Bot AI for Agar.io Game

Aditya Shirode, Anshul Deshkar, Ronak Ghadiya, Varun Jayathirtha
Department of Computer Science, North Carolina State University
{avshirod@,adeshka@,rghadiy@,vjayath@}ncsu.edu

Abstract

Agar.io is web-browser based multiplayer, real-time strategy game. The player (bot) and its opponents are circular cells who wander throughout the environment. You start as a solitary cell. The objective of the game is to grow larger in size, achieved by consuming smaller opponents and food (aka pellets), while staying away from bigger opponents to avoid being eaten. The environment also consists of viruses, which are stationary.

This project aims at developing a bot to play this game as efficiently as possible. The bot can recognize positions of other cells in the visible environment by continuous image processing. After perceiving the environment, it will apply various techniques discussed later in this report to achieve the game objectives as intelligently as possible. Various Game AI algorithms, combined with decision making, decide the next movement as input for the mouse pointer, and a decision to shoot or split. Our main goal is to use a combination of AI algorithms with tweaked and tested parameters to make the bot a good player of the game and achieve a set of defined goals.

Introduction

This game depends heavily on movement techniques such as seek, flee, collision avoidance, etc. and decision making as to which action gets priority and how arbitration works in different situations. We have come up with different techniques to handle these problems and would experiment on what combination of these techniques would be most efficient in achieving the goals of the game.

Background

Agar.io is a web-based MMO action game created by Matheus Valadares. Players start of as a single solitary cell in a map representing a petri dish; the goal is to gain as much mass as possible by swallowing smaller cells without being swallowed by bigger ones. The top ten players in an instance are given a position on the instance's Leaderboard. Currently, there is no real way to 'win', as matches go on indefinitely until the game instance restarts. For most players the objective is to hold the top position for as long as possible. You can play in four modes - FFA (Free For All), Teams, Experimental, and Party. In this project, we are considering FFA mode.

Objective

The objective of Agar.io is to grow your cell by swallowing food pellets and smaller cells, while avoiding bigger cells to stay alive. Ultimately, one's aim is to grow, whilst staying alive for as long as possible by using various abilities that the player has. Playing the game is the best way to understand the game mechanics. The details about the game that are mentioned in the next section are essential to the core game logic and hence play major role in the bot reaching its objective.

Game Environment

The speed of a cell is inversely proportional to the size of the cell. When a player eats smaller opponents, the player gains mass, and the size of the cell grows, making it slower. Cells gradually lose mass over time. Every player has two special abilities - a) Splitting in halves (by pressing 'W'), on doing so the two halves move side by side at a higher speed; b) Shooting out some of its mass (by pressing 'Space'). All playing objects need to avoid viruses (circular objects with jagged edge) smaller than themselves, as a significant collision with a virus would make them split into many smaller cells. The bot is always in the center of the visible screen and moves in the direction of the mouse pointer. If the player is split into many cells, all cells will still move toward cursor. The speed and steering behavior is handled by the game. As long as the mouse pointer is outside the bot's body, there is only one fixed speed. Once the mouse pointer goes within the region of the bot, it gets slower and stops when the mouse goes to the center of the bot. Also, the size of other objects is proportional to our size. After all cells of the bot are swallowed by enemy (or enemies), the game ends.

Tasks and Techniques

Our program detects the game environment, processes a screenshot to detect visible players' cells, and decide which direction to move in. Then it uses *java.awt.robot* class to move the mouse pointer and navigate the cell around.

We start the Java program for the bot, with an introduced delay equal to switching over to the browser and starting the game. There are various customizable settings in the game. For this bot to run, you have to turn the 'No skins', 'No colors', 'Dark theme', and 'No names' options ON. That can be done using the Settings option before starting the game.

The screen area to be considered is dependent on the screen resolution. Right now we are working for a single system with 1280x720p screen. At this resolution, on a Windows machine, you ignore the 85 pixels from top (Top of Browser window) and 50 pixels from bottom (Windows Task Bar). We are planning to make changes to the program screen detection class to make it work on any system/machine/resolution combination (Best way is to making it work on Full Screen mode in Chrome).

