Artificial Intelligence for Arcade Game Centipede

Gowtham Sathyan Arulselvan and Sudharsan Janardhanan
Department of Computer Science
North Carolina State University
garulse@ncsu.edu, sjanard@ncsu.edu

Abstract

To anyone who has been to an arcade, Centipede is one of the most popular games. Hitting its peak in terms of popularity in the early 1980s, Centipede[1] is a classic shooter game in which a bug-blaster, only capable of lateral movement, aims to eliminate all adversaries. Each[1] segment of the centipede becomes a mushroom when shot, and splits into two halves when shot anywhere besides the head. The player also encounters other creatures such as scorpions, spiders and fleas. The game ends when the bug blaster is hit by a creature or a projectile fired by a creature. We aim to develop an intelligent bot that can play the game automatically and survive the longest while scoring maximum points.

Introduction

This project aims to simulate AI techniques in automated game play for the classic arcade game Centipede. Our objective is to create AI that plays the game intelligently and strive to create a high score. We plan to incorporate AI techniques such as collision avoidance, intelligent decision making and path finding based on a finite set of states.

Background

Intelligent decision making has always been part of Artificial Intelligence in games. In the 1970s, a primitive type of decision making in the form if random movement was used in a similar shooter game called Space Invaders. Slowly, the decision making improved in games such as Pong where the trajectory of the ball was tracked to make informed choices in the game. Slowly as time went on, as games evolved, AI techniques too evolved in terms of sophistication.

Problem

We aim to build a utility-based agent that would not only have a goal to achieve, but a utility function that would evaluate the agent's performance as it is playing the game to dynamically determine its next set of actions so as to maximize its performance. The actions performed by the agent will be decided by a finite state machine and the utility function will rate the effectiveness of the performed action by comparing it with the environment before and after the action was performed. This will enable the agent to identify the next set of actions required in order to be more efficient.

Potential Benefits

Creating a bot for the game Centipede will help us learn a lot of techniques used in Artificial Intelligence. Despite it not having the need for any path finding techniques, it has the potential to include several sophisticated decision making algorithms, machine learning algorithms and collision avoidance techniques in a bid to have the bot last longer than any known high scores that have been achieved by humans. Since Centipede is one of the most renowned arcade games, it shouldn't be difficult to find out the record high scores in the history of the game. Attempting to beat such scores will enable us to learn efficient coding techniques, excel in decision making algorithms and create a bot that is worthy enough to play the game for a long time without being defeated.

Tasks and Techniques

Centipede, like any other retro 2D game like Tetris, has the incoming adversaries and obstacles initially move from the top to bottom after which depending on the character has them move in any of the four possible directions, namely up, down, left and right. The lateral movements are fixed up to the edges of the screens, meaning the centipede will turn once it hits the side of the screen. Similarly, the movement of the blaster is fixed on the x-axis bounded by the left and right edges of the screen. Hence, we do not need to worry about boundary conditions where either the adversary or the blaster exit the view port.

The properties of the game environment are:

- **Partially observable**: Once the game begins, the entire screen is available for the agent, but there will be adversaries moving into the screen from outside the view port as well.
- **Stochastic**: The movement of the centipede can be predicted, but the arrival of the next adversary or the location of the next placement of mushroom might not be predicted with accuracy.
- **Known**: The agent knows all possible legal movement actions it can perform.
- **Multi-agent**: The bot and the adversaries become agents.

- **Dynamic**: The environment can change between two actions performed by the agent.
- **Sequential**: The agent cannot perform actions based purely on the current state of the environment. The agent needs to keep track of all that has happened so far and how the environment has been changing.
- **Discrete**: There is a limited number of legally defined actions the blaster can perform.

At each iteration, the information available to the bot is:

- Location of the blaster
- Location of each part of the centipede (if any)
- Direction the centipede is headed
- Location of each mushroom
- Location of special adversaries (flea, spider or scorpion, if any)
- The direction the special adversaries are headed

Our aim is to identify the most important locations to be shot by the blaster in order to ensure prolonged survival of the blaster. This can be done by keeping track of location and movement of all adversaries and the locations of the mushrooms in order to destroy them in an efficient manner.

Centipede

The Centipede is the primary and the most common adversary in the game. It appears into the screen from above and will keep moving in a direction until it hits the edge or a mushroom after which it turns 90 degrees to the left or right in a random fashion. The aim of the centipede is to collide with the blaster and destroy it. Now, the aim of the bot is to destroy the centipede before it destroys the blaster.

