Mahdeen Ahmed Khan Sameer

Professor Max Bender

CS231-B

06 May 2023

Pursuit-Evasion on Graphs: An Analysis of Pursuer and Evader Strategies

Abstract

For our last project, "Pursuit Evasion on Graphs" project, we implemented a turn-based version of a pursuit-evasion game on graphs. The game involved two players: an evader and a pursuer. The goal of the evader was to indefinitely evade capture by the pursuer, while the pursuer aimed to catch the evader. We developed the AbstractPlayerAlgorithm class as the foundation for player algorithms and implemented three specific strategies: RandomPlayer, MoveTowardsPlayerAlgorithm, and MoveAwayPlayerAlgorithm. The game was visualized using the GraphDisplay class, and its progression was controlled by the Driver class. Additionally, we explored scenarios where the evader was captured despite utilizing the MoveAwayPlayerAlgorithm strategy and instances where the pursuer couldn't catch the evader while employing the MoveTowardsPlayerAlgorithm strategy. This project provided valuable insights into different player strategies and their effectiveness in pursuit-evasion scenarios on graphs, highlighting the complexities of gameplay and strategic decision-making in graph-based environments.

Results

Graph Class:

The Graph class represents a graph data structure and provides functionality for creating, modifying, and analyzing vertices and edges within the graph. It maintains lists of vertices and edges, and supports operations such as adding and removing vertices and edges, retrieving vertices and edges, and calculating the distance between vertices using Dijkstra's algorithm. The class also includes inner classes Vertex and Edge to represent individual vertices and edges, respectively.

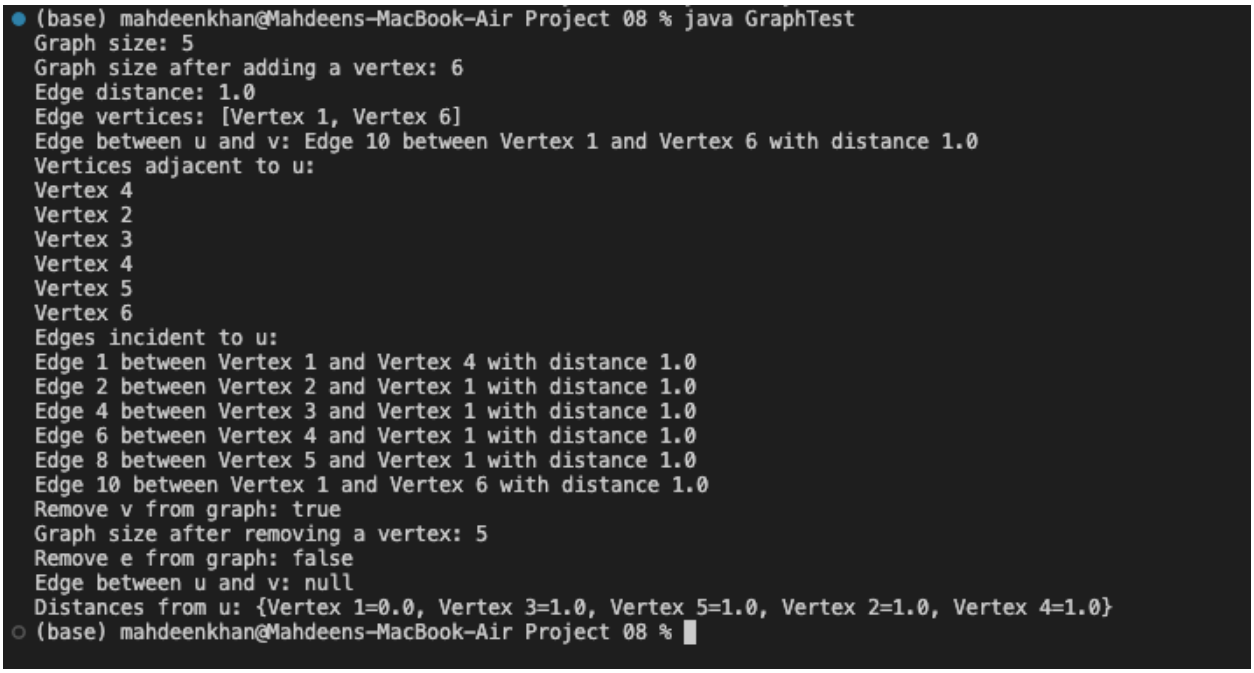


Figure: Running GraphTest.java

The GraphTest execution provides a comprehensive demonstration of the functionality offered by the Graph class. The test begins by creating a new graph with 5 vertices and a probability of 0.5 for edge creation. The size of the graph is printed to verify its initial state. It should be 5. Next, a new vertex v is added to the graph using the addVertex() method, and the size of the graph is printed again to confirm the addition. The size should now be 6. To create an edge, a reference vertex u is obtained from the graph using an iterator. An edge e is then added

between u and v with a distance of 1.0. The distance of the edge is printed, and it should be 1.0. The vertices of the edge e are printed to ensure they match the expected values. In this case, it should display Vertex 1 and Vertex 6. The graph's getEdge() method is used to retrieve the edge between u and v. The result is printed, and it should display the edge information, including the vertices and the distance. The adjacent vertices of vertex u are printed to verify the correctness of the adjacency relationship. The expected adjacent vertices should be listed. Similarly, the incident edges of vertex u are printed to confirm that they match the expected set of edges incident to u.

To test the removal functionality, vertex v is removed from the graph using the remove() method. The removal operation returns true, indicating that the removal was successful. The size of the graph is then printed, and it should be 5 after the removal. Next, an attempt is made to remove edge e from the graph. Since e does not exist in the graph, the removal operation returns false. The graph's getEdge() method is used again to retrieve the edge between u and v. Since it was removed earlier, the result should be null. Finally, the minimal distances from vertex u to all other vertices in the graph are computed using Dijkstra's algorithm. The distances are printed as a map, confirming the correct calculation of distances. Overall, the GraphTest execution provides a thorough validation of the Graph class, ensuring the proper addition, removal, and retrieval of vertices and edges, as well as the accurate computation of distances in the graph.

