maximum depth of its subtree and the number of children it will have. These properties are bounded by a fixed value in order to limit the growth of the tree. Finally, we recursively do the same thing for each of its children.

**Algorithm 1** Generate a random Behavor Tree

```
1:  procedure GENERATETREE(parent)
2:      nodeType ← randomly select from [Leaf, Composite]
3:      if nodeType == Leaf then
4:          if parent.childCount == parent.maxChildCount then
              return random action leaf
5:          else
              return random condition leaf
6:          end if
7:      else
8:          newNode ← randomly create a Sequence or a Selector
9:          newNode.maximumDepth ← random between 0 and MAX_DEPTH
10:         newNode.maxChildCount ← random between 0 and MAX_CHILDREN
11:         while newNode.childCount < newNode.maxChildCount do
12:             newNode.children.add(GenerateTree(newNode))
13:         end while
14:     end if
15: end procedure
```

Fig. 2. Overview of the algorithm to generate random behavior trees.

### *C. Evolving the Trees*

The generated trees we obtain are coherent but probably have no logic as the actions and conditions are linked together randomly. Thus, the goal of the evolutionary algorithm is to evolve from a random population to a population as fit as possible. The algorithm then goes as follow:

1. We generate a population of *n* Behavior Tree based Pac-Man controllers.
2. Each individual runs the game *m* times using their unique Behavior Tree so that we can get a representative average score for each of them.
3. We use the average score of each individual to sort the list of the individuals from the highest score to the lowest: we then *kill* the worse half, meaning that we remove the second half of the population, composed of the individuals that got the lowest scores.
4. The other half, that we keep, is thus composed of individual that got higher scores.
   We combine two-by-two those best individuals, that will serve as parents, to create offspring. We use two different types of combination: leaf combination and composite combination. They both follow a similar technique (see a visual result in Annex 1):
   a. We randomly choose a node in the first tree
   b. We randomly choose a node in the other tree (of the same type if we are combining leaves)
   c. We create two offspring: one which is the first tree where the first node (and its subtree) has been replaced by the second node (and its subtree), and vice-versa.

   The goal of this combination is to hopefully create trees that have the best parts of the best trees. We also considered using parents in a random order but decided it would only make the implementation harder and would probably have no impact on the result since the combination are random anyway.
5. We add mutated parents to the population. There are different types of mutations:
   a. Leaf removal: we select a random leave that is not an action and remove it from the tree.
   b. Task switching, we select a random leaf and we change its task, from one condition to another or from one action to another ;
   c. Composite switching, we select a random composite node and change its type ;
   d. Complex subtree regeneration: we select a random composite node and remove it and all of its subtree. We then randomly regenerate a node, which can then either be a leaf or a whole new subtree.

   The different mutations can be divided into two categories: a. and b. are the simple mutation while c. and d. are the complex ones.
6. We add a random number of completely new individuals to have more variations in the population. This was not planned originally but after multiple tests where the population was stagnating really quickly, we looked for new ways to bring more diversity. This appeared to be a good solution and gave good results (see in the next section for more). The number of new individual is generated between 0 and 15. These values have been arbitrarily chosen and were found to be a good balance between adding enough new parts without adding too many individuals.
7. We repeat from step 2 for *k* iterations.

The parameters we mention (*n*, the population size; *m*, the number of trials; *k*, the number of iterations) are not constant and can differ from one test to another. For the final tests, we used a population of 100 and 25 trials per individual for 100 iterations, as those values gave a good result.

## V. RESULTS

### *A. Performance*

The tree generator algorithm has been optimized to be as fast as possible while remaining flexible and easy to maintain (with lists to directly keep track of leaves and composite nodes for example), in order to speed up the tree search and the rest of the algorithm.

As an example, generating one million trees of large depth -- which is a much larger space than what we use for the actual algorithm -- takes about one second (from 0.7 to 1.2 second).

The evolutionary algorithm is obviously what takes most of the processing time but the performance greatly depends on the number of trials per generation and the size of the population: the higher those numbers are, the longer the training will take.

To reduce training time, we used Java's ExecutorService and Future objects to experiment multiple pacman controllers simultaneously and take advantage of multithread processors. This has drastically improved the computing time: going from about 6 minutes to about 24 minutes for 50 iterations, with an initial population of 100 individual and 25 trials per generation (as mentioned before these parameters values are the one that we use of the final tests).

