Game AI Decision Making Algorithms

Wei Zeng
University of Utah
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).

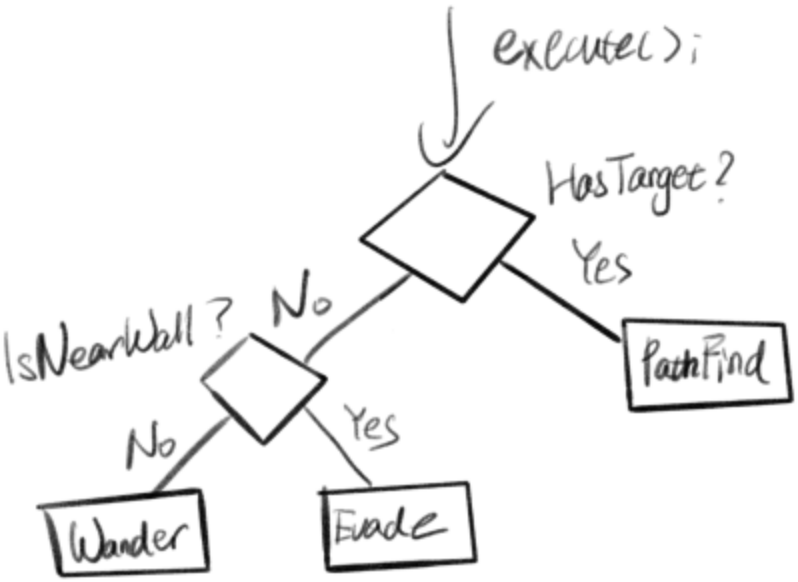


Figure 1: Decision tree design of my simulation.

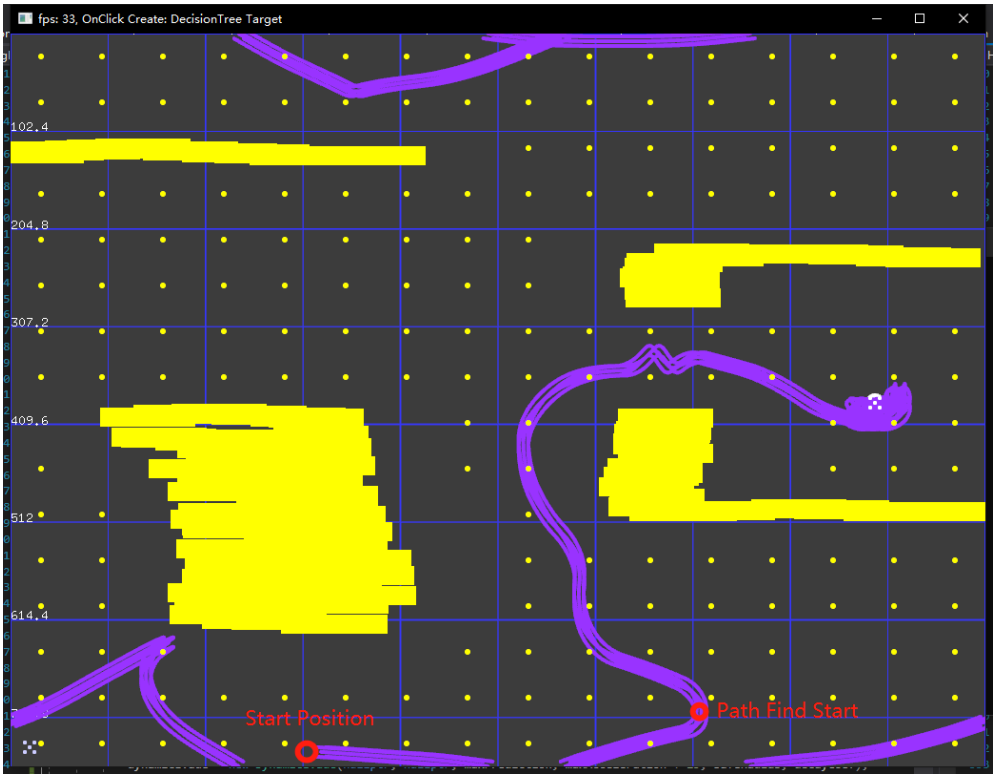


Figure 2: Example simulation of the decision tree controlled boid based on design in figure 1.