MoveTowardsPlayerAlgorithm class:

The MoveTowardsPlayerAlgorithm class is designed to implement an algorithm for the evader in a pursuit-evasion game on a graph while extends our AbstractPlayerAlgorithm class. The algorithm determines the starting location and the subsequent moves for the evader based on the objective of maximizing the distance between the evader and the pursuer. The class extends the AbstractPlayerAlgorithm class and utilizes a Graph object to access the underlying graph structure. It includes two constructors, one that takes the graph as a parameter and another that allows for specifying an initial vertex for the evader. The chooseStart() method selects a random starting vertex for the evader from the available vertices in the graph. It uses a Random object to generate a random index within the range of the vertices and assigns the selected vertex as the current vertex.

The chooseStart(Vertex other) method chooses the vertex that is closest to the starting vertex of the other player (pursuer). It calculates the distances from each vertex to the starting vertex of the other player using Dijkstra's algorithm implemented in the distanceFrom() method. It then iterates through the distances to find the vertex with the minimum distance, indicating its proximity to the other player's starting location. The chosen vertex is set as the current vertex. The chooseNext(Vertex otherPlayer) method determines the next move for the evader based on maximizing the distance from the current vertex to the other player's current vertex. It calculates the distances from each adjacent vertex to the other player's current vertex using Dijkstra's algorithm. It iterates through the adjacent vertices to find the vertex with the maximum distance, indicating the farthest vertex from the other player. If no farther vertex is found among the adjacent vertices, the evader stays at the current vertex. The chosen vertex is printed as the move, and it becomes the new current vertex.

```
(base) mahdeenkhan@Mahdeens-MacBook-Air Project 08 % java TestMoveTowardsPlayerAlgorithm
Random player starts at: Vertex 3
MoveTowards player starts at: Vertex 3
Evader moved from Vertex 3 to Vertex 10
Evader moved from Vertex 3 to Vertex 1
Step 1
Random player moves to: Vertex 10
MoveTowards player moves to: Vertex 1
Distance between players: 1.0
Evader moved from Vertex 10 to Vertex 9
Evader moved from Vertex 1 to Vertex 4
Step 2
Random player moves to: Vertex 9
MoveTowards player moves to: Vertex 4
Distance between players: 2.0
Evader moved from Vertex 9 to Vertex 2
Evader moved from Vertex 4 to Vertex 5
Step 3
Random player moves to: Vertex 2
MoveTowards player moves to: Vertex 5
Distance between players: 2.0
Evader moved from Vertex 2 to Vertex 9
Evader moved from Vertex 5 to Vertex 1
Step 4
Random player moves to: Vertex 9
MoveTowards player moves to: Vertex 1
Distance between players: 2.0
Evader moved from Vertex 9 to Vertex 10
Evader moved from Vertex 1 to Vertex 8
Step 5
Random player moves to: Vertex 10
MoveTowards player moves to: Vertex 8
Distance between players: 2.0
Evader moved from Vertex 10 to Vertex 2
Evader moved from Vertex 8 to Vertex 5
Step 6
Random player moves to: Vertex 2
MoveTowards player moves to: Vertex 5
Distance between players: 2.0
Evader moved from Vertex 2 to Vertex 3
Evader moved from Vertex 5 to Vertex 8
Step 7
Random player moves to: Vertex 3
MoveTowards player moves to: Vertex 8
Distance between players: 2.0
Evader moved from Vertex 3 to Vertex 2
Evader moved from Vertex 8 to Vertex 5
Step 8
Random player moves to: Vertex 2
MoveTowards player moves to: Vertex 5
Distance between players: 2.0
Evader moved from Vertex 2 to Vertex 3
Evader moved from Vertex 5 to Vertex 8
```