### *B. Scores and Evolution*

First, the result we had during the development of the project influenced how the final algorithm work. We indeed had diversity problems where the initial population would block the evolution because of the lack of better alternatives for each generation: that is when we decided to add more types of mutations (there was only task switching and leaf removal at the time) and new random individuals. As we can see in Figure 3, the best score for each generation stops evolving around the 15th iterations, which is early: a local maximum is reached.

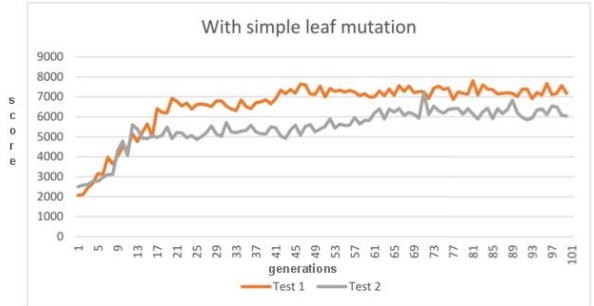



Fig. 3. The results obtained if the algorithm features only simple mutations (leaf removal and task switching)

As we can see in the figures 4 and 5, adding new types of mutations helped with this issue and population becomes stable at a later iteration and with better scores. Although, it helped, this is still an issue and the scores we get from one evolved population to another greatly depends on the length of the training as much as on the initial population.

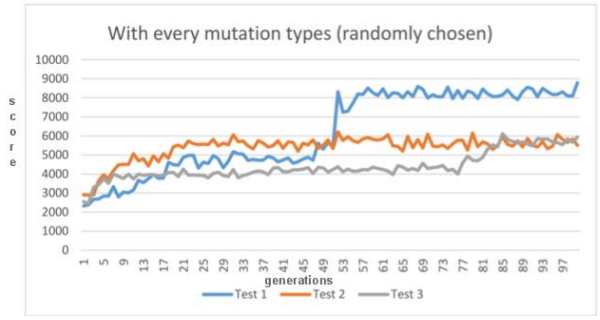



Fig. 4. The results obtained if the algorithm features every mutations we described, but only one is randomly chosen for each individual

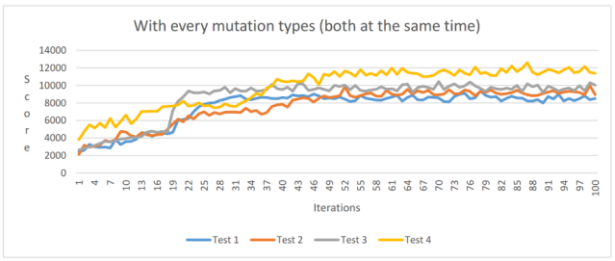



Fig. 5. The results obtained if the algorithm features every mutations we described and all the mutations are applied for each individual

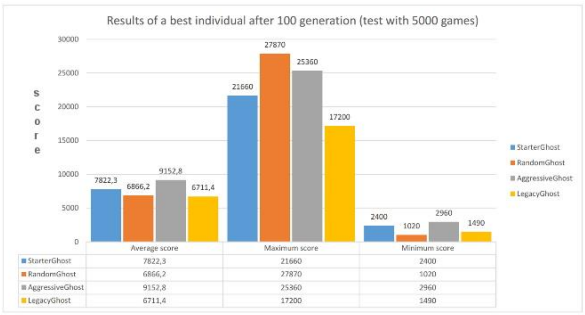



Fig. 6. Results from the best individual after 100 generations, trained with StarterGhost

In Figure 6 are the scores obtained with one best individual from a population that have been evolved for 100 generations. It is worth noting that the evolutionary algorithm is trained with a StarterGhost controller and that training with a different controller gives different results. For the final tests, the StarterGhost controller was chosen because it gave the best results compared to the other controllers (as we can see in Figure 7, which was trained with an AggressiveGhost controller). The reason for that might be that the StarterGhost behavior is the most balanced so that it works good for all the controllers whereas the AgressiveGhost behavior is really specific and the Behavior Tree generated will not be adapted to any other behavior.

