Multi-Agent System Application to Controlled Warehouse Delivery

Fatma Akcay
Harvard College
fatmaakcay@college.harvard.edu

Jiayun Fang
Harvard College
jfang@college.harvard.edu

May 4th, 2016

Abstract

This paper presents the design and implementation of a multi-agent swarm system for warehouse package delivery. The multi-agent system takes inspiration from AntNet and a drone-hive model for dispatching robotic "bees". The system is simulated on the University of California Berkeley Pacman game platform, with "ghosts" acting as the agents that carry packages. The system optimizes path finding by accounting for other robot traffic along its path. The performance of this system is tested against a basic path finding algorithm that does not take traffic into consideration as well as a test case that finds the best priority ordering to reduce traffic.

I. Introduction

Swarm intelligence in multi-agent systems are increasingly becoming more applicable as robotics have advanced to support greater computational power and higher level sensing. In this paper, we investigate the application of a multi-agent swarm system to warehouse delivery. The exact algorithm for efficient delivery was inspired by a combination of the Di Caro and Dorigo AntNet algorithm for routing telecommunication networks and Dantu, Kate, et. al. drone-hive model for dispatching robotic "bees". Since these two papers have focused on variable, uncontrolled environments, there remains a gap in applying swarm algorithms to uncontrolled environments like a warehouse. A warehouse in particular presents an urgent problem because of rapid growth of the role of robots in facilitating online retail. In line with general swarm intelligence, this delivery algorithm operates through multiple, independent agents that each seek to deliver a single package from a source to a destination. Communication between agents occurs through a global network that can be accessed and edited in real time by all agents.

We use aspects of the aforementioned algorithms to construct a model for dispatching packages from multiple locations in a warehouse that has a grided floor plan and may contain obstacles along the path. This algorithm also utilizes a central source of information that contains the packets to be delivered and several "hives" to dispatch our delivery robots, each of which is tasked with a single package delivery. As in a real warehouse, agents in this model dynamically change paths to avoid occupying the same physical space in the coordinate system as other agents. When many agents are crowded together in close proximity, this is identified this as congestion (defined computationally in section III) and agents reroute themselves. Delivery agents that are searching for paths to their package destination will have access to global congestion information and calculate routes that will be the fastest.

In order to test the correctness and compare the performance of our algorithm, we used the University of California Berkeley Pacman simulation for testing of artificial intelligence [1]. Modifications on the search and multiagent aspects of the simulation enabled us to create a grided environment very similar to that in a warehouse. In a simulated environment, we sought to investigate the impact of 1) varying the number of package sources 2) changing the distribution of agents throughout sources 3) warehouse layouts with various levels of set obstacles 4) taking traffic considerations in to path calculation. A full iteration is considered complete when all packages have been delivered and all agents have returned to a source. Some metrics we assessed included total performance time, per package delivery time, maximum number of nodes expanded in path computation, and number of collisions sensed.

II. Related Work

There are two sets of literature that we found relevant: models from biologically-inspired systems and existing applications in cloud robotics. In this section, research from each section is presented and contrasted with the purposes of this research project.

I. AntNet

AntNet is based on research on how the *lridomyrmex humilis* and *Lasius niger* ants use indirect communication in the form of stigmergy through laying pheromones to convey relative optimality of paths during environment exploration. Previous work with AntNet has been applied to packet switched networks[3]. In such applications, agents build a path from source to destination node while collecting explicit information about time length on the path and implicit information about the load status of the network. This information is back-propagated by another agent moving in the opposite direction and is used to modify routing tables of visited nodes. The routing problem is a stochastic distributed multi-objective problem. Decisions about routing can only be made locally with information about the current and future network states. This type of problem is well suited for a multi-agent system composed of two sets of homogeneous mobile agents, forward and backward. Agents in both sets are the same in structure, but they collect different information and have different tasks. At a basic level, the AntNet algorithm does the following:

1. A forward agent with destination node d is launched from start node s. It keeps a stack of every visited node and the time elapsed from its launching time until it reaches node d. It inserts this information (the path taken from s to d and the time it took) into a dictionary that it carries.

2. At every node, it decides which hop to take using a routing table. The route is selected proportionally to the goodness of each neighbor node and with a small exploration probability. If the best next node has already been visited, it chooses randomly over a uniform distribution of its other neighbors.

