**Artificial Intelligence for Game Developers Coursework 1 Report**

Lecturer: Mike Reddy

Filipe Serrazina, 17123437

University of South Wales

Introduction

According to Wikipedia: “Artificial Intelligence is intelligence demonstrated by machines” [1]. This intelligence is represented through a “Brain” that is composed of one or several algorithms that are adapted to analyse and solve the presented problem.

This project is going to look at Finite-State Machines (FSMs) and the A\* algorithm and how they differ in terms of Artificial Intelligence. In this project, we are also going to use an FSM to create an Artificial Intelligence (AI), with the Python language, designed to beat the default AI built into the provided “Soccer” game.

1. What is a Finite-State Machine?

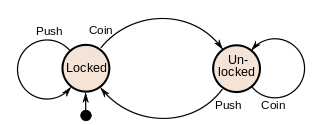
A finite-state machine is a computational model that can be implemented in several ways in order to represent a sequential logic that is organized through states. The simplest example of an FSM is a coin-operated turnstile as presented in the image below:

Figure 1 - FSM for a coin-operated turnstile [2]

As represented in Figure 1, there are two states in this FSM: “Locked” and “Unlocked”. The FSM starts in the locked state, as represented by the arrow coming out of the black circle on the bottom left. The FSM changes between theses two states through triggers, which in this case are “Push” and “Coin”, which represent the action of inserting a coin into the turnstile and pushing the bar of the turnstile. If a person attempts to push the turnstile without inserting a coin, it will remain locked. Once a coin is inserted, the turnstile FSM changes to the Unlocked state until the bar is pushed again.

These state machines can be much more complex than this example, using dozens of states and different triggers. This is the type of AI we will be using for the problem presented further in this report.

1. What is the A\* Algorithm?

The A\* algorithm (pronounced A-star) is a search algorithm that is usually used as a graph traversal and pathfinding method. It is an informed search algorithm, which means that it takes weights into account. Weights are values that influence the algorithm’s decision on what path to take, which generally is the cost or energy that is required to take a certain path.

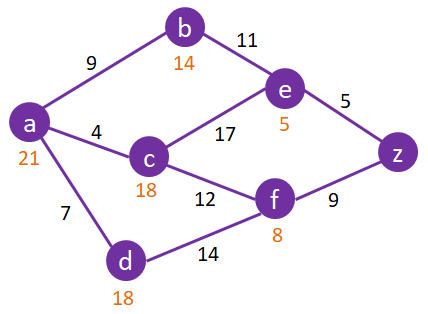
The weights for A\* are dictated by the formula f(n) = g(n) + h(n), where n is the next node on the path, g(n) is the cost from the start node to the n node and h(n) is a heuristic function that estimates the cheapest cost from node n to the goal [3]. A heuristic is an estimate of a path to reach a goal that is not always accurate [4]. The algorithm traverses all possible paths following the ones with the least cost until it eventually reaches its goal.

Figure 2 - Example of a graph analysed by an A\* algorithm

In the image above, we see an example of how A\* would see a graph, black numbers representing the g(n) cost, and orange numbers representing the total f(n).

Implementation

1. *Choosing an Algorithm*

For this project, we had to pick one game from a variety of games for which an AI should be made. In this report, the “Soccer” game was chosen.

The soccer game is, as the name implies, a virtual representation of the sport “soccer”. Given that the objective of the game is to make a series of decisions to reach the opponent’s goal, an FSM was chosen as the better suited algorithm instead of A\* or a combination of the two.

The contributing factors to this decision were the variety of states the game is going to be in, which are presented further in the report, and the fact that A\* just seemed like overkill given that avoiding the enemy players worked just fine without the need to finding the least dangerous or shortest path.

