

|  |
| --- |
| CAPTURE THE  FLAG |
|  |
| August 28  AMBROSE TREACY COLLEGE  By: Macsen Cunninghame and Nathan Perrier |



Contents

[EXPLORE 3](#_Toc112922020)

[Problem 3](#_Toc112922021)

[Project Scope 3](https://atcqldedu-my.sharepoint.com/personal/056671_atc_qld_edu_au/Documents/Desktop/final_GF_Capture_The_Flag-master/Macsen%20and%20Nathan's%20Capture%20the%20Flag%20Term%203%20Year%2010%20Digital%20Solutions%20Folio.docx#_Toc112922022)

[Limitations 3](https://atcqldedu-my.sharepoint.com/personal/056671_atc_qld_edu_au/Documents/Desktop/final_GF_Capture_The_Flag-master/Macsen%20and%20Nathan's%20Capture%20the%20Flag%20Term%203%20Year%2010%20Digital%20Solutions%20Folio.docx#_Toc112922023)

[Boundaries 3](https://atcqldedu-my.sharepoint.com/personal/056671_atc_qld_edu_au/Documents/Desktop/final_GF_Capture_The_Flag-master/Macsen%20and%20Nathan's%20Capture%20the%20Flag%20Term%203%20Year%2010%20Digital%20Solutions%20Folio.docx#_Toc112922024)

[Assumption 3](https://atcqldedu-my.sharepoint.com/personal/056671_atc_qld_edu_au/Documents/Desktop/final_GF_Capture_The_Flag-master/Macsen%20and%20Nathan's%20Capture%20the%20Flag%20Term%203%20Year%2010%20Digital%20Solutions%20Folio.docx#_Toc112922025)

[Machine learning 4](https://atcqldedu-my.sharepoint.com/personal/056671_atc_qld_edu_au/Documents/Desktop/final_GF_Capture_The_Flag-master/Macsen%20and%20Nathan's%20Capture%20the%20Flag%20Term%203%20Year%2010%20Digital%20Solutions%20Folio.docx#_Toc112922026)

[Analysis of classes and strategies 5](#_Toc112922027)

[Class Analysis 5](#_Toc112922028)

[BOT ANALYSIS 6](#_Toc112922029)

[PSEUDOCODE 8](#_Toc112922030)

[Evidence of screen output and code 14](#_Toc112922031)

[TEST 20](#_Toc112922032)

[User Testing 22](#_Toc112922033)

[Variable Table 22](#_Toc112922034)

[Social 24](#_Toc112922035)

[Personal 24](#_Toc112922036)

[Economic 24](#_Toc112922037)

[Legal 24](#_Toc112922038)

[Additional Evaluation 25](#_Toc112922039)

[Risks Taken 25](#_Toc112922040)

[What Was learnt 25](#_Toc112922041)

[Improvements and Limitations 25](#_Toc112922042)

[Real World Application 26](#_Toc112922043)

[References 27](#_Toc112922044)

# EXPLORE

## **Problem**

The challenge for this project is to program robotic tanks to engage in combat in two-person teams against teams from other classes to see whose code can outwit and outwit the other.

## **Project Scope**

* Draft Deadline: 19/08/2022
* Final Deadline: 31/08/2022

## **Limitations**

Advanced Mathematics does limit the number of features we can add to the code

## **Boundaries**

Boundaries that don’t need to be implemented:

* No cheating.
* No hacking.
* Can’t use outside parts of the code to disadvantage the other team or add an advantage to your team.

1. If a bot collides with an enemy bot on enemy territory, that bot goes to its team’s jail.
2. While jailed, bots cannot move.
3. If a friendly bot collides with a jailed bot, the jailed bot is released and can move again.
4. If a bot collides with its flag, that bot picks up the flag
5. If a bot brings its flag to its side, then its team wins
6. If no flag is taken after the time limit (2 minutes), then the team with the most points win

Points are rewarded for staying on enemy territory and being close to the flag. Points are rewarded using the following algorithm for each bot on its enemy’s side:

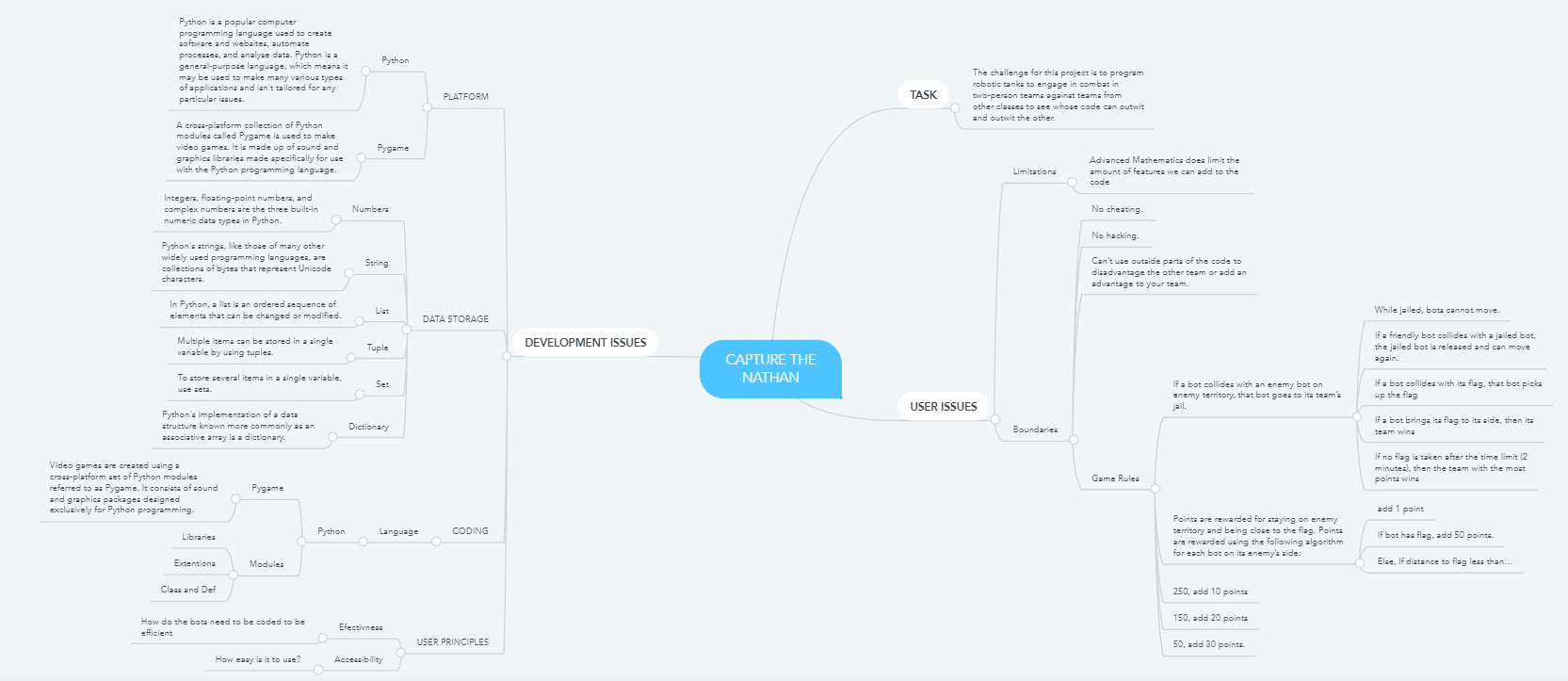
* add 1 point
* If bot has flag, add 50 points.
* Else, If distance to flag less than…
  + 250, add 10 points
  + 150, add 20 points
  + 50, add 30 points.

## **Assumption**

What is assumed to be implemented:

* Can the Bots move?
* Can the Bots sense other bots?
* Does the code run without any errors?
* Can the Bots defend properly?
* Is the code easy to update if changes are needed to be made?
* Does the code follow the specific guidelines?
* Laptop computers
* Internet
* Microsoft Visual code
* Online text editor
* MS Word
* Game Frame Capture the Flag files / environment

MIND MAP



|  |  |  |  |
| --- | --- | --- | --- |
| **PRESCRIBED CRITERIA** | | **SELF-DETERMINED CRITERIA** | |
| **PC1** | Must make a folio showing exploration, Analysis, Development, Synthesis and Evaluation. | SDC1 | Using arrays and lambda functions to sort the enemy’s distance, identification, and position |
| **PC2** | Must code 10 bots to compete in CTF | SDC2 | Using lists, arrays, and dictionaries to store data |
| **PC3** | Must use functions/ methods defined by parent classes and adhering to rules of engagement | SDC3 | Using vectors and linear equations to intercept the attacking enemy bots |
| **PC4** | Must create a 4 min video explaining your teams’ strategies | SDC4 | Machine Learning to avoid enemy bots when retrieving the flag |
| **PC5** | Must anticipate unseen enemy teams’ strategies and create code that will react accordingly | SDC5 | Modular Bots to increase or decrease the amount of defender or attackers |
|  |  | SDC7 | Updating the enemy bot’s locations every tick to determine the best path using weighted grid to get the best path to the flag |
|  |  | SDC8 | Creating attacking formations to efficiently get the flag |

Development Platform

## Machine learning

Machine learning is a subfield of machine learning (AI) and computer science that utilizes data and algorithms to mimic how people learn, progressively improving its accuracy. Machine learning is a critical component of the rapidly expanding discipline of data science. (Neural Networks , 2020). Algorithms are taught to generate classifications or predictions using statistical approaches. For example, an algorithm may be taught using images of dogs and other objects that have all been identified by humans, and the machine could then learn how to recognize images of dogs on its own. Python outperforms all other languages, with more than 60% of machine learning engineers utilizing and prioritizing it for development due to its ease of use. (Machine Learning, 2020). Neural networks, also known as artificial neural networks (ANNs) or simulated neural networks (SNNs), are a subset of machine learning that provide the foundation of deep learning techniques. Their name and form are inspired by the human brain, and they replicate the way real neurons communicate with one another. (How machine learning works, 2020)

# Analysis of classes and strategies

### Class Analysis

Diagram

Description automatically generated

3

Strategy analysis

Having looked at the example provided by Mr Crane our team has determined that using timings for our bots will be useful. we have made 4 main strategies. The first one is the defending strategy, where the bot tracks the closest enemy and drives towards them if they are less than halfway. The next strategy is the interceptor, where it intercepts the closest bot’s path. Additionally, there is the attackers where they drive towards the blue flag avoiding enemies if they are within a certain distance. Finally, there is the jailbreaker, this strategy detects if there is a bot in jail and if there is then it goes to the jail, avoiding getting stuck on the walls.