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. The design of my BT is shown down below in Figure 3. To simply explain, essentially, the monster character controlled by this BT would first check whether it could see the other character in the world (not blocked by any obstacles). If it does, it will keep chasing the character until the character gets out of sight (blocked out). If it could not see the character, it would randomly select a preset location to pathfind to simulate a patrolling kind of action. These pathfinding actions are interruptible, meaning that it could suddenly turn back to generate some unexpected catch.

There are quite a few challenges along the way of designing and implementing this BT. First of all, it was definitely the hardest to design a diagram that produces something

reasonably looking, in terms of in-game behavior, especially when the actions I have in such a simple simulation are so limited. Additionally, in terms of implementation, since the overall structure feels much like the previous Decision Tree and also they are sharing the same abstraction scheme (my blackboard), it was mostly about getting the inheritance clear and working together. It took some effort to debug since my hierarchy is so complex but through the process I was able to simplify some parts so it became more organized. Lastly, talking about the behavior of my monster, it was a nightmare in the beginning as some actions are not triggering and it caused lots of runtime bugs, mostly because of how I implemented my tasks and how it communicated with the blackboard. Initially I always had missing pointer references (nullptr) in some of my blackboard data since almost everything is taking data from it and a large amount of systems also write data to it. Well, essentially it became a bit hard to manage and I ended up needing to insert a lot of blackboard update calls here and there.

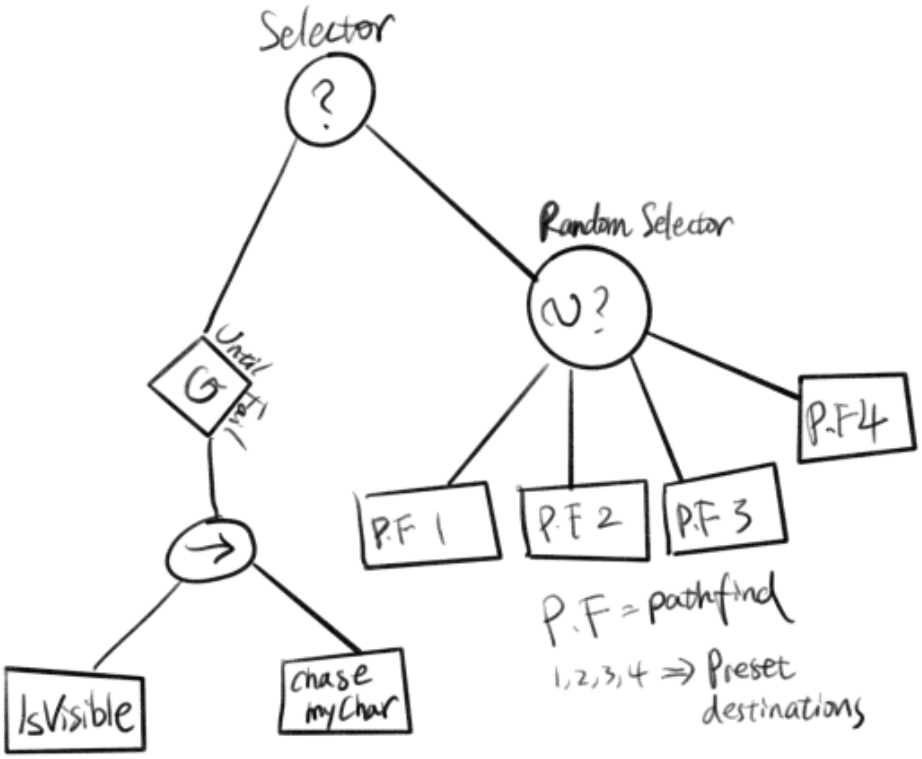


Figure 3: Behavior design of the Behavior Tree controlled character (Monster).

