Applying Path-Finding Techniques and Swarm Technologies to Package Delivery

Alex Miu
Department of Computer Science
and Electrical Engineering
University of Maryland, Baltimore County
Baltimore, Maryland
Email: ale18@umbc.edu

Josh McCarter
Department of Computer Science
and Electrical Engineering
University of Maryland, Baltimore County
Baltimore, Maryland
Email: jmccar1@umbc.edu

Itay Tamary
Department of Computer Science
and Electrical Engineering
University of Maryland, Baltimore County
Baltimore, Maryland
Email: itay1@umbc.edu

Jack Wang
Department of Computer Science
and Electrical Engineering
University of Maryland, Baltimore County
Baltimore, Maryland
Email: zhennan1@umbc.edu

***Abstract*—The goal of this project is to experiment with different swarm techniques and path-finding algorithms in order to build the most effective independently functioning character that can work with other characters to map out and navigate a maze. To develop this project, the character will perform two functionalities, the first being maze mapping, the second being maze navigation, represented as package delivery to different locations in the maze.**

I. Introduction

Most fields of robotics use several different kinds of artificial intelligence applications. Functionalities such as navigation might implement computer vision and path-finding, while healthcare robots might one day use natural language processing to analyze patient responses. A currently growing field of technology that applies multiple aspects of artificial intelligence is autonomous vehicles. Our motivation for this project was to apply autonomous vehicle technologies to building navigation. The environment we chose is a grid-based maze, and there will be multiple agents maneuvering through the grid. Having multiple agents in the maze adds the complexities of collision avoidance and target sharing (discussed later). These individual agents, which we refer to as cars, will navigate the maze independently.

In the first phase, the goal of a car is to map out the maze by discovering unknown grid points. A grid point will be discovered by determining which sides of the grids are walls of the maze, and which sides are open paths. Each independent car in the maze will be able to map out the grid space alone, but in this project they will not have to. When a car discovers a grid point in the maze, they update their own graph of the maze, but also attempt to update the graphs of all other cars. Since multiple cars may be in the maze at once, they will have to coordinate movement so as not to crash into each other, and not to duplicate effort by selecting the same target. We refer to the latter topic as target sharing, which we define as two independent cars planning on completing the same task. An example of target sharing in the first phase is when two cars have selected the same undiscovered grid point as their goal. When this occurs, the cars would decide which one will select a different undiscovered grid point as their target to avoid duplicating mapping effort. The first phase is completed when all reachable grid points in the maze have been discovered.

The second phase applies path-finding to the maze by changing the goals of the cars. Rather than mapping the maze, the cars will be tasked with navigating to specific locations in the maze, which we refer to as rooms. Cars will move to a task-assigning room in the maze to receive their target room. The locations of the rooms in the maze are discovered during the first phase, as room locations are predetermined and built into the maze. Once a car receives their target room (we call this picking up a package), it will determine the shortest path to the target room. When a car determines its path, it must also account for the current positions of all other cars in the maze, so as to avoid collisions. If two cars have conflicting paths, they will determine which car needs to move out of the path. An example of this is a room down a single-grid isle in the maze, which only has one entrance path. If the first car is trying to leave the isle while a second car is entering, the two will have to decide how to let one car pass. Cars will not be able to transfer phase two tasks to each other.

To summarize, the overall goal of this project will be to use independently functioning characters (cars) to first map out a maze, then to navigate it by delivering virtual packages to predefined locations (rooms) in the maze. The cars will use swarming technology and techniques to prevent target sharing and collisions. They will also implement path-finding algorithms to navigate the maze and reach their targets. Different path-finding algorithms and collision avoidance techniques will be tested to find the best combination that builds the most effective car.

II. Resources

A. Languages

For this project we intend to use python to develop the simulation and algorithm framework, and to use python to use the algorithms. When the physical characters are built, they will be controlled by C programming, which will improve control over hardware portions of the characters.

B. Data Sets

1) Maze Set: A set of pre-made mazes that will be used to test and debug algorithms during development.

C. Algorithms

1) A Pathfinding:* A* pathfinding is a best first search algorithm. It takes a node on a weighted graph and gives the shortest possible path from point A to point B. The algorithm will search through the graph and create costs based off of the weights of each of the edges that we assign. For our particular application we can give each edge in the graph a weight of one to start and then if we do some analysis and see that some paths are harder than others we can recalibrate the edge weights for more optimization. This is a very flexible algorithm that will always return the shortest path no matter the graph we pass it.

2) D Lite Pathfinding:* D* Lite operates much like A* pathfinding in that it will find the shortest path from A to B. Not only can this be used for pathfinding, it can be used to map unknown areas on a given map. D* Lite does this through greedy mapping, greedy mapping is when your robot/car is currently in a known cell it will move to the next closest unknown cell. This is very useful in the first phase of our project which would entail the cars moving out and mapping an area/maze before being able to efficiently deliver packages.

