CS231

12/8/23

Making a pursuer-evader game using Priority Queues and Heaps (Project 8 Report)

Abstract

I created a graph-based pursuit-evasion game. There are two players in the game: the pursuer and the evader, and they both have unique approaches. From a simple RandomPlayer to more calculated decisions like MoveTowardsPlayerAlgorithm and MoveAwayPlayerAlgorithm, I created a variety of player algorithms. The game's basic design is represented by a Graph class, which uses an adjacency list to manage vertices and edges to maximize efficiency. I investigated the performance of these algorithms by looking at average steps and capturing rates on graphs.

Exploration 1

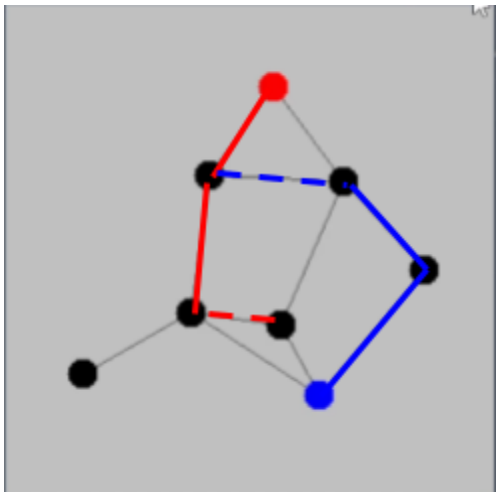
N (Vertices)	P (Probability)	Percent Caught	Steps
4	0.2	100%	1
6	0.2	80%	1
10	0.2	70%	3
4	0.5	70%	1
6	0.5	80%	2
10	0.5	40%	1
4	0.7	90%	1
6	0.7	90%	1
10	0.7	100%	2

As we can see from the data above, the probability of an edge does not have a big effect on the steps but has a huge impact on the percent of times that the evader gets caught. When the probability is either 0.2 or 0.7 the chance of the evader getting caught is much higher than when the probability is 0.5. The number of vertices has a negative correlation with the percent because as the number of vertices goes up the chance that the evader is caught goes down. The number of vertices also seem to have a positive correlation with the average number of steps.

Exploration 2

Video #1:

In this video you can see that the pursuer ends up catching the evader. But this could have run indefinitely if the evader had stayed on the square-like loop. Since the pursuer uses a greedy algorithm to chase the evader, it will always take the shortest path to the evader, therefore it will forever follow the evader either around the square, or go back and forth between two vertices.

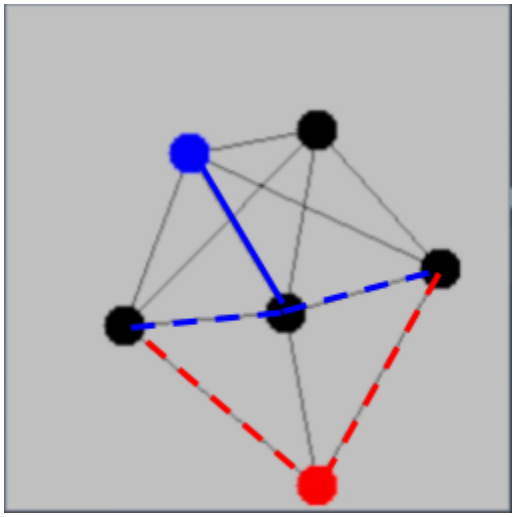


In this screenshot I show the path that could have been taken from the player's start points. The pursuer could have entered the square and the evader could have continued loop around it. The dashed line shows that they could have continued going back and forth between two vertices indefinitely.

Exploration 3

Video #2

In this video we can clearly see that the two players have fallen into an infinite loop. Due to the fact the the pursuer uses a greedy algorithm, it will never exit the loop. But this loop could have been avoided from the start.



If the pursuer had originally stepped into the middle instead of going around the sides, then no matter where the evader would go next, the pursuer would be able to catch them (shown by the dashed lines).

Extensions

videoEXTENSION

For my extension I created a new algorithm for the pursuer that follows the same principles as the MoveTowardsPlayer algorithm but only if it is further than 1 hop away from the evader. When it gets closer to the evader instead of simply following it, it will move to a random vertex nearby. I did this in order to try to avoid falling into a loop. Sometimes it works very well and is able to intercept the evader by making a random move, while other times if it would have just taken the shortest path it would of caught it much faster. Using this algorithm you will almost never fall into a loop but it can sometimes take very long for the pursuer to catch the evader in certain scenarios. In the video I provided you can see the pursuer take two seemingly random moves but ending up catching the evader. If it hadn't done these random steps it would have easily fallen into an infinite loop. (My extension algorithm is the file named "ExtensionAlgorithm.java")

Reflection

I used ArrayLists to effectively manage vertices and their connections in the pursuit-evasion game. Using an arraylist to manage edges and an adjacency list to manage vertices makes the game more efficient. Priority-based algorithms in the game were made possible by my use of a priority queue and heap, which enhanced the algorithms' performance.