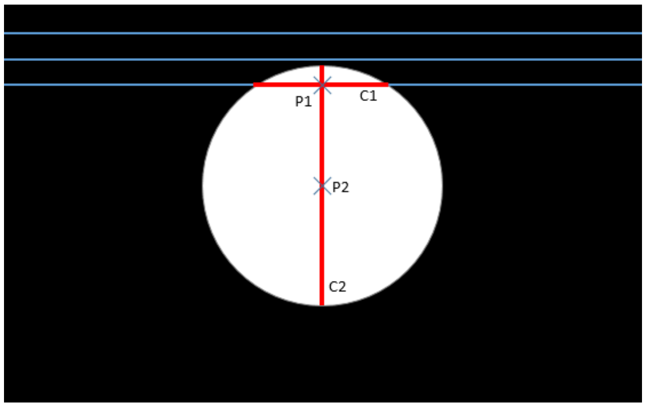
Image Processing

We consider only the specific game environment area in the screenshot that is the entire computer screen. Detection.java works on the black and white image obtained from the screenshot, containing various circles. The background is black. The bot (our self), other players and food pellets all are white circular pellets. Viruses are also white and circular, but with spiked border. We store the image as a two dimensional array of pixels, spread along the x- and y- coordinates.

The circle detection algorithm starts scanning from left top corner, going left to right. Since smallest items on the screen (pellets) have radius of at least 5 pixels, It can skip 5 pixels as it moves down, to reduce the computational complexity. In each line, it scans for the first occurrence of a white pixel which is surrounded by white pixel from all direction (indicating we have encountered some object). First such point, would be the point somewhere in the cell. To get the dimensions of the cell we need to locate the center of pellet and its radius. For this we traverse to the left and right till we encounter a color change to black. This line segment is a horizontal chord (c1) in the circle detected. From the center(p1) of this chord we move up and down till we detect a color change to black from white. This is a vertical chord(c2) in the circle (possibly the diameter). We take the length of the larger of the horizontal and vertical chords and assume that to be the diameter (i.e. half of it as the radius). The center(p2) of this vertical chord is considered as the center of the circle. After any circle is found, algorithm will set all of its points to black color so that these point will not considered again.

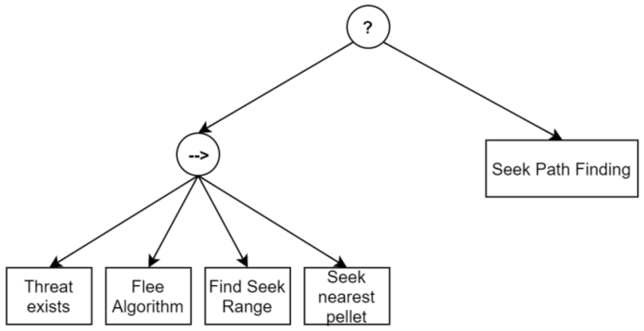
This algorithm will also work for cases where multiple enemies are overlapping. Since the cell's boundary has a grey pixel, our algorithm can easily find when any particular circle starts overlapping. In this case, it will find the vertical and horizontal chord only for the portion of first circle which is non-overlapped. It will select the bigger of the vertical and horizontal chord and will treat that cell as a circle of that diameter. Since, we are setting all points of the circle as black pixels, detection of second circle will not be difficult. The second circle will detected as normal.

Before starting to scan the image for enemies, viruses and pellets, the algorithm will scan the bot. Since the bot is always in the center of the screen the algorithm makes it a point to detect ourselves by moving to center of screen. As mentioned above, it will traverse in left and right direction to find the horizontal chord and similarly the vertical chord. We would store the bot's details in the first object of circles' list. After finding all circles, we store the detected circles in a list, with center coordinates, radius and a certain weightage value. This value is determined by logic described later in this report.

