|  |
| --- |
|  |
| AI for Games |
|  |
| Elliot Robinson (Q5130385), Gabriel Menezes (S6088315), Jonathan Archer (T7125507)  05-01-2021 |

Contents

[Introduction 1](#_Toc70769693)

[Results and Evaluation 2](#_Toc70769694)

[Flocking 2](#_Toc70769695)

[Decision Making 4](#_Toc70769696)

[Pathfinding 7](#_Toc70769697)

[Conclusion 8](#_Toc70769698)

[References 9](#_Toc70769699)

# Introduction

This report focuses on the implementation of several AI algorithms into a game based on the multiplayer online battle arena (MOBA) genre. The algorithms include:

* Decision making
* Flocking
* Path finding

The game is influenced by popular games such as League of Legends and DOTA 2 and uses a map with a very similar layout consisting of bases, a jungle, a series of towers and 3 lanes.

We discussed several different game types including real time strategy and role-playing games but felt these styles of gameplay may be too time consuming. Ultimately, we settled on the MOBA due to our team being more familiar with this style of game.

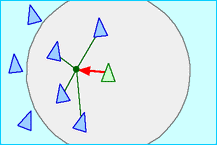
The AI algorithms were to be used on different areas of the game to allow each one to be demonstrated individually. The decision making is used on minions and champions to control where they are going and what to target. Flocking is being used on jungle minions allowing them to roam freely and group up in the jungle and the path finding is used to replace the NavMesh for the minions and champions.

# Results and Evaluation

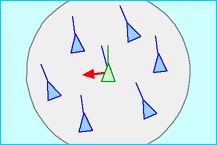
## Flocking

Flocking algorithms use a set of behaviours that can be applied to a set of agents. These behaviours make the agents appear to move as a sensible and coordinated group. The group can exhibit patterns which make it look like there is some form of flocking manager deciding where the agents go, however, each agent is making its own decisions based on the position of its neighbours.

The flocking behaviours should apply a movement to an agent based on how many neighbours are within a given radius of it and the direction in which they are moving. If an agent has no neighbours, it should continue to move in a forward’s direction. For each agent, we iterate over its neighbours and calculate its destination by combining the results from three core behaviours.

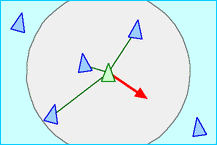


*Figure 1. Cohesion (Reynolds, n.d.)*

The cohesion behaviour *(Fig. 1)* tries to keep agents within the flock by finding the average position of the neighbours and moving towards it. This behaviour can sometimes cause the movement to look erratic and may need some form of steering or smoothing incorporated into it.  
  


*Figure 2. Alignment (Reynolds, n.d.)*

The alignment behaviour *(Fig. 2)* tries to get the agents to move in a common direction by finding the average heading of all its neighbours and moving in that direction.



*Figure 3. Avoidance (Reynolds, n.d.)*

The avoidance behaviour *(Fig. 3)* helps to prevent collisions and overlapping. Agents will determine if neighbours are too close and attempt to move away from them. The distance should be a radius smaller than that used to find nearby neighbours otherwise the other behaviours may not work.

Implementing the core behaviours works well in scenes with no obstacles, but in scenes where there are many, there can be issues with agents becoming stuck or not handling obstacles correctly. Our flocking was originally set up to only consider other agents and we needed a way to avoid objects in their path and steer away from them where possible. We managed to achieve this by using Unity’s NavMesh. Our function uses the agents position to find the closest point on the closest edge of the NavMesh and then calculates a distance between the two points. When the agent gets within a set distance, we can create a new movement vector that will make the agent avoid the edges. This works well in most cases as the NavMesh wraps around every obstacle. An alternative way we could have achieved this would have been to allow each agent to get a list of every nearby object. This list could then be filtered to get the obstacles and a movement vector calculated to move the agent away.

To help fine tune how our flocks work we implemented a composite behaviour class which allows us to apply a weighting to the individual core behaviours. This weighting allows us to increase or decrease how much of an impact each behaviour has on the agent’s movement.

