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. It is played on a fixed dimension frame in which player and other opponents are circular cells and they can wander throughout the environment. The objective of the game is to get bigger, achieved by consuming smaller opponents, and food, while staying away from bigger opponents to avoid being eaten. The environment also consists of viruses, which are stationary. All playing objects or cells need to avoid viruses smaller than themselves as significant collision with a virus would make them split into many smaller cells. The speed of a cell is inversely proportional to the size of the cell. When a player eats smaller opponents, the player increases mass and the size of the cell grows, making it slower. To counter this, every player has ability to split in half, the halves moving side by side, at a higher speed. The cell can also shoot out some of its mass. A player can do this to hit the virus and make the enemy on other side of virus split into several pieces. Movement of our player is decided by the mouse pointer direction, i.e. the player will move into the direction of the mouse pointer. Speed and steering behavior is handled by the game. During the game, the top 10 players in the room are shown on the leaderboard.

Our objective is to develop 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 AI techniques discussed later in this report to achieve the game objectives as intelligently as possible. Various game algorithms, together 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 the 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

Other than the basic goal of avoiding bigger opponents and chasing smaller opponents, there is no actual end game scenario. The playing goal can be divided into 2 parts - maximizing the size of the bot and maximizing survival time, i.e. avoid being eaten for as long as possible. Combined efficiently, achieving these goals will get the bot on the top 10 leaderboard. Staying on leaderboard as long as possible will be a sub-goal as well. We lose mass with time, and the amount of mass lost per unit time increases with our size, therefore as we grow bigger and we need to keep changing to a better strategy for either adding or sustaining our mass. This implies that we cannot achieve the goals by hiding and avoiding everything even if maximizing survival time is given priority.

The common strategy to play the game can be divided into 3 stages. At the start, the bot should prioritize eating most pellets in least amount of to time to grow faster. After bot is considerably big enough, it should prioritize eating smaller opponents directly or by splitting. If bot splits, it will be more vulnerable, so at this stage fleeing will gain some priority. As the bot will climb up on leaderboard, most other players in the room will be smaller in size. As it'll be losing mass at a higher rate now, it should implement strategies to sustain the mass, by seeking nearby bigger players or using virus to make other large players split into many pieces to maintain the top position.

Problem

The game does not provide a direct interface to input data needed for movement, or to perceive environment for decision making. So we need to use basic image processing by repetitively capturing the visible environment to get information, like position and size of enemy, position of virus, position of pellets, etc.

The basic movement algorithms are applicable in the context, but in an unconventional way, as we control the mouse that directs the character, instead of directly interacting with the character. So in most cases kinematic motion will work better, as for real players, movement of mouse is fast enough that we can ignore the acceleration and deceleration effect. This will instead be advantageous for the bot to push the ability to move mouse faster than a human player to be more effective. On the other hand this may result in a vibrating character due to conflicting decisions or a dilemma, e.g. trying to follow two smaller masses on opposite sides.

Deciding what action has more priority than the other would have to be implemented to have the bot perform more efficiently. As Arbitration for actions may change according to different situations.

Potential Benefits

The game requires the application of most movement, path finding, and decision making algorithms to make an effective bot. These techniques are implemented in most real-time strategy games and requires the bot to match the capabilities of human players which is the ultimate goal of building a good Game AI.

Tasks and Techniques

The tasks and techniques required can be divided into 2 parts. The first part is to create an interface with the game to get data which would be used by the second part that consists of AI techniques that the bot would use to give mouse and keyboard inputs to the game.

Basic Image Processing

This task is needed to retrieve information about the current status of game each instance to generate data needed to apply the various AI techniques. Most commonly available Image Processing APIs deal with processing real life photographs instead of basic shapes, and those that do detect circles assume the image has other shapes, or that circle is completely within the frame. Using these APIs will not be efficient for us because of the under-utilization. Our scenario has circles on a black background. So instead of detecting difference between circles and other shapes by number of vertices as most common algorithms do, we only need to find the position of the circles as we already know that they are, well, circular.

The circle position and radius detection algorithm we are using works as follows. The image is scanned from top left to bottom right to find black pixels (which indicate that the pixel is on the circle). Once we have a non-black pixel surrounded by non-black pixels, the image is traversed horizontally and vertically to find the limit of the current circle. The intersection point of these two lines is calculated, indicating the center of the circle.

To detect partial circles at the boundary, we can use the ratio of horizontal and vertical length that is calculated in the previous step, to get the position and size of the partial circle.

AI Techniques

We would need to modify AI Techniques such as seek, flee, collision avoidance, path finding, decision making, blending and arbitration to achieve the following sub-goals.

