1/19/2018

DAC619 - ASSIGNMENT

KANAGA MANIKANDAN SOLAIKANNAN – Q12389501

southampton solent university

**CONTENTS**

[Introduction: 1](#_Toc503792784)

[AI Algorithm: 1](#_Toc503792785)

[Advantages and Disadvantages: 2](#_Toc503792786)

[Advantages: 2](#_Toc503792787)

[Disadvantages: 2](#_Toc503792788)

[Comparison with other Algorithms: 3](#_Toc503792789)

# Introduction:

This is a report on the artificial intelligence implemented in a level created and developed in Unity Engine. The implementation of which AI algorithm, its justification along with its advantages and disadvantages, and how it was designed will be mentioned in this report.

# AI Algorithm:

The algorithm used for this level is the Decision Tree algorithm. The tree consists a branch of decision nodes and ends at action nodes. The AI is designed in a way the entity would focus on an object or another entity which is closest to it and decide and act upon it. A couple of decisions and actions that will be performed by the AI entity are listed below.

1. If the entity can’t see another entity, then randomly wander the level.
2. If a power attack pickup is located, the entity will move towards it if it hadn’t picked it up already.
3. If an entity can see another entity, it will move towards it and start attacking once they’re close enough to each other.
4. The entity will not engage its opponent, if its attack power is lower than that of the opponent’s. Instead, it will flee from combat
5. If the entity’s health drops below 25%, it will flee from combat and would start looking for a health kit and then, move towards it if it located one.
6. If the opponent is fleeing, the entity will continue to randomly wander the level

From the behaviours mentioned above, the AI entity decides and behaves accordingly. For some actions, it will have to perform a series of decisions before concluding to an action like for example, engaging the opponent. These decisions can be performed using the ‘if’ keyword in the code but they will be nested, and it will not be very efficient. To overcome this issue, the decision tree algorithm was implemented. The traversal of the tree will start from the root node and make its way down to an action node. The decision nodes will perform the ‘if’ checks and return either true or false. We can then, add two child nodes to this decision node, one of which will be used based on the result received from the decision. If the decision is positive, we will traverse the tree through the true child. With the help of this algorithm, if any changes/updates in the future had to occur, we can easily add new decisions and actions and rework the tree accordingly. This level of comfort wouldn’t be achieved with nested ‘if’ statements. It would be very hard to debug and would make the code untidy and unreadable.

An action node can either contain a single action or a sequence of actions. Sequential actions allow the entity to perform consecutive actions. The entity can perform the next action in the sequence only if its preceding action is completed. A delay can also be added between the execution of the actions.

# Advantages and Disadvantages:

## Advantages:

One of the advantages of this algorithm is that it’s simple to implement and understand. It uses a simple logic for performing various functions. In other words, it can be said that it’s a formalisation of nested ifs. It is also convenient to make changes to the tree when needed and wouldn’t disrupt the code structure.

## Disadvantages:

The major disadvantage of this algorithm is that it’s only applicable for games of low scope. If implemented for a complex situation, the tree would be enormously large and cause performance issues as the tree traversal would have to take place every frame. It would also start to get very hard to debug and identify bugs or errors. The other disadvantage is that it also requires a proper design which doesn’t accidentally trigger an infinite loop or allow back-tracking.

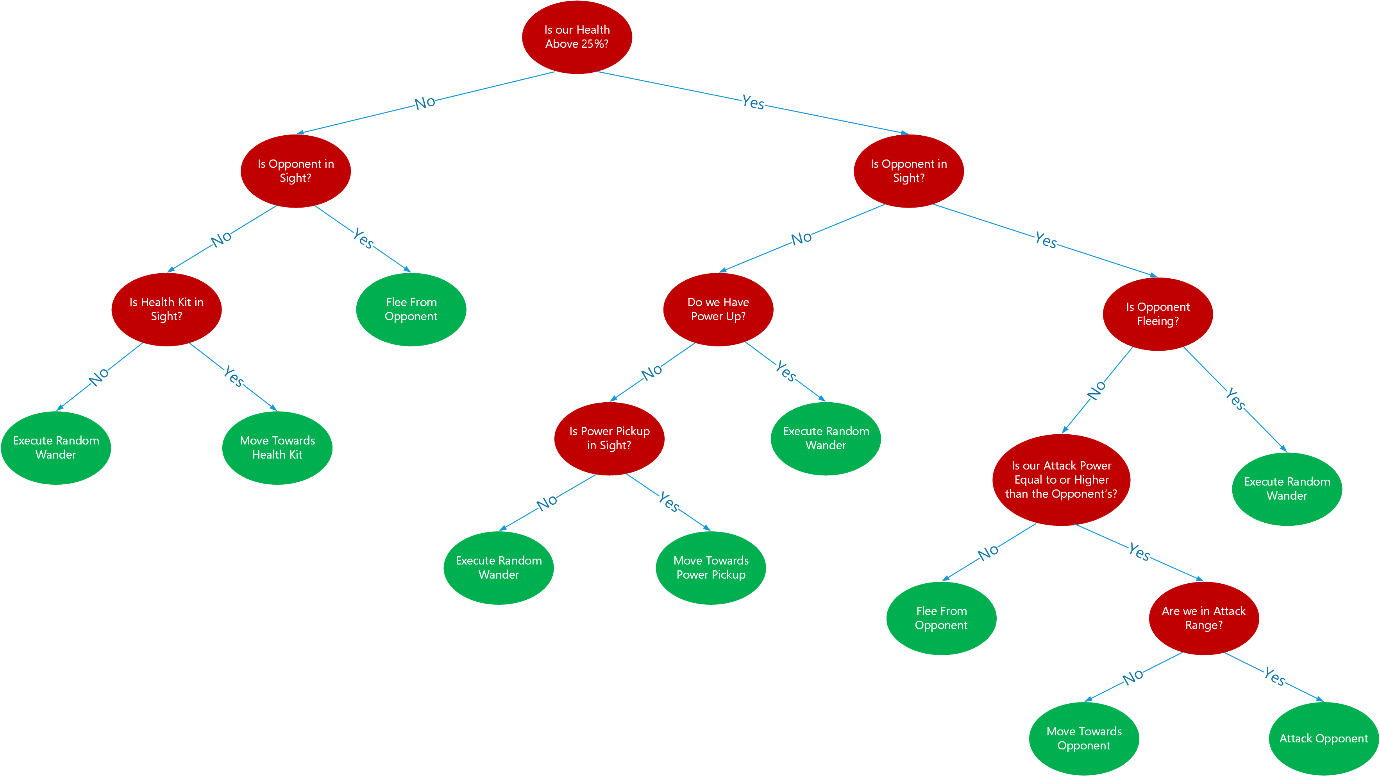
# Comparison with other Algorithms:

The decision tree algorithm will now be compared with 2 other algorithms. One of them is the state machine algorithm. The state machine algorithm allows the AI entity to be in various states and switch between states when needed to. It’s much simpler to implement than the decision tree but why this wasn’t chosen is because many states will have to implemented for this game level and each state will have to contain an if check or maybe nested ifs to enter another state from the current state. This will unnecessarily complicate the problem and will be difficult to make changes in the future. This algorithm can be used for much a simpler level.

The other algorithm is the neural network. Neural network is a very complex algorithm and it’s not required for this level. It uses high-level mathematics for calculations and processes which isn’t necessary. Since, the game level is slightly more complex for state machine and very simple for neural network, decision tree algorithm is more suitable and apt.

# Decision Tree Design:

The tree was designed based on the behaviours mentioned above. Decisions were prioritised based on how important they were. For example, it was necessary to perform a health check first and then check if another entity is in sight. The decision tree design can be seen below.



As you can see that when the entity’s health it low, it only checks if the opponent or a health kit is in sight and not anything else. By this way, we can avoid less important decisions and arrive at an action node as quickly as possible to avoid performance hiccups.

In the middle, we can see that we first check if the power up has been picked up and if not, we look for one. If we switch the order, it would still be acceptable but just like the previous case, we try to avoid checks where it’s possible.

At last on the right, we make a few decisions before attacking. We first check if the opponent is fleeing. If it does, then there is no need for further checks and continue to randomly wander the level. If not, then we perform the attack power check and if that’s positive, we move towards the enemy and attack when close.

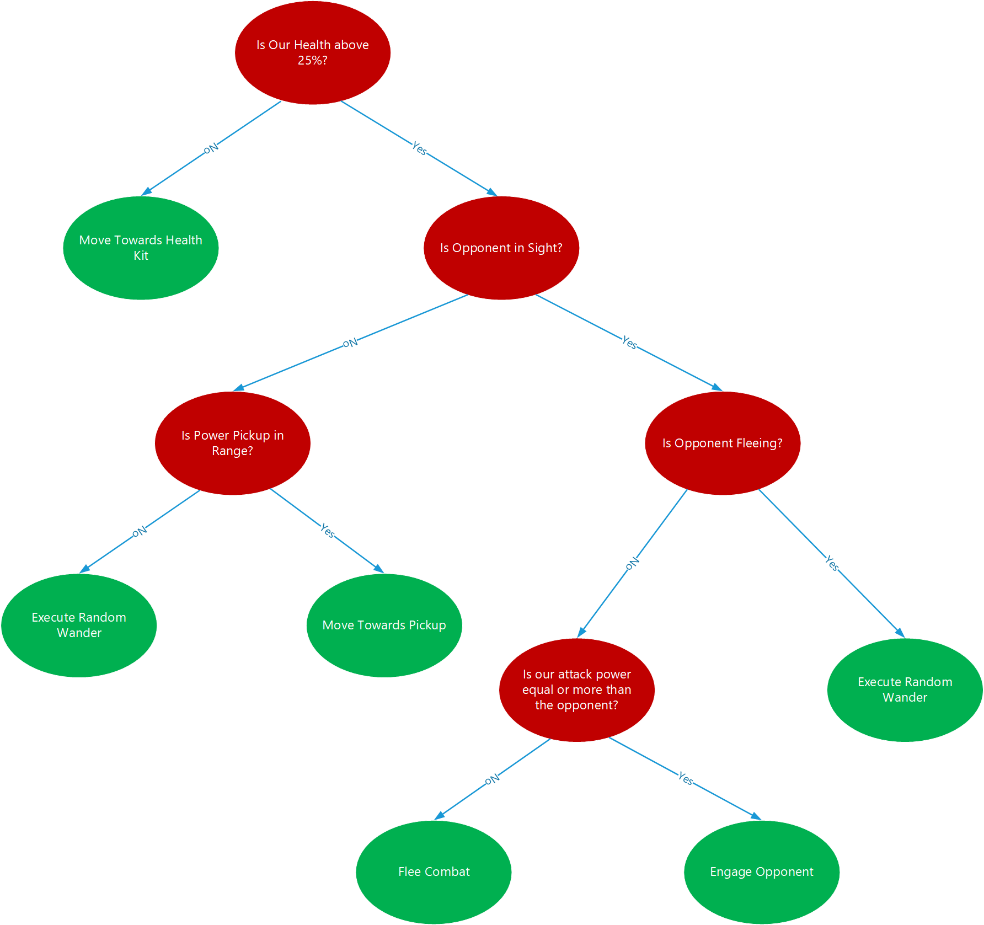
# Testing:

