Hierarchical path finding in Road System

Haoyang Liu
Rensselear Polytechnic Institute
Troy NY
liuh21@rpi.edu

***Abstract*—The Original Path finding algorithm can handle paths in a simple system. The improved hierarchical path finding can help trim down the complexity for an ordered, complex system.**

Keywords—path finding, hierarchical, Game AI

I. Introduction

In the most basic path finding algorithm used in Game AI, the algorithm finds the path by running A* on a single map system. It is fine for a simple system, but with the system complexity rising, the performance of the algorithm drops rapidly. In the designed hierarchical path finding system, The map is separated into different sections, so that the complexity of path finding can be dropped because the number of excessive searches is dropped.

In the example of this paper, I'll be using a road system to demonstrate how to use hierarchical path finding to navigate around.

II. The system

A. The Basic Idea

For a road system, it consists of Junctions and roads. This makes the road system a perfect testing ground for hierarchical path finding, as it can be separated into Junctions and roads in between.

B. Representations

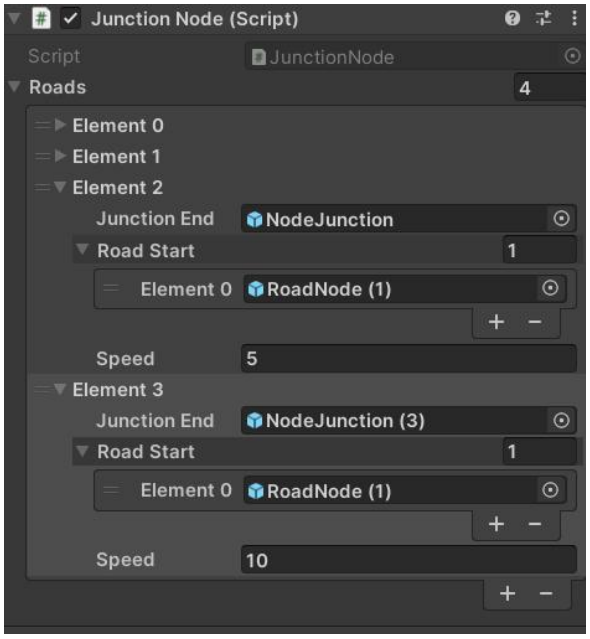
A real modeled road is helpful but not necessary for the system. The actual "road" is actually represented by nodes on the road and junctions. They're connected by one-way edges, so as a matter of fact, every road is actually one-way roads, and the two way road is actually 2 opposite road laid side by side. In that way, we can prevent the car driving on the wrong side of the road.

C. Information Storage

For the system, to secure accuracy, the nodes are manually placed on the map. So do the connections.

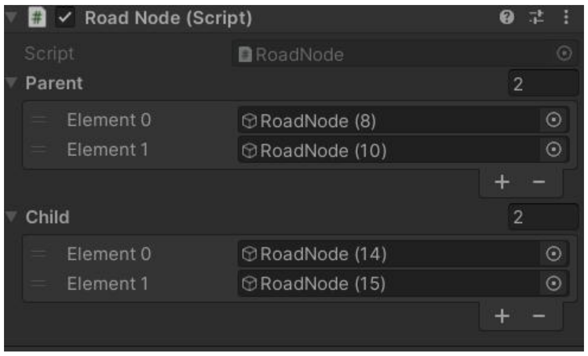
The road nodes consists of 2 lists of nodes: Parent and child. The child consists of the nodes the node connects to, both junction and in-road nodes. The child list has to be filled manually, but the parent list can be filled automatically in runtime by adding the node to its child's parent list.

The junction nodes are a bit different from the road nodes, since they have to be responsible for a higher level of path finding. At this moment, the data is stored in a new class named road. The Road will keep track of the end junction of the road, as well as the road nodes directly connected to the start junction.



```
Junction Node (Script)
Script            JunctionNode
Roads                                 4
  Element 0
  Element 1
  Element 2
    Junction End  NodeJunction
    Road Start                        1
      Element 0   RoadNode (1)
    Speed         5
  Element 3
    Junction End  NodeJunction (3)
    Road Start                        1
      Element 0   RoadNode (1)
    Speed         10
```

Fig 1 the sample of a junction node filled with necessary information



```
Road Node (Script)
Script         RoadNode
Parent                        2
  Element 0    RoadNode (8)
  Element 1    RoadNode (10)
Child                         2
  Element 0    RoadNode (14)
  Element 1    RoadNode (15)
```

Fig 2 the sample of a road node in the middle of a road (runtime)

III. Path finding algorithm

A. The Overview

In essence, the improved algorithm is just a modified A* algorithm, so the weight calculation is the same with A*, which is

$$f(x) = g(x) * Weight_g + h(x) * weight_h \qquad [1]$$

This part will not be elaborated in this paper.

