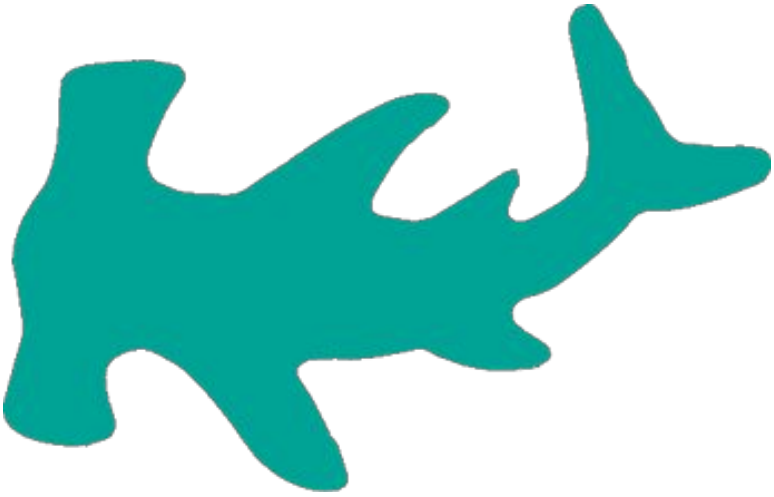


Hammerhead Tron AI

Alex Laurnet, Nick Bolles, David Dowling, Jon Scott

Introduction to Artificial Intelligence | December 12, 2015

Introduction:

The decision to focus on a stochastic "turn" based game provided a desirable mix between traditional logic techniques developed in the early days of artificial intelligence research, and contemporary probability techniques. We approached this problem with the following question in mind to guide our development: "How can a collection of AI agents successfully create forward thinking movements in a speculative environment".

By utilizing the concepts of the game Tron, there were several aspects that we could take advantage of. First, by leveraging an already existing framework, Tronament, we were able to focus specifically on the artificial intelligence aspects of the prototype rather than the graphic interface and human player controls. This framework also provided an environment where we could quickly swap different AI agents in and out and test various agents against one another.

Although solving this problem within a game sphere, this approach can be extrapolated into various stochastic domains. Although the goal states may be different, this approach can be used in several 2 dimensional movement scenarios. By modifying the goal states, this implementation would be used for movement of industrial robotic carts that transport materials from one area of the factory to another, creating optimal cleaning paths for robotic vacuums within a highly trafficked and crowded area. Although the ultimate goal states will need to be modified heavily, this approach could still have use within the financial sector. As described later, our approach uses three states to ultimately decide which direction to move next. This process can be abstractly compared to the decision of the three states of trade , buy, sell, or hold.

Approach:

The idea behind Hammerhead AI is to instigate human error while maintaining its own safety. For this to be successful, there must be several AI agents in place. First, the AI's ability to decide whether it needs to protect itself, attack, or put pressure on the opponent. In order to do this, we designed 4 AI agents: chase AI, bail AI, attack AI, and survive AI. Chase AI is designed to move our AI parallel to our opponent and leave no space between them and wait for the opponent to make a mistake. BailAI is designed to find the most open area on the playing field, and move in that direction in order to escape unsafe space. Attack AI is designed to work when the head of both players is within a five by five square; it then calculates the best move it can make based on a predetermined move set to essentially dominate that five by five square. Finally, the survive AI will attempt to survive as long as possible by finding the biggest open space and spiraling until it is out of room.

The AI agent needs to take in several pieces of data from the user and the game in order to make an intelligent decision. The first input that the AI agent needs is it's position on the game board, as well as its opponents position on the game board; both of which will be used in all of the heuristic functions. Besides the positions on the game board each state will take in additional inputs. For example, the chase AI will input the grid of trails, walls, and empty spaces for the whole board. Other inputs can include the enemy direction, which helps determine the enemy moves or the number of empty game board cells in each direction, which helps determine the best direction to go as a fallback for the heuristic. After the states receive these inputs they will use them to intelligently decide what the best move for their particular use case is.