3) Physically-embedded Particle Swarm Optimization: This swarm control technique decentralizes processing and decision making to the individual characters in the swarm network. It also ensures collaborative decision making, and synchronized actions, which for the purposes of this project would help prevent collisions and backups in single-lane pathways.

III. Related Works/Literature Review

A. Decision to Use D* Lite

One of the papers used to develop some of the initial designs for this project describes the original motivation behind creating D* lite which was being able to efficiently map out a space in ways that other pathfinding algorithms before it could not. Using aspects of lifelong planning A* pathfinding the authors were able to develop D* Lite to be more efficient and work in a plane with unknown terrain.

B. A* Pathfinding In Video Games

Another paper we found describes the use of A* pathfinding in video games which is essentially the same thing as our project. A* is the most popular pathfinding algorithm in game development because it is easy to prove that it is the most optimal for the problems that arise in game design.

IV. Preliminary Results

We broke the project down into four separate subsections which are maze generation, agent, swarm ai, and GUI. The goal for the mid semester report was to get an Agent in the maze and allow it to discover every point in a basic maze which required the framework of the maze class and a nearly full implementation of the agent class. We decided to come up with a few different simple schemes for pathfinding and discovery so we can set up the frameworks needed to support both D* Lite and A* pathfinding. So the simple discovery scheme we came up with looks at the vertices in the graph that are adjacent to the vertex that the agent resides on. Then it checks the edge weights that we have assigned and selects which vertex to move to based off the smallest weight. Currently we have each edge weighted as 1 but we are trying to implement a system that will change edge weights based off of what direction the agent is facing since in the real world going straight is more efficient than having to turn. If it cant find any undiscovered neighbors then it moves to the closest undiscovered node.

As far as the mazes we are testing on we wanted to test on what we consider the main building blocks of a maze. We think L shapes, T shapes, and boxes are the main building blocks of a maze. These patterns have things like terrain avoidance and backtracking which would present the biggest problem for the agent moving in the maze. Using our simple techniques we are able to get the agent to move around and discover these basic mazes. Obviously very simple methods like the ones we are using currently break down in terms of efficiency once you get to much bigger and more complex mazes and both A* and D* Lite help remedy this problem. The next step for us would be to polish up the framework of the program and then implement the bigger and more efficient pathfinding algorithms and then create a GUI to allow for more efficient testing.

V. Final Implementation

A. Maze Representation

The pathfinding algorithms used in our project require a weighted graph to operate and find paths on. So the maze class holds a list of nodes that are all connected using our encoding that we used to represent edges when generating the random mazes. Where there are no connections there are walls so the pathfinding algorithm is able to go around obstacles. Our graph representation of the maze is uniform in that the edge weights are all one since we try to have each node represent a unit length in the real world.

B. Agent Implementation

The agent utilizes a mixture of algorithms in order to discover points on a maze and path correctly and efficiently to nodes across the maze. Agents have a copy of the master maze that is updated accordingly each time they discover a new node in the maze. Each agent also has a path variable that is updated every time an agent is pathing to a new node in the maze. The last important variable of the Agent class is the LIFO queue that we use in the discovery algorithm that holds the closest undiscovered node at the front of the queue. We implemented a greedy mapping algorithm that is able to discover every node in a maze. The greedy mapping algorithm attempts to only look at nodes that are close to the current position of the agent to avoid traveling across the maze as much as possible. The Agent or Agents are initially placed at starting points in the maze. From that starting point they observe their surroundings and see what nodes that are connected have yet to be discovered. Since the edge weights are all set to one the Agent will select one of the neighbors randomly. Obviously with this random selection the discovery algorithm is not deterministic and will likely not return the same thing twice, especially as the number of nodes is increased in the maze. With the next target selected as the random selection the rest of the undiscovered neighbors are pushed on to a LIFO queue. We went with a LIFO queue because when you push a node on to the queue and then go to pop the front off, the node that is given will be the last undiscovered node that the Agent decided to not path to. This queue is utilized when an Agent gets put in a position where it has already discovered all the nodes that surround it. When this happens the Agent pops the first element off of the queue and paths over to that node. From there the agent might find more nodes to discover or it might have to resort to taking from the queue again. This process is repeated for each agent in the maze until the master maze has no undiscovered nodes remaining.