1. *The Finite-State Machine*

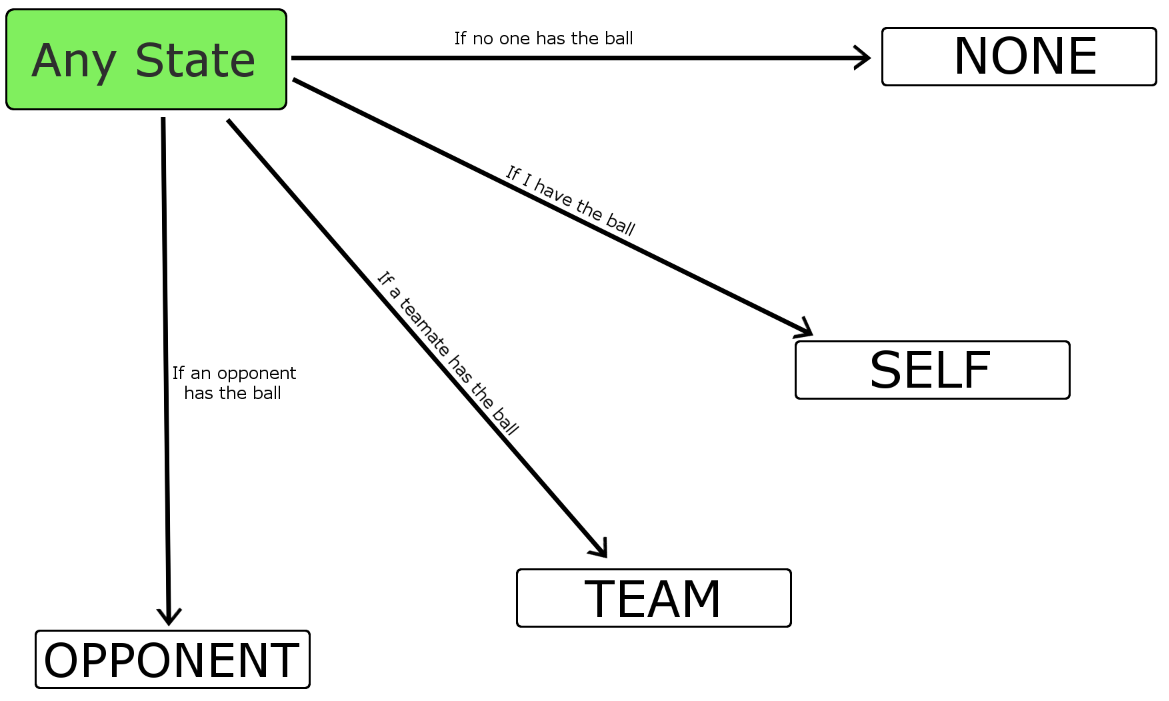
The image below is a visual representation of how the “brain” of this algorithm works.

Figure 3 - Finite-State Machine for the "Soccer" game algorithm

As we can see, there are a total of four states in our state machine: NONE, SELF, TEAM and OPPONENT. The algorithm is in a constant loop that evaluates the current situation it finds itself in and changes the current state accordingly. Let’s breakdown these states so we can see what exactly is happening when the machine is using them.

* **NONE:**

When the machine is in this state, it means that currently no one has the ball. Given that this is a rather simple algorithm and does not take any type of formations into account, it will just run towards the ball and hope that it can catch it before someone else does.

* **SELF**

When in this state, the machine-controlled player has the ball. This is the most complex state since it is the one that must make the most decisions. First, the algorithm checks if there are any targets in close range, this includes players and goals. If there are, it will simply pass or shoot to the closest one that is in a favourable position, which would be further into the field, in this case.

If there are no convenient targets to pass to, the player will keep moving towards the goal while avoiding the closest opponent by calculating the direction towards them and adding the inverse x of that result to our final direction.

As mentioned earlier, when it is in range of the goal, it will simply shoot at it.

* **TEAM**

This state mimics the default AI by positioning the player in a useful position to the current ball holder. Due to the way the game is built, this state is virtually never used since the player with the ball will always be marked as player-controlled. Nonetheless, it is nice to have this state, in case the game is ever modified for whatever reason.