Shooting a part of the centipede removes that part of it, except it can now split into two based on the location of the hit. In other words, unless the centipede is hit on its head, it will always split into two and the bot will now have two moving targets creating chaos. Hence, the most important aspect of the centipede is the requirement of striking it on its head. This will require a finite state machine that will keep track of the current and all possible future states (up to a limit) of the centipede so as to predict its location and enable us to time the shot and hit the head effectively.

Mushrooms

The mushrooms are static obstacles that can block the path of a blaster bullet from reaching an adversary. The killing of an adversary such as a centipede can result in a mushroom being formed at the location of its death apart from certain special adversaries. Once any mushroom is formed on the axis of movement of the blaster and the blaster touches it, the game ends. Hence, it is not only important to destroy mushrooms in order to be able to target the adversaries effectively, but it is crucial to target those mushrooms that are closer to the blaster than those that are farther away.

Special Adversaries

Shooting the head of the centipede earns the user 100 points whereas shooting any other segment earns the user 10 points. Barring the Centipede, there are other special characters in the game whose movement and points earned upon shooting varies as well.

Fleas[1] drop vertically and disappear on touching the bottom of the screen. They leave a trail of mushrooms if only a few mushrooms are present on the screen. They take two shots to kill and are worth 200 points.

Spiders[1] move in a zig-zag pattern on the screen and are worth a multiple of 300 points, with the multiplier ranging from 1-3 depending on how close the blaster shoots it.

Scorpions[1] move across the screen and turn every mushroom they encounter into poison mushrooms. They earn the most points for the user, contributing 1000. Centipedes[1] when in contact with a poisoned mushroom exhibit a special behaviour, they hurtle straight down towards the bottom of the screen and return to normal behaviour upon reaching it. A poisoned Centipede must be given a higher rank compared to other adversaries since it is more dangerous. It is fairly easy to destroy unless it is split into two segments, in which case it becomes hard to maneuver against.

Bug Blaster

The bug blaster is the character/sprite controlled by the AI agent. Its purpose is to shoot the adversaries and obstacles and survive for the longest while scoring the maximum amount of points possible. It can move on a fixed axis from left to right and vice versa and can shoot bullets in a constant direction of upward. Since the survival is the most important requirement, it is crucial to decide the next action as that might decide if the blaster is going to survive or not. Hence, we need a way to identify the best set of actions that can be performed from a given state and then choose the best among them to perform next. Therefore, we need to define heuristics.

Heuristics

In each time frame, the bug blaster has to make an informed decision as to which target to hit that would make the greatest impact. The bug blaster has to make a choice that maximises the amount of points gained while at the same time minimises the chance of risk to the blaster. In a scenario where multiple different creatures are present in the same position longitudinally, the bug blaster would choose the foe that rewards more points.

In a situation where the same amount of creatures are present, but a lower ranking creature is closer to the bug blaster and poses more risk would be shot first rather than a higher ranking creature that is farther away. These kinds of decisions require not only the current and previous states of the environment, but also the path cost of reaching a goal and the heuristics/cost of performing an action.

In our case, the cost of averting an imminent loss is going to be higher than scoring more points. We can define our heuristic as a function of distance of the adversary from the axis of the blaster, the direction in which it is moving (i.e.

higher if moving towards the blaster) and the points that can be obtained on destroying the said adversary.

Path Finding

Now that the heuristics are defined, we need a method to choose the order in which nodes will be visited. Path finding doesn't necessarily imply that a movement is required at each step. It simply defines the route to take to reach the goal state from the start state. In our terms, we need to identify the order in which the actions will be performed so that we can maximize our output, the score. The simplest methods would be employing a Breadth First or Depth First search, neither of which take into account heuristics. One way to overcome this would be to consider the heuristics as distances between the blaster and a foe thereby choosing the lowest heuristic as the shortest distance. But, considering the low amount of time available to compute and decide the next action, we would need to look at other algorithms that can dynamically compute next actions and select the action that would give the best outcome. Hence, we would be using Best First Search coupled with the A* search algorithms

Best First Search

In this algorithm, we would be choosing the action that would change the state to the next lowest heuristic among all possible next states from the current state. Best First Search is a Greedy algorithm, wherein the best solution at that point is chosen irrespective of whether it leads to the most optimal solution or not. The agent will look at the available data of locations of all adversaries and mushrooms, the direction of movements of each of the adversaries on screen and would compute the following two values:

- The points obtainable from killing the adversary - This is a straightforward calculation based on the type of adversary.
- The adversary that poses the highest threat. This is calculated based on the distance of the adversary from the bug blaster and the direction it is moving in.