The pathfinding algorithm that we implemented is A* pathfinding. A* is a pathfinding algorithm that was based off of Dijkstras algorithm but uses heuristic analysis in order to increase efficiency. The algorithm works by being fed a node to start on in our maze and a node to end at. The A* algorithm holds an open set and a closed set which hold nodes that are either waiting for search or have been searched. A* also uses special variables to test the fitness of a node, namely f-score, g-score, and h-score. G-score is the distance a current node is from the start and this is incremented every time we get further away from the start node. H-score is the heuristic that is used by A*, for a four way movement grid it is ideal to use the Manhattan Distance heuristic which is the sum of the absolute distance in the x and y directions from point a to point b(the legs of the right triangle with hypotenuse that connects a and b). F-score is simply the sum of g-score and h-score. So the main structure of A* is a loop that checks if the open set has no nodes left in it. From there we start evaluating nodes to find parts of the path. So first we take the node out of the open set that has the lowest f-score. From there we look into the neighbors of the node that we selected. For each neighbor we compute a hypothetical g-score which is just the g-score of the node we selected plus one since edge weights are all one. If the neighbor isnt in the open set then we should add it there and if the hypothetical g-score is greater than the actual g-score then we shouldnt evaluate the node. Otherwise we found a part of the path and the parent of the neighbor should be set to the node we selected and compute h-score and f-score for the neighbor to the end goal. This loop runs until the end node is found. From there to reconstruct the path you just need to trace the parents of all the nodes from the end goal.

C. Swarm Implementation

When multiple agents are introduced into a maze, the maze takes substantially less time to discover, but the potential for collisions arises. Defining that two agents cannot be in the same node in the maze means that agent management and collision avoidance must be implemented. This is the idea behind a swarm algorithm. Formally, it is meant to define communication and decision making between agents in regards to distributing the workload, duplicating effort, and avoiding inter-agent problems. The idea for this comes from mimicry of collective animals such as ants and bees. With the multi-agent ecosystem inside of the maze, such an algorithm is required to ensure that agents can work together and discover the maze more efficiently than if agents were acting independently.

The particle swarm optimization algorithm is a generic swarm algorithm that, instead of managing agents continuously, iteratively develops a solution for the swarm that optimizes the swarms total solution. This means that individual agents (referred to as particles) are continually asked to re-evaluate their solution with minor conditional changes until their individual solution most effectively benefits the swarm solution. Once the most optimal swarm solution has been achieved, the algorithm is complete. For the purposes of this projects, we used two different particle swarm optimization algorithms. The first algorithm is the Physically-Embedded Particle Swarm Optimization (referred to as PPSO), where all management and decision-making is decentralized, meaning each node can run independently or interact with other agents without relying on some queen or manager. The second algorithm, Extended-Particle Swarm Optimization (referred to as EPSO), has the same decentralized activity as PPSO, but restricts agents to only communicating with agents some distance n away, where n is referred to as the radius around some agent.

Both of these algorithms require some conditions or components of a solution. In this project, the solution is composed of potential collisions, duplicate targets, and target swapping. The algorithm iterates through the agents until none of these conditions will occur in the next cycle. A cycle is defined as a movement of all agents by one node. The algorithm runs every cycle to ensure that there are no collisions, and that effort is not duplicated. To prevent the duplication of effort, both algorithms check for target sharing, or duplicate targets. This occurs when both agents have chosen the same node as their target destination. If this were permitted, then multiple agents would waste cycles traveling to the same node. Therefore, only the agent nearest to the target location gets the target, an all other agents with the same target are told to re-select their targets. Distance is defined as the Manhattan distance between nodes.

There are four different collision types. Each of these is mitigated in the way described below:

1) Glance Collision: This kind of collision is similar to when there is an intersection between roads. Two agents are passing each others paths at a right angle, and so one agent must stop to allow the other to pass. Once the other agent has passed, the first agent can continue its path.

2) Head-On Collision: This kind of collision is when two agents run into each other heading in opposing directions. Rather than moving one agent to the side so the other can pass, agents will swap targets, therefore not needing to pass each other at all. Swapping targets is performed any time agents are closer to each others target locations than they are to their own target locations. By doing this head-on collisions will never occur.

3) Non-moving Collision: This kind of collision occurs when one agent runs into a non-moving agent on its path. This agent may be paused due to a glance collision or traffic jam, or it may be inactive if there are no more undiscovered nodes to claim. If the agent in the way is paused, it will resume at some point in the future, so the current agent pauses as well to wait. This is in part the solution to a traffic jam, discussed below. If the other agent is inactive, they swap targets and the previously-moving agent becomes inactive. This is similar to a hand-off of a package.

4) Traffic Jam: This occurs when multiple agents are stacked behind each other, perhaps trying to pass through a narrow path or the equivalent of a doorway. When this occurs, agents are naturally queued by arrival time to the traffic jam, and one agent is let through at a time.

D. User Interface

When the maze and number of agents have been decided, our program will display the UI which includes one window for the whole map and n (number of agents) sub-windows for each agent.