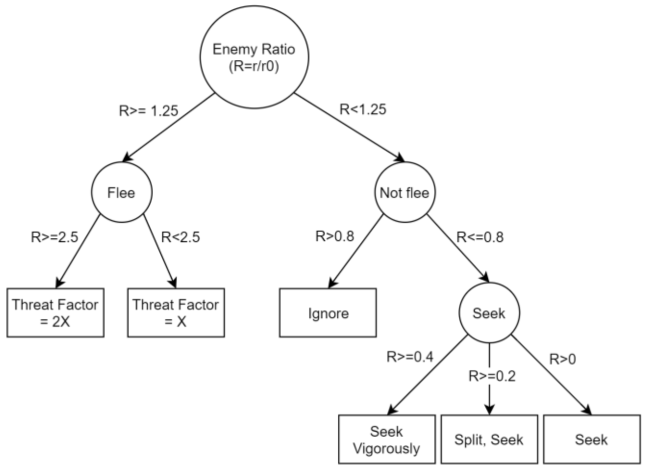


AI Techniques for Decision Making

The most important decision to be made in this game is when to flee from an enemy. We speculate that for the best performance, we must first stay away from danger and then try to consume pellets or smaller enemies. This decision is taken based on the threat level at every point in time. In most cases, there are threats based on which our 'flee' algorithm generates a 'flee vector'. The next step is to decide what to seek. The bot can seek pellets or smaller enemies in the general direction of the flee vector. For this there is another decision to be made. How much freedom does the bot have to waver from the general flee direction. This is decided by the amount of threat that is there in that region. The following Composite Task tree shows the flow of Decision Making in the program.



We use the following decision tree to determine which enemies is classified as Threat and which enemies to seek.



Flee Decision Making

Another important decision that is to be made in the game is to decide the intensity of flee, which depends on the radius ratios of the enemies to that of the bot itself. An enemy poses a threat when it has 1.25 times more mass than the bot. If this mass ratio is in range of 1.25 to 2.5, then our bot is a potential food for them. If the mass ratio is higher than 2.5 times, the enemy can split and plunge one if its halves towards us making them a more serious threat. So we need to flee from these enemies as quickly as possible. How far the enemy is also affects the fleeing factor. The closer the enemy, the more threat it possesses. If an enemy is too big, we do not have to worry about them too much as, we are not a target worth seeking for such an enemy. In general, threat from an enemy is inversely proportional to the distance and size of the enemy.

AI Techniques for Movement

After detecting all the players and pellets, we need to apply AI techniques to implement bot behavior in the game.

Flee

After image processing and circle detection part, we calculate the distance of each circle from the bot. After all the distances are found, a net ‘flee vector’ is generated by adding the displacement vectors of the enemies multiplied with their respective weight factors. This flee vector will give the direction in which the bot should flee. After normalizing, we can return the result as *(x,y)* coordinate pair. In case fleeing were the only objective of the game, this output could be passed to the actuator which sets the mouse pointer to that position after adjusting for the screen resolution.

Algorithm:

```
flee(circleList) {
Vector newLocation
  bot = circleList[0]
for enemy in circleList[1] → circleList[n-
1]:    // n: number of circles detected
             xDelta=(bot.x-
enemy.x)*enemy.weightage
             yDelta=(bot.y-
enemy.y)*enemy.weightage

// xDelta, yDelta each have the x and y com-
ponents of  the flee factor, need to normal-
ize
magntude=Euclidean_distance(xDelta,yDelta,0,
0)

// Find the normalized position and shift it
onto 200 pixel circle, to limit the mouse
movement to a designated game area
xDelta=200*xDelta/magnitude
yDelta=200*yDelta/magnitude

// Add offset to move the new location to
the center of the game environment
newLocation[0]= (bot.x + xDelta)
newLocation[1]= (bot.y + yDelta)
newLocation[2] = magnitude

return newLocation;
}
```

After calculating xDelta and yDelta, we calculate its Euclidean distance from the (0,0) point and store it as magnitude. For moving our bot player into the new calculated direction, mouse pointer should be in the direction from the center of the environment. This can be achieved by firstly, calculating the normalized vector and multiplying with the 200. This multiplication of 200 is done to shift the mouse pointer outside the bot's boundary. This value is chosen approximately and any value in range from bot's radius and half of dimensions of the game environment can be used. This new location point has to be given reference to the bot's current location and since bot is always at the center of the environment, we need to add offset of bot's location into newLocation. Magnitude of the flee vector is also stored because it will be needed to calculate the blended seek behavior. The calculated newLocation will be returned.