# BOT ANALYSIS

|  |  |
| --- | --- |
|  | The bots depicted in this diagram are coded in a technological manner. This requires programming the bots to intelligently utilizing linear equations and specialized math. The defending bots determine the coordinates the bot must go towards to effectively intercept the attacking bot as it moves in a specific direction and at a specific pace. |
|  | As the blue bot attack the red bot’s area, it detects the red defending bots and goes in the opposite direction until there’s an opening. |
|  | The blue bot is calculating the best path to the flag depending on where enemy bots are and what direction they go. It uses an algorithm to determine where each bot is going and what speed there going so the flag can be taken in the most efficient way. |
| Shape, rectangle  Description automatically generated | If the blue bot get the flag but a enemy bot is less than 200px away from the bot then it avoids the bot going in the opposite direction |
| Chart, line chart  Description automatically generated | If Blue 1 and Blue for are less than halfway and a enemy bot is in the blue half of the arena then the blue bot will intercept the bots using the same algorithm used in blue 2. |
|  | If Blue 3 detects a bot that is in the blue half of the arena it will find the closest bot to the red flag and turn towards that bot and drive towards it. |
| Rectangle  Description automatically generated with medium confidence | If Blue bots are in jail, then Blue 3 will go to the jail, avoiding the walls regardless of how many bots are in the blue half. If Blue 3 is in jail, then blue 5 will go to get all the blue bots out of jail. |
| Chart  Description automatically generated with medium confidence | If an enemy bot gets the flag, then all defending and intercepting bots will attack the bot. In addition, blue 5 will stop attacking at will try and get the bot before it crosses halfway. |
| Rectangle  Description automatically generated with medium confidence | If there are no enemy bots in the blue half of the arena, then all defending bots and blue 2 will return to the red flag and circle it until enemy bots attack. |