We encountered the issue of erratic movement caused by the cohesion behaviour and have updated our class to gradually change the agent’s vector towards the desired vector over time. This has helped to remove any jerky or sudden movements that were much more prevalent beforehand.

Our flocks can operate independently of each other. Each flock knows which members belong to it when it is created, we can use this information to filter out any neighbours that do not belong to the same flock and ignore them when calculating our movement. This allows for multiple flocks to be on screen at a given time. Two flocks may still encounter each other but they will both simply try to move past each other. This may not necessarily be needed for every game type and is more of a nice to have feature.

We originally planned to apply the flocking to the minions that travel down the lanes, however, due to the nature of MOBA style games, lane minions have very linear movement and this proved not to be a very good way of displaying how the flocking works. Instead we chose to move the flocks into the jungle where there are multiple obstacles, the agents would be able to roam more freely and eventually group up into a flock.

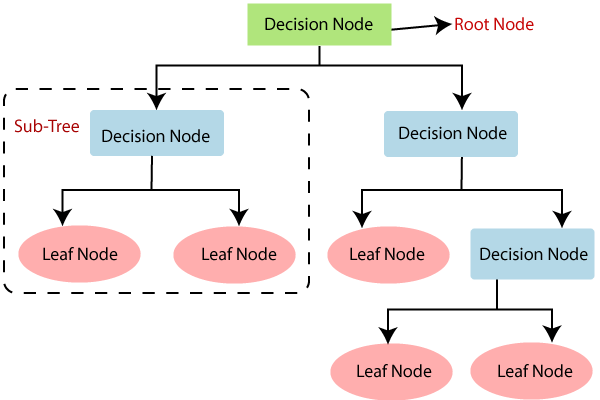
Although basic flocking algorithms perform well by themselves it is apparent that it is unable to handle all the issues related to a group’s movement. *(Danielsiek et al., 2008)* The pathing of the group still needs to be handled by other algorithms such as A\* which can cause issues when moving around the map. A way of handling this would be to use influence maps which can indicate where the boundaries of an area lie or where other agents are situated. If for example you had a 2 player strategy game the influence map could be used the identify where the enemies forces are most concentrated and either use this information to move towards or away from that location giving better direction to the flock. Combining these techniques can help to make a flock operate more intelligently.

## Decision Making

When deciding how to implement decision making, two different methods were looked at;

Decision Trees:

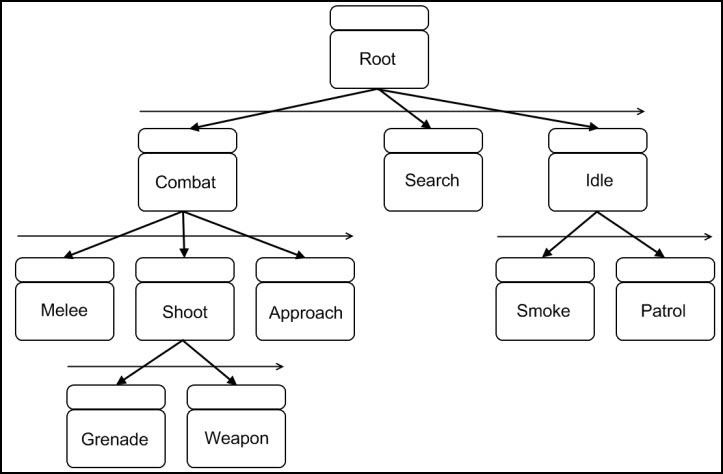
The decision tree works by creating branches of yes/no questions and the outcomes of those choices, this type of tree can also include a probability for outcomes to reinforce confident choices over time.



*Figure 4. Decision Tree (Anon, n.d.)*

Behaviour Tree:

A behaviour tree similar to the decision tree is a tree of hierarchical nodes which control the flow of decisions. The behaviour tree is also very similar to state machines; This is because the tree traversal is based on node states rather than a yes/no question. There are three main types of nodes used in a behaviour used to control the state. Composite nodes, which determine the fine state of a branch, Decorators that change the state of a child node and Execution nodes hold the action to be performed.