These four AI agents alone are not capable of making the decisions necessary to decide whether or not the move that each decides is the actual overall best move. In order to remedy this problem, we decided to implement an encompassing "master AI". The idea is that master AI will use heuristics on the game board, to choose which of the individual AI state agents to use. The heuristics include the Manhattan distance between the heads of each player, the safety of the opponent and our AI (which is determined by counting the number of occupied spaces in an 8x8 block around each player's head), and patterns of our opponents actions based on what state we're in and the average distance between the players heads.

Initially, the master AI agent was designed using a Bayesian Network for it's decision making model. At the time of design, the decision to use a Bayesian Network seemed perfect due to the statistical decisions made by this type of model. After research for actual implementation of a Bayesian Network into our code, we found a dynamic Bayesian Network model, a submodel of the overarching Bayesian Network family, that would provide an approach to making decisions as to which state is most likely to make the best move. This structured prediction model is the Hidden Markov model. For a successful decision to be made for master AI to determine the proper move, the Markov model takes into account the hidden properties (the potential states the master AI could enter) to guide the ultimate decision.

At each time step, we recalculate a discrete-time markov chain. Within this recalculation, we factor in the previous chain and the current state to update the statistical probabilities of a potential state change. This chain is the basis of our model that calculates the probability of which state master AI should transition to; ultimately determining which path the AI agent will recommend. The Hidden Markov model's decision will provide a heavier score to a state agent than any of the conditional

arguments will, making it the dominant deciding factor, while still accounting for conditional decision making. This allows for states to have a significant say in what the next move should be for the AI agent.

Once we established the ways our AI will interact with the opposing player, we needed an efficient way to get our AI into the predetermined positions while also factoring in obstacles, the path of each player. To accomplish the desired navigation, we chose to use the A* search algorithm. Each turn, the A* search algorithm constructs a graph of the game board, records whether each cell is traversable, and determines the best path to the goal cell while avoiding any obstacles. For the algorithm to determine whether a cell is an obstacle, the search algorithm weights cells based on whether they have been previously traversed. Then to determine the optimal neighbor node, the program calculates the optimal straight line distance and creates a bias using a static weighting heuristic.

In some situations the A* search may be unable to find a route to the goal coordinate. In this case another mode needs provide the goal coordinate. For example, the chase AI will not be able to find a route to the enemy if the enemy is blocked off. Because there is no way to be offensive or defensive in this situation, the survival mode will be initiated thus trying to survive as long as possible. The survival AI mode is another method of choosing a coordinate to pass to A*. Because the survival mode will spiral around the outside of the box that it is currently in, it should always be able to find a route via the A* search, unless there are no moves left, in which case the game is completed and the enemy wins.

Evaluation:

The only actual completed portion we were able get running was the A* search and Chase AI, and we found that Chase AI is a powerful AI on its own, but it's still predictable. However, in a completed state overall this would not be an issue. The way we developed the structure of our AI allows for the addition of an infinite multitude of state agents that can be factored into the scoring process, so where Chase AI fails, other state agents will be able to succeed, and as long as the scoring process is logically sound there is no reason the Master AI wouldn't be able to initiate the proper state agent at the right time. Additionally, with the fully functioning A* search building a state agent is an easy process. Anyone looking to build a state agent simply needs to pass the coordinates of a point into A* and it will either find the path, or tell you there is none, and does so in approximately 0.8 milliseconds. The more states that are added, the more tools are put in the hands of Master AI, making it exponentially less predictable and more effective every addition. Had we been given more time to build the project further, we would have 3 more states which through some research and logic determined to be essential to building a strong Tron AI. So when we demonstrate the surprisingly high rate of victory Chase AI had against both human and other simpler AI players, we can safely assume that we would be able to increase the margin of victory significantly if the other states were implemented, and Master AI is logically built.