DATA

## PSEUDOCODE

|  |  |
| --- | --- |
| FUNCTION NAME | PSUEDOCODE |
| **Blue 1 and 4 Tick**  **(Attackers and interceptors)** | SET halfway TO SCREEN\_WIDTH / 2 //halfway point of the arena  SET dict, dict\_positions TO {}, {} //creates empty dictionaries  SET dict\_positions, active\_bots,sorted\_dist,sorted\_bots,sorted\_positions TO self.sort(halfway,dict,dict\_positions, red\_flag, True) //sorts  the positions, distances and identifiers in order of closest or furthest  IF active\_bots AND (self.x IS LESS THAN halfway): //IF there are active bot's AND IF my bot is less than halfway  SET dict\_positions, active\_bots,sorted\_dist,sorted\_bots,sorted\_positions TO self.sort(halfway,dict,dict\_positions, red\_flag, True) //sorts  the positions, distances AND identifiers in order of closest or furthest  SET close\_coords TO (sorted\_positions[0][0], sorted\_positions[0][1]) //SET's the current location of the bot TO a variable  SET distance\_traveled TO self.point\_TO\_point\_distance(close\_coords[0],close\_coords[1], self.enemy\_coords[0], self.enemy\_coords[1])  //the distance the closest enemy travelled in a Tick  SET speed TO self.get\_speed(distance\_traveled) //calculate the speed of the enemy  SET my\_x, my\_y TO self.x, self.y //set the bots x,y positions IN a variable  SET x4 TO my\_x + ((my\_x - close\_coords[0])/2) //calculate x4 (setting it as halfway between the two bots)  SET active TO True  ELSE:  SET active TO False  SET sorted\_bots,sorted\_dist,sorted\_positions, active\_bots TO self.sort(halfway,dict,dict\_positions, self, active) //sorts the positions,  distances AND identifiers IN order of closest or furthest  IF self.wait\_count IS LESS THAN self.initial\_wait: //delays how long UNTIL the bot starts  self.wait\_count += 1  ELSE:  IF active EQUALS True: //if there are active bot's and if my bot is less than halfway  IF (self.x IS LESS THAN (halfway - 80)): //if the bot is less than halfway - 80 px  SET dist\_to\_self, dist\_to\_enemy,x4,y4 TO self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0], self.enemy\_coords[1], close\_coords[0], close\_coords[1], active) //calculate the equations needed TO get the intercept  IF (dist\_to\_self EQUALS dist\_to\_enemy) or ((abs(dist\_to\_enemy – dist\_to\_self)) <= 45): //if the distances are equal and it is an  isosceles triangle then drive towards the intercept at the same speed of the bot  self.drive\_to\_enemy(speed,x4,y4)  ELSE:  IF y4 IS MORE THAN OR EQUAL TO SCREEN\_HEIGHT: //if y4 is larger than the screen height SET it TO the max screen height  AND make distances equal  SET dist\_to\_self TO dist\_to\_enemy  SET y4 TO SCREEN\_HEIGHT  self.drive\_to\_enemy(speed,x4,y4)  ELSE IF y4 IS LESS THAN OR EQUAL TO 0: //IF y4 is smaller than the screen height SET it TO the lowest screen height  AND make distances equal  SET dist\_to\_self TO dist\_to\_enemy  SET y4 TO 0  self.drive\_to\_enemy(speed,x4,y4)  ELSE:  IF (dist\_to\_self IS MORE THAN dist\_to\_enemy) AND (abs(dist\_to\_enemy – dist\_to\_self) > 45): //IF it's not an isoceles  SET x4 TO my\_x + ((abs(x4 - close\_coords[0]))-(dist\_to\_self-dist\_to\_enemy)) // recalculate x4, moving it left  SET dist\_to\_self, dist\_to\_enemy,x4,y4 TO self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0], self.enemy\_coords[1], close\_coords[0], close\_coords[1], active) //recalculate equations now that x4 has been changed  ELSE IF (dist\_to\_enemy IS MORE THAN dist\_to\_self) AND (abs(dist\_to\_enemy – dist\_to\_self) > 45): //IF it's not an isoceles  SET x4 TO my\_x + ((abs(x4 - close\_coords[0]))+(dist\_to\_enemy-dist\_to\_self)) // recalculate x4, moving it right  SET dist\_to\_self, dist\_to\_enemy,x4,y4 TO self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0], self.enemy\_coords[1], close\_coords[0], close\_coords[1], active) //recalculate equations now that x4 has been changed  self.drive\_to\_enemy(speed,x4,y4) //drive forward, TOwards the intercept, matching enemies speed  ELSE: //IF my bot is further than halfway - 80px turn back towards red flag  self.go\_to\_flag()  ELSE:  IF sorted\_dist[0] IS MORE THAN 180: //IF the distance TO the closest bot is more than 180  IF self.has\_flag: //IF the bot has the flag back TO it's own half  self.turn\_towards(0, self.y)  self.drive\_forward(FAST)  ELSE:  self.drive\_to\_flag() //drive TO the blue flag  ELSE:  IF self.point\_to\_point\_distance(blue\_flag.x, blue\_flag.y, self.x,self.y) IS LESS THAN 180 AND sorted\_dist[0] IS LESS THAN 180: //IF  the distance between the bot AND the blue flag is more than 180 AND the closest bot is less than 180 away  self.turn\_towards(-sorted\_positions[0][0], -sorted\_positions[0][1], Globals.FAST) //go the opposite direction  self.drive\_forward(FAST)  ELSE:  self.drive\_TO\_flag() //drive TOwards blue flag  IF self.has\_flag: //IF bot has the flag  self.turn\_towards(0, self.y) //go back TO their half  self.drive\_forward(FAST)  IF sorted\_dist[0] IS LESS THAN 200: //IF bot is less than 200 away from the closest enemy  self.turn\_towards(-sorted\_positions[0][0], -sorted\_positions[0][1], Globals.FAST) //avoid enemy  self.drive\_forward(Globals.FAST)  IF active EQUALS True:  SET self.enemy\_coords TO (close\_coords[0], close\_coords[1]) //update enemy last coordinates with the current coordinates |
| **Blue 2 Tick**  **(Interceptor)** | SET halfway TO SCREEN\_WIDTH/2 //halfway  SET dict, dict\_positions TO {}, {} //create empty dictionaries  SET dict\_positions, active\_bots,sorted\_dist,sorted\_bots,sorted\_positions TO self.sort(halfway,dict,dict\_positions, red\_flag) //sorts the  enemies distance and position from shortest TO furtherest  IF active\_bots: //IF there are active bot's  SET close\_coords TO (sorted\_positions[0][0], sorted\_positions[0][1])  SET distance\_traveled TO self.point\_to\_point\_distance(close\_coords[0],close\_coords[1], self.enemy\_coords[0], self.enemy\_coords[1])  //the distance the closest enemy travelled IN a Tick  SET speed TO self.get\_speed(distance\_traveled) //calculate the speed of the enemy  SET my\_x, my\_y TO self.x, self.y //SET my bots x,y positions IN a variable  SET x4 TO my\_x + ((abs(my\_x - close\_coords[0]))/2) //calculate x4 (setting it as halfway between the two bots)  SET active TO True  ELSE:  SET active TO False  IF active EQUALS True:  IF (self.x IS LESS THAN (halfway - 80)):  SET dist\_to\_self, dist\_to\_enemy,x4,y4 TO self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0], self.enemy\_coords[1], close\_coords[0], close\_coords[1]) //calculate the equations needed TO get the intercept  IF (dist\_to\_self EQUALS dist\_to\_enemy) or ((abs(dist\_to\_enemy – dist\_to\_self)) IS LESS THAN OR EQUAL TO 45): //IF the distances are  equal and it is an isoceles triangle then drive towards the intercept at the same speed of the bot  self.drive\_to\_enemy(speed,x4,y4)  ELSE:  IF y4 IS MORE THAN OR EQUAL TO SCREEN\_HEIGHT: //IF y4 is larger than the screen height SET it TO the max screen height and make distances equal  SET dist\_to\_self TO dist\_to\_enemy  SET y4 TO Globals.SCREEN\_HEIGHT  self.drive\_to\_enemy(speed,x4,y4)  ELSE IF y4 IS LESS THAN OR EQUAL TO 0: //IF y4 is smaller than the screen height SET it TO the lowest screen height and  make distances equal  SET dist\_to\_self TO dist\_to\_enemy  SET y4 TO 0  self.drive\_to\_enemy(speed,x4,y4)  SET y4 TO 0  self.drive\_to\_enemy(speed,x4,y4)  ELSE:  IF (dist\_to\_self IS MORE THAN dist\_to\_enemy) and (abs(dist\_to\_enemy - dist\_to\_self) IS MORE THAN 45): //if it's not an isosceles  SET x4 TO my\_x + ((abs(x4 - close\_coords[0]))-(dist\_to\_self-dist\_to\_enemy)) # recalculate x4, moving it left  SET dist\_to\_self, dist\_to\_enemy,x4,y4 TO self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0], self.enemy\_coords[1], close\_coords[0], close\_coords[1]) //recalculate equations now that x4 has been changed  ELSE IF (dist\_to\_enemy IS MORE THAN dist\_to\_self) and (abs(dist\_to\_enemy - dist\_to\_self) IS MORE THAN 45):  SET x4 TO my\_x + ((abs(x4 - close\_coords[0]))+(dist\_to\_enemy-dist\_to\_self)) //recalculate x4, moving it right  SET dist\_to\_self, dist\_to\_enemy,x4,y4 TO self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0], self.enemy\_coords[1], close\_coords[0], close\_coords[1]) //recalculate equations now that x4 has been changed  self.drive\_to\_enemy(speed,x4,y4) //drive forward, towards the intercept, matching enemies speed  ELSE: //if my bot is further than halfway - 80px turn back towards red flag  self.go\_to\_flag()  ELSE:  IF self.point\_to\_point\_distance(red\_flag.x, red\_flag.y,self.x,self.y) IS MORE THAN 50: //if there are no active bots, turn back towards red flag, and circle it  self.go\_to\_flag()  ELSE:  self.turn\_towards(red\_flag.x, red\_flag.y, MEDIUM)  self.drive\_forward(SLOW)  IF active EQUALS True:  SET self.enemy\_coords TO (close\_coords[0], close\_coords[1]) //set the last coordinates of the enemy to the current coordinates |
| **Blue 3 Tick**  **(Defender and jailbraker)** | SET stuck TO self.is\_self\_stuck() //checks TO see IF the bot is stuck  SET closest, distance\_to\_flag TO self.closest\_to(red\_bots, red\_flag) //get's the closest bot TO the flag  SET closest\_to\_self, distance TO self.closest\_to(red\_bots, self) //get's the closest bot TO self  SET red\_flag TO red\_flag //SET's the red flag as a variable  SET halfway TO SCREEN\_WIDTH / 2 //SETs hafway as half of screen width  IF self.point\_to\_point\_distance(32,32,self.x,self.y) IS MORE THAN halfway - 100: //IF the bot is further than halfway  self.go\_to\_red\_flag() //turn back TOwards the red flag  ELSE:  FOR friend IN blue\_bots: //FOR ever blue bot  IF friend.jailed: //IF the a blue bot is jailed  self.go\_to\_jail(friend) //go TO jail  IF stuck EQUALS True:  self.turn\_left() //IF stuck turn left  FOR bot IN red\_bots: //FOR every red bots  IF bot.has\_flag and bot is not self: //IF bot has flag  self.turn\_towards(bot.x, bot.y, FAST) //turn towards the bot  self.drive\_forward(FAST) //drive forward  IF distance IS LESS THAN halfway - 80: //IF the closest enemy bot is less than halfway then go to enemy bot  self.turn\_towards(closest.x, closest.y, FAST)  self.drive\_forward(FAST)  ELSE:  IF self.point\_to\_point\_distance(red\_flag.x,red\_flag.y,self.x,self.y) IS MORE THAN 50: //IF distance TO flag is more than 50 then go  TO flag  self.go\_TO\_red\_flag()  ELSE:  self.turn\_towards(red\_flag.x, red\_flag.y, Globals.MEDIUM) //else slowly circle flag  self.drive\_forward(SLOW) |