3. If a cycle is detected, where the ant is forced to visit a node it has already visited, the cycle's nodes are popped from the agent's stack.

4. When the destination node is reached, the forward agent creates another backward ant to transfer all of its memory to.

5. The backward ant takes the same path as the forward ant and updates the routing table with information about the goodness of choosing the next node when the destination is node d. It also updates a list of the mean trip time and variance from node i to node j (nodes it has visited). This represents the memory of the network.

This general process allows for information of the state of the dynamic network to be updated as various forward agents explore new paths [3].

Aspects of the AntNet algorithm that we

used include 1) ants that transition from forward to backward agents 2) indirect communication to other ants by updating routing grid 3) dynamic adaptation of one individual's routes based on information from others. These aspects are well suited for the warehouse scenario because agents are assumed to be independently acting once dispatched from the source but are able to adapt their actions based on knowledge from other agents. Possible user needs include accidents or obstructions in certain areas of the warehouse work grid.

However, our algorithm differs from AntNet because information will not be communicated through agents but rather through a global routing map that is dynamically updated and accessible to every robot through wireless connection. Given that we hope to apply this to a warehouse scenario, each robot is solving a shortest path problem rather than path discovery. We assume a grid that is predetermined with no unplanned obstacles appearing after the delivery path has been calculated. We also do not do any exploration in this algorithm and each robot knows the route to their destination upon receiving the package.

II. AntNet Variations

There are variations of the AntNet algorithm, as it can be applied to numerous problems. One such variation is the "Threshold based AntNet", which modifies the algorithm to efficiently use information for road networks. The travel times between sources and destinations are used as threshold values, which is used to judge the 'goodness' of new paths as they are found. This algorithm assumes that a discovered good route at time t is good unless proven otherwise. The threshold for a path between s and d can be calculated in a variety of ways. One of these methods involves running the classic AntNet algorithm to calculate the best route between s and d with the corresponding time. The threshold can also be calculated by using information about the maximum speed for each link in the network to calculate the shortest path between two nodes.

A good route is defined as one that is within the bounds of the calculated threshold. When a good route is discovered from node s to d, the maximum probability for each pair from s to i ($i = 1, ..., d$) is updated in the routing table. The algorithm then stops searching for another good route from s to i, and the integrity of the good route is checked after a period of time Δt. This check is done by a third group of agents, check ants, which periodically check the travel time of a good route against the corresponding threshold [2].

III. BeeHive Algorithm

Similar to the AntNet algorithm, bee swarms are used as a model to solve distributed multi-objective problems. One model that borrows from the examples of bees is Karma, a micro-aerial vehicle (MAV) swarm software and management system that mimics a hive and bee swarm structure. This hive-drone model uses centralized computer (hive) to assign tasks to agents. Agents are located at the central hive, and are dispatched as the hive determines their need. Agents are simple and can only gather data on their flight and update the central hive upon their return. Hence, no real-time updates occur. The global state is mapped in Cartesian coordinates and agents are given directions to a specific coordinate. Agents do not have the ability to determine their exact location, only estimates [4].

We use aspects of the Karma system to coordinate our robotic swarm. Our global state is represented in a grid-like structure, and our package-queue is like the central hive in this model. Each robot is assigned the next package in the queue, and the shortest path from the source of the package to its destination will already have been calculated for the robot.

IV. Cooperative, Autonomous Robots

While the theoretical modeling of biological multi-agent systems has been developing, so too has the application based research on multi-vehicle systems (MVS). Astronomy, defense, and health care have all benefited from advances in MVS. Amazon's high profile acquisition of Kiva Systems, the robotics startup that built an end-to-end system for orchestrating autonomous robots, quickly led to the integration of MVS into a workflow very close to the everyday consumer. Kiva's main innovation is in constructing inexpensive robots that could lift inventory pods which contained some set of stored packages and transport them to stations where human workers can pick the items to be shipped [5]. We were inspired by the work of Kiva and sought to incorporate biologically inspired elements to the delivery algorithm.

Like Kiva, our robots operate in a controlled environment where steps are well-defined and routes can be pre-calculated. In addition, our robots are cooperative as opposed to self-interested in that they coordinate together to accomplish the singular goal of delivering all packages. While they are independent in terms of conducting operations within their individual robot systems, each robot seeks to reduce problems of congestion or alert others to obstacles that would enable an overall more efficient system.