A series of tests were taken during this project and the results can be seen below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Task No. | Task | Expected Result | Actual Result | Status (Pass/Fail) | Notes |
| 1 | Decision Tree Traversal | The tree should traverse correctly and arrive at an action node | The tree traversed correctly and performed the correct action | Pass | - |
| 2 | Random Wander | The entities should randomly wander when there is no one in sight | The entities randomly wandered when required | Pass | - |
| 3 | Move Towards Pickup | The entity should move towards pickup when located | The entity only picks up when it’s too close | Fail | The entity would only pick up when it’s in pickup range which was very short. This was fixed by using the perceived objects list to locate a power pickup and making the entity move towards it. |
| 4 | Move Towards Health Kit | The entity should move towards the health kit when located | The entity moves directly towards it even if it’s not in sight | Fail | The logic was implemented incorrectly by directly calling the move function. This was fixed by using the perceived objects list and making the entity move towards it when in sight. |
| 5. | Move Towards Agent | The entity should move towards the agent when in sight | The entity only moves toward it when they’re too close | Fail | A range was used to make them move towards each other and it was very short. This was also fixed by using the perceived objects list and move when in sight. |
| 6 | Attack Agent | The entity should attack the other agent when close | The entity attacks but health gets reduced very quickly | Fail | The attack function was called every frame which made the health deplete very fast. This was fixed by adding a delay before the action gets executed. |
| 7 | Flee Combat | The entity should flee if its attack power is lower than the enemy or when low in health | The entity flees when needed to | Pass | - |
| 8 | Resume Random Wander | The entity should resume random wander if its opponent is fleeing | The entity resumes random wander correctly | Pass | - |
| 9 | Multiple Objects | The entity should interact with the closest object/entity if more are in sight | The entity detects the closest object/entity properly and behaves appropriately | Pass | - |

# Design Iterations:

The presented level had only one of each from the entity’s point of view. There was another entity, one power pickup and one health kit. Implementing AI logic just for these objects specifically didn’t sound very efficient. So, the AI script was modified to have a knowledge of multiple entities and collectibles. Initially, it was designed in a way that the script would add all the game objects of a specific tag to respective lists and loop through each list and find out which is the closest one. It was later realised that this implementation is very expensive as there could be more than 20 or 30 and it would loop through all. Not only the code was expensive but also illogical because the entity would be aware of a health kit’s position that may be placed outside the level. The decision tree looked like the one below for the logic mentioned before.



It was then realised that the entities had a sensing script attached to them which would add game objects into a dictionary which are in sight and then remove them when out of sight. But unfortunately, only the names of the game objects in the dictionary can be used which wasn’t applicable for the design. So, a new list was created, and this was used to add and remove game objects appropriately. Instead of looping through every game object in the scene, it was improved to loop through the objects that are in sight and find the closest one. This improved performance and was also more logical.

# Self-Evaluation:

The presented level was a good stepping stone for me to implement a simple AI algorithm. The AI agents only do a couple of actions such as moving and attacking depending on the situation. An algorithm which was easy to implement, easily understandable, and apt for this game level was the decision tree algorithm. Having the algorithm chosen, a draft design had to be created to get started. With some base design, the code for the decision tree was implemented along with the action classes and performed my first test. It was successful and was a good start for the rest of my project. The decision tree went through so many iterations because a decision or an action was forgotten, and the tree had to be changed appropriately. Only a small number of problems were encountered which weren’t too severe and was also completed in a short span of time.

Improvements to this project can be done be allocating a cone vision to the agents for extra precision in locating game objects. Implementing a patrol route to the agents can also help the agents to move around the level more accurately rather than moving back and forth at times. It can also be improved by adding a recharge timer to the health kits so that they’re not active all the time. As of now, if the agents fight near health kit, they would automatically pick up the health kit and fight indefinitely.