*Figure 5. Behaviour Tree (Anon, 2013)*

Decision trees are great when there are a set of questions that govern decisions. For example, the tree works well when asking, ‘is an enemy in range?’ if yes, attack the enemy, otherwise do not. The problem is that if we now change the question to, ‘which enemy is closer?’ the decision tree no longer works and that is where a behaviour tree comes in. The behaviour tree can look through any number of enemies and determine which is closer. That flexibility in questions and actions is why the behaviour tree was implemented for this demo.

As briefly mentioned before, a behaviour tree has several different types of nodes used to control both the flow of the tree and the state of nodes and branches in the tree. The first type of node implemented was the composite node. There are many different composite nodes, but for this project, only two were implemented.

Selector:

Selector nodes execute their children from left to right. They stop executing when one of their children succeeds. If a Selector’s child succeeds, the Selector succeeds. If all the Selector’s children fail, the Selector fails.

Sequence:

A sequence nodes execute their children from left to right. They stop executing when one of their children fails. If a child fails, then the Sequence fails. If all the Sequence’s children succeed, then the Sequence succeeds.

The next type of node to be implemented was the decorator; again, there are many decorators, but only one was necessary for the game being made.

Inverter:

The Inverter node works similarly to the not operator, effectively flipping the state of its child node. So if the child returns success, it becomes a failure.

Finally, the execution nodes were made. There is nothing special about these nodes, as they contain the behaviour or actions to perform the ai. Because the behaviour tree works with states, though, it is essential to endure all nodes return one of these three states

Success:

The node has successfully performed its action. E.g. attacking an enemy unit.

Wait:

The node is still processing the action. E.g. walking to a location.

Failure:

The node has been unsuccessful in its execution. E.g. attacking an enemy that was just killed.

Combining all of these different nodes makes the behaviour tree.

For this demo, only one behaviour tree is used for all the agents. The reason is that most of the agents would share similar behaviours, such as attacking a tower or another champion/minions. The only difference in the behaviour would be from retreating when health is low or waiting by a tower to regroup with champions. The whole behaviour tree can be seen below.

Selector

Chooses the action to take

TargetChampSeq

Determine if a Champion shoould be targeted

IsChampionInRange?

Check if a champion is in range

AttackChampion

Set the Champion in range to be the target

TargetMinionSeq

Determine if a minion should be targeted

IsMinionInRange?

Check if a minion is in range of attack

AttackMinion

Set the minion in range to be the tagret

TargetTowerSequence

Determine if a tower shoould be targeted

IsInGroup?

GoToEnemyTower

Find the closes enemy tower and make that the target

RegroupSeq

Determine if it is necessary to regroup

Inverter

Invert isInGroup

IsInGroup?

Check to see if the unit is in a group of 3 or more

FindClosestOwnedTower

Find the closest tower belonging to the unit and wait for more units

Overall, the implementation of the algorithm works quite well. It is flexible and scalable as tree and behaviours can be chained together to create complex actions. However, because the entire system is purely scripted at the moment, it is incredibly time-consuming to create even a simple behaviour tree. One solution to this would be to create a user interface for the behaviour system similar to what is used in the Unreal Engine.

There were also a few problems with how the behaviour system interacted with Unity. The main problem was that many nodes require knowledge of game objects that can be destroyed. Meaning that sometimes, when another unit destroys a game object, they break for a moment until the garbage collector can clean up the memory and the list can be cleared and updated.

Another unexpected behaviour that was not directly caused by the behaviour tree was that champions would sometimes choose to go through the jungle when attacking a tower. However, this behaviour seems to be due to the navmesh picking the jungle as the most efficient route. This caused another unforeseen problem with the flocking agents. The Champions that wandered into the jungle would effectively never come out, as the attacking enemies take precedence over attacking a tower. Meaning that the Champion would stay in the jungle, killing any flock the cross its path. As this was noticed very late in development, the fix was not to allow champions to attack flocking agents.

## Pathfinding

While looking at different Pathfinding algorithms there were many options I could have went with for the purpose of our MOBA Style game, these being;

A\* Pathfinding:

A\* is an Algorithm designed to create a Weighted grid in order to find the optimal path between two points, it is often used for its efficiency and optimal paths.

