Game AI Decision Making Algorithms

Wei Zeng
University of Utah
380 S 400 E APT 414
Salt Lake City, UT, USA
(858)6665965
u1330357@utah.edu

ABSTRACT
In this report I would mainly discuss my implementation of a 2D decision making simulation using structures including Decision Tree, Behavior Tree and a Goal-Oriented Action Planner.

Keywords
Game AI, Decision Tree, Behavior Tree, Goal-Oriented Action Planner

INTRODUCTION
When people talk about AI in game, we generally think of how algorithm-driven agents behave in a scripted way or in other words, how they make decisions with respect to the surrounding environment. In this report, I will look specifically at this aspect of Game AI technologies and discuss how I design and implement various decision making algorithms on a monster chase mini simulation game.

DECISION TREE
One most basic method to achieve decision making behavior would be via a decision tree. As we could already see in its name, the decision tree is constructed just like the tree data structure, with a slightly different design in what we store as nodes.

In my implementation, I have a base "DTNode" parent class with two other special node classes named "DecisionTreeNode(DTN)" and "ActionTreeNode(ATN)". DecisionTreeNode is a type of node that evaluates whether a certain condition in the game world is true or false. This kind of node stores a type of class called "Decision", which is also a parent class allowing different decision implementations, being responsible for checking the conditions. An example of a DTN could be a "Has Target" check (see in figure 1) which checks whether the destination target exists or not in the game world. Similarly, ATN holds action objects that perform specific actions based on implementation such as "Wander" and "Pathfind" (see figure 1).

The biggest challenge of implementing a decision tree is actually how we could compose world data into an abstraction scheme and allow access to the abstraction scheme between nodes, decision and actions. In my design, I took an approach that encapsulates all required data into a custom class that I named blackboard. The blackboard is initiated at the top level of the hierarchy and all other systems share this blackboard via pointers that

grant access to it. Essentially, in my decision tree (behavior tree as well in the later section), I pass down this blackboard down the tree layers to each child class and nodes.

For the trees, the blackboard contains a large amount of world data. Specific to the below decision tree, here are all the relevant data I used: a vector of all boids in the world, a vector of all obstacles, pointer to path find destination, pointer to my character, pointers to movement algorithm classes to support movement actions, tile graph of the world and lastly all the constant parameters for movement algorithms.

Now, let me introduce a bit about how my decision tree controls the character. Upon execution of the DT-controlled character, we check whether the destination target exists or not. If it does not, we check if we are next to any obstacles. If we are, we execute an evade action to evade the walls. Else we perform a wander action. Going back up, if there is a target in the world, we simply pathfind to the destination (check figure 1 for better understanding and figure 2 to see a snapshot of the behavior).

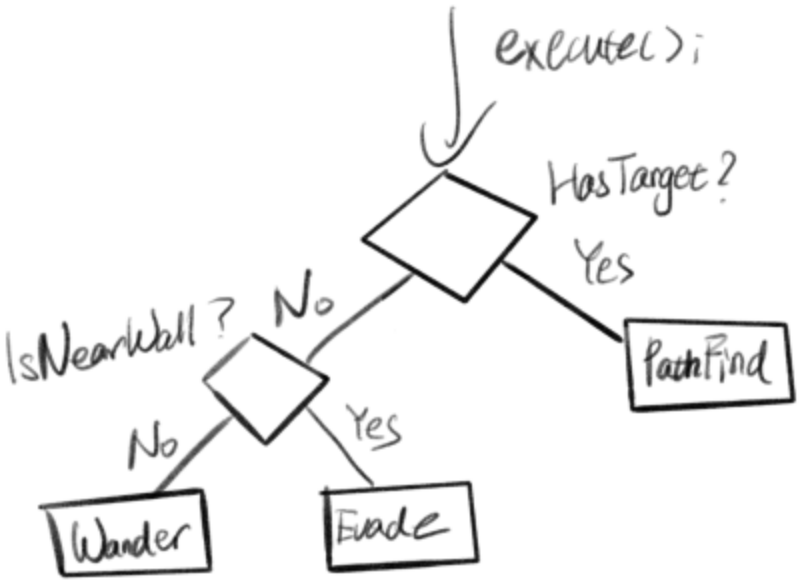

Figure 1: Decision tree design of my simulation.