While simulation on real hardware is outside the scope of this paper, it is our hope that our simulation adequately models behavior of MVS such as Kiva. In order to accomplish this, we took into account strict considerations of physical space occupation in that at each step of time no two agents were allowed to occupy the same place in the grid. In addition, we sought to have multiple sources to simulate places where multiple robots could be dispatched. Our simulation also models the multi-agent system well in that all agents move simultaneously and can communicate with each other about traffic and obstacles. The paralleled nature of this delivery paradigm as opposed to the traditional batching system is a key factor in MVS increasing productivity. In addition, our system can continue to operate should one of the agents fail, in line with Kiva's asset of not having one point of failure. This is especially critical a large-scale warehouse environment because operations ceasing as as result of a single location or single agent failure could be disastrous for profits and operations.

III. Implementation

Before we begin the discussion of the algorithm, we first define the main structures present in the project. An `Agent` is the main actor in delivering a `Package` which contains a coordinate for its destination and a level of priority. The `Agent` is an object representation of a robot that contains a controller and makes decisions about its actions based on certain game state conditions such as its current location in the coordinate, adjacent coordinate occupations, past nodes traversed (for routing table updates), etc. In addition we use a `Grid` as a shared coordinate system that stores some data about the current game state such as where each `Agent` currently is and where the walls are. The `GameState` is the complete compilation of all such data and the status of the delivery queue. It is updated with each passing of time and is initialized from the layout given when running Pacman. The `Queues` contain a list of queues that are assigned to `Sources`, with one queue relating to a specific source. This framework was constructed with the warehouse distribution structure in mind in which specific categories of items may be compartmentalized in different parts of the warehouse. Each queue contains some number of packages that are queued by priority, akin to packages that may be expedited or overnight shipped. In addition, the `GameState` keeps track of a `RoutingTable` that keeps track of "costs" associated with each `Action` or step wise movement within the grid. If one `Agent` detects some kind of collision with another `Agent`, the value for that action in the table is incremented to indicate higher cost of that path.

Our complete algorithm can be divided into two main sections: assignment and transport. In the assignment phase we consider how agents receive packages, how package sources are set up, as well as how this plays out in the code. In the transport phase, we discuss how routes care calculated, how agents sense congestion around them, and how we incorporated the Pacman simulation with these structures.

I. Package assignment

Package assignment occurs at specific, prescribed locations called Sources in the simulation. Sources are the only places in the grid in which Agents are allowed to overlap in their physical movement. In our simulation, we make several assumptions about the setup of the inventory: 1) that the packages themselves do not affect the physical space of the queue 2) that Agents are able to pick up their packages immediately after entering the source location 3) multiple agents entering one queue will continue to obtain packages in order of priority. We acknowledge that some of these are unrealistic in an experimental set up; we were constrained by the time and Pacman simulation set up of the project but we hope these could be addressed in future work.

Agents receive packages from the source they are closest to by popping the next highest priority package from the heap. With each package, the agent receives a destination and then proceeds to run a search function with costs. The path calculated contains a list of one-step directions (from North, East, South, West). The path of least cost is stored in the agent and it takes its first action by popping off the first action from the path.

In order to compare the performance of different kinds of search and of cooperative agents that take into account each others' movements, we used two main search functions: 1) breadth first search with each step cost being a static value of 1 and 2) uniform cost search with each step cost updated as agents are near collision in their paths. While the first time a search is run is at package assignment time, at every subsequent time step the agent recalculates the path using whichever respective search function in order to change paths based on real time data of nearby agents.

After Agents finish transporting their Package to its corresponding destination, Agents return to the source that is closest to where they had dropped off their package. They then compute using the same search algorithm–as if the source was a package destination–a dynamic path of getting there. Once they arrive at the source, they are then reassigned a package and continue to cycle through assignment and transport until all packages at all sources have been delivered. To end the simulation, all agents return to whatever source is closest to the last package they delivered.

In order to build the package assignment structure in the Pacman simulation, we used the `GhostAgent` specified in Pacman simulation and integrated it with the `SearchAgent`. We store the destination information in the Agent and the sources in the GameState data. This allowed us to utilize the graphics built into the simulation to test the behavior of our agents and see whether they were effectively delivering the packages.