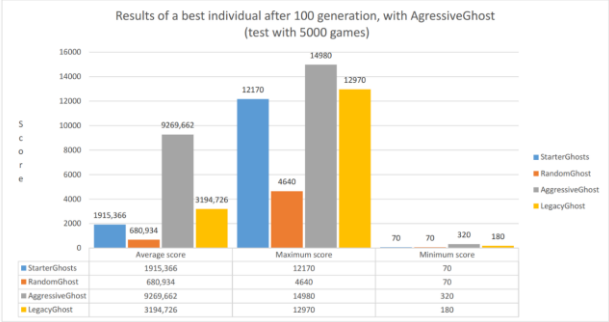



Fig. 7. Results from the best individual after 100 generations, trained with AgressiveGhost

In the figure below is the evolved behavior tree after 100 generations.

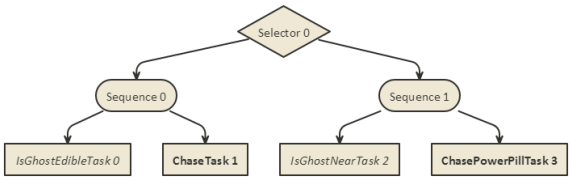



Fig. 8. Evolved behavior tree of the best individual.

## VI. DISCUSSION

Our goal for the project was to see if an evolutionary algorithm with limited restrictions could produce well performing and meaningful behavior trees, and the results were positive, however not perfect. The evolved trees could achieve decent scores but not as good as a humanly designed tree. The design of the well performing trees did not always make perfect sense, compared to the intentions of the actions and conditions.

However, it was also interesting to see how the evolutionary algorithm could come with alternative solutions that performs surprisingly well. For example the behavior tree in Figure 7, which is the tree with highest fitness after 100 generations, is performing quite well compared to the actual logic if the tree. It consists of two sequences with a single condition and action in each, meaning if both the conditions return false, Ms. Pac-Man will not do anything (more correctly, it will move in the same direction as before and eventually stand still at a wall). The second sequencer will be activated if a ghost is near, which will make Ms. Pac-Man move towards a powerpill. If she consumes the powerpill, the first sequencer will make her chase the ghosts.

The actions and conditions were designed by us in a single context, meaning it is difficult to say if these results could be copied to other games or domains. Also, in relation to the actions and conditions, we preferably should have designed more of them so the algorithm could produce even more varied behavior trees. Furthermore, the actions and conditions that we did make, were perhaps intended too much to be linked in a certain way.

It could also be interesting to compare with an other research on evolving behavior trees for Ms. Pac-Man, to see if more restrictions and control over the evolutionary algorithm could lead to better results. However, this would make most sense for ourselves to do that, so the possible actions and conditions stays the same.

## REFERENCES

[1] A. E. Eiben & J.E. Smith, “What is an Evolutionary Algorithm?” in *Introduction to Evolutionary Computing*, 2nd ed. Springer, 2007.

[2] C. Simpson, 'Gamasutra: Chris Simpson's Blog - Behavior trees for AI: How they work',*Gamasutra.com*, 2015. [Online]. Available: http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php. [Accessed: 03- Dec- 2015].

[3] M. Colledanchise et al., “Learning of Behavior Trees for Autonomous Agents,”*arXiv:1504.05811v1 [cs.RO]*, April 2015.

[4] R. Pereira & P. Engel , “A Framework for Constrained and Adaptive Behavior-Based Agents ,”*arXiv:1506.02312v1 [cs.AI]*, June 2015.

[5] K. Scheper et al., “Behaviour Trees for Evolutionary Robotics,”*arXiv:1411.7267v2 [cs.RO]*, August 2015.