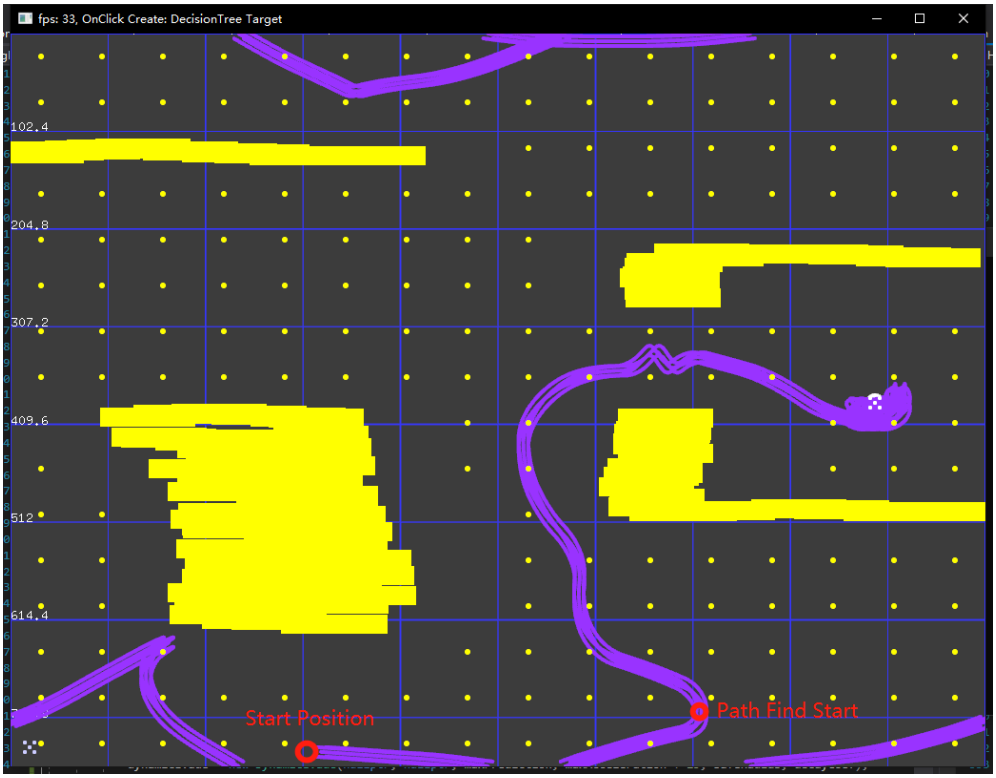

Figure 2: Example simulation of the decision tree controlled boid based on design in figure 1.

BEHAVIOR TREE

Behavior Tree (BT) is another type of decision making algorithm that allows more customization and therefore more power into building complex behaviors. In a Behavior Tree, we view each node as a "Task" which is a base class that supports a wide range of functionalities. Similar to the Decision Tree, the Task nodes in BT have Decision and Action variants which pretty much do the same thing as in DT. However, in BT, node could also become a so-called Composite node that allows special ways of executing its children nodes.

In my BT, I implemented four types of Composites including a Selector, a Sequencer, a Until-Fail Decorator and a Nondeterministic Random Selector.

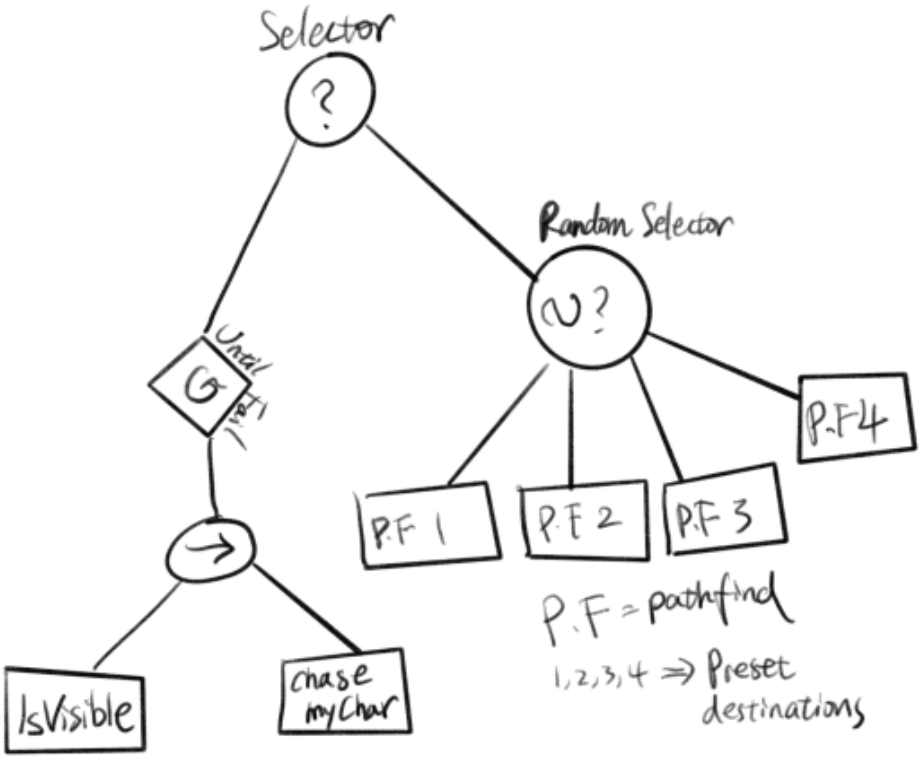

Figure 3: Example behavior of the Behavior Tree controlled character (Monster).

