Decision Making

Mit Doshi
University of Utah
Mit.doshi557@gmail.com

Abstract

This document is a report of the implementation of various decision-making algorithms using tile graphs. I have used different algorithms like decision trees, behavior trees, and goal-oriented action planners to created different forms of AI bots having a different task to perform using the Path-finding and movement algorithms created before.

Algorithms

Decision Tree

Decision Tree is a data structure created using tree/graphs data structure. The difference is that Decision Trees are divided into two parts.

- Decision Node.
- Action Node.

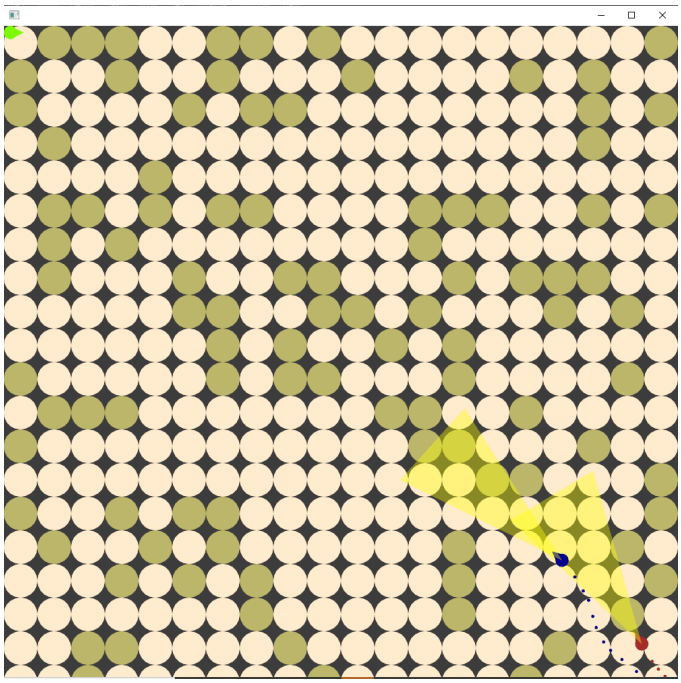
Decision Node is the driver of the decision tree, whereas the action nodes are the task that is expected. In Decision trees, action nodes are always leaf nodes, whereas decision nodes are intermediate nodes. The decision is always affected by the world state, and the action node affects the world state. The action inside a decision tree is handled by an action manager, which handles frame by frame handling of action to provide the desired outcome.

For the desired outcome of the decision tree, there should be a meaningful world state. The decision nodes must have two children who return output in the form of true and false.

The action manager is an important part of decision trees as it needs to schedule the execution of each action. Actions have special categories on which the action manager needs to perform. Following are the categories:

- Priority.
- Exipiry Time.
- Interrupt other actions.
- Run together with other action.

These actions help to create a priority in the action manager, which helps it to maximize its effectiveness.



Working DT-boid and BT-boid

Destroyer Boid

I have changed my representation of my player boid to accommodate the AI feature inside it. I have added features like storing the color that represents it, storing the energy it has to work. The AI boids that I have created for this assignment spent energy to perform any activity, for example, moving, after spending all the energy, the boid needs to sleep to rest and recharge it's energy up to a certain level to work again. The destroyer boid moves around the world, looking for obstacles to destroy. So the actions associated with this boid are as follows:

- Wander aimlessly in the world.
- Look for any obstacles.
- Reach the obstacle.
- Destroy Obstacle.
- sleep to gain energy.

To make it move visually readable. The destroyer boid has multiple colors. Aquamarine when it's wandering and looking for an obstacle. Red when it's destroying the obstacles and black when it's out of energy. Destroying an obstacle removes itself from the world, and that world space can then be used as a location that players can move.

The actions have a strict priority; sleeping has the highest priority above all. Also, sleeping can never be interrupted by any other actions but can interrupt all other actions.