The algorithm can be changed to match many different scenarios and types of games.

* Completed Algorithm
* Efficient
* Can be Molded to most game types
* Not complicated to employ
* Speed of Algorithm Dependant on Calculation of h

Djiskstra’s Algorithm:

Dijkstra’s is the algorithm in which A\* was built off of, there are many variants and versions but all of them are designed to find the shortest path between two set nodes, however it is very wide-use and not entirely malleable to given situations.

* Nodes are permanent therefore a grid only has to be created once, making it easy to find new paths
* Efficient for large scale problems such as larger maps
* Search is blind making the Algorithm waste time which could be employed better
* Cannot handle negative edges
* Requires tracking of visited nodes

Best First Search:

Best First Search also known as an “Informed Search” is a more simplistic version of the other two, but it can get the job done.

* More efficient than it’s similar Depth First/Breadth First search algorithms
* Middling levels of time consumption
* Not entirely reliable

In the end I chose to go with A\* Pathfinding due to my familiarity with it being the highest and more papers being available on it than any of the other algorithms meaning I could read up on it more.

In a simple Explanation A\* runs through a graph of already calculated values, each with a weighted value and iteratively finds paths while calculating the point total to run through that path, once all possible paths have been found, then you can tell your system to decide whether to use the path with the shortest value or not. In some cases you may want to find the longest possible path (Attempting to fill something instead of trying to get through it).

Due to the complexity of a MOBA Style game I had opted to try and make the pathfinding script only run when needed, such as a unit action or first movement as a method of optimization, as it would be highly favourable over constantly running to try find the shortest path, instead only after actions, such as when an attack is finished, the old path is finished or a nearby event altering their needed path, this could work in conjunction with the decisions making agent in effect.

Unfortunately in the end while the algorithm and setup worked in the test scenario, the implementation in the game world fell short, at first it was intensely laggy due to it trying to create a path all the time, as well as not recognizing what I had set as walls in the previous test, or the Start/End positions. After lots of attempts to get it working I had adhered to the fact that I was under qualified to get it working to the required specifications.

# Conclusion

Individually the algorithms work well and can be demonstrated without many issues. Each algorithm was designed in its own test scene prior to being implemented into the main map which created some problems when we decided to finally bring everything together. It may have been beneficial to try and implement the algorithms into the main map from the beginning instead of creating them in separate tests scenes.

Decision making and flocking did not work together as well as we expected them to. We encountered issues where the scripts would try to override each other causing the AI to either move directly to a location without flocking or flock together and not move anywhere. We worked around this problem by separating the two algorithms and using them on different areas of the game.

While the pathfinding worked fine in the test scene this did not function as intended in the main map and we had to resort to using the NavMesh in place of it.

Overall, we were happy with MOBA as our genre of choice however we agree that the flocking may have worked better using a different game type.

Despite the issues encountered the algorithms still work in the final map and it is possible to play a full game.

# References

Anon., 2013. *Cryengine.* [Online]   
Available at: <https://docs.cryengine.com/display/SDKDOC4/Coordinating+Agents+with+Behavior+Trees>  
[Accessed 28 5 2021].

Anon., n.d. *JavaTpoint.* [Online]   
Available at: <https://www.javatpoint.com/machine-learning-decision-tree-classification-algorithm>  
[Accessed 28 5 2021].

Reynolds, C. n.d. *Boids*  
Available at: <https://www.red3d.com/cwr/boids/>  
(Accessed, 2021)

Anon., 2021. *GeeksForGeeks*. [Online]

Available at: <https://www.geeksforgeeks.org/a-search-algorithm/>

(Accessed, 2021)

Amit Patel., 2009. *RedBlobGames*. [Online]

Available at: <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>

(Accessed, 2021)

Danielsiek, H., Stüer, R., Thom, A., Beume, N., Naujoks, B., Preuss, M. 2008.   
*Intelligent Moving of Groups in Real-Time Strategy Games*

Available at: <https://www.csse.uwa.edu.au/cig08/Proceedings/papers/8075.pdf>  
(Accessed, 2021)