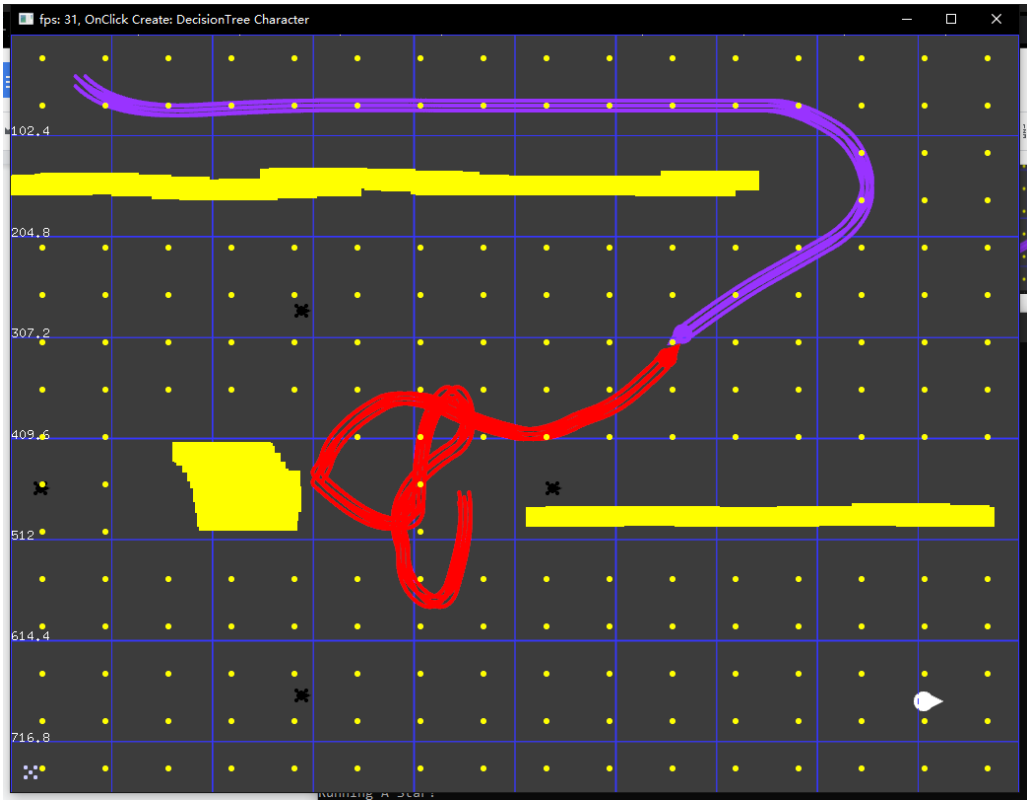

Figure 4: Example behavior of the Behavior Tree controlled character (Monster).

GOAP

For the A* algorithm, I designed two simple heuristic functions to test its performance. First one being a constant guess with h(x) = 1. In my custom graph, this heuristic function is both admissible and consistent. Since the weights of all edges in my graph are significantly larger than 1, this is definitely not an overestimation. As for consistency, it satisfies the triangle inequality since weights are much larger than 1 which means 1 + a, where a >= 1, is always larger than 1.

The second heuristic is a random guess function. Its output was capped to be larger than 100 and smaller than 1300. For this heuristic function, it is hard to say whether it is admissible or consistent because it really depends on the output of the random function. It could produce admissible and consistent results for some certain edges but most of the time it should violate at least one of the rules.

Conclusion

After all the simulations, in conclusion, I found it most challenging to create a user-friendly yet at the same time runtime efficient data structure for graphs. In our simulation the usage of graphs is so ubiquitous that the impact of its performance, in

terms of searching for specific vertices, removing one, adding one etc, become significantly large over the simulation. During implementation and designing my structure, I oftentimes try to get O(1) via using more memory to store extra data for searching but at the end got to a point where some other system completely ruins the time saved due to some extra operations needed to manage that data.