| **Blue 5**  **Tick**  **(Attacker, Jailbreaker, Flag Defender)** | SET halfway TO SCREEN\_WIDTH / 2 //sets halfway as half of screen width  SET red\_flag TO red\_flag //sets the red flag as a variable  SET dist\_to\_flag TO self.dist\_to\_enemy\_flag() // gets the distance of the closest enemy TO flag  SET stuck TO self.is\_self\_stuck() //checks TO see IF the bot is stuck  IF stuck EQUALS True: //IF the bot is stuck then turn left  self.turn\_left()  IF (Globals.blue\_bots[2].jailed) and dist\_to\_flag IS MORE THAN 225: //IF blue 3 is IN jail and the bots distance TO the flag is more than 225  FOR bot IN red\_bots: //for every red bot  IF bot.has\_flag and bot IS NOT self: //IF bot has flag  self.turn\_towards(bot.x, bot.y, FAST) //go towards bot  self.drive\_forward(FAST)  FOR friend IN blue\_bots: //for every blue bot  IF friend.jailed: //IF bot is jailed  self.go\_to\_jail(friend) //go TO jail  ELSE:  SET blue\_flag TO blue\_flag //SET the blue flag TO a variable  SET closest, dist TO self.closest\_to\_me() //get the distance and identIFier of the closest bot TO the bot  IF dist IS MORE THAN 180: //IF distance TO the bot is more than 180  IF self.has\_flag: //IF the bot has the flag  self.turn\_towards(0, self.y) //turn back towards the blue side  self.drive\_forward(FAST)  ELSE: //IF the bot doesn't have the flag then turn towards the blue flag  self.turn\_towards(blue\_flag.x,.blue\_flag.y, FAST)  self.drive\_forward(FAST)  ELSE:  IF (self.x IS LESS THAN halfway - 100) and (closest.x IS LESS THAN halfway): //IF the bot is less than halfway then go towards the closet  bot  self.turn\_towards(closest.x, closest.y, FAST)  self.drive\_forward(FAST)  ELSE:  IF self.point\_to\_point\_distance(blue\_flag.x, blue\_flag.y, self.x,self.y) IS MORE THAN 180: //IF the distance between the blue flag and  the bot is more than 180 then avoid the closest bots  self.turn\_towards(-closest.x, -closest.y, FAST)  self.drive\_forward(FAST)  ELSE:  self.turn\_towards(blue\_flag.x, blue\_flag.y, FAST) //turn towards the blue flag and drive fast  self.drive\_forward(FAST)  IF self.has\_flag: //IF the bot has flag then turn towards the blue half  self.turn\_towards(0, self.y)  self.drive\_forward(FAST)  SET closest, dist TO self.closest\_to\_me() //update the closest bots  IF dist IS LESS THAN 200: //IF dist of the closest is less then 200 then avoid it  self.turn\_towards(-closest.x, -closest.y, FAST)  self.drive\_forward(FAST) |
| **Calculate() (blue 2, 1, 4)** | DEFINE FUNCTION calculate(self,x4, my\_x,my\_y, last\_sorted\_positions\_x, last\_sorted\_positions\_y,sorted\_positions\_x,sorted\_positions\_y):  SET m TO self.gradient(last\_sorted\_positions\_x,last\_sorted\_positions\_y,sorted\_positions\_x,  sorted\_positions\_y) //get's the gradient  SET c TO sorted\_positions\_y - m\*sorted\_positions\_x //rearranges y=mx+c TO get c  SET y4 TO abs(m\*x4 + c) //get the y4 value by substituting x4 into y=mx+c  SET dist\_to\_self TO self.point\_to\_point\_distance(my\_x,my\_y,x4,y4) //gets the distance  between self and intercept  SET dist\_to\_enemy TO self.point\_to\_point\_distance(sorted\_positions\_x,sorted\_positions\_y,x4,y4) //get's the distance  between enemy and intercept  RETURN dist\_to\_self, dist\_to\_enemy, x4, y4 |
| **Gradient()**  **(blue 2,1,4)** | DEFINE FUNCTION gradient(self,x1,y1,x2,y2):  TRY:  SET gradient TO (y2 - y1) / (x2 - x1) IF (x2 - x1) != 0 else 0 //calculates the gradient of two points if it doesn't EQUALS 0, else it set's the  gradient as 0  EXECPT ZeroDivisionError: //if an error occurs due to division by zero then RETURN infinite  SET gradient TO math.inf  RETURN gradient |
| **Sort()**  **(blue 1,2,4)** | DEFINE FUNCTION sort(self, halfway,dict,dict\_positions, object, active):  FOR enemy IN Globals.red\_bots: //for each enemy IN red bots  SET enemy\_distance TO self.point\_to\_point\_distance(object.x,object.y, enemy.x, enemy.y)  //get the distance between the object and the enemy  SET dict[enemy] TO enemy\_distance // put the enemy's indentifier and the distance  into dict  SET dict\_positions[enemy] TO [enemy.x,enemy.y] // put the enemy's indentifier and it's  position into dict\_positions  SET dict TO {key: val FOR key, val IN sorted(dict.items(), key TO lambda ele: ele[1])}  //sort dict values from lease TO most  SET dict\_positions TO {key: val FOR key, val IN sorted(dict\_positions.items(), key TO lambda ele:  ele[1])} //sort dict\_position values from least TO most  SET sorted\_bots TO list(dict.keys()) //put the values and keys into lists  SET sorted\_dist TO list(dict.values())  SET sorted\_positions TO list(dict\_positions.values())  IF active EQUALS False: //if there are active bot's and IF my bot is less than halfway  SET active\_bots TO {} //return an epemty dictionary  RETURN sorted\_bots,sorted\_dist,sorted\_positions, active\_bots  ELSE:  SET i TO 0  SET active\_bots TO {} //create an empt dictionary  WHILE i IS LESS THEN len(dict.keys()): //for every element IN the dictionary  IF sorted\_positions[i][0] IS LESS THAN halfway: //if the current bots (bot[i]) x value is less  than halfway  SET active\_bots[sorted\_dist[i]] TO [sorted\_positions[i][0],sorted\_positions[i][1]] //add  the distance of that bot and it's position TO the active\_bots dictionary  SET i TO i + 1  SET sorted\_bots TO list(dict.keys()) //put the values and keys into lists  SET sorted\_dist TO list(dict.values())  SET sorted\_positions TO list(dict\_positions.values())  RETURN dict\_positions, active\_bots,sorted\_dist,sorted\_bots,sorted\_positions |
| **Drive\_to\_enemy()**  **(blue 2, 1, 4)** | DEFINE FUNCTION drive\_to\_enemy(self,speed,x4,y4):  self.turn\_towards(x4,y4, FAST) //turn towards the intercept point  IF speed EQUALS 1: //matches the speed of the bot  self.drive\_forward(SLOW) //drive forward at the same speed as the enemy bot  ELSE IF speed EQUALS 2:  self.drive\_forward(MEDIUM)  ELSE:  self.drive\_forward(FAST) |
| **Get\_speed()**  **(blue 2, 1, 4)** | DEFINE FUNCTION get\_speed(self, distance):  SET speed TO distance //the distance between two points correlates with the speed of the bot  IF speed IS LESS THAN OR EQUAL TO 3: //if speed is less than or equal to 3 then the bot is going  slow  RETURN 1  ELSE IF speed IS MORE THAN 3 and speed LESS THAN OR EQUAL TO 7: //if speed is more than 3  but less than or equal to 7 then the bot is going medium speed  RETURN 2  ELSE IF speed IS MORE THAN 7: //if speed is more than 7 then the bot is going fast  RETURN 3 |
| **Closest\_to()**  **(blue 3,5)** | DEFINE FUNCTION closest\_to(self, object, target):  TO closest TO object[0] //set closest TO the first object  TO shortest\_distance TO self.point\_to\_point\_distance(closest.x,closest.y,target.x,target.y) //get  the distance between the object and the target  FOR objects IN object: //for every object  IF self.point\_to\_point\_distance(objects.x,objects.y,target.x,target.y) IS LESS THAN  shortest\_distance: //if the distacne between the new object and the target is less  TO closest TO objects //set closest TO the current object  TO shortest\_distance TO self.point\_to\_point\_distance(closest.x,closest.y,target.x,target.y)  //set the distance TO the distance between the new object and the target  RETURN closest, shortest\_distance |
| **Go\_to\_jail()**  **(blue 3, 5)** | DEFINE FUNCTION go\_to\_jail(self,friend):  SET stuck TO self.is\_self\_stuck() //check TO see IF bot is stuck  IF stuck EQUALS True: // IF bot is stuck  self.turn\_towards(friend.x,friend.y, FAST) //turn towards friend  ELSE IF self.x IS LESS THAN OR EQUAL TO 20 and (self.angle IS MORE THAN OR EQUAL TO 80 and self.angle IS LESS THAN OR EQUAL TO 100): //if the bot is stuck on wall with the lowest x axis  self.turn\_towards(friend.x, 0, FAST) //turn towards friend.x and 0  ELSE IF self.y IS LESS THAN OR EQUAL TO 20 and ((self.angle IS LESS THAN OR EQUAL TO 10 and self.angle IS MORE THAN OR EQUAL TO 350) or self.angle < 0): //if the bot is stuck on wall with the  lowest y axis  self.turn\_towards(0, friend.y, FAST) //turn towards 0, friend y  ELSE:  self.turn\_towards(friend.x,friend.y, FAST) //if the bot is not stuck then slowly navigate TO  the friend IN jail  self.drive\_forward(SLOW) |
| **Is\_self\_stuck()**  **(blue 3,5)** | DEFINE FUNCTION is\_self\_stuck(self):  IF not self.jailed: //if the bot is not jailed  IF self.x LESS THAN OR EQUAL TO self.width: //if the bot's x value is less than it's width then  it is stuck  RETURN True  ELSE IF self.x IS MORE THAN OR EQUAL TO SCREEN\_WIDTH - self.width: //if the bot's x value  is more than screen width - the bots width then it is stuck  RETURN True  IF self.y LESS THAN OR EQUAL TO self.width: //if the bot's y value is less than it's height then  it is stuck  RETURN True  ELSE IF self.y IS MORE THAN OR EQUAL TO SCREEN\_HEIGHT - self.height: //if the bot's y value  is more than screen height - the bots height then it is stuck  RETURN True  ELSE:  RETURN False //else the bot is not jailed |