Avoid bigger enemies

The bot needs to look out for enemies which are 20% bigger in radius than itself and try to move away from them. Because the speed of an enemy is inversely proportional to its size, the bot needs to be wary of those enemies which are just big enough to eat it. It can be a little less worried about the much bigger enemies are they are bound to be significantly slower than itself.

We fix a threshold radius for each enemy depending on its size, and when that threshold radius is breached, the bot can begin to flee from the enemy. Although we need to be careful of the fact that a much bigger enemy can split to instantly gain more speed and catch up with us, one design aspect is to fix the threshold radius keeping this possibility in mind.

Avoid virus

The smaller virus is terminal to the bot. So, we need to stay away from smaller viruses. I case we are unable to, we end up splitting and hence become vulnerable to more enemies. Although if the virus is bigger than the bot, it can harmlessly pass through it. Avoiding the virus can be done using obstacle avoidance algorithms.

Seek smaller players

In Agar, one of the main objectives is to score points by eating smaller opponents. Our bot has to seek the direction where it can consume more opponents and score points. There can be many strategies to calculate the direction in which our object should go. We are considering the one in which our object will seek the nearest smaller opponent.

There are many ways to find the distance between our object and the opponent. In our case, we consider the Euclidean Distance. We will calculate the distance from our object's center to the opponent's center. Another way is to

find the distance gap between the boundaries of objects. The latter is not effective as eating the opponent requires the bot to reach close to the center of the smaller opponent. So, in this case, the distance between the centers of objects will be the right choice. Distance and direction of the nearest opponent will be calculated in each update, and any changes in the environment will be taken into account to update the new target to seek.

As the movement part is handled by the game itself, objective of the bot is to find the direction of target to seek. Once this direction is found, the corresponding mouse movement will be done to set the direction of object and the object will seek the target.

Split to escape from larger players

One of the most interesting features of the game is splitting. This can be leveraged in many situations to our advantage. One such situation is escaping enemies. If the enemy is too close for comfort, the bot can split to increase its speed and let at least one half escape the approaching enemy. The only downside is that the two halves need some time to join back, which makes the bot a bit more vulnerable to being eaten.

Split to eat smaller players

Another interesting use of the splitting ability is to catch a smaller enemy which is fast enough to be out of reach normally. In such a case, we can split and fling one of the halves into the enemy's direction catching it off the guard. One of the parameters to consider here is if the split is worth it. It is, if the food we are chasing is about 20% smaller than half our size. Otherwise, if the target is too small, the gain is too less justify the risk in splitting. On the other hand if target is too big, the split will be wasted. We would also need to factor in if there are other enemies of large enough to split and eat us after we have split.

Shoot virus to split other players

One way to make enemy split into many pieces is to shoot some mass at the virus and split the virus into the opponent. In this situation, when the opponent is lying exactly behind the virus, we can shoot the virus to launch a new virus to the opposite side where the enemy is. This is especially useful when the bot is sufficiently large to risk weight to shoot virus and have the need to eat more mass to maintain.

To identify this situation, our strategy is to look in the direction of virus for enemies below threshold radius. If we find such an enemy, the bot can get closer to the virus using techniques mentioned above and then shoot at the virus.

Path finding

At the beginning of the game, we need to eat the food pellets lying around us to increase our mass. We need to do this as efficiently as possible, meaning eating as many food pellets as possible in shortest amount of time. For that, we implement a weighted pathfinding algorithm, finding the shortest path to a pellet or finding the path with the most pellets and shortest distance.

Blending and Abstraction

As the priority of the bot may change in different situations and different stages of the game there is a need of decision making to implement Blending and Abstraction effectively. We also need to implement a mixture of steering algorithms, and tweak them as needed to obtain the required performance.

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 leaderboard, 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 performance.

Number of cells eaten

This indicates the number of enemy cells the bot has consumed. 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 performance, 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.

Conclusion

Agar.io looks like a simple game to play but there are many strategies involved which the bot should excel at in unison to become a good player. Although identifying enemies and either seeking or fleeing constitute the basic strategy, we will need to implement advanced strategies like virus avoidance and splitting decisions in order to fulfill the goals. There are some parameters that would need tweaking to get better performance. Some of them are, the threshold distance from which to flee away from bigger enemies, proportionate maximum weight before splitting, maximum distance to avoid viruses, etc. It is difficult to estimate the accuracy of our algorithms in terms of time and statistics at this point of time, but we will consider the possible changes in strategies to improve results. To begin with, the most vital task is to process the screen capture of current game situation, apply the enemy detection algorithm, and get the required location data in a timely manner. In our view, the strategies and techniques stated above are a good direction to start with and as we proceed we can come up with more ideas to improve the bot for Agar.io.

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].