Detecting and avoiding borders

We also have to avoid being too close to the border to avoid getting trapped. The challenge in this is that the game does not have an explicit border. Only when a cell crashes into a border, it gets squished to show to the player that he/she has reached the border. There are no objects outside the border. It is essentially empty black area. This makes it difficult to detect the border using basic image processing. A workaround for this scenario is, we first identify the leftmost, rightmost, topmost, bottommost circles. Suppose the leftmost pellet is at a distance 'x' from the left end of the screen, and this value 'x' is such that x > SCREEN_WIDTH/4 , we can say that we are near the left edge of the environment. Similarly we can conclude about the bot being near the other borders.

Now, to avoid being trapped, we introduce a 'fake enemy' centered at the boundary of the arena so that the bot experiences a repulsion from that 'fake enemy'. This helps us ensure that the bot tries to stay away from the border. In case the bot is approaching a corner, it experiences repulsion from two 'fake enemies' and hence gets repelled diagonally opposite to the corner.

Range of Seek

The bot's main objective is to stay alive. So, it gives the highest priority to flee behavior. After calculating the flee vector, the bot can try to eat smaller pellets which are in the general direction of the vector, to gain mass. To eat a pellet, the bot will have to seek in the direction of the smaller pellet and when smaller pellet is covered by bot's boundary, it will be eaten. The bot does not have to consider all the pellets in the environment. It just needs to consider those pellets which are in the conical range of Flee vector's direction. The size of this cone (angle of the cone) is decided by the threat level in the general direction of the flee vector. More the threat, smaller the size (range) of this cone.

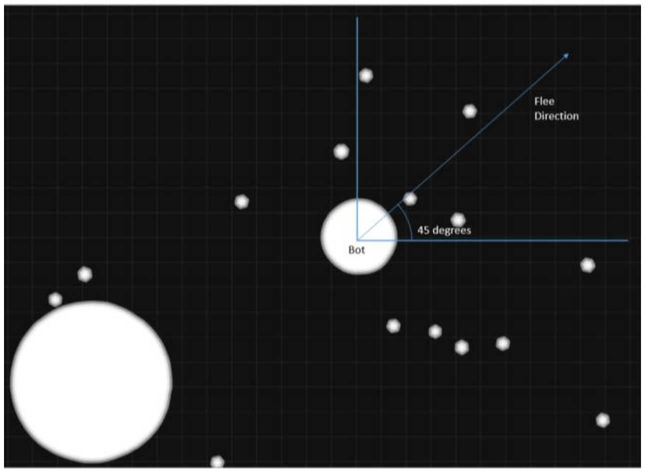
Algorithm:

```
angle = angleFind-
er(newLocation[0],newLocation[1]);
range = rangeFinder(angle);

angleFinder(x,y)
{
    bot= circleList.get(0);
    angle=Math.atan2(y-bot.y,x-bot.x);
    return angle;
}
rangeFinder(angle)
{
    rangeFactor=0;
      bot=dct.circleList.get(0);
    for (i = 1; i < circleList.size(); i++)
    {
      enemy = circleList.get(i);
      if( enemy.r >= bot.r*1.25 )
      // Enemy 25% bigger than us
      {

  if((angle+Math.PI/4)>angleFinder(enemy.x,e
nemy.y) && (angle-
Math.PI/4)<angleFinder(enemy.x,enemy.y))
          rangeFactor+=enemy.r;
      }
    }
    rangeFactor/=bot.r;
    maxRange=20*Math.PI/180;
    minRange=0;
    return Math.max(0,maxRange-((maxRange-
minRange)/5)*rangeFactor);
}
```

The first step is to find the 90 degree range centred in the direction of flee. This done by selecting 45 degree range on both sides of the direction of flee. In this 90 degree range, enemies who can consume us need to be considered, then we need to find the summation of the ratio of the enemy's radius to the bot's radius. This summation value will be needed to decide the final conical range for seek behavior.



