Multi-Agent Surveillance

Bianca Caissotti di Chiusano, Robert Leal, Piotr Lewandowski
Asha de Meij, Husam Mekhallalati, Kaiwei Ni, Piero Rubini Cansino

***Abstract*—Modern game theory has been shaped by the influence of Multi-Agent Surveillance Systems, providing the motivation and inspiration for the further investigation needed to answering multiple research questions. This paper asks: "How different exploration algorithms perform under varying circumstances?". This paper aims to investigate the use of the unique algorithms implemented for multi-agent entities (labled as "Guards" and "Intruders") within a "Cops and Robbers" style game. The algorithmic approaches presented in this paper include the implementations of a Brick and Mortar algorithm, an AStar Search algorithm, an Angle - Marker (Cooperative Approach) Based Search algorithm, and and Angle - Rotation (Selfish) Based Search algorithm, where experiments and tests were performed to determine the computation efficiencies and overall agent performance with respect to each agent's ultimate end goal. For a Guard entity, the end goal is defined by exploring the state-space before patrolling to "catch" an Intruder, where the Intruder's end goal is to reach the target-node before being caught by the Guard. The agents utilize these state-based approaches for path-making and perform indirect communication with one another via the use of dropping pheromones on the map environment. Thus, this paper will also outline the influence of pheromones and their impact on the agent entity's performance in regard to achieving their respective goals. It should also be noted that this paper will test, evaluate, and analyze the influence of map constraints for the agent entity performances via walls, the use of teleporters, the number of agents being used on both sides, the vision range of an agent, and the agent's walking speed within the experiments. In addition, this paper will also evaluate the effect of exploration (map knowledge) on the surveillance agents' ability to catch intruders and the impact of varying vision range affect the SAs' ability to catch intruders.**

I. Introduction

In the real world, many cognition problems require an end result of knowledge that are built upon large groups of people. In the sectors of technology, finance, and economics, the results of coordinated actions are all present between large groups of entities. For example, in the scenario of a self-driving car, the decisions of an agent are based on the result of several other agents within the same scenario. Therefore, it becomes important to ask: how can we mimic multi-agent behavior between artificial intelligence (AI) agents?

A crucial part in achieving proper mimicking of multi-agent behavior is to exploit the set of coordinated actions needed to complete a specific task at hand that the agent entities must handle. This means agents must first explore their environments using their available set of actions in order to build their knowledge base before exploiting their actions to display mature decision-making and intelligence. As a result, the decisions that become exploited from an intelligent agent can be used to mimic the use and results of coordinated actions that are present within large groups of people and their respective cognition problems in the real world. Therefore, it becomes important to discuss the scope, the agent's environment, the available objects and abilities, and the agents' ultimate end goal in order to understand the set of necessary coordinated actions needed to satisfy the task at hand.

Thus, this paper intends to explore the scope of multi-agent surveillance with two agent entities known to be either a "Guard" or an "Intruder" that spawn within the environmental setting of a "Cops and Robbers" style game. To put the game environment into perspective, the game environment is composed of a two-dimensional string array, representing the map of the game, which houses a coordinate system used for spawning multiple objects such as teleporters, pheromones dropped by the agent entities, walls, and the Guard and Intruder agent entities themselves.

Respectively, the Guard and Intruder each have their own end goals within the game environment. When a Guard spawns on the map, it has no knowledge of the game environment, meaning its first coordinated task is to explore the map in order to build knowledge of the game environment. The Guard utilizes its vision to build a "mind-map" that is also structured as a two-dimensional string array, in order to help it gain a sense of knowledge of it's relative location in correspondence to other objects (walls, the target, and the teleporter) and entities (the Intruder) it may see while exploring. Once the Guard finishes exploring the game environment, it will start patrolling the map, searching through its now known state-space for the Intruder. If the Intruder is spotted at any point, the Guard will immediately start chasing the Intruder in order to catch it. If the Intruder is caught before the Intruder reaches the target node, the Guard will have succeeded, satisfying its ultimate end goal. In the case of the Intruder, when the Intruder spawns on the map, it only knows the direction vector from it's relative position in correspondence to the target node. Like the Guard, the Intruder also utilizes it's vision to build knowledge on the objects and entities present on the map, except it's end goal is to reach the target without being caught by the Guard. If the Intruder manages to reach the target node untouched, it will have satisfied its ultimate end goal.