GENERATE

## Evidence of screen output and code

|  |  |
| --- | --- |
| Evidence of Code | Annotations |
| **BLUE BOT 1 and 4 (Attackers and interceptors)** | |
| def tick(self):          halfway = Globals.SCREEN\_WIDTH / 2                  #halfway point of the arena          dict, dict\_positions = {}, {}                       #creates empty dictionaries          dict\_positions, active\_bots,sorted\_dist,sorted\_bots,sorted\_positions =  self.sort(halfway,dict,dict\_positions, Globals.red\_flag, True)   #sorts the positions,  distances and indentifiers in order of closest or furtherest          if active\_bots and (self.x < halfway): #if there are active bot's and if my bot is less  than halfway              dict\_positions, active\_bots,sorted\_dist,sorted\_bots,sorted\_positions =  self.sort(halfway,dict,dict\_positions, Globals.red\_flag, True)   #sorts the positions,  distances and indentifiers in order of closest or furthest              close\_coords = (sorted\_positions[0][0], sorted\_positions[0][1])     #set's the current  location of the bot to a variable              distance\_traveled = self.point\_to\_point\_distance(close\_coords[0],close\_coords[1],  self.enemy\_coords[0], self.enemy\_coords[1])   #the distance the closest emeny traveled  in a Tick              speed = self.get\_speed(distance\_traveled)       #calculate the speed of the enemy              my\_x, my\_y = self.x, self.y                     #set my bots x,y positions in a variable              x4 = my\_x + ((abs(my\_x - close\_coords[0]))/2)   #caculate x4 (setting it as halfway  between the two bots)              active = True          else:              active = False              sorted\_bots,sorted\_dist,sorted\_positions, active\_bots =  self.sort(halfway,dict,dict\_positions, self, active)    #sorts the positions,  distances and identifiers in order of closest or furtherest            if self.wait\_count < self.initial\_wait:     #delays how long until the bot starts              self.wait\_count += 1          else:              if active == True:    #if there are active bot's and if my bot is less than halfway                  if (self.x < (halfway - 80)):       #if the bot is less than halfway - 80 px                      dist\_to\_self, dist\_to\_enemy,x4,y4 =  self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0], self.enemy\_coords[1],  close\_coords[0], close\_coords[1], active)   #calculate the equations needed  to get the intercept                      if (dist\_to\_self == dist\_to\_enemy) or ((abs(dist\_to\_enemy - dist\_to\_self)) <=  45):  #if the distances are equal and it is an isoscele**s** triangle then drive  towards the intercept at the same speed of the bot                          self.drive\_to\_enemy(speed,x4,y4)                      else:                          if y4 >= Globals.SCREEN\_HEIGHT:         #if y4 is large**r** than the screen  height set it to the max screen height and make distances equal                              dist\_to\_self = dist\_to\_enemy                              y4 = Globals.SCREEN\_HEIGHT                              self.drive\_to\_enemy(speed,x4,y4)                          elif y4 <= 0:     #if y4 is smaller than the screen height set it to the  lowest screen height and make distances equa1                              dist\_to\_self = dist\_to\_enemy                              y4 = 0                              self.drive\_to\_enemy(speed,x4,y4)                            else:                              if (dist\_to\_self > dist\_to\_enemy) and (abs(dist\_to\_enemy - dist\_to\_self)  > 45):         #if it's not an isoceles                                  x4 = my\_x + ((abs(x4 - close\_coords[0]))-(dist\_to\_self-  dist\_to\_enemy))   # recalculate x4, moving it left                                  dist\_to\_self, dist\_to\_enemy,x4,y4 =  self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0],  self.enemy\_coords[1], close\_coords[0], close\_coords[1], active)  #recalculate equations now that x4 has been changed                              elif (dist\_to\_enemy > dist\_to\_self) and (abs(dist\_to\_enemy –  dist\_to\_self) > 45):       #if it's not an isoceles                                  x4 = my\_x + ((abs(x4 - close\_coords[0]))+(dist\_to\_enemy-  dist\_to\_self))              # recalculate x4, moving it right                                  dist\_to\_self, dist\_to\_enemy,x4,y4 =  self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0],  self.enemy\_coords[1], close\_coords[0], close\_coords[1],  active) #recalculate equations now that x4 has been  changed                                self.drive\_to\_enemy(speed,x4,y4)  #drive forward, towards the intercept,  matching enemies speed                    else:       #if my bot is further than halfway - 80px turn back towards red flag                      self.go\_to\_flag()              else:                  if sorted\_dist[0] > 180:        #if the distance to the closest bot is more than 180                      if self.has\_flag:           #if the bot has the flag back to it's own half                          self.turn\_towards(0, self.y)                          self.drive\_forward(Globals.FAST)                      else:                          self.drive\_to\_flag()    #drive to the blue flag                  else:                        if self.point\_to\_point\_distance(Globals.blue\_flag.x, Globals.blue\_flag.y,  self.x,self.y) > 180 and sorted\_dist[0] < 180:  #if the distance between  the bot and the blue flag is more than 180 and the closest bot is less  than 180 away                          self.turn\_towards(-sorted\_positions[0][0], -sorted\_positions[0][1],  Globals.FAST)   #go the opposite direction                          self.drive\_forward(Globals.FAST)                      else:                          self.drive\_to\_flag()                    #drive towards blue flag                          if self.has\_flag:                       #if bot has the flag                              self.turn\_towards(0, self.y)        #go back to their half                              self.drive\_forward(Globals.FAST)                              if sorted\_dist[0] < 200: #if bot is less than 200 away from the  closest enemy                                  self.turn\_towards(-sorted\_positions[0][0], -sorted\_positions[0][1],  Globals.FAST)   #avoid enemy                                  self.drive\_forward(Globals.FAST)          if active == True:              self.enemy\_coords = (close\_coords[0], close\_coords[1])      #update enemy last  coordinates with the current coordinates | 1) Declares the necessary variables that will be used in functions and in the tick  2) sorts the distances, positions and identification of the enemy bots by passing the data through the use of the sort function  3) checks to see ig there are bots in the active\_bots dictionary and if the bot (self) is less than halfway  4) declares the necessary variables used for intercepting enemy bots and sets active to True  5) if there are noo bots in active\_bots and the bot (self) is not less than halfway then it set’s active to False and resorts all necessary variables differently due to active being False  6) adds 1 to the variable that holds a count until it equals the wait variable  8) calculates the linear equations needed to find the intercept and returns the possible intercept points and the distances to those points  9) if the distances are equal or are within a certain bias then go to the intercept points  10) else if the y values given by the linear equations is larger than the screen height then set the y value to max screen height and set the distances to equal and if the y value is smaller than the minimum screen height then set the y value to 0 and make the distances equal  12) if dist\_to\_self is more than dist\_to\_enemy it means that to get the intercept point the x4 value has to be moved to the left (needs to be less than the current x4 value) or if dist\_to\_enemy is more than dist\_to\_self then the x4 value should be moved to the right (or should be more than the current x4 value)  14) once the new values and equations are given, the blue bot travels towards the intercept point although, if the bot is further than the halfway point then turn back to the red flag to prevent the bot from going to jail while intercepting  16) if active is not True then check if the closest bot is more than 180px and if the bot(self) has the flag then drive back towards the blue half of the arena else go towards the blue flag  18) else if the distance between self and the closest bot is more than 180, check if the distance between self and blue flag is more than 180, if it is then turn away from the enemy bot by switching the enemies coordinates to negative  19) if the bot (self) is within 180px of the blue flag, then just drive straight towards the flag and if the flag is retrieved then go back the blue half of the arena, avoiding enemies if they come within 200px  20) update enemies last coordinates now that the tick has finished to the current coordinates of that same tick |
| **BLUE BOT 2 (Interceptor)** | |
| def tick(self):  halfway = Globals.SCREEN\_WIDTH/2  #halfway          dict, dict\_positions = {}, {}          dict\_positions, active\_bots,sorted\_dist,sorted\_bots,sorted\_positions = self.sort(halfway,dict,dict\_positions, Globals.red\_flag)  #sorts the enemies distance and position from shortest to furtherest          if active\_bots: #if there are active bot's              close\_coords = (sorted\_positions[0][0], sorted\_positions[0][1])              distance\_traveled = self.point\_to\_point\_distance(close\_coords[0],close\_coords[1], self.enemy\_coords[0], self.enemy\_coords[1]) #the distance the closest emeny traveled in a Tick              speed = self.get\_speed(distance\_traveled) #calculate the speed of the enemy              my\_x, my\_y = self.x, self.y #set my bots x,y positions in a variable              x4 = my\_x + ((abs(my\_x - close\_coords[0]))/2) #caculate x4 (setting it as halfway between the two bots)              active = True          else:              active = False          if active == True:              if (self.x < (halfway - 80)):                  dist\_to\_self, dist\_to\_enemy,x4,y4 = self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0], self.enemy\_coords[1], close\_coords[0], close\_coords[1]) #calculate the equations needed to get the intercept                  if (dist\_to\_self == dist\_to\_enemy) or ((abs(dist\_to\_enemy - dist\_to\_self)) <= 45): #if the distances are equal and it is an isoceles triangle then drive towards the intercept at the same speed of the bot                      self.drive\_to\_enemy(speed,x4,y4)                  else:                      if y4 >= Globals.SCREEN\_HEIGHT: #if y4 is lager than the screen height set it to the max screen height and make distances equal                          dist\_to\_self = dist\_to\_enemy                          y4 = Globals.SCREEN\_HEIGHT                          self.drive\_to\_enemy(speed,x4,y4)                      elif y4 <= 0:                   #if y4 is smaller than the screen height set it to the lowest screen height and make distances equal                          dist\_to\_self = dist\_to\_enemy                          y4 = 0                          self.drive\_to\_enemy(speed,x4,y4)                        else:                          if (dist\_to\_self > dist\_to\_enemy) and (abs(dist\_to\_enemy - dist\_to\_self) > 45): #if it's not an isoceles                              x4 = my\_x + ((abs(x4 - close\_coords[0]))-(dist\_to\_self-dist\_to\_enemy)) # recalculate x4, moving it left                              dist\_to\_self, dist\_to\_enemy,x4,y4 = self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0], self.enemy\_coords[1], close\_coords[0], close\_coords[1]) #recalculate equations now that x4 has been changed                          elif (dist\_to\_enemy > dist\_to\_self) and (abs(dist\_to\_enemy - dist\_to\_self) > 45):                              x4 = my\_x + ((abs(x4 - close\_coords[0]))+(dist\_to\_enemy-dist\_to\_self)) # recalculate x4, moving it right                              dist\_to\_self, dist\_to\_enemy,x4,y4 = self.calculate(x4,my\_x,my\_y,self.enemy\_coords[0], self.enemy\_coords[1], close\_coords[0], close\_coords[1]) #recalculate equations now that x4 has been changed                            self.drive\_to\_enemy(speed,x4,y4) #drive forward, towards the intercept, matching enemies speed                else:   #if my bot is further than halfway - 80px turn back towards red flag                  self.go\_to\_flag()            else:              if self.point\_to\_point\_distance(Globals.red\_flag.x,Globals.red\_flag.y,self.x,self.y) > 50:  #if there are no active bots, turn back towards red flag, and circle it                  self.go\_to\_flag()              else:                  self.turn\_towards(Globals.red\_flag.x, Globals.red\_flag.y, Globals.MEDIUM)                  self.drive\_forward(Globals.SLOW)          if active == True:              self.enemy\_coords = (close\_coords[0], close\_coords[1])  #set the last coordinates of the enemy to the curRent coordinates | 1) Declares the necessary variables that will be used in functions and in the tick  2) sorts the distances, positions and identification of the enemy bots by passing the data through the use of the sort function  3) checks to see if there are bots in the active\_bots dictionary if there are it declares the necessary variables used for intercepting enemy bots and sets active to True  5) if there are no bots in active\_bots then it sets active to False  8) if active is True, then it calculates the linear equations needed to find the intercept and returns the possible intercept points and the distances to those points  9) if the distances are equal or are within a certain bias then go to the intercept points  10) else if the y values given by the linear equations is larger than the screen height then set the y value to max screen height and set the distances to equal and if the y value is smaller than the minimum screen height then set the y value to 0 and make the distances equal  12) if dist\_to\_self is more than dist\_to\_enemy it means that to get the intercept point the x4 value has to be moved to the left (needs to be less than the current x4 value) or if dist\_to\_enemy is more than dist\_to\_self then the x4 value should be moved to the right (or should be more than the current x4 value)  14) once the new values and equations are given, the blue bot travels towards the intercept point although, if the bot is further than the halfway point then turn back to the red flag to prevent the bot from going to jail while intercepting  15)else if active is not True then it return back to the red flag as there are no bots to intercept |
| **BLUE BOT 3 (Defender)** | |