II. Package transport

Package transport is implemented in our system through the `SearchAgent` and the core game play classes. As the Agents and the packages with them moved simultaneously, the game state was updated to include exact locations of agents and walls. This information is used by the `SearchAgent` to determine its next legal move. Each Agent recalculates its optimal path to its goal at each step - this detail was crucial to implement collision detection and avoidance.

We define a collision to be the state in which two or more agents are one step away from occupying the same physical space. This is also a measure of traffic congestion, as the more collisions occur, the more congested a state is. It is important to note that while Agents recalculate their shortest path from their current location to their goal, this path will not change unless an unforeseen obstacle appears in their path. Collision is just this - it forces agents to take paths that deviate from the shortest path.

In order to account for collisions in our system, when a collision was detected, the location of the agents involved were marked as walls - effectively stopping either agent from moving to a position occupied by the other. As each Agent recalculates its shortest path, the new positions that have been marked as walls in the layout are illegal positions for them to move into. Thus the shortest path that is calculated will not include those positions.

In order to stop Agents from recreating the same congested state, the cost of taking that path had to be increased in order to discourage current Agents as well as future Agents from taking that path. For this, we used the `RoutingTable`, implemented as a dictionary of `startState` and `goal` tuples, initialized with values of 1.0, which give the cost of traveling to the `goal` using the `startState`, which includes information about the location of Agents and walls in the current state. When collisions were detected, the cost of traveling to the `goal` of the agents involved from their `startState` was increased by 1.25. This discourages Agents from taking paths that are heavily congested, allowing for the optimal use of the available paths. The BFS test cases did not make use of the `RoutingTable`, instead using a constant cost function where every action is equally weighed as 1.0.

IV. Data and Results

In order to test the performance of our algorithm, we used several metrics 1) total delivery time 2) delivery time per package and 3) number of total collisions experienced (collision defined in subsection I). We examined these metrics across grids of varying sizes, grids with different density of walls, agents distributed over sources differently, number of sources for constant number of packages, and using different search functions. In addition to our cooperative, traffic factoring model, we compared our performance to a basic breadth first search without taking into account traffic as well as a know-all case in which packages were ordered in a way so as to minimize collisions. During all tests, we held the number of agents constant at 4 because 1) we sought to control for the number of multi-agents in order to conduct specific tests in between different setups and 2) there were limitations in the Pacman simulation in accommodating more than 4 agents.

I. Layout

We created three square layouts that we used to measure the effect of the size of the space on our results. All small layouts are 7*x*7 (Figure 1), medium are 20*x*20 (Figure 2), and large are 40*x*40 (Figure 3).