In the algorithm, the agent first finds the way between junctions, then search for ways in between the junctions, and this builds up the basic idea for the function.

B. Find the closest node

In order to start the path finding, we first have to find the closest node in the system. Finding the closest road node is simple, but then we have to find the right connected junction so that the path finding can correctly connect the target and the agent.

For the agent side, since it is trying to move away from the road node, the target junction it going to find is the child of the road node.

For the target, since the agent is trying to move toward the target node, the junction it needed is the parent of the road node.

```
Junction FindParentJunction(Position){
    Node closestNode; //find the closest road node
    While (closestNode is not a Junction){
        ClosestNode = ClosestNode.parent[0];
        //reach to the parent of the node
        }
    Return closestNode;
    }

Junction FindChildJunction(Position) {
    Node closestNode; //find the closest road node
    While (closestNode is not a Junction){
        ClosestNode = ClosestNode.child[0];
        //goes to the child
        }
        Return closestNode;
    }
```

C. Finding path of Junctions

The first step for the path finding is to find the correct sequence of junctions, or in other word, path finding with the node of Junctions. Everything is mostly similar to the A* algorithm. To store the values of the A* navigation without interfering the original information storage, A different class to store information related to A*, called JunctionAStar, is introduced.

The class consists of the following information:

name	Function
Junction	The junction this class refers to
g	The g value of A* path finding for this class
h	The h value of A* path finding for this class
f	The f value of A* path finding for this class
parent	The parent for the Junction (in A*)

In addition to storage, the class is also needs to be comparable so it can be able to find the “same” junction and see if the class needs to be replaced.

For Equals(), JunctionAstar compares the Junction with the other class. If the Junction is the same, then the two class is indicated to be referring to the same object, no matter of the other results.

For CompareTo(), it compares the f value of the 2 JunctionAstar. This functionality is used to see if the current path found to a Junction is actually a more advisable path than the previous one, also to sort the list to pick the next node to search;

The pseudo-code for the path finding comes as below:

```
List<JunctionAStar> junctionAstars;
//all the Junctions we have fully explored
List<JunctionAStar> active = { //the starting
Node};
//all the Junctions that is queued to explore
JunctionStart;
JunctionEnd;

NavigateJunction(){
    While ( active.count > 0 && Not reached
End){
        Active.sort();
        AstarJunction current = active[0];
        Active.removeAt(0);
        //pick the smallest Node, and remove
    from the list
        If (current is the End Junction){
            //switch to searching the road in
        between the Junctions.
            Return;
            }
        Foreach (road connect from the
    Junction){
            //Construct a new JunctionAStar
        class
            JunctionAStar j;
            j.g = current.g + junction.g;
        //filling in the junction, g, h, f, parent
    information of the Junction.

        If (active.contains(j)){
            //find and compare the j
            If ( current j is smaller){
            //Replace j in active;
            }
        }
        Else add j to active;

        }
        Add current to junctionAstars;
    }
```

D. Transition between Junction and Roads

Now we have the Junctions aligned, but we still need the road nodes in between. In the previous step, we just treated them as the connection between the Junctions. To find the best path, we just need to:

1. Trace the parent of the Junction, as normal
2. Treat the two Junctions as start and end of the path finding
3. Add the start Junction to the final path
4. Pass it to road path finding
5. Repeat the steps in a new pair of Junctions.

E. Finding path in the roads

Path finding in the roads is similar to finding path between the Junctions. The most need to caution is that the child of a road node can be a junction, and if it is not the one we wanted, we simply skip it.

Also, we'll use the similar class, RoadAStar, to store information about A*. The basic information is:

name	Function
node	The node this class refers to
g	The g value of A* path finding for this class
h	The h value of A* path finding for this class
f	The f value of A* path finding for this class
parent	The parent for the Road Node (in A*)

Similarly, this class also needs to be comparable for implementation of A*.

The Equals() is use to see if the road node has been add to the list; the CompareTo() is use to compare the f values for optimized selection of nodes.

The pseudo-code for path finding comes as below:

```
List<RoadAStar> roadAStars;
//all the roadNodes that are fully explored
List<RoadAStar> active = {all the nodes the
parent Junction directly connected to}

NavigateRoad(){
    While(active.count > 0 && the path is not
found){
    Active.sort();
    RoadAStar r = active[0];
    Active.removeAt(0);
    Foreach (road node that is child to the
current node){
        RoadAStar ra;
        Ra.g = r.g + road.weight;
        //calculate and record the road node, g,
    h, f and parent for ra
        If (active.contains(ra)){
            //compare ra and the one in list
        (r1)
            If (ra < r1){
                //replace r1 with ra
                }
        }
        Else add ra to active;
        }
        Add r to roadAStars;
    }
}
```

F. The endpoint and the starting point

With all the algorithms above, we can find the way from one junction to another, but our agent and target is connected to the road node instead of the Junction. So at this time, we need it to find a connection between a road node and a corresponding junction. It can be achieved with the NavigateRoad() in the previous part. All we need to change the start or the ending point of the algorithm.