Knowing these details that are relative to this paper's exploitation creates a discussion for how to achieve the multi-agent's end goals, which now allows this paper to explore the set of coordinated actions needed to be taken in order to do so. This allows the paper to transition to the discussion among some of algorithms and strategies within the State-Of-The-Art that were used for intelligent path-making, wall detection, chasing, teleporter use, indirect pheromone communication,

and decision-making by the Guard and Intruder entities to obtain their respective goals.

II. State Of The Art

The State-Of-The-Art presented in this paper is circulated around the topics of graph idleness, the Brick and Mortar algorithm utilized by the Intruder for exploration towards the target node, and the Guard's AStar Heuristic Search for chasing the Intruder along the path of minimum-cost.

The path-making for the Guard is based on algorithmic approach suggested in the paper: *Multi-Agent Patrolling: An Empirical Analysis of Alternate Architectures* by authors Aydano Machado, Geber Ramalho, Jean-Daniel Zucker, and Alexis Drogoul published in July 2002. The paper *Multi-Agent Patrolling: An Empirical Analysis of Alternate Architectures* discusses locally shared graph idleness, which introduces the concept of "instantaneous node idleness". Instantaneous node idleness is defined as "the number of cycles that a node has remained unvisited", which is used within the implementation that will be presented in this paper. In short, this paper adopts the concept of instantaneous node idleness through the initialization of a "last-seen" array, which is used by the Depth-First Search Algorithm within the Guard's patrolling stage by a score system stored in the last-seen array. Thus, when Patrolling, the Guard will always pick the most rewarding move based on the scores indexed in the last-seen array.

On the topics of Brick and Mortar, along with AStar Heuristic Search, this paper will further discuss in full detail the concepts and implementation of each approach below in the Methodology.

III. Methodology

The Methodology that this paper will present contains the Logic of the Game Controller, where the Game Loop will be defined along with the agent entities, the map, and HashMap structure for clarity. Next, the Cool Search Algorithm (based off Depth First Search) used as the main exploration algorithm will be discussed alongside with its pseudocode. Additionally, the Patrolling Algorithm via Depth First Search will be presented, where the implementations of scores via the last-seen array referencing locally shared graph idleness will be introduced. Following from the Patrolling Algorithm, the AStar Heuristic Search Algorithm used by the Guard when chasing the Intruder will be explained. Additionally, the Angle - Rotation and Angle - Vector Based Algorithms utilized by the Intruder when traversing towards the target node will be highlighted. Lastly, the Brick and Mortar Algorithm for Intruder exploration towards the target node will be discussed as the last multi-agent strategy.

A. Logic: The Game Controller

The Game Controller initializes an entire map for the game simulation as a two-dimensional string array. Moreover, the Game Controller initializes a HashMap for the Guard to use when performing exploration on the map. The reason for the use of HashMaps is because the map is not known to the entities at any point in time. Therefore, to allow to the Guard to remember all the tiles seen, it became necessary to utilize a data type that can be expanded easily. Thus, HashMaps with array lists inside became the main design point from the Game Controller for creating the backbone and structure for the logic within the Guard's exploration of the map and Patrolling for the Intruder.

Additionally, the Game Controller is responsible for handling all the entity interactions with the map and the objects present on the map, which is done through the Game Loop. Nonetheless, the Game Loop continuously updates the GUI and has an additional responsibility for getting all the moves from all entities on the map before executing entity moves. The Game Loop will do this until the game ends, meaning either the Guard(s) have caught the Intruder(s), or the Intruder(s) have reached the target node.