Once the calculations are done, a combination of these values will be used to determine the final heuristic value for each possible action. The action with the lowest heuristic value will be chosen as the next move.

Although this method can produce good results, there is a certain factor that we do not consider while making the choice, which is the path cost. Since Best First Search only takes into account the heuristic, we do not need to calculate the path cost which could prove handy in choosing better solutions. This is where we would need A*.

A* Search Algorithm

When the blaster is moved on its axis, although can be considered negligible, there is a small time lag between one position and another. In other words, there is a time δt that the blaster takes to move from one position to another. This negligible amount can become significant when the actions require the blaster to move to longer distances in consecutive moves. Due to this, the predicted location of any adversary in the next state might not be the same when the blaster reaches the required position, as the time difference would change the current and next states. Hence, we would include the time taken to reach the goal location as the path cost and add it to the heuristic to get the final value of the function on which A* would be implemented. This would ensure that the agent not only chooses the best possible adversary or obstacle to shoot, but will also predict the possible locations of all those adversaries and obstacles and identify the one that would create the biggest impact for the longevity of survival.

Collision Avoidance

The ideal scenario would be for the blaster to kill every foe before it reaches the bottom of the screen, but with foes such as Fleas that drop vertically to the bottom of the screen, it becomes hard to exterminate them early on, especially as the difficulty increases as the levels progress. Another prime example would be when a centipede touches a poisoned mushroom and hurtles down towards the bottom of the screen, it might not be possible to destroy the entire centipede before it reaches the blaster. In such cases, collision avoidance and evasion need to be given the highest priority and hence, the lowest heuristic. This is done since preserving the life of the bug blaster is of paramount importance.

Evaluation Methods

Since Centipede requires a combination of several actions such as movement, combat, evasion and decision-making, we would need to experiment with the various techniques mentioned so far to figure out how to design the best performance evaluation metric.

The first experiment would be to identify which of A* or Best First Search would give better results. Since there is a minor difference between the two algorithms, we need to identify if it is worth doing the extra computation and if the difference in performance is minimal, we would be better off choosing the efficient one over the optimal algorithm. All this is in consideration of the fact that we need the actions to be generated very quickly.

The next experiment would be to test and improve the evasion techniques. As mentioned above, the evasion of imminent death is the most important, but if that leads to death in the next time stamp, it is not a great solution. This would entail that the evasion algorithm needs to identify a location that the blaster can evade to in order to protect itself from the imminent threat but at the same time will also be safe from other immediate threats at least until the next AI cycle begins.

Apart from deciding which state/location to move to and what to shoot at, we need to evaluate the timing of the shots taken by the blaster. The agent needs to keep accurate track of the states, current and future, of all the adversaries and obstacles and shoot at the target ahead of it reaching its final state. Since we need the blaster to ideally shoot at the head of a centipede, we will measure the efficiency of the agent by keeping track of the number of times it manages to strike the head compared to the number of times it hits the body.

Once we reach the local maximum of satisfaction with the performance of the codes individually, it would be time to put the entire agent to a test. Now, although we can still measure the performance measures stated above, we would be focusing on the longevity of survival of the agent on its own and the points it can rack up in the process.

In the initial stages, the agent would need to prioritise the number of points since the levels would be easier and the agent will have considerably more time to respond. As the levels progress, the difficulty increases and hence the focus shifts towards survival than the points earned. Hence, our performance measure would initially be inclined towards points and would slowly incline away from it and towards duration of survival as the levels progress.

We will also be attempting to collect historical data from the internet regarding high scores in the arcade and compare against the points that our agent can score. Doing so, we can glean information as to when survival would overtake points earned when calculating our performance metric. Our objective is to attempt to beat the high score records that we can find online.

Conclusion

Although Centipede is a pretty straightforward and simple game to play, having been conceptualised in the birth of the arcade era in the 1970s, designing an AI for it is far from such. Quite a few decisions must be taken when designing its AI. The hardest decisions lie in areas such as the trade off between complexity and simplicity, the choice of heuristic used for each action that the blaster makes and designing an appropriate metric that takes into context difficulty level, points earned and time survived. We hope to navigate these difficulties by the process of experimentation, adequate research and using proven scientific methodologies to solve this problem.

References

[1]https://en.wikipedia.org/wiki/Centipede(videogame)
[2]https://www.gamedev.net/tutorials/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942/
[3]https://www.geeksforgeeks.org/agents-artificial-intelligence/
[4]https://www.geeksforgeeks.org/a-search-algorithm/