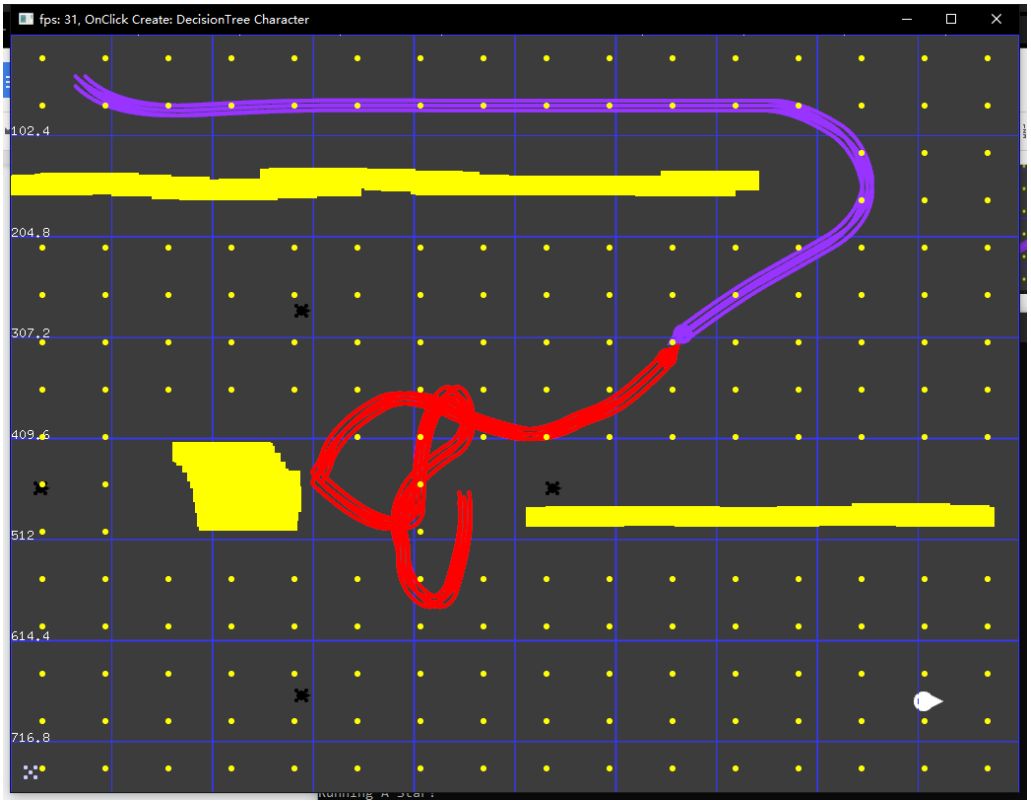


Figure 4: Example behavior of the Behavior Tree controlled character (Monster). Monster in Red, DT character in purple and destination in white.

GOAP

For the last part and the most fun part (also the most time-consuming and problematic), I implemented a simple goal-oriented action planner to control my character (previously controlled by the DT). Just for fun, I am using a different abstraction scheme to represent the state of the world.

First, let me briefly discuss how the new abstraction scheme is designed (please refer to the lower section of **Figure 5**). I created a vector of integers encoding four conditions each representing: 1. Is the character spotted by the monster? 2. Has the character arrived at the target destination to escape? 3. Does the escape destination exist? 4. Is the character far enough from the monster?. Each index could have 3 values from -1 to 1. A value of -1 represents that the condition is not important to evaluate for the given goal or action. A 0 or 1 simply represents false and true. A "WorldStates" class is responsible for maintaining these states at each frame and the GOAP can dynamically retrieve a vector representation as we discussed above to perform action planning. Compared to the previous abstraction scheme, this new one is much simpler and convenient in terms of passing around and doing comparisons.