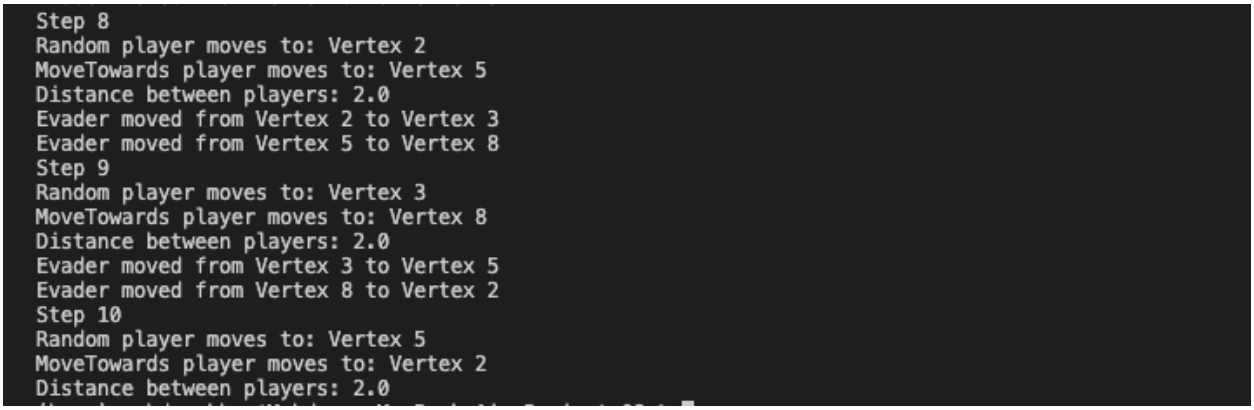


Figure: Running TestMoveTowardsPlayerAlgorithm class

The TestMoveTowardsPlayerAlgorithm class provides a simulation of a pursuit-evasion game on a graph using the MoveTowardsPlayerAlgorithm. The game involves two players: a MoveTowardsPlayerAlgorithm and a RandomPlayer. The goal of the MoveTowardsPlayerAlgorithm is to maximize the distance from the RandomPlayer while selecting its moves. In the simulation, a graph is created with 10 vertices and an edge probability of 0.5. The MoveTowardsPlayerAlgorithm and RandomPlayer are initialized with the same graph. Each player selects a starting vertex using the chooseStart() method, with the MoveTowardsPlayerAlgorithm considering the RandomPlayer's starting position. The game simulation consists of 10 steps. In each step, the players take turns choosing their next moves. The RandomPlayer selects a random adjacent vertex to move to, while the MoveTowardsPlayerAlgorithm determines the vertex that maximizes the distance from the RandomPlayer's current position. The chosen moves and the calculated distance between the players are printed for each step. Analyzing the simulation results, we observe that the MoveTowardsPlayerAlgorithm attempts to increase the distance from the RandomPlayer by strategically selecting vertices. However, due to the random nature of the game and the graph's structure, the distances between the players fluctuate between 1.0 and 2.0 throughout the

simulation. As a result, the TestMoveTowardsPlayerAlgorithm class effectively demonstrates the behavior of the MoveTowardsPlayerAlgorithm in a pursuit-evasion game on a graph, showcasing its ability to choose moves that aim to maximize the distance between the players.

Overall, the MoveTowardsPlayerAlgorithm class provides an algorithm for the evader in a pursuit-evasion game on a graph, aiming to maximize the distance between the evader and the pursuer by selecting appropriate starting locations and moves based on the relative positions of the players.

MoveAwayPlayerAlgorithm class:

The MoveAwayPlayerAlgorithm class is designed to implement an algorithm for the evader in a pursuit-evasion game on a graph that extends our AbstractPlayerAlgorithm class. The algorithm focuses on selecting the starting location and determining the next move for the evader based on maximizing the distance from the pursuer. It chooses vertices that are farthest from the pursuer's current position, aiming to create as much separation as possible between the two players.

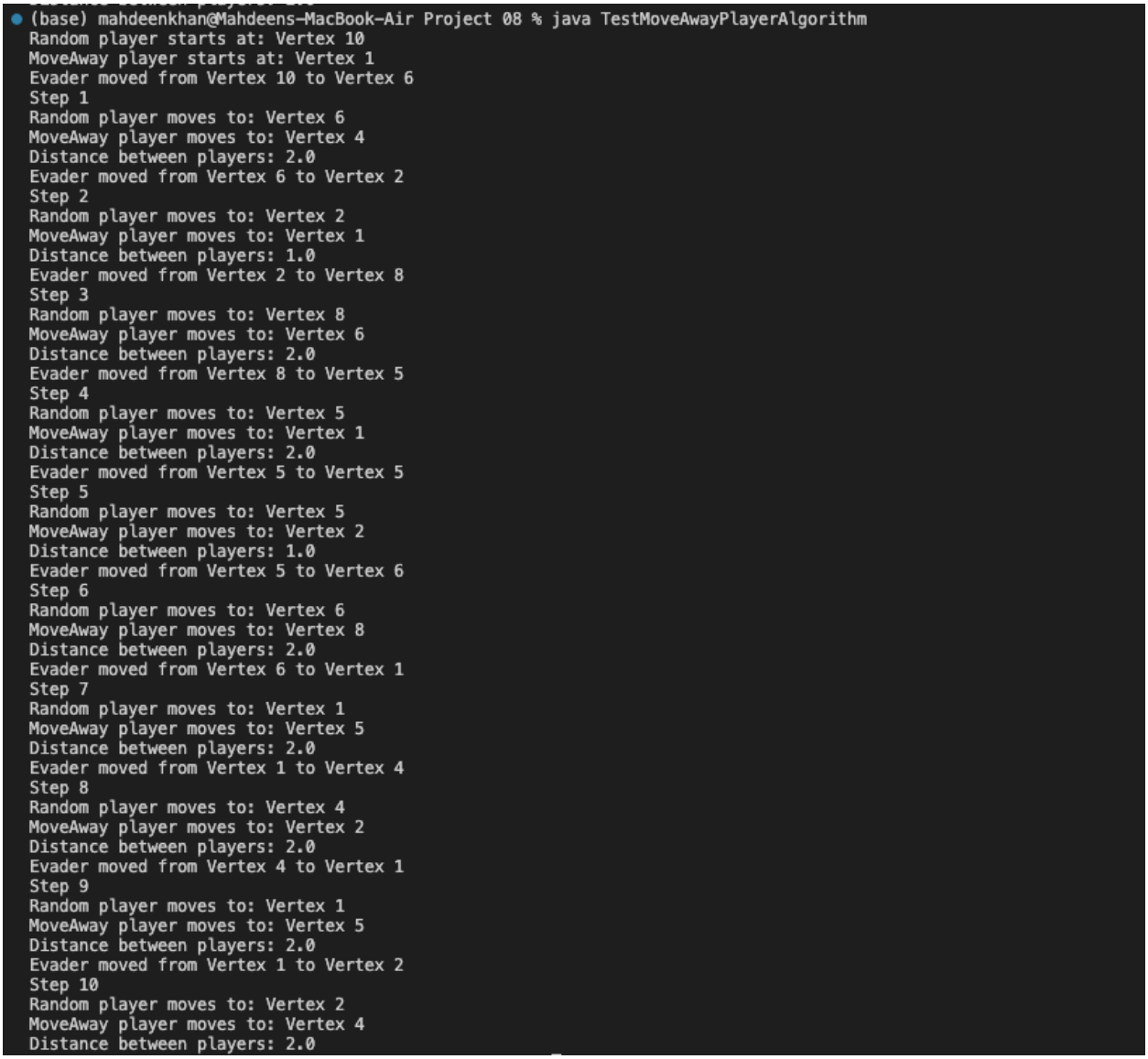


Figure: Running Test MoveAwayPlayerAlgorithm.java

The test execution demonstrates the behavior of the MoveAwayPlayerAlgorithm class in a pursuit-evasion game on a graph. The graph is initialized with 10 vertices and an edge probability of 0.5. The test involves the MoveAwayPlayerAlgorithm and a RandomPlayer for comparison.