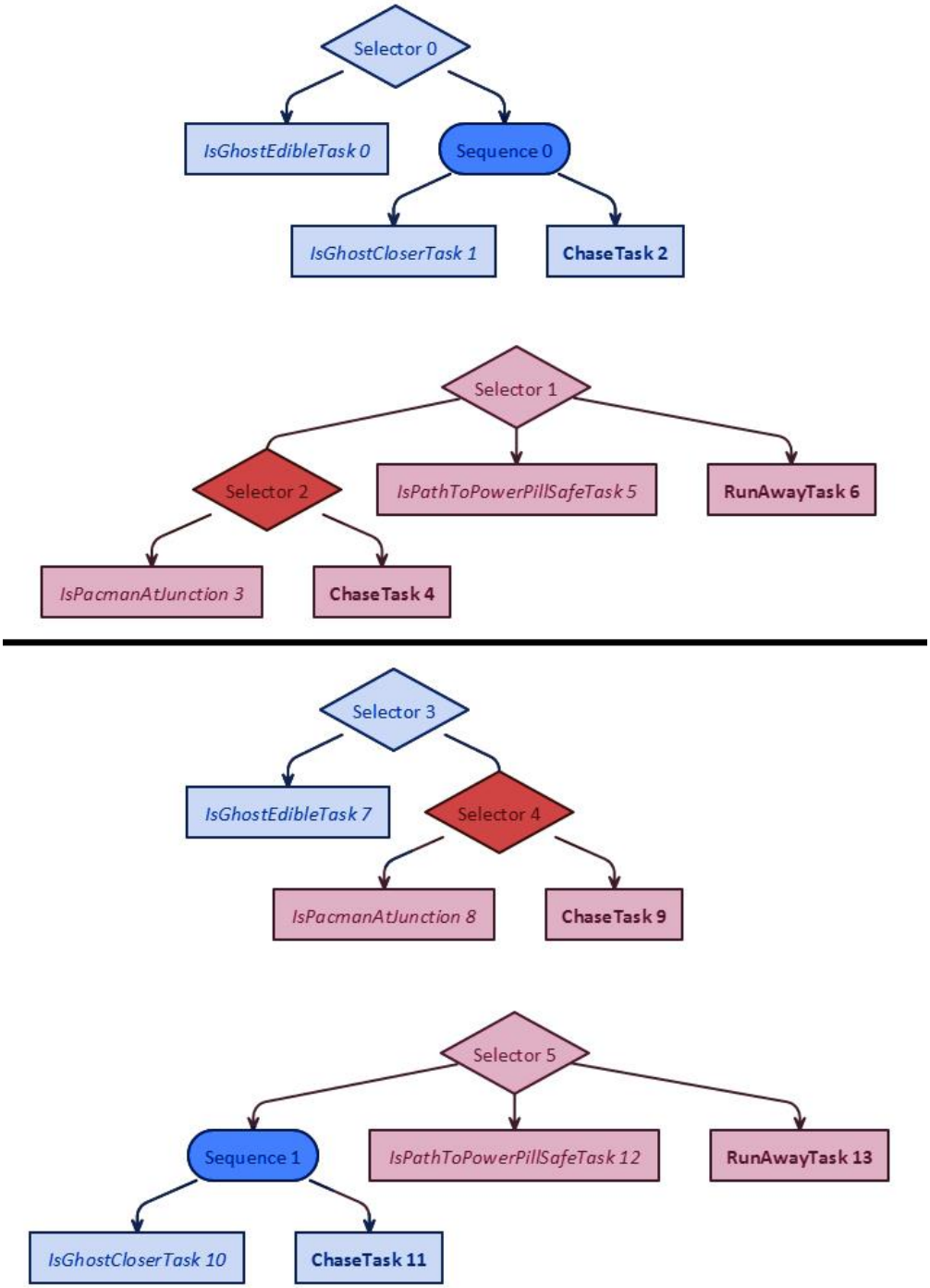
[6] C. Lim et al., “Evolving Behaviour Trees for the Commercial Game DEFCON,” pp 100-110 in Volume 6024 of the series Lecture Notes in Computer Science, 2010.

[7] D. Perez et al., “Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution,” pp 123-132 in Volume 6624 of the series Lecture Notes in Computer Science, 2011.

[8] S. Risi, 'Lecture 2: Common “Non-modern” AI Techniques', IT University of Copenhagen, 2015.

[9] Arcade-museum.com, 'Ms. Pac-Man - Videogame by Midway Manufacturing Co.', 2015. [Online]. Available: http://www.arcade-museum.com/game_detail.php?game_id=8782. [Accessed: 03- Dec- 2015].

# ANNEX 1




This is a visual example of composite combination in the algorithm. The upper part consists of the behavior trees of the two parents, one in blue and one in red. The two composites chosen for combination is shown by a darker color. The lower part consists of the two children created by switching the two composites.