For the whole map window, we have two frames, the left frame displays the map and the right frame displays buttons. In the left frame, we use a grid layout, and use background colors to represent the state for the location. A black background is for locations that are unreachable, gray nodes are for locations that the node is undiscovered in, green for locations where the node is discovered, and red for locations of agents. The max number of pixels for width and length is 800 pixels depending on the number valid nodes locations. If maze is small enough, then there are 50 by 50 pixels for each location. Three buttons in the right frame include one step, start, and pause. One-Step button lets each agent move one step. Start button will keep each agent moving until the pause button is clicked.

For each agent window, their algorithm is the same. The window uses grid layout, and 5 by 5 locations will display where each location is with 50 by 50 pixels. The center is the agent location, and other places are the locations around the agent. We have different images for different states of the location. For example, location that is without a node, we will use cloud to represent it. So the user can easily know the situation of each agent.

E. Maze Generation

Although no Artificial Intelligence techniques were directly implemented in the generation of sample mazes, this stage provided crucial data sets for the training of our system. There were several considerations taken during the maze generation process with scale, complexity, and variation among the most important. The maze had to mimic the physical layout of a confined space (room) where furniture, support beams, and other obstructions of varied dimensions exist. To accurately simulate such a space, we decided to generate these mazes in a similar way to how a physical room is built. First, the size and dimensions of the room are determined. Next, the walls (outline) of the room are built in accordance to the determined dimensions. Finally, furniture and other obstacles are brought in and usually placed in a uniform or planned distribution but can also be placed randomly. We decided to simulate the latter as it is more general and widely-applicable for the purposes of our system. With that, the maze was generated as follows in a JavaScript program within the NodeJS environment:

We specify the area or square length of our desired space. The fundamental unit for our system was cm2. The scale was determined to provide the most favorable balance between flexibility and computational resource consumption.

With the desired area, the program randomly generates the length and width dimensions that produces the desired area. For the purposes of variation, the proportion between length and width (and vice-versa) is randomly determined to be between 60 - 100 percent of one another, where 100 percent indicates the confined space will have the dimensions of a perfect square.

Using a two-dimensional (technically three-dimensional) row-major matrix that corresponds to the position coordinates of the room, we generate an empty space. The width of the room is defined as the row (vertical) and length is defined as the column (horizontal). Each element or cell within the matrix represents 1 cm2 of our space, where each cell is an array of four bits, each bit corresponding to a barrier or wall in four directions: up and down (horizontal lines), left and right (vertical lines); respectively. A 1 signals a barrier, while a 0 indicates no barrier, at its respective position to the cell.

Example. A confined square space of 4 cm2 given by 2 cm X 2 cm is represented as:

[1, 0, 1, 0] [1, 0, 0, 1] (10101001) [0, 1, 1, 0] [0, 1, 0, 1] (01100101)

An empty cell/space is defined by: [0, 0, 0, 0] (0000)

Once each cell is defined as an empty space, we have a confined space of our desired area. We then generate obstacles, this adds complexity to our space. We randomly generate obstacles based on the distribution specified by the complexity scalar. The greater the complexity scalar, the more likely a space will be occupied by an obstacle and consequently inaccessible by an agent. If a space is randomly selected to be the starting point of an obstacle, the script generates the corresponding barriers along the outside outermost row and column cells. The internal cells within these walls become inaccessible to agents. As an obstacle is essentially a smaller maze: its shape, size and dimensions can also be varied and randomly generated.

An agent is treated as a temporary obstacle of 1 cm by 1 cm (1111) taking up 1 cm2.

After generating the unique confined space and its internal obstacles, the script produces a text file with all the information necessary to be reparsed by the Swarm program to create a graph that simulates a physical space. This script can be set to run any number of times, allowing us to produce large data set of uniquely configured spaces.

The Swarm program creates an instance of the Maze object which holds the graph simulating our space. The Maze class takes in the name of the text file corresponding to the desired space and parses the bits into a new two-dimensional matrix holding the nodes of our graph. Every line in the text file corresponds to each row, and every four bits (or characters) correspond to a column where the node is held. Once initialized with its relative position to the origin point (0, 0) of our space. The nodes are then connected accordingly with their adjacent node if there is no barrier (1) in between them. After connecting all the nodes, we have a grid of nodes and our graph. As stated earlier, an obstacle is essentially a smaller maze and through the connection process, all the internal empty space of an obstacle also become interconnected, creating separate individual graphs within the obstacle. These internal graphs are not connected to the outside space as barriers separate them. With this implementation, agents can map out internal physically-unconnected areas if desired and allow for greater flexibility in wider application. For the purposes of our system, we simply want to map the greater outer space, and so we must designate these nodes of the corresponding graph as valid. A condition for our system is the graph containing the origin of the space, point (0, 0), will always be the desired graph and that each individual agent must have at least one path connecting it to every other agent. With this criterion met, a simple Breadth-First Search is done starting at the origin node. Any node connected to the origin will be visited and subsequently marked as valid. Once all the nodes of our desired space are accounted for, we can randomly place agents in a valid starting point as well as extract spatial data of our space. The graph is then handled by the Swarm program.