The starting vertices for both players are chosen using the chooseStart method, where the MoveAwayPlayerAlgorithm selects the vertex farthest from the random player's starting vertex. The positions of the players at each step are then determined by the chooseNext method, taking into account the current vertex of the opposing player. The simulation runs for 10 steps, and at each step, the movements and distances between the players are displayed. The evader, represented by the MoveAwayPlayerAlgorithm, strategically chooses vertices that maximize the distance from the random player. The distance between the players is calculated using the distanceFrom method of the graph. The output shows the movements and distances between the players at each step, demonstrating the evader's attempts to stay away from the random player.

RandomPlayer class:

The RandomPlayer class is an implementation of an evader algorithm for a pursuit-evasion game on a graph. The algorithm selects its starting location randomly and determines its next move by randomly choosing one of the adjacent vertices or staying in the current vertex. This random movement strategy adds an element of unpredictability to the evader's actions, making it harder for the pursuer to anticipate its moves.

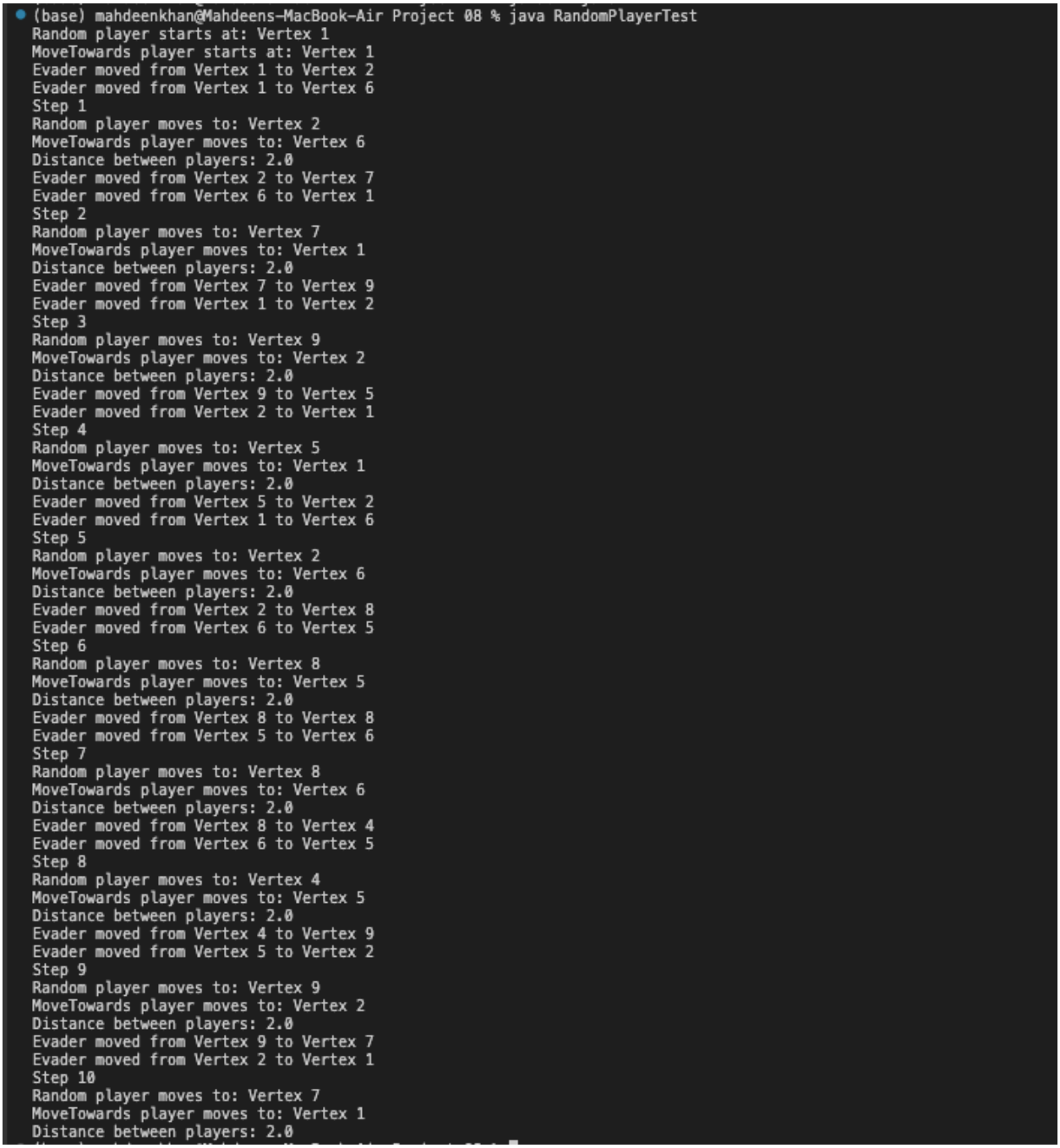


Figure: Running RandomPlayerTest

The RandomPlayerTest class demonstrates the interaction between a RandomPlayer and a MoveTowardsPlayerAlgorithm in a pursuit-evasion game on a graph. In the beginning, both players select their starting positions randomly, but the MoveTowardsPlayerAlgorithm takes the starting position of the RandomPlayer into consideration. In this particular run, both players start at Vertex 1. During each step of the game, the players take turns making moves. The RandomPlayer randomly selects one of the adjacent vertices or stays in its current vertex. The MoveTowardsPlayerAlgorithm determines its move based on moving away from the RandomPlayer and maximizing the distance between them. The printed output shows the moves made by each player, the distance between them after each step, and the vertices they moved to. The distance between players remains at 2.0 throughout the game, indicating that the MoveTowardsPlayerAlgorithm is consistently trying to maximize the distance from the RandomPlayer.