* **OPPONENT**

When in this state, it means that the opponent currently has the ball. At the moment, this state is the same as the “NONE” state and the player will simply run after the ball holder until it reaches them. The ball holder is always slower than the rest of the players, so the AI usually manages to catch up.

1. *Integrating the FSM with Existing Code*

After the FSM was implemented in the “update” function of the MyAI class, the “eval” function was created to determine which state the state machine should be in.

Both functions are executed on the final couple of lines of the Player Class’ “update” function, in order to find the player’s movement target, instead of doing so by detecting the player input.

This is done, by first, running the eval function to update the state machine’s current state, and finally run the FSM which gives the player its new movement target.

Results

Overall, this project was a success. Out of 30 tests, 10 against each difficulty, the AI won 100% of the times, although the “Hard” difficulty did manage to score a couple of goals in some games.

This FSM was built has a Perfect AI, meaning that even though it has an outstanding success rate, it does not behave human-like, and it is easy to see that it is a machine that always makes a quick decision. Because of this, this AI is not suited to be put up against a human player, as it would quickly become frustrating for the player. However, since the goal of this project was to make it beat the default AI of the game, it was a huge success. Here are a few videos demonstrating the AI in action:

**EASY DIFFICULTY:** <https://www.youtube.com/watch?v=7eqTrqdZSII&feature=youtu.be>

**MEDIUM DIFFICULTY:** <https://www.youtube.com/watch?v=5K4PjzxaKu0&feature=youtu.be>

**HARD DIFFICULTY:** [https://www.youtube.com/watch?v=9geGQ26WpKw&feature=youtu.be](https://www.youtube.com/watch?v=9geGQ26WpKw&feature=youtu.be%20)

Conclusion

Although it is possible that the FSM could’ve performed better with the help of A\* pathfinding, the “Results” section showed that there was no need for one. There is a chance that this happened because the default AI wasn’t that good. Maybe with a stronger opponent, different techniques would have had to been used.

Overall, this project was a great experience in order to understand AI better and the different algorithms that can be used to achieve said intelligence. It was also very beneficial in order to grasp a lot of techniques used in Python, Pygame and Game Development in general.

Code Appendix

**AI Class**

# (17123437 - Filipe Serrazina) Main AI of the coursework

class MyAI:

def \_\_init\_\_(self, controlled\_player):

self.player = controlled\_player

# STATES = "OPPONENT", "TEAM", "NONE", "SELF" // These determine who has the ball

self.state = "NONE"

def eval(self):

# No one has the ball

if game.ball.owner is None:

self.state = "NONE"

# I have the ball

elif game.ball.owner == self.player:

self.state = "SELF"

# A teammate has the blayer

elif game.ball.owner.team == self.player.team:

self.state = "TEAM"

# An opponent has the ball

else:

self.state = "OPPONENT"

def update(self, speed):

ballPos = game.ball.vpos

# If no one has the ball or the opponent has the ball

if (self.state is "NONE") or (self.state is "OPPONENT"):

# Move towards the ball/opponent and get the ball

targetDir = ballPos - self.player.vpos

return self.player.vpos + normalize(targetDir) \* speed

# If I have the ball

elif self.state is "SELF":

# Helpful variables for quick use

myPos = self.player.vpos

goalPos = game.goals[0].vpos

dirToGoal = myPos - goalPos

distToGoal = magnitude(dirToGoal)

team = game.teams[self.player.team]

targetable\_players = [p for p in game.players + game.goals if

p.team == self.player.team and targetable(p, self.player)]

# Do we have a targetable object?

if len(targetable\_players) > 0:

# Get the closest one and pass them the ball

target = min(targetable\_players, key=dist\_key(self.player.vpos))

game.debug\_shoot\_target = target.vpos

# If we don't have any targetable objects

else:

# Then we don't have a target at all

target = None

# If we don't have a target

if target is None:

# Make an unreasonable target position

targetPos = Vector2(100000, 100000)