Now, it is time to introduce the goals of the planner, the actions that it supports and most importantly how it picks what action to perform. In **Figure 5** down below, we could see the two goals of the planner, namely "Escape Room" and "Avoid Monster" both of which should be quite self-explanatory from their name. LOI stands for level of importance which is the key of how we pick which goal to focus on at each frame. Each goal is given a fixed LOI although you might notice that "Avoid Monster" is given two possible values of LOI. This is because I want to allow dynamic goal changes in the game. As I said, LOI is important due to the fact that the planner always picks the goal with a smaller value of LOI (representing a higher priority) to focus on.

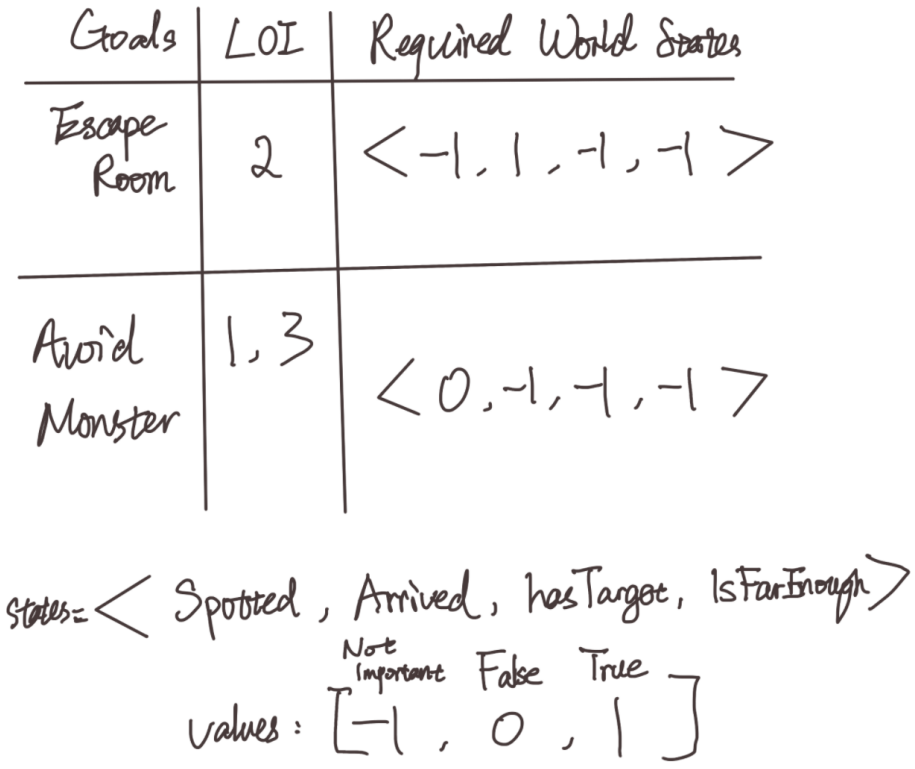
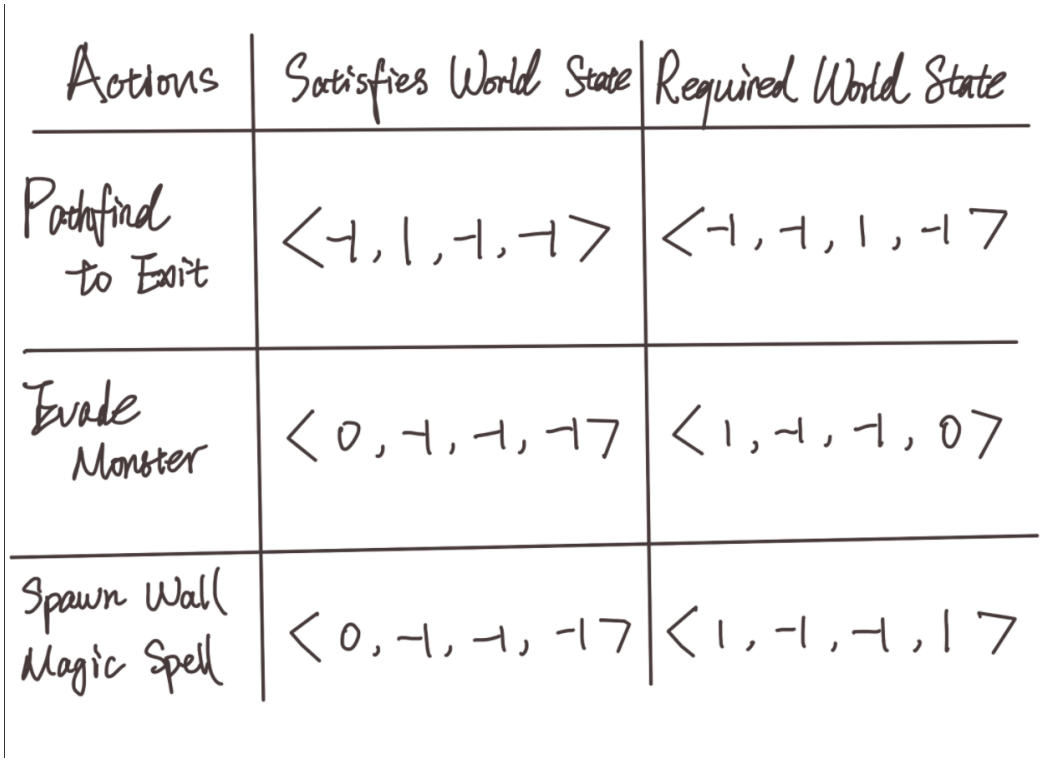