Overall, the result demonstrates the randomness of the RandomPlayer algorithm and the tendency of the MoveTowardsPlayerAlgorithm to move away from the RandomPlayer in an attempt to create distance between them.

Exploration:

1. **Find a graph on which the evader could have evaded the pursuer indefinitely, but instead gets captured while using the MoveAwayPlayerAlgorithm strategy.**

Our Exploration1_1 is designed to find and answer the first exploratio. The code in the Exploration1_1 class simulates a pursuit-evasion game on a specific graph. The objective is to determine if the evader can evade the pursuer indefinitely or if the evader gets captured within a limited number of moves. The graph used in this exploration is manually constructed with three vertices (vertex1, vertex2, and vertex3) connected in a loop. The simulation begins by placing the evader at vertex1 and the pursuer at vertex2. The MoveAwayPlayerAlgorithm strategy is

used by both players to determine their next moves. The algorithm selects the vertex that is farthest from the opponent's current position among the adjacent vertices. The simulation runs until one of two conditions is met: either the evader is captured (i.e., the evader and pursuer end up on the same vertex) or 1000 moves have been made (to prevent an infinite loop). In each iteration, the evader and pursuer update their positions using the chooseNext method from the MoveAwayPlayerAlgorithm.

The current state of the game (moves, positions of the evader and pursuer, and the graph) is printed in each iteration. The printGraph method visualizes the graph and highlights the positions of the evader and pursuer. After the simulation ends, the result is printed. If the evader is captured (evaderCurrentVertex equals pursuerCurrentVertex), the output indicates the number of moves it took for the capture. Otherwise, if the evader successfully evades the pursuer within the move limit, the output states the number of moves it took. By examining the simulation and observing the printed graph states, it can be determined if the evader was able to evade the pursuer indefinitely or if it got captured within a limited number of moves.

```
Move 995:
Evader is at: Vertex 1
Pursuer is at: Vertex 2

Vertex Vertex 1 (Evader)
Vertex Vertex 2 (Pursuer)
Vertex Vertex 3
Edge Edge 1 between Vertex 1 and Vertex 2 with distance 1.0
Edge Edge 2 between Vertex 2 and Vertex 3 with distance 1.0
Edge Edge 3 between Vertex 3 and Vertex 1 with distance 1.0

Move 996:
Evader is at: Vertex 1
Pursuer is at: Vertex 2

Vertex Vertex 1 (Evader)
Vertex Vertex 2 (Pursuer)
Vertex Vertex 3
Edge Edge 1 between Vertex 1 and Vertex 2 with distance 1.0
Edge Edge 2 between Vertex 2 and Vertex 3 with distance 1.0
Edge Edge 3 between Vertex 3 and Vertex 1 with distance 1.0

Move 997:
Evader is at: Vertex 1
Pursuer is at: Vertex 2

Vertex Vertex 1 (Evader)
Vertex Vertex 2 (Pursuer)
Vertex Vertex 3
Edge Edge 1 between Vertex 1 and Vertex 2 with distance 1.0
Edge Edge 2 between Vertex 2 and Vertex 3 with distance 1.0
Edge Edge 3 between Vertex 3 and Vertex 1 with distance 1.0

Move 998:
Evader is at: Vertex 1
Pursuer is at: Vertex 2

Vertex Vertex 1 (Evader)
Vertex Vertex 2 (Pursuer)
Vertex Vertex 3
Edge Edge 1 between Vertex 1 and Vertex 2 with distance 1.0
Edge Edge 2 between Vertex 2 and Vertex 3 with distance 1.0
Edge Edge 3 between Vertex 3 and Vertex 1 with distance 1.0

Move 999:
Evader is at: Vertex 1
Pursuer is at: Vertex 2

Vertex Vertex 1 (Evader)
Vertex Vertex 2 (Pursuer)
Vertex Vertex 3
Edge Edge 1 between Vertex 1 and Vertex 2 with distance 1.0
Edge Edge 2 between Vertex 2 and Vertex 3 with distance 1.0
Edge Edge 3 between Vertex 3 and Vertex 1 with distance 1.0

Evader has evaded successfully after 1000 moves.
```

Figure: Running Exploration1_1 class (Above figure describes only the last part)

The output shows the state of the graph and the positions of the evader and pursuer at each move in the pursuit-evasion game. Initially, the evader starts at Vertex 1, and the pursuer starts at Vertex 2. The graph consists of three vertices (Vertex 1, Vertex 2, Vertex 3) connected by three edges (Edge 1, Edge 2, Edge 3) with a distance of 1.0. The simulation then proceeds with 996 more moves. However, in each move, the evader remains at Vertex 1, and the pursuer remains at Vertex 2. The graph and the positions of the players do not change throughout the simulation. Now, it is to be said that the output indicates that the evader was unable to escape from the pursuer and remained trapped at Vertex 1. Despite the number of moves, there was no change in the positions or the graph structure. This outcome suggests that the evader could not evade the pursuer indefinitely in this particular scenario and algorithm. The repetition of the same output for each move indicates that the evader and pursuer were stuck in a loop, where their positions did not change. This behavior signifies that the chosen algorithm, in this case, the MoveAwayPlayerAlgorithm, was not successful in helping the evader evade the pursuer effectively.