The Decision tree for the destroyer is a binary tree that has a condition that evaluates either true or false, which helps the tree to guide towards action at the end. The destroyer ignores the tile graph because it can destroy obstacles so even though the graph has inaccessible locations the destroyer can reach it by destroying all the obstacles in between

The destroyer decision tree works on three conditions:

- Do the boid has enough energy.
- Is there a obstacle nearby.
- Is the destroyer adjacent to the obstacle.

The First condition checks whether the boid has enough energy to act, and without enough energy, the tree makes the boid go to sleep, else make the boid check the second condition. The second condition checks if there are any obstacles nearby if any obstacles are not found, the tree will make the boid wander around searching for an obstacle. Once an obstacle is found, it checks the third condition, which is to check if the destroyer is next to the obstacle or not. If the destroyer boid is far, it will send action to move towards the obstacle once the destroyer boid is next to the obstacle, destroy action would be called to destroy the obstacle.

For decision trees to work, it also needs the destroyer boid's information as well as the world space information regarding the locations of all obstacles. The creation of the first part was easy as the boid itself is then responsible for performing the actions, and thus it's observing all the time. The second part, however, was difficult as my world state changes every time I restart the application. So, to make the code robust enough that can be reusable, I used Lambda functions. All decision nodes have function pointers called "IsConditionTrue" that evaluate the condition assigned to them, thereby creating different results even though the same function is called. This helped me to modularized my code and made it such that I can use a single function call to create a different implementation. The action doesn't need the world information as they change it, so I have used different class for each action; this helped me to organize my decision tree.

The result of the test was as expected. The destroyer boid would slow start destroying all the obstacles that come in its path. If the program is executed for a long time, then we can notice that most of the obstacles in the world are destroyed, and then the destroyed boid will wander in the middle section of the world, the reason for this as the target to wander are calculated using a binomial distribution.

Behavior Tree

A behavior tree is one of the most used decision-making algorithms. Behavior trees are more like a graph rather than a tree, and every node is active. It is a re-entrant structure, where the tree knows which node it has visited the last time and then executes from the last visited node in the next frame. This eliminates the need for an action-manager, as each node can execute itself when needed. More complex behaviors can be defined using behavior trees as it allows node dependencies. All the data in a behavior tree is stored in a structure called blackboard. A blackboard stores any data need for the behavior tree, including the world states that affect the conditions in the three. All the data is stored using a string mapped value for accessing the data. So I have used a map structure with strings as keys and void pointer as a value. Using void pointers allows the storage of any data; the drawback being both reading and writing operation must cast the data to the same format to prevent memory corruption.

Unlike Decision Tree, who has two types of nodes, the behavior tree has multiple types, including the condition and action task. The other types of task present in behavior tree are:

- Sequencer Task.
- Repeat Untill Fail Task.
- Selector Task.

Each task returns it’s execution status to its parent task, this enables the re-entrant behavior as the parent rembers what the current status of its child are and based on that the execution continues.

Killer Boid

For the Behavior of my Killer Boid, I am using three types of composite nodes: Repeater, Sequencer and Selector. Repeater repeats its children, be it a success or a failure. I wanted the AI to keep running the same behavior through and through and thus, used a repeater. A selector is like an OR condition. It evaluates all its children until one of them is successful, in which case it also returns a success. It fails only when all its children fail. Here I used a selector for deciding if the Boid should sleep or not. If it doesn’t have any energy, the first child returns Success/Running status, goes into sleep and the Selector returns success without running its second child. This logic works beautifully as I don’t want the Boid to be doing anything while sleeping. If in case it does have energy, the first child returns failure,

and the selector then goes to the second child which, logically, makes AI do his wakeful activities. The second child itself is a sequencer. It is like logical AND. It returns Success only when all the children also return Success. The Sequencer is responsible for the agent to get a target and then walk towards the target. If the agent fails to get a target, then it shouldn't be walking anywhere. But if it does, then it should walk towards it. Here, the Get Target action does two things; First it scans for a nearby area and see if the player is around him. If he is, then it makes him as a target. But if there isn't, then it picks a random spot on the map and makes that as a target. If the energy level of Boid is greater than 70% and it is around obstacles, then it may even pick an obstacle as its target. In the second child, 'Walks To Target'. it walks towards the target. The Killer Boid does use the NavMesh and instead of using Wander, uses Path Following. After reaching the target, it picks a suitable action to execute depending on the target.