Figure 5: Goals for the GOAP with the Level of Importance (LOI) and Required World State and the explanation of the abstraction scheme.

Because the simulation is fairly simple, I am only supporting three types of actions in the GOAP: "Pathfind to Exit", "Evade Monster" and "Spawn Wall Magic Spell" (See **Figure 6**). The first two are again self-explanatory and I know you might be wondering what this "Spawn Wall Magic Spell" is. This is something that I originally thought that it might be fun to play with but eventually became a nightmare to debug. But anyway, "Spawn Wall Magic Spell" allows the GOAP controlled character to literally spawn an obstacle in-between the monster and itself. I will use this as an example to explain what "Satisfies World State (SWS)" and "Required World State (RWS)" mean. SWS in this case is <0, -1, -1, -1>, meaning that the magic spell is able to satisfy the need of not being spotted by

the monster. RWS in this case is <1, -1, -1, 1>, meaning that in order to cast the spell, the world must have already accomplished the state where the character is spotted by the monster and they are far from each other (so we do not cast a wall into ourselves).



Actions	Satisfies World State	Required World State
Pathfind to Exit	<-1, 1, -1, -1>	<-1, -1, 1, -1>
Evade Monster	<0, -1, -1, -1>	<1, -1, -1, 0>
Spawn Wall Magic Spell	<0, -1, -1, -1>	<1, -1, -1, 1>

Figure 6: Actions available in the GOAP with all of their SWS(Satisfied World States) and RWS(Required World States).

So, how do all these things come together to make a decision? It is really straightforward. It first goes through the goals and finds the one with the highest LOI. It then goes through all actions supported and compares the RWS of the current goal against the SWS of the action. If there is a match, meaning that we found an action that could take us towards the goal state, it checks the RWS of the selected action to see whether the action is good to be performed. If both checks are valid, it fires the action. In the below capture (**Figure 7**), I simulated a GOAP controlled character against a BT controlled monster. As you could see, our GOAP agent has been spamming a lot of magic spells to block out the monster. It also got to a point where the monster actually approached closely to the GOAP agent triggering the evade action, helping it to eventually escape.