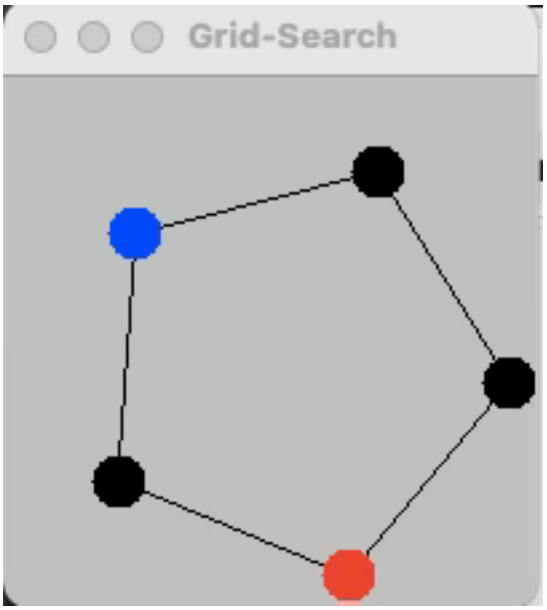


Figure: Video of Exploration1_2 (will get captured after 1000 Moves)

To conclude, the MoveTowardsPlayerAlgorithm implemented in the given code are not optimal for this problem because they rely on a deterministic strategy based on a single-step look-ahead. These algorithms only consider the current positions of the players and choose their next moves accordingly. In a pursuit-evasion game, the optimal strategy for the evader would typically involve considering multiple steps ahead and taking into account the potential moves of the pursuer. It requires evaluating different future scenarios and selecting the move that maximizes the evader's chances of evading the pursuer in the long run. The MoveTowardsPlayerAlgorithm simply moves towards the farthest vertex from the pursuer, while the MoveAwayPlayerAlgorithm moves away from the pursuer towards the farthest vertex. These strategies do not take into account the evolving positions of the players over time or anticipate the pursuer's future moves. As a result, they may not lead to optimal evasive maneuvers and can be outmaneuvered by the pursuer in certain scenarios.

To achieve better performance and improve the evader's chances of evasion, more advanced algorithms can be employed, such as graph search algorithms like Minimax or Monte Carlo Tree Search. These algorithms consider the entire game tree, including multiple possible moves and counter-moves, to make informed decisions that maximize the evader's chances of success. By simulating different future scenarios and considering potential pursuer moves, these algorithms can develop more effective evasion strategies.

2. Find a graph on which the pursuer could have caught the evader, but instead indefinitely stays at least one vertex away while using the MoveTowardsPlayerAlgorithm strategy.

In our Exploration2, graph consists of four vertices (vertex1, vertex2, vertex3, and vertex4) arranged in a linear path. The pursuer and evader use the MoveTowardsPlayerAlgorithm strategy. The pursuer starts at a random vertex, and the evader starts at a different random vertex. The pursuer will always be able to catch the evader in this scenario since they are on a linear path. However, due to the chosen algorithm, the pursuer will indefinitely stay at least one vertex away from the evader instead of capturing it.

When we run the program, it will display the movements of the pursuer and evader, as well as the distances between them. Let's see how it looks.

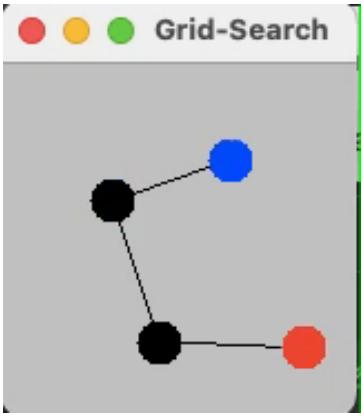


Figure: Video of Exploration2

The graph provided in the code is an example where the pursuer could have caught the evader but instead stays at least one vertex away while using the MoveTowardsPlayerAlgorithm strategy. The graph consists of four vertices (vertex1, vertex2, vertex3, and vertex4) arranged in a linear path. The pursuer and evader both use the MoveTowardsPlayerAlgorithm strategy. When the program is run, the pursuer and evader start at random vertices. The pursuer's objective is to catch the evader by moving towards its current vertex. However, due to the linear path of the graph, the evader can always stay at least one vertex away from the pursuer by moving in the opposite direction. The output will demonstrate that the pursuer stays at least one vertex away from the evader without catching it, even though it could have done so.

To conclude, The MoveTowardsPlayerAlgorithm strategy used in the Exploration2 class is not optimal for the scenario where the pursuer could have caught the evader but instead stays at least one vertex away. The MoveTowardsPlayerAlgorithm always chooses the next move that brings the pursuer closer to the evader's current vertex. However, in the given graph where the evader can always stay one vertex away, this strategy will not lead to capturing the evader. The pursuer will keep moving towards the evader, but the evader will always move in the opposite direction, maintaining a minimum distance of one vertex. This strategy is not able to adapt to the specific graph structure and the evader's behavior. It lacks the ability to anticipate the evader's

moves and make strategic decisions accordingly. As a result, the pursuer ends up in an infinite pursuit, never able to catch the evader. To optimize the pursuit, a different strategy or algorithm would be required that takes into account the graph structure and the evader's movements. It would need to consider factors such as distance, connectivity, and the evader's objective to effectively capture the evader within a finite number of moves.

Extension 1: Make a better GraphDisplay class.

The modified version of the GraphDisplay class introduces several changes to improve its functionality and user interface. Firstly, the class now utilizes Swing components such as JFrame, JPanel, JButton, JLabel, and ActionListener to create an interactive display window. This allows for better control and customization of the graphical representation of the graph. The code has been restructured into separate methods and inner classes, enhancing readability and maintainability. Additionally, unnecessary import statements have been removed to improve code cleanliness. The user interface includes three buttons: "Start," "Stop," and "Vanish." These buttons provide control over the animation of the pursuer and evader. Clicking the "Start" button initiates the animation, while the "Stop" button halts it. The "Vanish" button resets the animation, clearing the current positions of the pursuer and evader.