| def tick(self):  stuck = self.is\_self\_stuck()        #checks to see if the bot is stuck          closest, distance\_to\_flag = self.closest\_to(Globals.red\_bots, Globals.red\_flag) #get's the closest bot to the flag          closest\_to\_self, distance = self.closest\_to(Globals.red\_bots, self)     #get's the closest bot to self          red\_flag = Globals.red\_flag         #set's the red flag as a variable          halfway = Globals.SCREEN\_WIDTH / 2  #sets hafway as half of screen width          if self.point\_to\_point\_distance(0,Globals.SCREEN\_HEIGHT/2,self.x,self.y) > halfway - 100:   #if the bot is further than halfway              self.go\_to\_red\_flag()   #turn back towards the red flag          else:              for friend in Globals.blue\_bots: #for ever blue bot                  if friend.jailed:            #if the a blue bot is jailed                      self.go\_to\_jail(friend)  #go to jail              if stuck == True:                  self.turn\_left()    #if stuck turn left                for bot in Globals.red\_bots:                            #for every red bots                  if bot.has\_flag and bot is not self:                #if bot has flag                      self.turn\_towards(bot.x, bot.y, Globals.FAST)   #turn towards the bot                      self.drive\_forward(Globals.FAST)                #drive forward                if distance < halfway - 80:                             #if the closest enemy bot is less than halfway then go to bot                  self.turn\_towards(closest.x, closest.y, Globals.FAST)                  self.drive\_forward(Globals.FAST)              else:                  if self.point\_to\_point\_distance(red\_flag.x,red\_flag.y,self.x,self.y) > 50:      #if distance to flag is more than 50 then go to flag                      self.go\_to\_red\_flag()                  else:                      self.turn\_towards(Globals.red\_flag.x, Globals.red\_flag.y, Globals.MEDIUM)   #else slowly circle flag                      self.drive\_forward(Globals.SLOW) | 1) Declares the necessary variables that will be used in functions and in the tick  2) if the distance between 0 and half of the max screen height to the bot (self) is more than halfway – 90 then go back to the red flag  3) else check if any blue bots are in jail, if there are then get the bots out of jail  4) if the bot (self) if stuck then turn left  5) for every red bot, check if an enemy has the red flag, and if one does then drive towards the enemy  6) if the distance between self and the closest enemy is less than halfway – 80 then track towards the enemy  7) else if the distance is more than halfway – 80, check if the distance between self and the red flag is more than 50. If it is then go to the flag, else circle the flag slowly |
| **BLUE BOT 5 (Attacker, Jailbreaker, Flag Defender(chaser))** | |
| def tick(self):  halfway = Globals.SCREEN\_WIDTH / 2          #sets hafway as half of screen width          red\_flag = Globals.red\_flag                 #set's the red flag as a variable          dist\_to\_flag = self.dist\_to\_enemy\_flag()    # get's the distance of the closest enemy to flag          stuck = self.is\_self\_stuck()                #checks to see if the bot is stuck            if stuck == True:   #if the bot is stuck then turn left              self.turn\_left()  for bot in Globals.red\_bots:    #for every red bot            if bot.has\_flag and bot is not self:    #if bot has flag                 self.turn\_towards(bot.x, bot.y, Globals.FAST) #go towards bot                 self.drive\_forward(Globals.FAST)          if (Globals.blue\_bots[2].jailed) and dist\_to\_flag > 225:    #if blue 3 is in jail and the bots distance to the flag is more than 225                for friend in Globals.blue\_bots: #for every blue bot                  if friend.jailed:       #if bot is jailed                      self.go\_to\_jail(friend) #go to jail            else:              blue\_flag = Globals.blue\_flag   #set the blue flag to a variable              closest, dist = self.closest\_to\_me()    #get the distance and identifier of the closest bot to the bot              if dist > 180:      #if distance to the bot is more than 18-                  if self.has\_flag:   #if the bot has the flag                      self.turn\_towards(0, self.y)    #turn back towards the blue side                      self.drive\_forward(Globals.FAST)                  else: #if the bot doesn't have the flag then turn towards the blue flag                      self.turn\_towards(Globals.blue\_flag.x, Globals.blue\_flag.y, Globals.FAST)                      self.drive\_forward(Globals.FAST)              else:                  if (self.x < halfway - 100) and (closest.x < halfway):  #if the bot is less than halfway then go towards the cloeset bot                      self.turn\_towards(closest.x, closest.y, Globals.FAST)                      self.drive\_forward(Globals.FAST)                  else:                      if self.point\_to\_point\_distance(Globals.blue\_flag.x, Globals.blue\_flag.y, self.x,self.y) > 180:  #if the diance between the blue flag and the bot is more than 180 then avoid the closest bots                          self.turn\_towards(-closest.x, -closest.y, Globals.FAST)                          self.drive\_forward(Globals.FAST)                      else:                          self.turn\_towards(Globals.blue\_flag.x, Globals.blue\_flag.y, Globals.FAST) #turn toawrds the blue flag and drive fast                          self.drive\_forward(Globals.FAST)                          if self.has\_flag: #if the bot has flag then turn towards the blue half                              self.turn\_towards(0, self.y)                              self.drive\_forward(Globals.FAST)                              closest, dist = self.closest\_to\_me() #update the closest bots                              if dist < 200: #if dist of the closest is less then 200 then avoid it                                  self.turn\_towards(-closest.x, -closest.y, Globals.FAST)                                  self.drive\_forward(Globals.FAST) | 1) Declares the necessary variables that will be used in functions and in the tick  2) if the bot (self) if stuck then turn left  3) for every bot in red bots, if a bot has the flag then go towards the flag  3) if blue bot 3 is in jail and the distance from the bot (self) to the blue flag is more than 225 the get the bot(s) out of jail  4) else Declares the necessary variables that will be used to attack  5) if the distance between self and the closest enemy is more than 180, then check if self has flag, if so then go back to the blue half of the arena, else go to the blue flag  6) else if the distance between self and the closest enemy is less than 180, check if the distance between the bot (self) and the blue flag is more than 180, if it is then avoid the closest bot  7) else go towards the blue flag and check if the bot has retrieved the flag, if it has then go back to the blue half of the arena. Although, if a bot is less than 200px away then avoid the closest bot |
| **Calculate() (blue 2, 1, 4)** | |
| def calculate(self,x4, my\_x,my\_y, last\_sorted\_positions\_x,  last\_sorted\_positions\_y,sorted\_positions\_x,sorted\_positions\_y):  m = self.gradient(last\_sorted\_positions\_x,last\_sorted\_positions\_y,  sorted\_positions\_x,sorted\_positions\_y) #get's the gradient          c = sorted\_positions\_y - m\*sorted\_positions\_x #rearranges y=mx+c to get c          y4 = abs(m\*x4 + c) #get the y4 value by substituting x4 into y=mx+c          dist\_to\_self = self.point\_to\_point\_distance(my\_x,my\_y,x4,y4) #get's the distance between self and intercept          dist\_to\_enemy = self.point\_to\_point\_distance(sorted\_positions\_x,sorted\_positions\_y,x4,y4) #get's the distance between enemy and intercept          return dist\_to\_self,dist\_to\_enemy,x4,y4 | 1) calculates the gradient of the enemies last position (enemies position in the last tick) and the enemies current position  2) get c by rearranging y=m(x4) + c and putting the enemies current x,y coordinates into the equation  3) calculates y = m(x4) + c now that c has been found and turns it into a positive integer  4)gets the distance of the bot(self) to the intercept points given and the distance of the enemy to the intercept points  4) returns the values calculated |
| **Gradient() (blue 2,1,4)** | |
| def gradient(self,x1,y1,x2,y2):  try:              gradient = (y2 - y1) / (x2 - x1) if (x2 - x1) != 0 else 0   #calculates the gradient of  two points if it doesn't == 0, else it set's the gradient as 0          except ZeroDivisionError:   #if an error occurs due to division by zero then return infinite              gradient = math.inf          return gradient | 1) try’s to calculate the gradient of the points passed into the function if the result of (x2 - x1) does not equal 0, if it does then set the gradient to 0  2) if a ZeroDivisionError does occur when calculating the gradient then set the gradient to infinity |
| **Sort() (blue 1,2,4)** | |
| def sort(self, halfway,dict,dict\_positions, object, active):        for enemy in Globals.red\_bots:                  #for each enemy in red bots              enemy\_distance = self.point\_to\_point\_distance(object.x,object.y, enemy.x, enemy.y)  #get the distance between the object and the enemy              dict[enemy] = enemy\_distance                # put the enemy's indentifier and the distance into dict              dict\_positions[enemy] = [enemy.x,enemy.y]   # put the enemy's indentifier and it's position into dict\_positions            dict = {key: val for key, val in sorted(dict.items(), key = lambda ele: ele[1])}                        #sort dict values from lease to most          dict\_positions = {key: val for key, val in sorted(dict\_positions.items(), key = lambda ele: ele[1])}    #sort dict\_position values from least to most            sorted\_bots = list(dict.keys())  #put the values and keys into lists          sorted\_dist = list(dict.values())          sorted\_positions = list(dict\_positions.values())                if active == False:         #if there are active bot's and if my bot is less than halfway              active\_bots = {}        #return an epemty dictionary              return sorted\_bots,sorted\_dist,sorted\_positions, active\_bots          else:              i = 0              active\_bots = {}                    #create an empt dictionary              while i < len(dict.keys()):         #for every element in the dictionary                  if sorted\_positions[i][0] < halfway:    #if the current bots (bot[i]) x value is less than halfway                      active\_bots[sorted\_dist[i]] = [sorted\_positions[i][0],sorted\_positions[i][1]]   #add the distance of that bot and it's position to the active\_bots dictionary                  i = i + 1                sorted\_bots = list(dict.keys()) #put the values and keys into lists              sorted\_dist = list(dict.values())              sorted\_positions = list(dict\_positions.values())                return dict\_positions, active\_bots,sorted\_dist,sorted\_bots,sorted\_positions | 1) for every red bot calculate the distance from that bot to the object passed into the function add the enemy’s identification and distance to the object to dict and add the enemies identification and coordinates to dict\_positions  2) sort both dictionaries using lambda expression, sorting the data from smallest integers to largest (closest to furthest)  3) put the keys and values of the dictionaries into lists so that they can be called upon using arrays  4) checks if active is false, and if it is then return an empty and return all necessary lists and dictionaries as there are no enemy bots or that the bot (self) if in the red’s half of the arena  5) if active is true then create an empty dictionary and cycle through all the values (enemies) in dict and check if the enemy is in the blue half of the arena, if it is then add the bot’s indetification and coordinates to the active\_bots dictionary    6) relist the keys and values of the dictionaries into lists so that they can be called upon using arrays and return all the necessary list and dictionaries |
| **Drive\_to\_enemy() (blue 2, 1, 4)** | |
| def drive\_to\_enemy(self,speed,x4,y4):          self.turn\_towards(x4,y4, Globals.FAST)   #turn towards the intercept point          if speed == 1:  #matches the speed of the bot              self.drive\_forward(Globals.SLOW)          elif speed == 2:              self.drive\_forward(Globals.MEDIUM)          else:              self.drive\_forward(Globals.FAST) | 1) turn towards the intercept point given by the equations  2) checks if the speed == 1,2 or 3 and matches the speed accordingly |
| **Get\_speed() (blue 2, 1, 4)** | |
| def get\_speed(self, distance):          speed = distance  #sets the speed to the distance          if speed <= 3: #if speed is less than or equal to 3 then the bot is going slow              return 1          elif speed > 3 and speed <= 7: #if speed is more than 3 but less than or equal to 7 then the  bot is going medium speed              return 2          elif speed > 7: #if speed is more than 7 then the bot is going fast              return 3 | 1) sets the speed to the distance between two points as correlates with the speed of the enemy bot  2) checks if the speed matches the values of one of the speeds set in globals.py and returns a number depending of the speed |
| **Closest\_to() (blue 3,5)** | |
| def closest\_to(self, object, target):          closest = object[0] #set closest to the first object          shortest\_distance = self.point\_to\_point\_distance(closest.x,closest.y,target.x,target.y) #get  the distance between the object and the target          for objects in object: #for every object              if self.point\_to\_point\_distance(objects.x,objects.y,target.x,target.y) <  shortest\_distance: #if the distacne between the new object and the target is less                  closest = objects #set closest to the current object                  shortest\_distance = self.point\_to\_point\_distance(closest.x,closest.y,target.x  ,target.y) #set the distance to the distance between the new object and the target          return closest, shortest\_distance | 1) set closest to red bot 1 and set shortest\_distance to the distance between red bot 1 and the target passed into the function  2) for every object check if the distance between the object and target is less than the shortest\_distance and if it is update closest to that object and shortest\_distance to the distance between that object and the target  3) once all objects have been iterated over return closest and shortest\_distance |
| **Go\_to\_jail() (blue 3, 5)** | |
| def go\_to\_jail(self,friend):          stuck = self.is\_self\_stuck()        #check to see if bot is stuck          if stuck == True:                   # if bot is stuck              self.turn\_towards(friend.x,friend.y, Globals.FAST)  #turn towards friend          elif self.x <= 20 and (self.angle >= 80 and self.angle <= 100): #if the bot is stuck on wall  with the lowest x axis              self.turn\_towards(friend.x, 0, Globals.FAST)    #turn towards friend.x and o          elif self.y <= 20 and ((self.angle <= 10 and self.angle >= 350) or self.angle < 0): #if the  bot is stuck on wall with the lowest y axis              self.turn\_towards(0, friend.y, Globals.FAST)    #turn towards 0, friend y          else:              self.turn\_towards(friend.x,friend.y, Globals.FAST) #if the bot is not stuck then slowly  navigate to the friend in jail              self.drive\_forward(Globals.SLOW) | 1) Call upon is\_self\_stuck() to check if the bot (self) is stuck and check if stuck is true if it is then turn towards the blue bot in jail  2) if the bot is stuck on wall with the lowest x axis and the bots (self) angle is more than 80 but less than 100 turn towards friend.x and 0  3) if the bot is stuck on wall with the lowest y axis and the bots (self) angle is less than 10 and more than 350, then turn towards 0 and friend.y  4) else just turn towards the blue bots in jail and drive forward slow |
| **Is\_self\_stuck() (blue 3,5)** | |
| def is\_self\_stuck(self):          if not self.jailed:           #if the bot is not jailed              if self.x <= self.width:  #if the bot's x value is less than it's width then it is stuck                  return True              elif self.x >= Globals.SCREEN\_WIDTH - self.width:   #if the bot's x value is more than  screen width - the bots width then it is stuck                  return True              if self.y <= self.height: #if the bot's y value is less than it's height then it is stuck                  return True              elif self.y >= Globals.SCREEN\_HEIGHT - self.height: #if the bot's y value is more than  screen height - the bots height then it is stuck                  return True              else:                  return False     #else the bot is not stuck | 1) checks to see if the bot (self) is not in jail  2) checks if the bots x coordinate is less than its width (lowest screen width (0) + self.width) and if it is then it returns true as the bot is stuck  3) if the bots x coordinate is more than the maximum screen width – the bots width then return true as the bot is stuck  4) if the bots y coordinate is less than its height (lowest screen width (0) + self.height) then the bot is stuck  5) if the bots x coordinate is more than the maximum screen height – the bots height then return true as the bot is stuck  6) if the bots coordinates do not match these conditions return false as the bot is not stuck |