1) Entities: Upon creation of an entity, When the game is running, entities will get the vision from the game controller and use a chosen strategy to decide on its next move.

2) Map: The map is contained within a two-dimensional string array, where each object is represented by a different symbol. Each symbol representation representing each object on the map is read by the GUI to properly display it to the user.

- "E" represents an entity
- "W" represents a wall
- "G" represents a guard
- "I" represents an intruder
- "T" represents the teleporter
- "V" represents the target for the intruder
- "X" represents undiscovered coordinates

B. Cool Search Algorithm (Guard)

The main exploration algorithm (the Cool Search Algorithm) is a modified Depth First Search algorithm. After extensive research, more multi-agent algorithms were discovered; however, they were not picked due to the involvement of executing a leader-electing approach with non-direct communication. In simpler terms, it felt like programming direct communication in an in-direct way, which led to all these ideas to be discarded. Thus, the Cool Search Algorithm was implemented, which simulates the possible outcomes of executing all of the available moves. At each new depth is creates X new children from each parent node, where X is the amount of possible moves to execute. The algorithm uses new information gain per depth as a heuristic. Each node's value is the maximum value from all their children added up to their own individual value. The individual value is acquired by simulating the vision at their current position and using that to update it's maps in memory, where each newly seen point is equal to 1 point. The details of the algorithm are given below.

Algorithm 1 Cool Search Tree Root

Require: List of all available moves M, Position int[] xy, Rotation rot, Empty Tiles and Walls Hashmaps, max depth $dmax$

1: **for** Move m in M **do**
2: Create a child node with the move m, deep-cloned copies of the HashMaps, position and rotation and depth 1.
3: Add the value of that child node to a list of values.
4: **end for**
5: **return** The move, whose corresponding child node has the highest value.

Algorithm 2 Cool Search Tree Node

Require: Move m, position $int[]xy$, Empty Tiles and Walls Hashmaps, Rotation rot, current depth d, max depth $dmax$, List of all Available moves M

1: Execute the given move, in result possibly changing xy and rot.
2: Simulate the vision an agent would receive at current xy and rot, where the agent assumes that every tile he has not seen is empty.
3: Calculate the node's value, by updating the Hashmaps with the simulated vision, where each new empty tile seen is one point.
4: Divide the node value by current depth.
5: **if** d == $dmax$ **then return** The current node value.
6: **else**
7: **for** Move m in M **do**
8: Create a child node with the move m, deep-cloned copies of the HashMaps, position and rotation and depth $d + 1$.
9: Add the value of that child node to a list of values.
10: **end for**
11: **return** The node's value summed with the max value from the children.
12: **end if**

1) Cool Search Strategy (Guard): While this strategy is running, explored tiles, wall obstacles, objects on the map, and visited points are all stored within HashMaps and ArrayLists. Through the vision acquired, the agents update their HashMap values. The entity chooses the best move based on the biggest information gain/turn. The agent will simulate its movement, vision, and updating of its HashMaps to therefore determine its information gain, which can be defined as the newest empty tiles seen by its simulated vision. The behavior of the entity utilizing "cool search" uses Depth First Search to find the best move. If there is no information gain at the current depth, the entity will create two-dimensional string representation storing the entity's findings so far. Based on this current string representation, the entity will try to use that to find possible future exploration points. If they exist, the entity will start path-making towards them. In the latter case, where the agent does not find future exploration points but has discovered a teleporter, the agent will path-make to the teleporter and use it. If there is absolutely nothing else for the agent to explore, the agent will become "stuck" in the sense that it is done.