In figure above, 90 degree range is selected by 45 degree on both sides of the flee vector. Then in this 90 degree range, all the enemies who pose a threat to the bot are selected, let us call them p1,p2,…pn. Next step is to find the summation of the ratio of radius of the enemies to the radius of bot

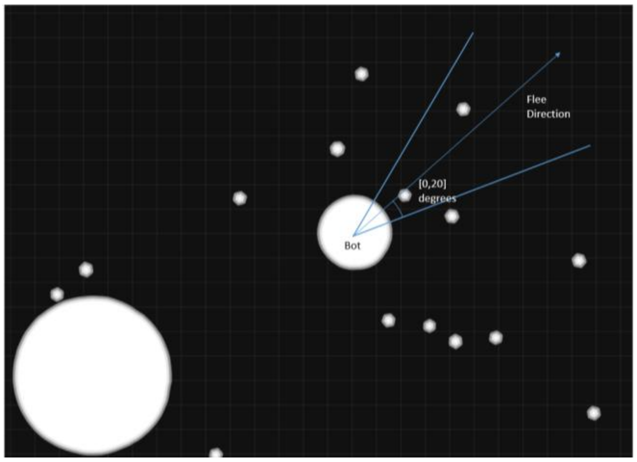
$$\text{ratio} = \sum_{i=0}^{n} \frac{r_i}{r_b}$$

Here, ri = radius of enemy , rb= radius of bot.

The idea of considering the ratio of bigger enemies while selecting the range is needed, because these bigger enemies which may fall in the range of flee may pose a significant threat and our seek range needs to be narrower to avoid being consumed by a bigger enemy. The summation of ratios will be used in equation to find the range of seek.

range of seek = maximum seek range – 4*ratio

Here, we have chosen the Maximum seek range and minimum s seek range as 20 degrees and 0 degrees respectively. This means that when there is very little threat in the general direction of flee, we can have the freedom of seeking pellets in the range of flee direction +/- 20 degrees. If there is a high enough amount of threat in that direction we decide that the range is zero because, in such a high threat scenario consuming pellets should be the least of our concerns. So, final range of seek will be from 0 to 20 degree on both side of the direction making it total [-20,20] degree range.
As shown in figure, the bot will try to look for smaller pellets which are in its range of flee direction. All the other pellets will be ignored. After every iteration, the positions of bot and enemies change, so the flee direction is also modified which gives a new range of seek to consider.



Seek Nearest pellet
This method is used to find the nearest pellet in the seek range discussed above when there is at least one threat on the screen. This is done by finding the pellet with minimum Euclidean distance from the bot and assigning final seek position to the coordinates of this pellet. This (x,y) value is normalized to 200 radius around the bot.

Seek Pathfinding
This method is used to optimize the amount of pellets eaten per unit time when there is no other threat on the screen. This is done by using a modification to the pathfinding algorithm. Steps for this algorithm are as follows:
• The graph is created by assuming that the initial graph is fully connected.
• Then the minimum spanning tree is extracted from this graph.
• Then a pathfinding algorithm is used to find path for each pellet taking the bot as the starting point.
• The best set of pellets to follow would be the list of pellets in a path with least total cost and most vertices, i.e. (total cost of path / no. of vertices in path).

The drawback of this algorithm would be that it is too computation intensive. We would compare it with performance of other methods to get similar yields such as k-means to find the cluster of pellets that are most advantageous to seek.

A faster method to detect best set of pellets to seek would be dividing the screen into rectangular areas and finding the one with the most number of pellets. Although this method is not efficient, it is much faster than the other two methods discussed above.

Virus detection and usage

For testing purposes, initially we have assumed virus as just another enemy cell. Next step would be detecting the virus as a separate type of object that does not need to fled from and should not be seeked. Instead:

• it can be used to hide when the bot is smaller than the virus,

• it can be ignored when the bot is the same size as the virus (1.25>SizeRatio>0.8) and

• it should be avoided when the bot is bigger than the virus.

Moreover, another way to use the virus is to make it to split enemy cell into several pieces. This is done by aiming mouse towards the virus with enemy on the other side and using 'W' key to shoot 7 small cells towards virus. The virus then duplicates and the new virus moves on the opposite direction of our shooting direction. This can be used to force virus into enemy and splitting it so that it can be consumed.