F. Program Drivers

Because there are two different use cases for our code, there are two different drivers to run it. The UI Driver utilizes the visualization done by the user interface to help present the maze and the agents movements to the user. However, testing for large amounts of data with a UI requires human interaction. Instead, the second driver does not use a user interface, and performs various tests on the maze changing initial variables, such as the maze, the number of agents, and the radius if the Extended-Particle-Swarm-Optimization algorithm is used to manage the swarm. A third testing program was written in addition to these to automatically run multiple pre-generated maze files. Output data is stored in a CSV file. For information on how to use the driver, please visit the README.txt file in the codebase.

VI. Initial Testing

A. Discovery and Pathfinding

Initially we had very few mazes for proof of concept tests that we expected to translate into bigger mazes once we implemented the final discovery and pathfinding algorithms. We tested these algorithms on basic maze structures to test aspects of discovery and to make sure A* was working properly. These initial mazes were an L maze, T maze, and a square maze with a pocket in the center. Each of the initial mazes first proved that the Agent was operating correctly and could individually discover every node in the maze. The most important factors to analyze were undiscovered nodes being queued for later exploration, and the paths to those nodes being properly returned by the A* algorithm.

B. Swarm Collision Testing

After the agent class had been tested and proved error-free, the swarm algorithm needed testing. This involved designing custom mini-mazes that would force various collision cases and other conditions that needed to be caught and handled by the swarm algorithm. Examples of tests include mazes that force parallel movement, head on collisions, glance collisions, and target swapping. Each condition was first tested with two agents, then a varying number of agents. The purpose of these tests, much like the purpose of the tests for the Agent class, was to prove that the swarm algorithm works, and can adjust the agent movements to prevent collisions and make collaboration more efficient. After fixing a few errors, these tests came back successful, showing the swarm algorithms would properly manage a variety of agents.

C. Complete Initial Test

Finally, once both algorithms were error free and proven to work, the final stage was to piece everything together. The first test with the new driver was with a small maze, generated by our maze generation program, and four agents. The driver ran the maze once with one agent, once with two agents, once with three agents, then once with four agents. It then finished and generated the output report file, a CSV file with all of the potential collision cases, number of agents, and other information about the running environment of the maze. This test showed that the generated mazes work, and that the agents were able to navigate them, and that the swarm algorithms was able to manage the agents. With just this one run, we were able to see stark changes in the various outputs given a change in the number of agents. A full analysis of the results we found is presented in the Final Results section.

VII. Final Results

A. Maze Data-Set

Since there was no reliable way for us to retrieve pre-generated mazes that we could find easily online we decided to write a script that would make mazes with randomly generated obstacles ourselves. The script creates mazes of the same size but with varying dimensions(same number of nodes in the maze, just different length and width). Then it adds obstacles randomly throughout the open space and encodes the maze into a text file that is read by the Maze class in the python code. In total the script created 15,000 mazes with 1600 nodes, 210 mazes with 100,000 nodes, 100 mazes with 400,000 nodes, and 100 mazes with 800,000 nodes. Each node is meant to represent a unit length of any type centimeter, meter, foot, etc.

B. Test Results

The tests we ran set a varying number of agents in a set of mazes, and kept track of various events that occurred in each run. The variables set include the number of agents in the maze, for which each maze had one agent through the max number of agents. We also set the maze itself, selecting a random set of mazes from our larger generated set. Some of the output variables we kept track of include the number of cycles it took to complete a maze with n many agents, the collisions/events and types of collisions/events that occurred at each cycle, and the number of agents that were active each cycle. These data points are collected, organized, analyzed, and presented in the remainder of this section.

We pulled from a set of 15,000 test mazes that all had a maximum number of nodes that was equal to 1600. Each maze has randomly generated obstacles so each maze is different from the others in terms of connectivity. We were interested in different aspects of the discovery phase for a given maze since the discovery phase is the most computationally expensive and we want to make discovery cost as little as possible.

The first statistic we were interested in was the average number of cycles it took our test cases to run discovery in for up to 64 different agents. So we tested each maze with 1,2,3,...,64 agents and recorded the number of cycles it took to finish. We collected all the number of cycles for n-many agents and took the average and plotted the average against the number of agents to end up with figure[1]. We expected a linear difference in time as the number of agents increased before we improved our testing framework but obviously this shows a much greater relationship. This graph is very close to exponential decay in the number of cycles as a function of the number of agents. This is somewhat expected for a maze of this size since as the number of agents increases each agent can only do so much because so many other nodes are being discovered by the other agents.