C. Depth First Search Algorithm (Guard)

The Guard agent's main purpose after completing its exploration of the map is to catch the Intruder. This task can be divided into two sub-tasks:

- Patrolling the map
- Chasing the intruder once it finds it

1) Patrolling Algorithm - Depth First Search (Guard): The Guard works with a Depth First Search algorithm to revisit areas of the map it has explored.

In the Guard agent's memory (represented as a HashMap) is a map where each location is assigned an integer value that increases every turn the agent is not seeing it.

Using a recursive Depth-First Search algorithm, the Guard runs through all its possible moves:

- Walking
- Turning in one of the other 3 directions besides the current direction
- Using a teleporter if it's facing one

It then simulates the state of the Guard with its new rotation, where the score is then calculated to determine which move returns a higher score*. Once the Guard calculates the score of the squares it sees, it then sets the integer values of the squares to 0. It then (up to a certain depth) does these steps repeatedly. Once it reaches its final depth, it returns the moves that net the highest values.

2) Score:* The score is calculated from all the positions (henceforth referred to as squares) the Guard sees. For example, if the Guard has a vision range of 5, it sees 5 squares ahead of it with a "width" of 3, so it also sees the squares on both side of it. Every square possesses an integer value that increases every turn, the Guard seeks to maximize its score by finding the moves that return the highest integer sum from the squares.[2]

D. A* Heuristic Search - Chasing Algorithm (Guard)

When the Guard is exploring the map, it is using its vision to be able to store the locations of the entities present on the map. Within the code-base, the Guard will continue to build it's own mind-map using the Cool Search Strategy algorithm. However, once the Guard sees the Intruder, the Guard will stop exploring the map (utilizing the Cool Search Strategy) and switch over to the A* Heuristic Search algorithm for chasing. The Guard will then store its current position and the Intruder's position (via a calculation of coordinates based on its vision). Once the positions of both the Guard and the Intruder are known, the Guard will solely focus on "chasing" the Intruder, becoming its primary goal.

A* takes input of the start and ending coordinate positions (Guard and Intruder) before creating its path-making decision to traverse through the map. A* will take the given map as

input and create a grid of cells, where a ”Cell” represents an object that can be associated with a specific score value. This score value is based on a open set of nodes that need to be evaluated, which are stored in a priority queue. As a result, the cells with the lowest cost will go first in the priority queue. In addition, A* also creates a closed set of cell nodes that have already been evaluated for completeness. It should be noted that the Manhattan distance is used as the heuristic for A* to assign score values to each node (also know as the Cell object in the grid map). Nonetheless, A* will continuously update the cost if needed as it processes through the grid, heavily penalizing diagonal moves (as we only allow all agent entities in the game environment to move up, down, left, and right). This means diagonal moves will never be allowed in the final path-making of the guard, meaning a path either exists with the allowed movements or does not exist. Thus, this will allow the Guard to create a path of Cell objects, which are stored as coordinates in a Linked List data structure, that direct it towards the Intruder in order to capture it based on the path of least cost.

E. Angle-Based Algorithm: Angle - Marker (Cooperative) and Angle - Rotation (Selfish) (Intruder)

The Intruder agent is an intelligent agent that uses either a calculated direction or the exploration algorithm to decide on its next move. Before each move, the direction of the target is calculated and passed to the Intruder. This is given by the angle of the vector from the Intruder's current location to the target. Based on the calculated angle, the Intruder will know its next local rotation. As the direction is based on a vector between two points, which does not take into account the presence of walls, the intruder is likely to get stuck. Thus, if it encounters a wall, it will acknowledge that it is stuck and will start to explore the environment using the Guard's Cool Search algorithm. Once the Intruder is not stuck anymore, the direction will be calculated again. Similarly to the Guards, the Intruder can spawn anywhere in the map, with any given global rotation. Nonetheless, it will believe that its starting coordinates are (0,0) and that it is facing forward. After each move, its local idea of the map it has been exploring will update. This is also based on its given walk speed and eye range. Once the intruder sees the target it will place itself on top of it and the game will end.