In conclusion, the GOAP agent behaves in a very unique and entertaining way thanks to the magic spell ability and we could definitely see the advantages of having such a structure of decision making method.

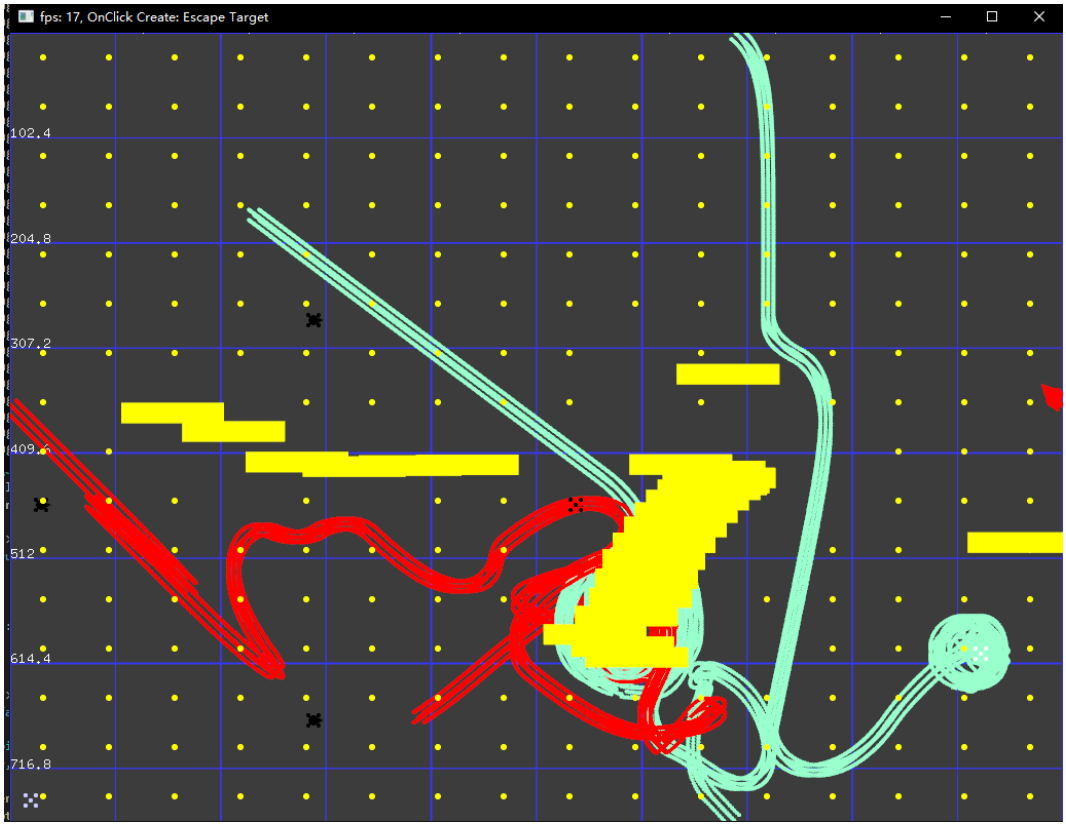


Figure 7: Example behavior of the GOAP controlled character in green, BT monster in red, obstacles in yellow and destination in white (lower right corner).

Conclusion

After all the simulations, I found it most challenging to figure out the overall structure, in terms of implementation, of how we design these algorithms, especially the abstract schemes and how we share the abstraction schemes across different hierarchies. It was also challenging to design actions/decisions/goals/composites that all combine into a reasonable behavior. Inheritance was the definite core of how I could make all these structures work but it was also tricky sometimes to debug. There are also parts and features that I think I could definitely expand on in the future with games that support more actions and gameplay. For example, running an A* on the GOAP action picking algorithm with a heuristic based on how many RWS are needed. In conclusion, these are for sure some challenging topics to implement but also at the same time very customizable and interesting to design.