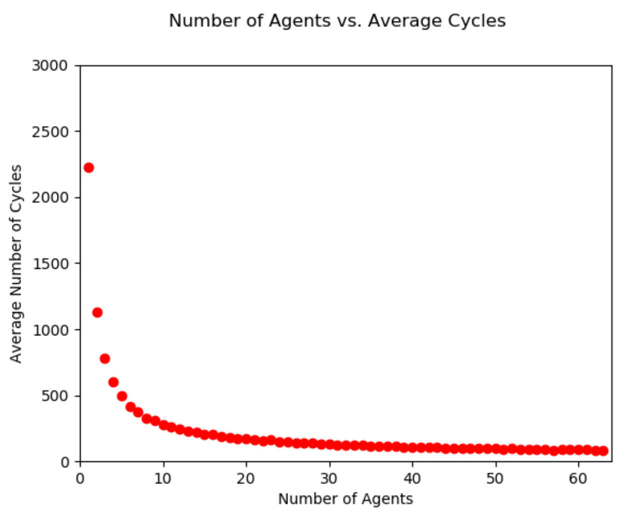

Fig. 1. Average Number of Cycles Needed to Discover Maze

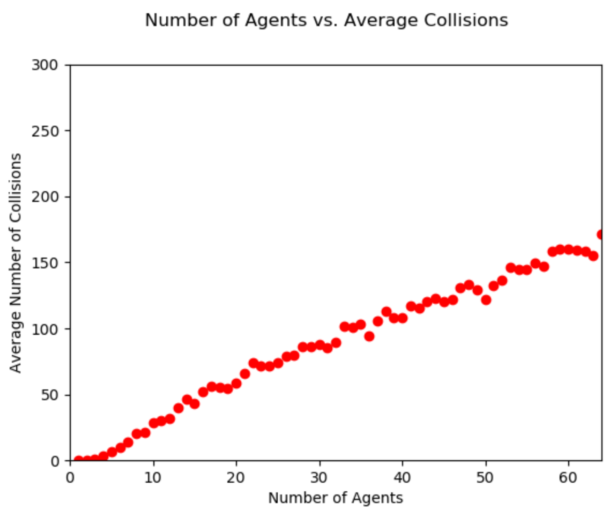

Fig. 2. Average Number of Total Collisions

Another statistic we were interested in was the relationship between the number and type of collision cases were handled during discovery since most of the collision cases involve recomputing an agents path which requires some computation that could be intensive. Figure 2 displays the average number of collisions for a given number of agents over our test cases. The relationship is linear which makes sense since as the number of agents in a maze the number of nodes an agent should be expected to evaluate should be divided by n for the number of agents in the maze. But agents will still try to evaluate as many nodes as possible so the number of collisions should be closely related to the number of agents. Figure 3 displays the average frequency of the five different collision cases. We wanted to see this since some cases of collision require more computation than others. For example swapping targets is the most frequent collision case and it is also has the highest computational cost since you have to recalculate