EVALUATE

## TEST

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TEST** | **TEST DATA** | **EXPECTED OUTCOME** | **ACTUAL OUTCOME** | **RECOMMENDATIONS** |
| A Test to get the best path to the blue flag | def get\_path(self,grid,end,start,weights,active\_bots, blocked,dict\_path):          weight\_pos = list(weights.keys())          weight\_dist = list(weights.values())          dict\_temp = {}          if start == end:              # self.drive\_backward()              print(len(grid))              print("START: ", start, "END: ", end)              print("---------------------- WORKED! ---------------------")              return dict\_path          else:              if end in blocked and ((abs(start[0] - end[0]) <= 120) and (abs(start[1] - end[1]) <= 120)):                  print(len(grid))                  print("START: ", start, "END: ", end)                  print("---------------------- WORKED! ---------------------")                  return dict\_path              else:                  find = True                  while find == True:                      i,j=0,0                      while i <= len(weights):                          if start == end or (end in blocked and ((abs(start[0] - end[0]) <= 120) and (abs(start[1] - end[1]) <= 120))):                              find = False                              break                          else:                              if (start[0] == (weight\_pos[i][0])) and (start[1] == (weight\_pos[i][1])): #works                                  x=0                                  while x <= len(weights):                                      if (((weight\_pos[i][0] + 40) == weight\_pos[x][0]) and ((weight\_pos[i][1] + 40) == weight\_pos[x][1]) or ((weight\_pos[i][0] + 40) == weight\_pos[x][0]) and ((weight\_pos[i][1]) == weight\_pos[x][1]) or ((weight\_pos[i][0] + 40) == weight\_pos[x][0]) and ((weight\_pos[i][1] - 40) == weight\_pos[x][1]) or ((weight\_pos[i][0]) == weight\_pos[x][0]) and ((weight\_pos[i][1] - 40) == weight\_pos[x][1]) or ((weight\_pos[i][0] - 40) == weight\_pos[x][0]) and ((weight\_pos[i][1] - 40) == weight\_pos[x][1]) or ((weight\_pos[i][0] - 40) == weight\_pos[x][0]) and ((weight\_pos[i][1]) == weight\_pos[x][1]) or ((weight\_pos[i][0] - 40) == weight\_pos[x][0]) and ((weight\_pos[i][1] + 40) == weight\_pos[x][1]) or ((weight\_pos[i][0]) == weight\_pos[x][0]) and ((weight\_pos[i][1] + 40) == weight\_pos[x][1])):                                          temp = list([weight\_pos[x][0],weight\_pos[x][1]])                                          j = j + 1                                          print(j)                                          if temp not in blocked: #test                                              print("not blocked")                                              dict\_temp[weight\_dist[x]] = (temp) #test                                          if j == 8: #test                                              dict = {key: val for key, val in sorted(dict\_temp.items(), key = lambda ele: ele[0])}                                              print(dict)                                              weighted\_pos = list(dict.values())                                              weighted\_dist = list(dict.keys())                                              start = [weighted\_pos[0][0], weighted\_pos[0][1]]                                              dict\_path[weighted\_dist[0]] = [weighted\_pos[0][0], weighted\_pos[0][1]]                                              print(dict\_path)                                              print("START: ", start, "END: ", end)                                              self.get\_path(grid,end,start,weights,active\_bots, blocked,dict\_path)                                              find = False                                              break                                      x=x+1                              i=i+1 | The algorithm would search over the grid and would get the surround blocks of the start position of the blue bot. if the block is not blocked (is not on or near a enemy) then Once all these blocked have been gathered then it finds the one with the lowest weight (distance to the blue flag). Once it has been found then it updates the start variable to the grid with the lowest weight and adds it to the path dictionary. Then this algorithm is repeated until the start block equals the end block or is near the end block if the end block has an enemy on it. | While it would sometimes work, it would often freeze the game while running due to the while loops and the amount of data being processed. In addition, sometimes it would not be able to find the end block and would be stuck in a constant loop that would crash the game. | If a more efficient code was used that used arrays instead of dictionaries it would reduce the amount of necessary code but would also be more complex. In addition, if the while loops were changed and additional conditions to detect when it cannot find the end would prevent the game from crashing and freezing |
| A test to get the gradient of two points | def gradient(self,x1,y1,x2,y2):          gradient = (y2 - y1) / (x2 - x1)          return gradient | Returns the gradient of two points | While most of the time it did return the gradient, when there were no bots in active\_bots and this function was still being called upon an error would occur stating ‘ZeroDivisionError’ | To prevent this you can put conditions in to detect if the denominator is zero and also put a try and except in to further prevent this issue from occurring |
| A test to sort  different distance based on the object given when the function gets called | def sort(self, halfway,dict,dict\_positions, object, active):        for enemy in Globals.red\_bots:                  #for each enemy in red bots              enemy\_distance = self.point\_to\_point\_distance(self.x, self.y, object.x, enemy.y)  #get the distance between the object and the enemy              dict[enemy] = enemy\_distance                # put the enemy's indentifier and the distance into dict              dict\_positions[enemy] = [enemy.x,enemy.y]   # put the enemy's indentifier and it's position into dict\_positions            dict = {key: val for key, val in sorted(dict.items(), key = lambda ele: ele[1])}                        #sort dict values from lease to most          dict\_positions = {key: val for key, val in sorted(dict\_positions.items(), key = lambda ele: ele[1])}    #sort dict\_position values from least to most            sorted\_bots = list(dict.keys())  #put the values and keys into lists          sorted\_dist = list(dict.values())          sorted\_positions = list(dict\_positions.values())                if active == False:         #if there are active bot's and if my bot is less than halfway              active\_bots = {}        #return an epemty dictionary              return sorted\_bots,sorted\_dist,sorted\_positions, active\_bots          else:              i = 0              active\_bots = {}                    #create an empt dictionary              while i < len(dict.keys()):         #for every element in the dictionary                  if sorted\_positions[i][0] < halfway:    #if the current bots (bot[i]) x value is less than halfway                      active\_bots[sorted\_dist[i]] = [sorted\_positions[i][0],sorted\_positions[i][1]]   #add the distance of that bot and it's position to the active\_bots dictionary                  i = i + 1                sorted\_bots = list(dict.keys()) #put the values and keys into lists              sorted\_dist = list(dict.values())              sorted\_positions = list(dict\_positions.values())                return dict\_positions, active\_bots,sorted\_dist,sorted\_bots,sorted\_positions | If the red flag was given as the object then the function should get the distance to the enemy from the red flag | The function would provide incorrect distances. Instead the variable to calculate the distance would calculate the distance from self to the object, instead of enemy to the object | To fix this if self.x and self.y were changed to enemy.x and enemy.y it would calculate the proper distance |

## 

## User Testing

|  |  |
| --- | --- |
| Can the user open the program? | [✓] |
| Can the bot move? | [✓] |
| Can the bot get blue bots out of jail? | [✓] |
| Can the bot change speed? | [✓] |
| Can the bot turn left, right and towards an object? | [✓] |
| Can the bot retrieve the flag? | [✓] |
| Can the bot get enemy bots out? | [✓] |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **PRESCRIBED CRITERIA** | | | **SELF-DETERMINED CRITERIA** | | |
| **PC1** | Must make a folio showing exploration, Analysis, Development, Synthesis and Evaluation. | [✓] | SDC1 | Using arrays and lambda functions to sort the enemy’s distance, identification, and position | [✓] |
| **PC2** | Must code 10 bots to compete in CTF | [✓] | SDC2 | Using lists, arrays, and dictionaries to store data | [✓] |
| **PC3** | Must use functions/ methods defined by parent classes and adhering to rules of engagement | [✓] | SDC3 | Using vectors and linear equations to intercept the attacking enemy bots | [✓] |
| **PC4** | Must create a 4 min video explaining your teams’ strategies | [✓] | SDC4 | Machine Learning to avoid enemy bots when retrieving the flag | [×] |
| **PC5** | Must anticipate unseen enemy teams’ strategies and create code that will react accordingly | [✓] | SDC5 | Modular Bots to increase or decrease the amount of defender or attackers | [✓] |
|  |  |  | SDC7 | Updating the enemy bot’s locations every tick to determine the best path using weighted grid to get the best path to the flag | [×] |
|  |  |  | SDC8 | Creating attacking formations to efficiently get the flag | [×] |