Some additions outside of the "essential" state agents(Chase AI, Bail AI, Attack AI, Survive AI) that we thought of for expansions would be a Cut AI which would move vertically or horizontally for multiple moves to section off parts of the game space leaving the opponent in a smaller space than our AI, Box AI to start building a box around the opponent in an attempt to close them in, and a Neutral AI that simply moves to open spaces similar to Bail AI, but monitors for player moves to react to. Other things

we would likely work on past the final build is a more robust Master AI structure. The Master AI at this point is fairly simple and makes decisions mostly based off scoring, and Markov. Later versions could have a more accurate scoring system through trial run testing including Markov's portion as well. The addition of machine learning that can be saved outside of the client instance would allow for our AI to be trained every time it runs and scored based on victories and losses. These ideas would take much longer than the time we have, but they are certainly possible, and would make our AI even more effective.

Results:

```
153     that.move = function() {
154         that.find();
155         try{
156             move = that.chase();
157         }catch(e){
158             console.log("Cannot find route to enemy");
159             //Just find a safe direction to go now
160             return that.getSafeDirection();
161         }
162         if (!move){
163             debugger;
164             that.chase();
165         }
166         if(move.x != my_x){
167             if(move.x > my_x){
168                 that.direction=tronament.EAST;
169             }
170             else{
171                 that.direction=tronament.WEST;
172             }
173         }
174         if (move.y != my_y){
175             if(move.y > my_y){
176                 that.direction=tronament.SOUTH;
177             }
178             else{
179                 that.direction=tronament.NORTH;
180             }
181         }
182         return that.direction;
183     }
```

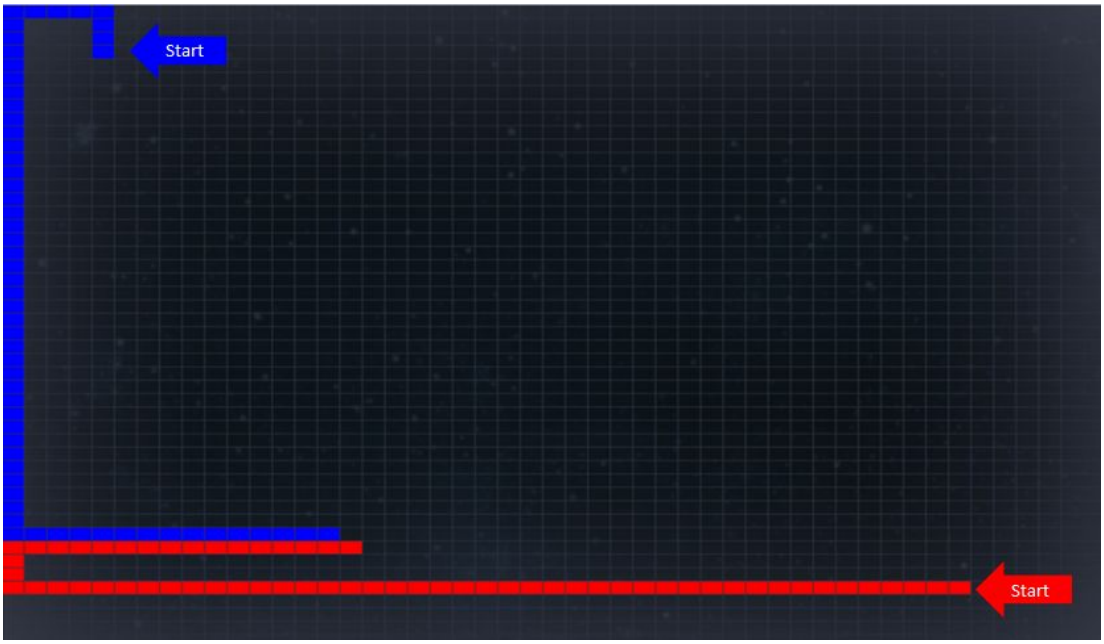
Chase Ai's movement function. This function will try to chase the closest enemy player and if it can't it falls back to just getting a safe direction. With the addition of the master AI, instead of falling back to just finding a safe direction the master AI will decide to go into another mode, such as survival mode.

```
66  that.chase = function(){
67      /* favors is a string that's either up, down, left or right that, based on the vert/hori weight decide
68      ** whether to hug the line above, below, to the left of, or two the right of the opponent.  The weighting
69      ** will measure distance between players and in what direction, then favor will determine which distance
70      ** is shorter, then favor that direction to hug the line.  from there we will make
71      ** a coordinate point.
72      */
73      var x = closestEnemyX,
74          y = closestEnemyY,
75          dx = 0;
76          dy = 0;
77      var invalid = [];
78      switch (that.closestEnemy.getDirection()) {
79          case tronament.NORTH:
80              dy = -1;
81              break;
82          case tronament.SOUTH:
83              dy = 1;
84              break;
85          case tronament.WEST:
86              dx = -1;
87              break;
88          case tronament.EAST:
89              dx = 1;
90              break;
91      }
```