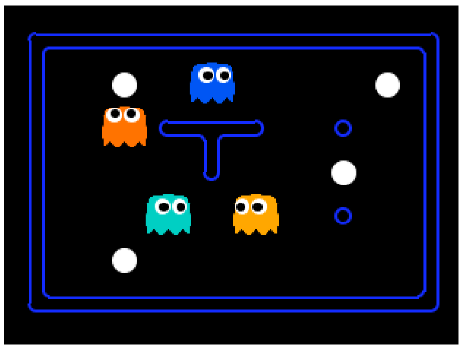

Figure 1: *Small layout with 4 sources, dense walls.*

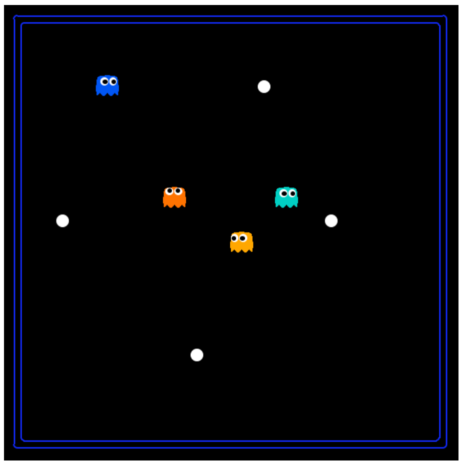

Figure 2: *Medium layout with 4 sources, no walls.*

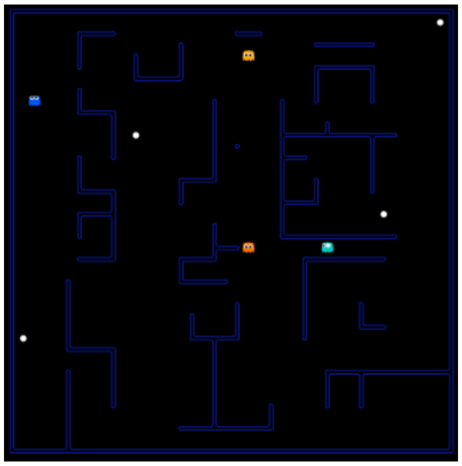

Figure 3: *Large layout with 4 sources, dense walls.*

II. Test Cases

Test cases were manually created to compare across all variables identified in section IV. This included creating package delivery sites for each type of layout (small, medium, large), as well as varying the number of sources and the distribution of Agents at the sources. Additionally, the layouts of the walls were created to the effect of the density of walls. This set of test cases were run on BFS, Ant/UCS, and the "Know All" implementations.

II.1 BFS

This test case runs with breadth first search as the algorithm to find shortest path, and every weight was considered to be 1. This was a reasonable implementation of a test case that realistically should do poorly compared to the Ant-UCS implementation because it does not keep the collision heuristic. Because the cost of each node was kept constant, the routing table was not used in the BFS implementation. Collisions were still avoided in the same way, they just did not impact the cost of a given path or node. All other parts of the Ant/UCS implementation were kept the same.

II.2 ANT-UCS

Our implementation of the multi-agent swarm system (referred to as "ANT" in test cases and graphs) included using UCS and a routing table that updated based on collision. This is described in full detail in section III.

II.3 KNOW-ALL

The "Know-all" test case was planned to be the most optimal in terms of time and efficiency. The idea is that the layouts and package queues are constructed in a way such that collision is minimized and Agents don't expand many nodes. While this is certainly possible to construct, many of our "Know-all" test cases were not the most optimal they could be because they were manually constructed by adding packages to the queues in a way that reduced collision. However, while not the most optimal "Know-all" cases, these test cases performed better than the other two.

II.4 Graphs

We compared the correlation between all of the variables of significance in our test cases. The graphs are presented in the following pages and a discussion of the results follows.

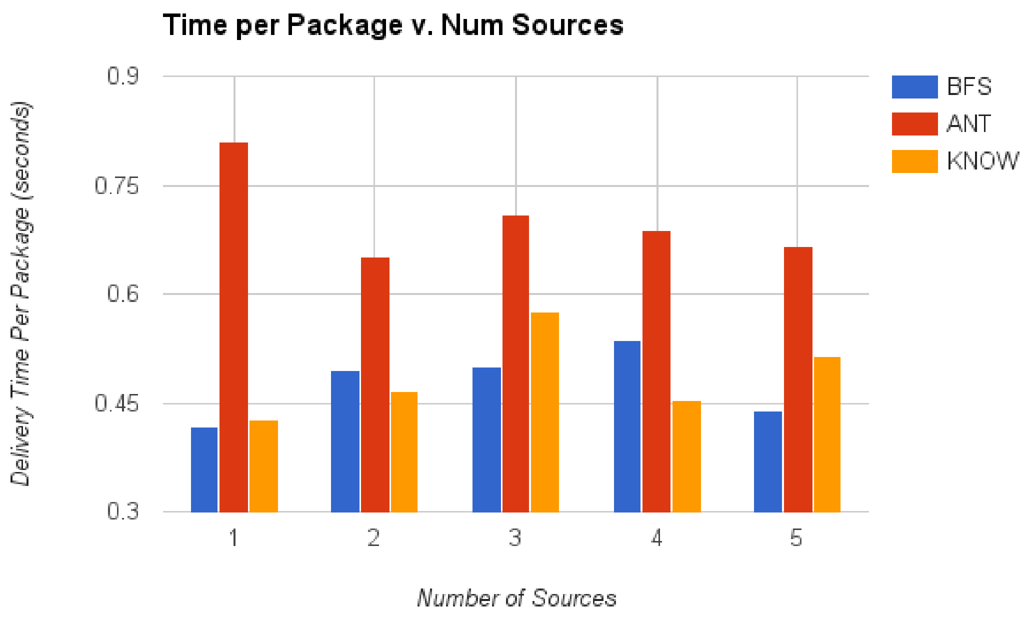

Figure 4: *Delivery time per package compared across all test cases and varying number of sources.*