## Variable Table

|  |  |  |
| --- | --- | --- |
| **VARIABLE NAME** | **VARIABLE TYPE** | **DESCRIPTION** |
| Halfway | integer | This variable stores half the screen width. It is used to see if enemy bots are in our half or not. |
| Closest | string | It is the identification of the closest bot and is used to locate and track the bot |
| dist | integer | It is the distance from self and the closest enemy bot, and can be used to either stay away from the enemy bot or chase it |
| dict | dictionary | Stores each enemy and their distance, it then gets sorted in order to lowest distance to largest |
| Dict\_positions | dictionary | Stores each enemy and their coordinates, and is sorted from closest to furthest |
| Sorted\_dist | list | Stores a list of sorted distances from a given point to each enemy bot |
| Sorted\_postitions | list | Stores a list of sorted coordinates of the enemy bots |
| Active\_bots | dictionary | Active bot’s stores bots that are in a particular area of the arena, and stores their distance and position |
| Dist\_jail | integer | Stores the distance of the closest friendly bot to jail, and is used to make the closest bot go to jail |
| stuck | Boolean | Checks to see if the bot is stuck on the edges of the arena |
| M (gradient) | integer | Calculates the gradient of the last and current position of the enemy bot |
| x4, y4 | integer | These are the coordinates of the intercept. x4 is calculated by picking precise points along the x axis of the blue bot and enemy bot. y4 is calculated by putting x4 into y=m(x4) + c |
| speed | integer | The speed variable is used to get the speed of the enemy bot. the speed is calculated by finding the distance between the enemies last position and current position |
| my\_x, my\_y | integer | These variables are used to store the blue bots position at the start of the tick in order to get precise calculations. This is due to the blue bot possibly moving and changing it’s x,y values |
| active | Boolean | This is used to tell the bot when to intercept bots and when to attack the blue flag or return to the red flag depending on the bot |
| Dist\_to\_self and dist\_to\_enemy | integer | These variables store the distance between the blue bot and the intercept point and the enemy bot and the intercept point. It is used to determine if the intercept given by the equations is equal in distances (to see if it creates an isosceles triangle) |

Impacts

## Social

When AI takes over repetitive or dangerous tasks such as jobs in warfare, it frees up the human workforce to do work they are better equipped for. This allows people to better connect with people outside of a bloodshed envirournment. Additionally, Jobs outside of warfare as a result of the use of AI will appear due to the need for designers, engirneers, construction workers, and much more.

## Personal

With the adoption of artificial intelligence there would be a higher chance of wars, due to the limited risk of human life, because everything would be controlled by robots. Additionally, there would be an increase of wars due to the ability for robots to be specialized in particular task. Although, due to Robots not having conscious thoughts, PTSD and other mental disorders caused from war would be vastly lowered.

## Economic

Due to the vast capabilities of AI and robots these bots would outweigh the use of humans in warfare positions. This would irradicate the jobs of millions of people and heavily impact the economical structure of the country using the weaponry but the country being invaded by it too. To counteract the loss of jobs the need for additional funding to provide the people who lost their job to the use of AI in warfare will aid them to find a new job.

## Legal

The legal implications of AI in warfare is limitless due to the ethical status of the situation. Due to the large impacts a war with AI weaponry can cause, the implication of legislations will have to subsidize the use of this machinery. Such laws could consist of how many a country can have at any one time, the deployment of this machinery, and much more. Additionally, algorithmic openness, cybersecurity flaws, unfairness, bias, and discrimination, lack of contestability, issues with legal personhood, issues with intellectual property, detrimental effects on workers, privacy and data protection issues, liability for harm, and a lack of accountability are just a few of the legal issues with machine learning

S

# Additional Evaluation

* Extensions
* Pitfalls-what held you back when coding.
* Success criteria you did not achieve and why – impact of these on the A.
* Any failed tests and how you fixed them ‘
* Any learn in the project – creating functions / modules, inputs, and outputs, how to create extended datatypes, the 4 main basic datatypes.

## Risks Taken

Syntax errors were a significant risk to the overall success of running the program. Additionally, common errors while coding was an apparent issue due to the only debug information being ‘blue exception error’. This became a time-consuming exercise due to having to spend extra time finding the issue. Lack of knowledge and understanding on how to efficiently fix errors limited the capability of the code. However, the errors helped discover new parts of Python programming so the program could run without errors. Machine learning was also a major risk to the finalizing of the code due to the amount of time spent on it. After researching much about machine learning, however, it was limited because when testing the code, the machine learning always froze the game and made the files really big and hard to run. Path finding algorithms turned out to be a vast risk. Unfortunately, after spending hours upon hours trying to find a way to make it work, in the end it didn’t work, risking time for completing other tasks in this assignment. Lastly, while bots that intercept enemies was achieved, due to the lack of knowledge in this department, finding a solution was extremely difficult. After an abundance of research and testing, it was decided that vectors were not a viable solution and instead the use of linear equations were applied. Finally, the time taken trying to develop attacking formations, reduced the time that could have been put into the research and development of pathfinding algorithms or machine learning as it was not able to be done due to the limited amount of knowledge.

## What Was learnt

The program utilizes python to code different robots to use strategies to efficiently capture the opposing flag. This means that the code has successfully met the prescribed criteria. The coding experience was challenging, as the code needed to be continuously developed to remove errors and keep the robots functioning. Some difficult errors to resolve were organizing inefficient code. The issues were resolved by using functions, lists, arrays, and dictionaries which were advised as part of the Self determined criteria. Correcting these errors shortened and structured the code that made it neat and efficient. Another error that caused problems was. To solve this problem variables had to be globalized throughout the code. Using the newfound knowledge helps learning about more efficient ways to program, making programming easier in the future. Trying to solve the path finding was the most complex aspect of the code, learning how to use path finding algorithms will continue to aid in many other tasks involving robots in the future. Unfortunately, the lack of knowledge and the limited amount of time, prevented the path finding technology to be fully functional. Mathematical equations were used to find intercepts using linear equations, which proved to be one of the more difficult tasks which got completed.

## Improvements and Limitations

The following improvements would have assisted the development of the code and shortened how long it took to write the program, and can be linked back to the self-determined criteria:

* Take more care with indents and syntax errors from the beginning of the programming, to eliminate errors and save time. This would also help to develop the program in a better time frame.
* Run the program regularly to error check. Checking for error frequently is helpful to get through the code a complete it quicker.
* Trying to get feedback early on in production would have been useful instead of asking for help at the last second.
* Adding machine learning to improve the efficiency of the code, this would have not only have improved the code, but it would also have displayed a great understanding of advanced coding. Unfortunately, this was challenging to add due to the complexity and the lack of time giving to complete such a complex task.
* Pathfinding is a high level of AI understanding of machine technology to find the best and most efficient method of moving from A to B. This would have not only have improved the code, but it would also have displayed a great understanding of advanced coding. Unfortunately, this was challenging to add due to the complexity and the lack of time giving to complete such a complex task.
* There was a difficult task with preventing bots from getting stuck on the edges of the map. The code does prevent this issue; however, it was quite difficult and complex. Finding an easier way to complete this would have shortened the time and made it more reliable.
* Globalizing functions, so the functions don’t have to be copied and pasted into each bot’s file. This is technically against the rules; however, it would have saved time to complete other parts of the code.
* Having a modular “closest” function that incorporates all versions would be helpful, instead of having several variants of that one type. Currently there are a few more versions but can be simplified down to one; however, the limited time was a major factor. In addition, it would also improve the efficiency and making the code “look better.”

## Real World Application

Being able to program AI systems to execute activities without involving any human intervention is tremendously helpful in a future where AI-heavy lifestyles are becoming more and more common. The warfare industry will not need to be compiled with the deaths of millions of solders with the use of machine learning. As AI can react in a fragment of a second will pin-point accuracy, the implications for AI capable machine sin the miliary is immense. Additionally, the jobs created by the use of AI in warfare will increase economic growth globally. Machine learning and artificial intelligence algorithms will soon perform many dangerous jobs and responsibilities totally, or at the very least in part, even outside of the military. However, algorithmic openness, cybersecurity flaws, unfairness, bias, and discrimination, lack of contestability, issues with legal personhood, issues with intellectual property, detrimental effects on workers, privacy and data protection issues, liability for harm, and a lack of accountability are just a few of the legal issues with machine learning and prevent it from entering dangerous spaces were these issue are critical.

# References

Haponik, A. (2019, October 18). *Machine learning in Economics – How is it Used?* Retrieved from addepto : https://addepto.com/blog/machine-learning-in-economics-how-is-it-used/#

*How machine learning works*. (2020, January 7). Retrieved from DataRobot: https://www.datarobot.com/blog/how-machine-learning-works/#:~:text=How%20does%20machine%20learning%20work,and%20involves%20minimal%20human%20intervention.

*Machine Learning*. (2020, July 15 ). Retrieved from IBM Cloud Learn Hub: https://www.ibm.com/cloud/learn/machine-learning

*Neural Networks* . ( 2020, August 17 ). Retrieved from IBM Cloud Learn Hub: https://www.ibm.com/cloud/learn/neural-networks#:~:text=Neural%20networks%2C%20also%20known%20as,neurons%20signal%20to%20one%20another.

Rodrigues, R. (2020). *Legal and human rights issues of AI: Gaps, challenges and vulnerabilities*. Retrieved from Science Direct : https://www.sciencedirect.com/science/article/pii/S2666659620300056

Singh, V. (2020, February 18). *How Machine Learning Is Changing the World*. Retrieved from Data Science Central : https://www.datasciencecentral.com/how-machine-learning-is-changing-the-world/#:~:text=Machine%20learning%20is%20changing%20the%20world%20by%20transforming%20all%20segments,shopping%2C%20food%20ordering%2C%20etc.