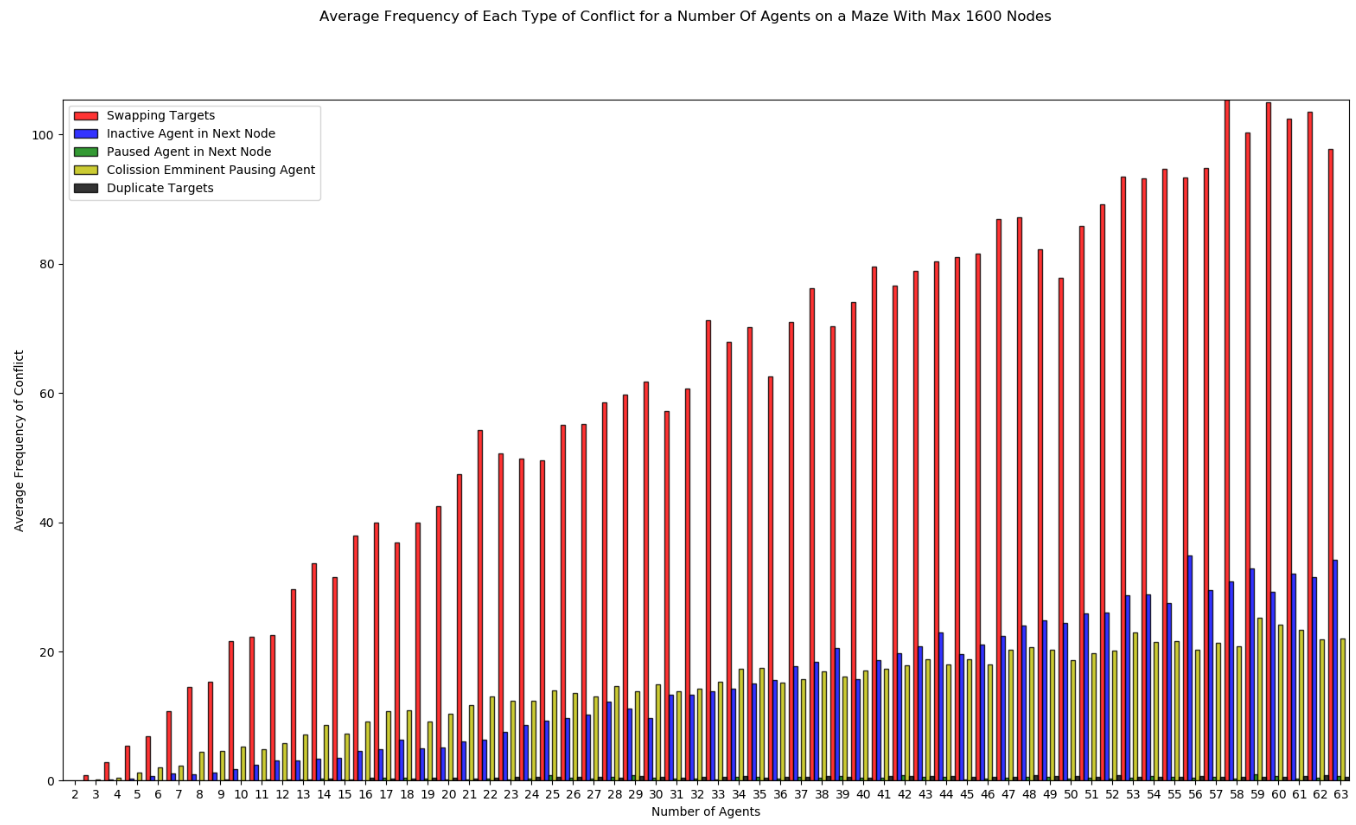

Fig. 3. Average Frequency That Specific Special Event Occurs

paths for the two agents that swapped targets. This result show us that down the line we might want to streamline swapping somehow to make it more efficient so the algorithms run faster. Another interesting result is that once the number of agents was greater than 35 agents start to become inactive more often. Since the size of the maze is relatively small you would expect this to happen since as was said previously the agents are expected to have to traverse a smaller number of nodes.