To implement this, we would follow these steps:

• Detect virus

• Detect enemy with size equal or more than us (SizeRatio>0.8) on the opposite side of virus from the perspective of the bot.

• Check if distance from bot to virus and virus to selected enemy is below a defined threshold.

• Point mouse towards virus and press 'W' 7 times

• Resume normal state of flee and seek.

Split to consume

Each player can use 'Space' key to split into double number of cells which can be used to eat other cells as splitting gives momentum to catch other player cells that are faster. It cannot be overused as a player with more cells to control is more vulnerable.

First step to achieve control over splitting behaviors is the ability to detect all the cells of the bot even if there are more than one. The current implementation assumes that there is only 1 cell for the bot. To do so, we would be using a name like '_' (underscore) to detect all our cells (each cell would have that name displayed.

The following steps can be used for the Split to consume behavior:

• Detect appropriate enemy cells that require the Split and consume behavior (0.4>=SizeRatio>=0.2)

• Check threshold for max distance from such detected cells.

• Check threshold for max number of enemy of size 1.25 to 2.5 times the bot (2.5>SizeRatio>=1.25). (Estimated threshold = ~2)

• Check threshold for max number of enemy of size 2.5 to 5 times the bot (5>SizeRatio>=2.5). (Estimated threshold = ~0)

• Place mouse in direction of target and press 'Space' key.

Split to escape

Similar to the Split to consume behavior, splitting can be used to escape high levels of threats. This is useful to gain speed and escape faster. Sometimes it involves sacrificing one of the 2 cells to the enemy after splitting

The following steps can be used for the Split to consume behavior:

• Check if magnitude of flee from 2.2.1 is above a threshold value. (This would indicate that there is high amount of threat concentrated in one direction)

• Check threshold for min number of enemy of size 2.5 to 5 times the bot (5>SizeRatio>=2.5). (Estimated threshold = ~1)

• Press 'Space' key to split (Mouse direction would be handled by the flee algorithm in 2.2.1).

Blending and Arbitration (Manage Multiple and Different Size Cells)

As mentioned earlier, the current implementation needs to be updated to be able to detect multiple cells of the bot. Once detected, other algorithms need blending and arbitration on the thresholds they use as it is not advantageous to use techniques like split to consume, virus usage, etc. when the bot is already split into 2 or more cells.

Moreover, when 2 or more cells of the bot are of different size, the flee and seek algorithm would give different results for different cells of the bot.

We would need to decide by experimentation to see what would give the best results out of different techniques for blending and arbitration. For example these methods would need to be tested to get the best result: Adding flee vectors of all cells with equal weight vs adding flee vectors of all cells with preference to larger cells vs ignoring flee vectors of all cells except the largest cell.

Evaluation Methods

There are two kinds of indicators of performance of the bot- direct and indirect.

Direct Indicators

Total survival time

Another important parameter to judge the performance upon is total survival time. Although sometimes the bot will have to split and sacrifice a part of itself, it will end up surviving the attack rather than succumbing to enemy without splitting. This may take the bot off the leader-board, but the bot can survive longer and try to reach the leaderboard again.

Time on the leaderboard

Presence on the leaderboard is an indication of a good player. The longer the bot can stay on the leaderboard, the more consistent its performance can be assumed. So, the duration on leaderboard is a good measure of the bot's per-formance.

Number of cells eaten

This indicates the number of enemy cells the bot has con-sumed. Higher the number, higher the dominance of the bot over other bots. This indicates a better performance. Therefore, in general number of cells (enemies) eaten is a good measure.

Indirect Indicators

Food eaten

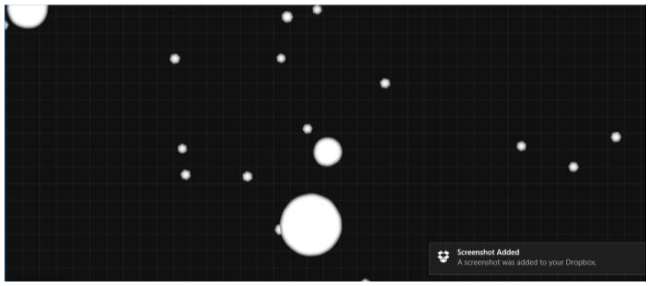
This indicates the number of food pellets that the bot has consumed. Food pellets do not evade the bot, and are easy to consume, therefore they are not a real measure of per-formance, although a higher count is generally a good sign.