On that basis, two separate Intruder algorithms were then implemented, a Cooperative based approach and a Selfish, more aware but slower Intruder. Both Intruder algorithms use the Angle-Based approach mentioned previously; however, the cooperative Intruder releases markers when it sees the goal. When these markers are then seen by the other Intruders on the map, they (except the one Intruder releasing them) start to use the exploration algorithm, as to distract the Guards from the Intruder approaching the goal (the one releasing markers). Meanwhile the Intruder that spotted the target moves towards it as fast as possible. The second (Selfish) Intruder does not use markers, instead it has increased awareness of its surroundings as it does a 360 turn after every move. If it does spot a Guard it attempts to move away until the Guard is no longer in sight, then it reverts to the Angle-Rotation movement.

F. Brick and Mortar Algorithm For Exploration Towards Target (Intruder)

The Brick and Mortar algorithm enables that multiple agents can work together to explore a variety of terrain, although the agents have no prior knowledge of the map. In this algorithm, the map is represented as a grid of cells. Each cell can be in one of the following states:

- Wall: This cell cannot be traversed by any agent, since the agent is blocked by an obstacle.
- Unexplored: This cell has not been visited by any agent.
- Explored: An agent has traversed this cell at least once but may be traversed again to reach other unexplored cells.
- Visited: No agent can traverse a visited cell. This cell has been explored and is not needed to reach other unexplored cells.

The fundamental concept behind Brick and Mortar is to gradually thicken existing walls by labeling the cells around them as visited. It's worth noting that visited cells are the same as wall cells in that they can't be reached anymore. In the description of Brick and Mortar, inaccessible cells are referred to walls and visited cells, whereas accessible cells refer to unexplored or explored cells. The goal of Brick and Mortar is to gradually thicken the blocks of inaccessible cells while keeping the accessible cells connected.

This algorithm consists of two discrete steps: the marking step, and the navigation step. During the marking step, the state of the current visited cell is updated by an agent who chooses between the explored and visited states. A cell is marked as visited if it does not block the path between two accessible cells. The cell is marked as explored if such a path is nonexistent. In the navigation step, the agent's goal is to visit an unexplored cell that is likely to be marked as visited. This is done by selecting the neighboring cell with the most inaccessible cells i.e., walls or visited cells. If there is no such cell, the agent goes to one of the unexplored cells. In case all cells in the 4 directions are inaccessible, the target exploration task is finished. The details of the algorithm are given below.[1]

Algorithm 3 Brick and Mortar

Marking Step
if the current cell is not blocking the path between any two *explored* or *unexplored* cells around **then**
    mark the cell as *visited*
else
    mark the cell as *explored*
end if
Navigation Step
if at least one of the four cells around is *unexplored* **then**
    for each of the *explored* cells see how many wall or *visited* cells are around it, then go to the cell with most of them, which is most likely to be marked as *visited* in the marking step
else
    if at least one of the four cells around is *explored* **then**
        go to one of them. Avoid selecting the cell where you came from unless it is the only candidate. Instead select the first *explored* cell in an ordered list of adjacent cells, e.g [North,East,South,West] The order of cells in the list depends on the agentID, so that different agents disperse in different directions.
    else
        terminate. All adjacent cells are inaccessible, i.e. *visited* or *wall* cells.
    end if

IV. Experiments and Results

These will be discussed at the end of the document.

V. Discussion

In smaller maps, the guards seem to perform better when not in patrolling mode (releasing pheromones), a possible explanation to this would be that since the guards would favour moving into tiles with no pheromones the guards do not converge and surround agents. However in both cases when the number of intruders does not exceed the number of guards the guards end up capturing all of the intruders, albeit taking more time when patrolling (releasing pheromones). At least one intruder makes it to the goal when there are more intruders than guards.