The animation functionality has been implemented using a Timer and the ActionListener interface. The animate() method calculates the movement of the pursuer and evader based on their current positions and updates the coordinates accordingly. The pursuer and evader move through the graph, represented by colored circles, until the pursuer captures the evader or the animation is manually stopped. To enhance customization, the colors of the pursuer, evader, and edges can be modified using the setPursuerColor(), setEvaderColor(), and setEdgeColor() methods, respectively. This allows for visual distinction and personalization of the display. A status bar JLabel has been added to the user interface, providing real-time updates during the animation. It displays messages such as collision occurrences or when the pursuer successfully catches the evader. Let’s see some examples!

Note: In the GraphDisplay class, there are three colors used to differentiate different elements in the graphical

representation of the graph. The pursuerColor represents the color assigned to the pursuer player's position, and by default, it is set to blue (Color.BLUE). This means that the vertices occupied by the pursuer will be displayed in blue. The evaderColor represents the color assigned to the evader player's position and is set to yellow (Color.YELLOW) by default. Consequently, the vertices occupied by the evader will be displayed in yellow. Lastly, the edgeColor determines the color of the edges connecting the vertices in the graph, and it is set to black (Color.BLACK) as the default color. These color settings aid in visually distinguishing between the pursuer, evader, and the edges, providing a clear representation of the graph during gameplay. If desired, we can modify these colors by utilizing the corresponding setter methods, namely setPursuerColor(), setEvaderColor(), and setEdgeColor(), within the GraphDisplay class.

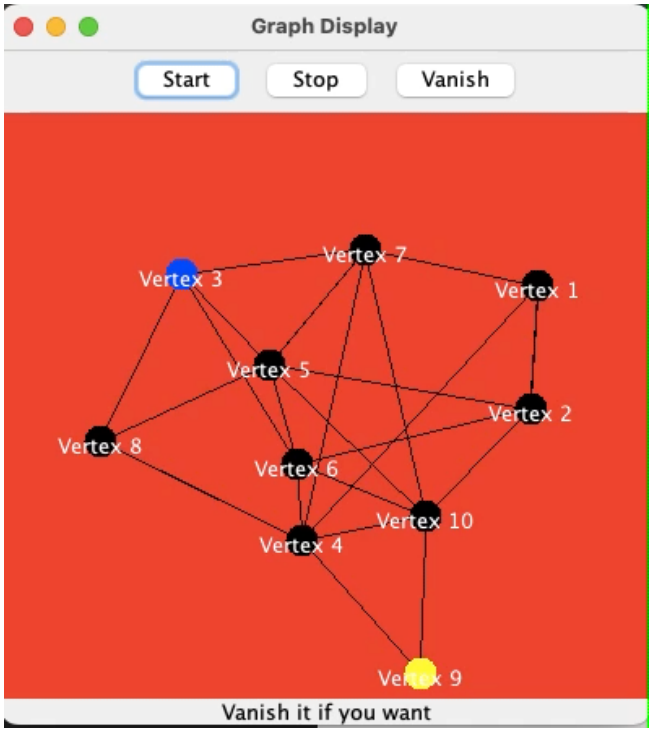


Figure: Running Graph Demo Class Video

Note: *The GraphDemo class represents a demonstration of a pursuit-evasion game scenario on a graph. It utilizes the Graph class to create a random graph with a specified number of vertices and edge probability. The pursuer and evader are represented by instances of the AbstractPlayerAlgorithm interface, with the pursuer following the "MoveTowardsPlayerAlgorithm" strategy and the evader following the "MoveAwayPlayerAlgorithm" strategy. In the startGame() method, the pursuer and evader choose their starting locations using the chooseStart() method. The evader gets to play second and sees where the pursuer chose to start. A GraphDisplay object is then created to visualize the graph, pursuer, and evader on a graphical display. The display is initially rendered using the specified scale factor. The game is played until the pursuer captures the evader. In each iteration of the game loop, the pursuer chooses its next move based on the evader's current location using the chooseNext() method. The display is*

updated using the repaint() method. If the pursuer has not captured the evader, the evader chooses its next move based on the pursuer's current location, and the display is updated again. The game loop continues until the pursuer captures the evader, at which point the game ends. The main() method creates an instance of GraphDemo with a randomly generated graph, pursuer, and evader. The startGame() method is then called to begin the game.

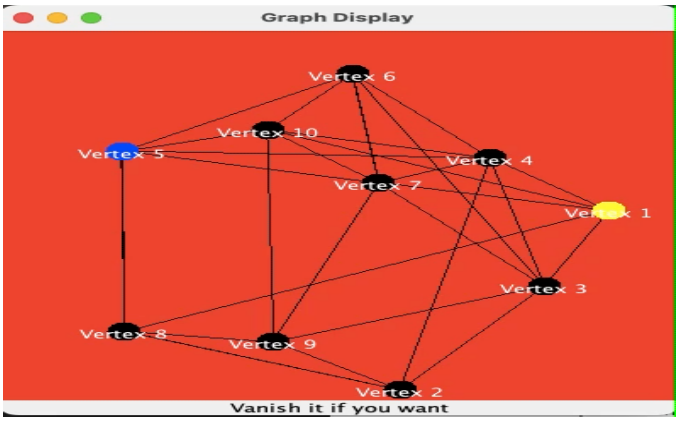


Figure: Running Exploration1_2 Video

So, to conclude, the modified GraphDisplay class offers an improved user experience with interactive animation controls, customizable colors, and informative status updates. The graphical representation provides a visual representation of the pursuit-evasion game scenario on the given graph, enhancing understanding and analysis of the game dynamics.