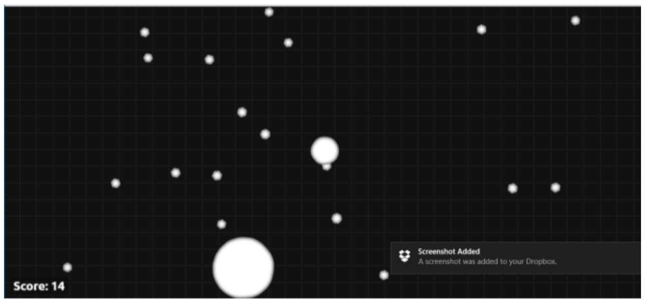
Highest Mass

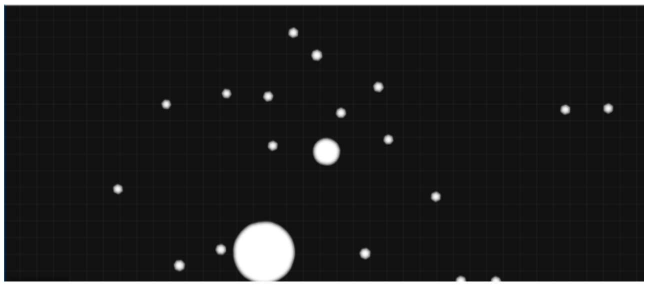
This is the highest mass achieved in its lifetime. This again is not always a good indicator, as bot can choose to remain at a lower mass in order for it to be fast enough to evade enemies. But in many cases it can be a vague measure to gauge the performance of the bot.

Results

Screenshots showing working program:







Conclusion and Future Work

With just the basic flee and seek behaviors implemented, the bot is in a good enough shape to survive for a while in the game. We plan to implement various modified path searching methods, to traverse around efficiently while seeking. The list of things that are yet to be implemented also contains - Decision when to split, Detecting and con-trolling motion of various split parts, Using Viruses to bot's advantage, Shooting mass to move faster/ split virus, 'Teaming' up with another player/ another instance of the bot.

References

Millington, I. and Funge, J. (2009). *Artificial intelligence for games*. Burlington, MA: Morgan Kaufmann/Elsevier.

Agar.io, (2016). *Agar.io*. [online] Available at: http://agar.io/ [Accessed 22 Feb. 2016].

Agarioguide.com, (2016). *Game Mechanics | Agar.io Skins*. [online] Available at: http://www.agarioguide.com/game-mechanics/ [Accessed 22 Feb. 2016].

AgarioZone, (2015). *Agario Strategies - 10 agario strategies to help you survive*. [online] Available at: http://agariozone.com/agario-strategies/ [Accessed 22 Feb. 2016].

Redditcom. (2016). Reddit. Retrieved 2 April, 2016, from https://www.reddit.com/r/agario.

Appendix: Glossary

• Pellet - Small hexagon/circle shaped object that is not controlled by anyone and is stationary. It is generated ran-domly across the map. It has the least radius among all objects in the game. It can be consumed by any player in the game.

• Virus - Stationary green (white in black and white mode) object with spiked border. It is not controlled by anyone. It has a complex mechanism as compared to other objects as described previously.

• Cell - Any object that is not a pellet or a virus. It is circle shaped. It can belong to any player or be a stationary cell not controlled by anyone in some cases.

• Player - Any controller of a cell or a group of cells. A player only controls more than one cell if his cell splits due to certain reasons.

• Bot - The player that we are controlling, i.e. the program is controlling.

• Enemy - All the other players in the environment

• Consume - When one cell A1 of player A overlaps one cell B1 of player B, where cell A1 has bigger radius than B1 (ra1 = 1.25 * rb1). Then B1 no longer exists and that mass is added to the mass of A1.

• Mass - Value of each cell (and pellets) that is directly proportional to the radius of the cell.