Here, the 'Get Target' and 'Walk to Target' uses Blackboard effectively. If the Boid has already picked a target, we don't want the Boid to pick a new target every frame. We want it to pick a new target only when it has reached the old target. But you may ask, why do we visit the 'Get Target' node every frame? We can just return Running from 'Walk To Target' until it actually reaches the target and lock the AI from visiting any other node. We can do that, but this prevents the Boid from taking care of other important activities such as sleeping or scanning the Player. We want to do those two every frame. So 'Walk To Target' always return Success. This prevents locking the tree while we are walking. But then how do we solve our initial problem? Answer: We use the blackboard. Once we have a target inside 'Get Target', we store that piece of information inside the blackboard. Next frame, we check if we have a target or not. If we do, we do nothing, otherwise we get a new target. Inside 'Walk to Target', we walk every frame and check if we have reached our target. If we do, we inform the blackboard about it and this makes the 'Get Target' get a new target next frame. There is one additional thing. The Player is the priority of this Boid. Even if the Boid already has a target, it will still always seek the Player's position upon locating him. But what if the Player is always in range of the Boid? Do we use A* every frame to go towards the player? No, we don't! We update the path which the Boid has already computed when it first saw the player. Since the Player can only walk adjacent tiles, the Boid only needs to update the path by adding the new location of the player at the end of it. But if we lose the player in between, then we use A* to compute the path again. This improves the performance significantly.

Another interesting thing to note here is how the 'Get Target' and 'Walk To Target' are segregation of potential singular tasks. Get Target picks between three targets: the player, an obstacle or a random non obstacle tile. In a Decision tree, this would have been in three separate action nodes. Doing so, however, didn't yield any significant benefit, except that this made evaluating and debugging the Behavior tree a lot easier. The same is with the 'Walk To Target' which not only walks towards the target but also decides what action to execute upon reaching the target. Those actions include, eating the player, digging an obstacle or just chilling at a random location.

The Behavior tree, just like the Decision Tree, needs the world state at all times. I again used function pointers to handle that. This has two benefits: All the tasks in my behavior tree are context independent and reusable, and my Behavior Tree has access to all the World information. The Boid has a scanner attached on its head for scanning the player. The range and the width of this scanner is directly proportional to the depleting energy of the Boid. Hence, the lesser energy the Boid has, the easier it is to run away from the Boid as it will have a much shorter range to search for the player. The scanner is shown in game by a yellow alpha-blended triangle. Since digging takes a toll on energy which affects the scanner range, the Boid only digs when above 70% energy and rarely indulges in that. Also, since the Player is the priority, the Boid will only sleep or chase once the player is found. So, if the player ever spawns trapped behind a wall of obstacles, the Boid will be stuck, looking at the player from the other side, trying to rocket through that wall. He won't dig or walk elsewhere in that case. Other than that, the Boid behaves perfectly as desired. Once the Boid successfully manages to eat the player, they both respawn at their initial locations. Compared to the Decision Tree, I didn't have to make different action node classes for different behavior and I didn't have to deal with a separate action manager and priorities. Although, logically, I still had to think equally, even moreso, as I had to make sure I am not locking the tree in some node which should be interruptible. That was easily done in Decision Tree by just setting the interruptible flag as true.

Goal Oriented Action Planner

Goal Oriented action planner is decision making algorithm that works to obtain a world state using a list of actions. GOAP uses A* algorithm to ceate a graph that is generated as needed. We use bit manipulation to handle each world state, and this world state is then used to achieve the desired goal.

For GOAP I have used my destroyed boid, and that added a goal to kill the player to it. For that I am using bit manipulation to have world states like 'isPlayerFound', 'reachedPlayer', 'hasEnergy'.

I have not been able to put much effort to GOAP, and my Planner is still not working everytime, but I have tried doing it