For the target part, it needs to find a connection from junction to a road node. For this part is simple. We just need to replace the end target with the road node.

For the agent side, a little bit of modification is needed. Since we start at the road Node instead of Junction, there is no conflict in data structures when running A*. So, instead of adding the connected road nodes, we can just add the road node itself to the active list.

G. Combine the results.

For now, we have gone through all the algorithms for the agent to find the correct path, but for now, the path is not constructed, with all the information all over the place. What we also need to do is to connect them all. This can actually be done when the A* just finished searching and starts tracing back. We will use a list *route* to store the sequence.

To elaborate, In the beginning, the road from target to the closest Junction is pushed to the list. Then for the part in between, We first push the Ending Junction to *route* at the beginning of the list, the we start the algorithm to find the road between the Junction and the next Junction. Likewise, push the road to the beginning of the list. Repeat until the final Junction is reached. At this moment, all is left is the connection from the junction to the agent, which is another list of road nodes, just push it to the front of *route.*

IV. Testing

To demonstrate the outcome for the algorithm, a road system with 2-lane 2-way roads, 2-lane one-way roads and a 4-lane 2-way road is built in unity (see fig 3).

The nodes in between the road is hand laid on the map and manually connected, and this part took a lot of effort and time.

For the result of the path finding, the car agent can find the route to the target correctly, but it was not guaranteed to be the shortest path space-wise. It would not search for other road nodes else than the one closest to the target, which sometimes may not be the best solution, since it may need to cross several more junctions. Also, in current system, the agent cannot drive to the opposite side of the road, unless through a junction, which is not usually the case in normal traffic rules. The biggest problem comes from the roads end and start at the same junction. The agent can get lost at the junction, for the road is confusing for the current search method. Other than these, the agent can run perfectly along the roads to the target.

V. Flaws and possible improvements

The method provides a good base for a hierarchical path finding in a road system, but in the test, it also shows room for improvements:

- The road is not optimized, it still can be shortened.
- The agent is following strictly to the road path, and unable to make U-turns in the road.
- All the nodes are hand laid and manually connected, which is time-consuming, and possibility can be processed automatically when redesigned.
- The agent can only driving on the road in a uniform speed, and the speed information should as be able to be conveyed to the agent, for it to adjust and follow the speed limitation.

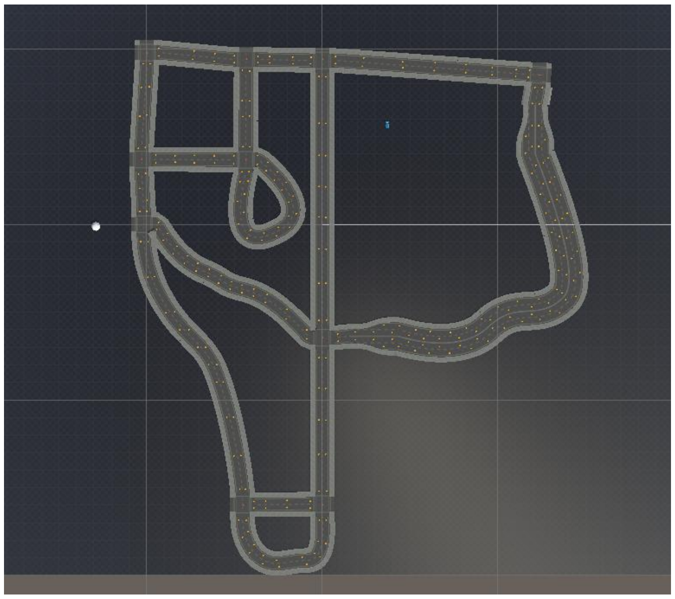


Fig 3 The road system.

- There can be "characteristics" for the agents. We can assign driver with "rude" characteristic, making it able to drive on the opposite way.
- For a system with multiple agent, it would not perform well, as it only calculates the shortest path, without avoiding other agents. A continuous path finding can be implemented to make it possible.
- For the detail level of the design, it cannot separate lanes for the agents, for example, straight-only and left-turn only lanes.

VI. Conclusion

In the paper, we have discussed the hierarchical path finding for a road system. It has proven useful for the system by differentiating between roads and junctions, helping to reduce the depth of the search area, but still remains with rooms for further modification and improvements. This wasn't meant to be a final product, and was never planned for a excel performance, but to provide a base for further works. I hope this algorithm can help create basic idea for creating a road system, and then modify it to fit the theme or idea of the planned game.