Chase AI's chase function. This function finds where the enemy is, what direction it is moving and finds the closest point that it can get to the enemy. This function attempts to get in front of the enemy by one square if the square is empty.

```
126
127         while(!result){
128             checkBounds(dx,dy);
129             //Add this set to the invalid array so that it is not tried again
130             invalid.push({x:dx,y:dy});
131             var result = that.goto(closestEnemyX+dx,closestEnemyY+dy);
132         }
133         return result;
134     }
135
136     that.goto = function(x,y){
137         that.updateGrid();
138
139         var result = new GraphSearch(grid,new GridNode(my_x,my_y,grid[my_x][my_y]),new GridNode(x,y,grid[x][y]));
140         if (result || result.nextMove() || result.nextMove()[0]){
141             return result.nextMove()[0];
142         }else{
143             debugger;
144         }
145
146
147     }
148
```

Once the Chase AI has a destination point, it calls the goTo function to determine the optimal next move based on the A* search. If the next move is null, then a path is not available. In theory master AI will determine the next best move for survival based on the inputs of other AI agents.



Example of Chase AI(red) in action against Demo AI (an AI based on random moves with a simple safety net). You can see Chase AI is able to match up next to it's opponent with ease.

The statistical results of our Chase AI alone were good despite predictability. Chase AI was victorious in 8 out of 10 trial runs against a human player, and victorious 17 out of 20 times against a simple AI based around random moves with a simple built in safety

feature. The majority of the time we believe Chase AI to have lost is when it is cut off from the other player. This is due to the fact that once it is cut off from the player as it stands now, Chase AI reverts back to the Demo AI (the simple AI stated above) because A* doesn't have a path for it, and the other state agents are absent at the moment. We believe that once the rest of the AI is finished, this will no longer be a problem, and margin of victory will be much higher.

Conclusion:

This project has brought several interesting factors of AI development to light. We learned that a large amount of time is needed to planning and designing each phase so that implementation can go smoothly. The decisions that must be made at design time have a very profound dependency on how successful the final AI agent will behave. On the flip side, we had to be careful that we did not get bogged down with every possible state and into the details to where we were essentially paralyzed with the overwhelming amount of factors to consider. This realization was crucial to our progression forward to the final AI agent.

Another goal of this project was to continue to keep a modular framework that can be easily changed. We built our Maser AI with that it mind. When adding a new state to the master AI, all one needs to do is tell master AI how to weight each output from the newly developed AI agent. This modularity is what will help this AI grow and become a stronger AI as time goes on.

REFERENCES:

javascript-astar 0.4.0:

http://github.com/bgrins/javascript-astar

Freely distributable under the MIT License.

Implements the astar search algorithm in javascript using a Binary Heap.

Includes Binary Heap (with modifications) from Marijn Haverbeke.

http://eloquentjavascript.net/appendix2.html

Other resources for algorithm implementation:

https://en.wikipedia.org/w/index.php?title=Hidden_Markov_model&oldid=691299185

https://en.wikipedia.org/w/index.php?title=Markov_chain&oldid=694804754

https://en.wikipedia.org/w/index.php?title=Bayesian_network&oldid=694678697

https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=693709201