In bigger maps, when the intruders are concentrated on either side, the guards manage to capture all the intruders, whether it is patrolling or exploring. However if the intruders are placed in varying locations (sparsely) exploration performs better than when the guards are set to patrolling. The guards starting with patrolling only succeed in capturing all the agents when they are placed close to each other on either side of the map.

We can conclude from the results of the first two experiments that for the patrolling algorithm to work the guards need to have knowledge of the map which can only be achieved through exploration.

From experiment three, we can see that an increase in map size naturally increases the number of turns the intruders need to find the goal, as it needs to perform more moves to reach a further goal or reach it from a more distant target.

Experiment four, tries placing the guard and intruder at varying starting locations, it seems that the closer the starting coordinates are to each other the higher the chance of the guard capturing the intruder is.

In the fifth experiment we try changing the starting coordinates of the intruder in the x and y axis. A decrease in distance in the y axis, results in a higher decrease in the number of turns needed to reach the goal than a decrease in distance in the x axis.

When experimenting with different eye ranges, it is noticed that the higher the eye range the faster the guards capture the agents and the less likely it is for them to go into patrolling mode. That is because the guards are able to see the intruders at greater distances and thus begin chasing them at earlier stages of the simulation.

VI. Conclusion

The research presented led to an overall improved understanding of designing a multi-agent surveillance system. In conclusion it seems like knowledge of the environment is crucial for an exploration surveillance agent's performance, guards perform much better when they start out the simulations exploring the environment and only patrol when they have explored all of it, or stop and chase when they have seen an intruder. It is also very evident that the bigger the eye range the faster the SAs are at accomplishing their task of capturing the intruders, that is mainly due to faster exploration and an increased ability to start chasing from further aways positions, it is also less likely for them to be completely done with exploration before seeing a target and chasing it with higher eye ranges. We can conclude then that there are many factors affecting the performance of exploration agents and any minor changes can dramatically change the outcomes of simulations. Moreover, this paper also highlighted the use of multiple agents (for both guards and intruders) and their impact on performance and optimally within the system. Therefore, for possible future extensions of the team's multi-agent surveillance system, the group can look at how modern game theory is influencing multi-agent surveillance systems, specifically through reinforcement learning.

VII. References

[1] Ferranti, E., Trigoni, N., Levene, M. (2007). Brick Mortar: An on-line multi-agent exploration algorithm. Proceedings 2007 IEEE International Conference on Robotics and Automation. https://doi.org/10.1109/robot.2007.363078

[2] Menezes, R., Martins, F., Vieira, F. E., Silva, R., Braga, M. (2007). A model for terrain coverage inspired by ant's alarm pheromones. Proceedings of the 2007 ACM symposium on Applied computing - SAC '07. https://doi.org/10.1145/1244002.1244164

[3] Machado, A., Ramalho, G., Zucker, J., Drogoul, A. (2003). Multi-agent patrolling: An empirical analysis of alternative architectures. Multi-Agent-Based Simulation

II, 155-170. https://link.springer.com/chapter/10.1007/3-540-36483-8$_1$1

Num. Intruders	Num. Guards	Num. Turns	Guard Success	Started Patrolling	Intruders left
1	4	30	Yes	Yes	0
2	4	31	Yes	Yes	0
3	4	30	Yes	Yes	0
4	4	26	Yes	Yes	0
5	4	18	No	Yes	3
6	4	16	No	Yes	5
7	4	16	No	Yes	6
8	4	16	No	Yes	5

Table I: Patrolling Guard Success As Number of Intruders Increase

Num. Intruders	Num. Guards	Num. Turns	Guard Success	Started Patrolling	Intruders left
1	4	25	Yes	No	0
2	4	25	Yes	No	0
3	4	23	Yes	No	0
4	4	20	Yes	No	0
5	4	19	No	No	3
6	4	16	No	No	5
7	4	16	No	No	6
8	4	16	No	No	5

Table II: Exploration Guard Success As Number of Intruders Increase