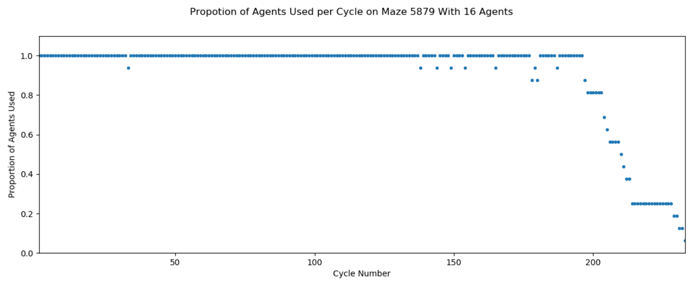

Fig. 5. Active Agents Per Cycle Out of 16 Agents

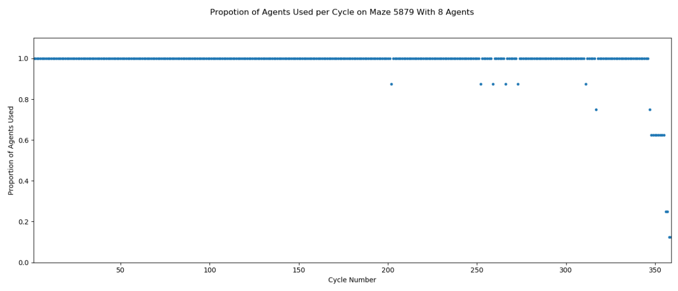

Fig. 4. Active Agents Per Cycle Out of 8 Agents

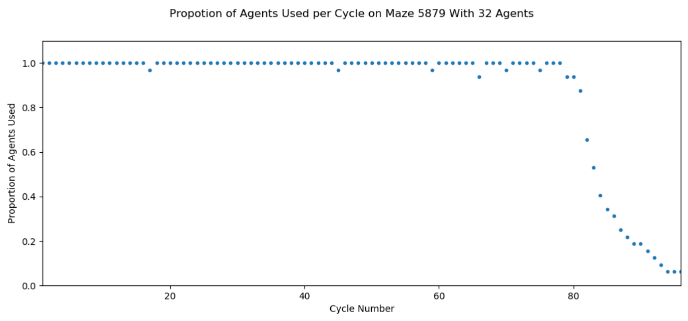

Fig. 6. Active Agents Per Cycle Out of 32 Agents

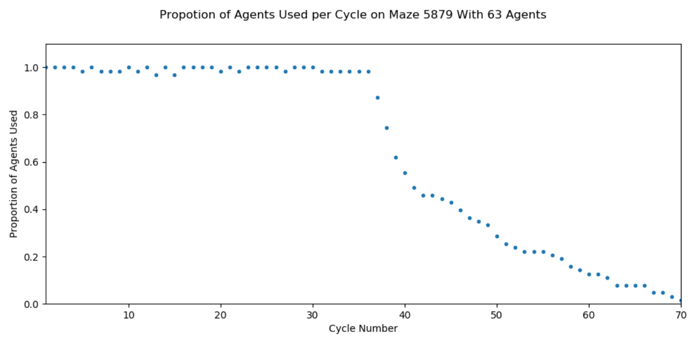

Fig. 7. Active Agents Per Cycle Out of 64 Agents

The final statistic we were interested in was the proportion of agents that are used in each individual cycle when running 8, 16, 32, and 63 agents on a random maze in the set of 32 test mazes. When running a low number of mazes you would expect there to be very few times where all the agents are not running in the maze at the same time. This is shown in the graph of 8 agents in the same maze where some pauses happen before the end when almost all the agents deactivate within the last 40-50 cycles. As the number of agents gets doubled the plots start to smooth out but there are are still very steep drop offs that occur at some point. This indicates that the number of agents that are currently in the maze can efficiently solve a maze that would take the number of cycles before the drop off which means you could find the optimal number of agents to solve a maze if you know the amount of cycles it will take to discover the maze.

VIII. Conclusion

This projects goal was to find the most efficient combination of agents, pathfinding algorithms, and swarm algorithms that provided the quickest solution to mapping out an undiscovered area. To complete the project, we generated random mazes of varying size, and used algorithms that fit our implementation. The agents used that A* pathfinding algorithm to navigate the maze, and the swarm was based off of Particle Swarm Optimization algorithms to iteratively solve any issues between agents and help improve discovery efficiency.

After running the tests, analyses showed that there is a correlation between number of agents, size of a maze, and how fast the maze can be fully discovered. Many agents in a small maze will solve the maze very quickly but that comes at the cost of complexity and needed computational power. The opposite of this statement is also true, if you have a large maze to map and you have very few agents the cost is low but the time to solve is very long. This leads us to believe that you can find the optimal number of agents for any maze of a given size if you have an idea of how many cycles it should take to complete a maze and how willing you are to make computations.

With these results in mind, it is important to ask what they can be applied to. At the base level, the results can show the ideal number of agents for mazes of varying size. This statistic can be applied to search robots. If, in the case of a natural disaster, emergency services introduces a number of search robots to scan an area for survivors, they can look at the size of the search area and know the optimum number of search bots to deploy in order to minimize collisions and discovery time. Such a statistic can be used for situations that care more about minimizing specific kinds of collisions (maybe due to physical limitations of the robot). With emergency search robots as just one example, these statistics can be applied to any physical search operation over an area of known size.

If there had been more time, either total or during the duration of the project, we would have done two major things. The first is implement and fully test additional pathfinding and swarm algorithms, so as to compare the efficiencies with different combinations of pathfinding and swarm algorithms. Doing this, we expect to see different combinations perform best with a certain range of agents, dependent on the size of the maze. These observations could help select which algorithms to used based on the situations number of agents and area of operation. Secondly, we would have performed many more tests, on much larger mazes and with many more agents. These tests take a long time to run, but would have shown how massive swarms behave with these algorithms, and in massive areas. Doing so would have provided us with the upper limits in capacity and capability for the different swarm algorithms and maze sizes, giving us an even more accurate relationship between maze size and number of desired agents.

References

[1] Z. Hu and J. Li, *Application and implementation of A* algorithm in picture matching path-finding* 2010 International Conference on Computer Application and System Modeling (ICCASM 2010) Taiyuan, 2010.

[2] S. Koenig and M. Likhachev, *D*Lite*, In Proceedings of the AAI Conference of Artificial Intelligence (AAAI) 2002.

[3] M. Couceiro and P. Vargas and R. Rocha and N‘ Ferreira, *Benchmark of swarm robotics distributed techniques in a search task*, 3rd ed. Elsevier BV, 2013.

[4] X. Cui and H. Shi, *A*-based Pathfinding in Modern Computer Games*, IJCSNS International Journal of Computer Science and Network Security School of Engineering and Science, Victoria University, Melbuorne, Australia, 2011.