# If we do have a target

else:

# Get its position

targetPos = target.vpos

# Get the direction to the target

dirToTarget = targetPos - myPos

# Get the distance to the target

distToTarget = magnitude(dirToTarget)

# If we're close enough to the target

if distToTarget < 300:

if target.vpos.y < self.player.vpos.y:

# Give them the ball and stop

game.ball.shoot(team, target)

#self.state = "NONE"

return self.player.vpos

# If we're not close to the target, just keep moving towards the goal

# Bit of the FSM that handles opponent player avoidance, due to the aggressive way

# the AI plays, this is more of a hindrance than actual help

# Get the closest opponent player

opponent\_players = [p for p in game.players if p.team != self.player.team]

closest\_opponent = min(opponent\_players, key=dist\_key(self.player.vpos))

# Get the direction towards them

playerPos = self.player.vpos

oppPos = closest\_opponent.vpos

dirToOpponent = closest\_opponent.vpos - self.player.vpos

if self.player.vpos.y > closest\_opponent.vpos.y:

dirToOpponent.y = 0

dirToOpponent = normalize(dirToOpponent)

else:

dirToOpponent.x = 0

dirToOpponent.y = 0

dirToGoal = normalize(dirToGoal)

dirNormalized = normalize(dirToGoal + dirToOpponent)

return self.player.vpos - dirNormalized \* speed

elif self.state is "TEAM":

if self.player.active():

# If I'm near enough to the ball, try to run somewhere useful, and unique to this player - we

# don't want all players running to the same place. Target is halfway between home and a point

# 400 pixels ahead of the ball. Team 0 are trying to score in the goal at the top of the

# pitch, team 1 the goal at the bottom

direction = -1 if self.player.team == 0 else 1

self.player.home.x = (ballPos.x + self.player.home.x) / 2

self.player.home.y = (ballPos.y + 400 \* direction + self.player.home.y) / 2 **Player Class’ Update Function**

def update(self):

# decrement holdoff timer

self.timer -= 1

# One of the main jobs of this method is to decide where the player will run to, and at what speed.

# The default is to run slowly towards home position, but target and speed may be overwritten in the code below

target = Vector2(self.home) # Take a copy of home position

speed = PLAYER\_DEFAULT\_SPEED

# Some shorthand variables to make the code below a bit easier to follow

my\_team = game.teams[self.team]

pre\_kickoff = game.kickoff\_player != None

i\_am\_kickoff\_player = self == game.kickoff\_player

ball = game.ball

if self == game.teams[self.team].active\_control\_player and my\_team.human() and (

not pre\_kickoff or i\_am\_kickoff\_player):

# This player is the currently active player for its team, and is player-controlled, and either we're not

# currently waiting for kickoff, or this player is the designated kickoff player.

# The last part of the condition ensures that in a 2 player game, player 2 can't make their active player

# run around while waiting for player 1 to do the kickoff (and vice versa)

# A player with the ball runs slightly more slowly than one without

if ball.owner == self:

speed = HUMAN\_PLAYER\_WITH\_BALL\_SPEED

else:

speed = HUMAN\_PLAYER\_WITHOUT\_BALL\_SPEED

# (17123437 - Filipe Serrazina) AI Operations

self.ai.eval()

target = self.ai.update(speed)

**Helper Functions**

# (17123437 - Filipe Serrazina) Helper function used to calculate the magnitude/length of a vector2

def magnitude(v):

return math.sqrt(pow(v.x, 2) + pow(v.y, 2))

# (17123437 - Filipe Serrazina) Helper function used to normalize a vector2

def normalize(v):

vmag = magnitude(v)

if vmag == 0:

vmag = 0.1

return Vector2(v.x / vmag, v.y / vmag)

References

1. <https://en.wikipedia.org/wiki/Artificial_intelligence>
2. https://en.wikipedia.org/wiki/Finite-state\_machine
3. [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)
4. <https://en.wikipedia.org/wiki/Heuristic>