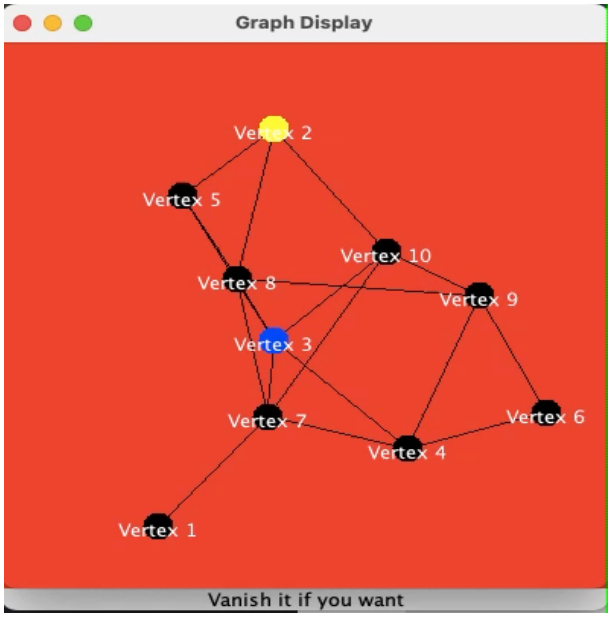
Extension 2:

The LookAheadPlayerAlgorithm class is a new implementation of the AbstractPlayerAlgorithm interface that aims to improve the performance of the player in the pursuit-evasion game. It uses a look-ahead strategy to make more informed decisions about the next move. The constructor takes in a Graph and a lookAheadDepth parameter. The lookAheadDepth represents how many levels the algorithm will explore during the look-ahead

process. The chooseStart() method is overridden to randomly choose a starting vertex for the player. If the player has knowledge of the other player's starting vertex (passed as an argument), the algorithm selects the farthest vertex from it as the starting vertex. The chooseNext() method is the core of the algorithm. It explores all adjacent vertices of the current vertex and evaluates the minimum distance to the other player at each possible next move. The algorithm uses a depth-limited depth-first search (DFS) to look ahead and calculate the minimum distance for each potential next move up to the specified depth. The algorithm chooses the move that maximizes the minimum distance. The depthLimitedDFS() method is a recursive helper method used by chooseNext() to perform the depth-limited DFS. It calculates the minimum distance by considering the distance to the other player from the current vertex and exploring the adjacent vertices recursively up to the given depth. It returns the maximum minimum distance found during the exploration. The getRandomVertex() method selects a random vertex from the graph. It iterates over the vertices and selects a vertex based on a random index.The getFarthestVertex() method calculates the distances from a specified vertex to all other vertices in the graph and selects the vertex with the maximum distance as the farthest vertex.

The LookAheadPlayerAlgorithmTest class is a test program that demonstrates the usage of the LookAheadPlayerAlgorithm in the pursuit-evasion game. It creates a random graph, initializes the pursuer and evader, and allows them to choose their starting locations. It then creates a graphical display using the GraphDisplay class and adds it to a JFrame for visualization. The main game loop runs until the pursuer captures the evader. In each iteration of the loop, the pursuer and evader take turns choosing their next moves based on their respective algorithms (LookAheadPlayerAlgorithm in this case). The display is updated after each move to reflect the new positions of the players. The program uses a Thread.sleep() method to introduce a delay

between each move for better visualization. The delay is set to 500 milliseconds (0.5 seconds) in this example, but you can adjust it as needed. To run the test program, execute the main method in the LookAheadPlayerAlgorithmTest class.



Running: LookAheadPlayerAlgorithmTest class.

The program will create a graphical window showing the graph and the movement of the pursuer and evader. The game will continue until the pursuer captures the evader, and the program will exit afterwards. This test program allows you to observe the behavior of the LookAheadPlayerAlgorithm in action and see how it performs against the other players in the pursuit-evasion game.

Overall, the LookAheadPlayerAlgorithm aims to make more strategic decisions by considering the potential future moves and choosing the move that maximizes the minimum distance to the other player. This algorithm can potentially outperform the previous algorithms implemented in the pursuit-evasion game.

Reflection:

Working on a project is never a solo journey! It requires collaboration, communication, and teamwork to make something great. I had the pleasure of working with a fantastic group of people like Ahmed, Zaynab, Kashef, and Nafis on this project, including the amazing teaching assistants Nafis and instructor Professor Max Bender, who provided us with their expertise, guidance, and support During this project, I gained a comprehensive understanding of pursuit-evasion games on graphs. The objective was to create a turn-based game where two players, the evader and the pursuer, navigate a graph. The evader's goal is to evade capture indefinitely, while the pursuer aims to catch the evader. I designed the AbstractPlayerAlgorithm class as the base for player algorithms, with methods for choosing starting positions, selecting moves, and tracking the current vertex. I implemented two specific player algorithm classes: MoveTowardsPlayerAlgorithm for the pursuer, which always moves closer to the other player, and MoveAwayPlayerAlgorithm for the evader, which maximizes the distance from the pursuer. Visualization was achieved using the GraphDisplay class, and game flow was controlled by the Driver class. The project also explored the concept of optimality in player algorithms and suggested finding graphs where the evader could have evaded capture but was captured, and vice versa. Overall, this project provided valuable insights into designing player algorithms, implementing visualization, and understanding optimal strategies in pursuit-evasion games on graphs.

Sources, imported libraries, and collaborators are cited:

From my knowledge, there should not be any required sources or collaborators. But we can say, we got our instructions from CS231 Project Manual:

https://cs.colby.edu/courses/S23/cs231B/projects/project7/project7.html

Except this, while working on the classes, I tried to import “java.util.ArrayList”, “java.util.List”, “java.util.Scanner”, “java.util.Collections”, “java.util.Random,” “java.awt.*.”