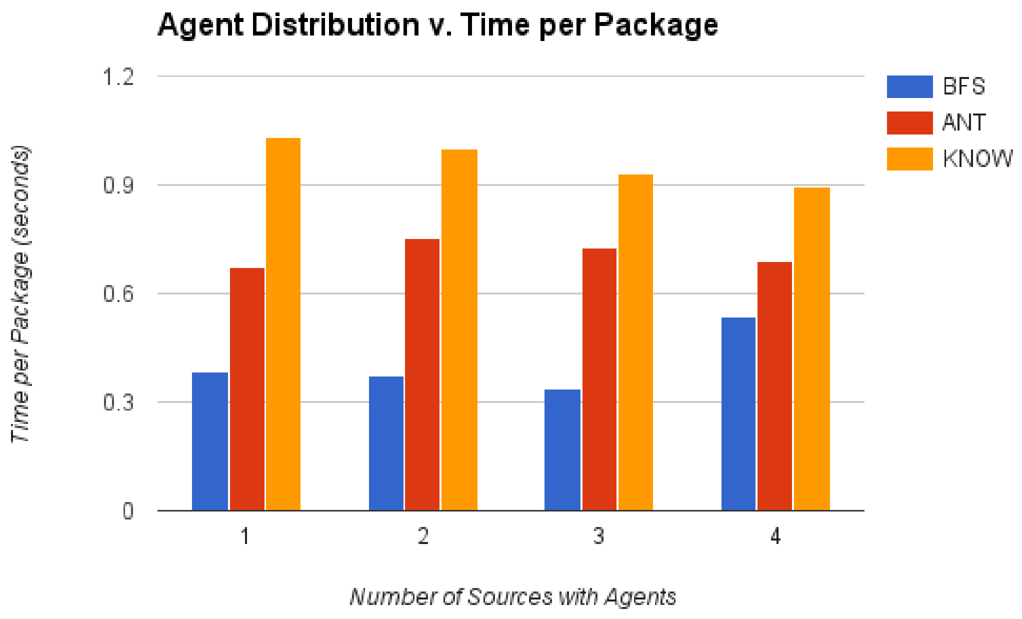

Figure 5: *Delivery time per package compared across all test cases and varying distribution of Agents.*

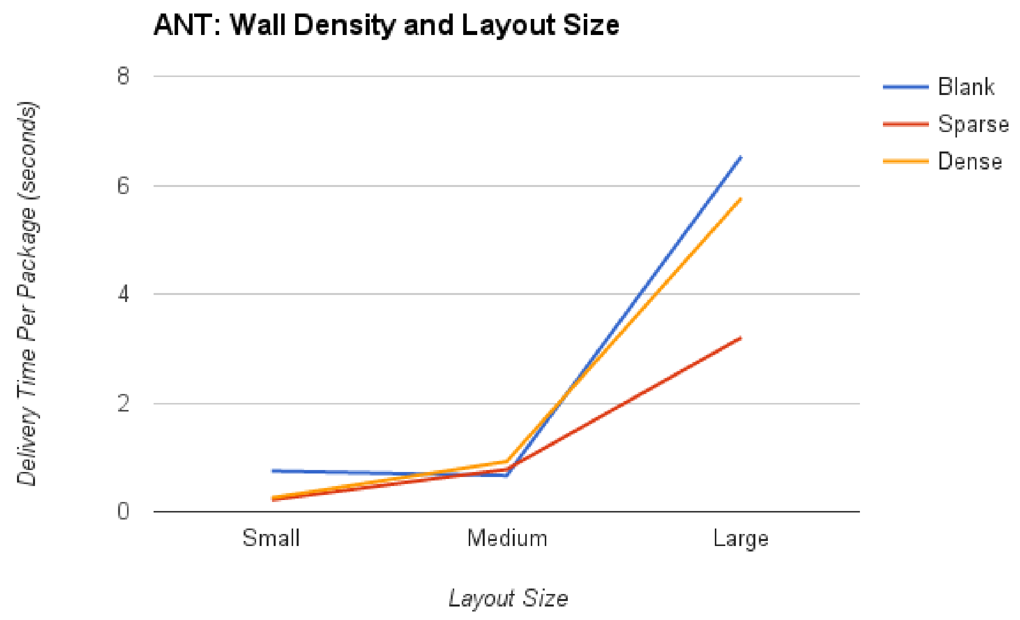

Figure 6: *Delivery time per package for Ant/UCS test case for all layout configurations*

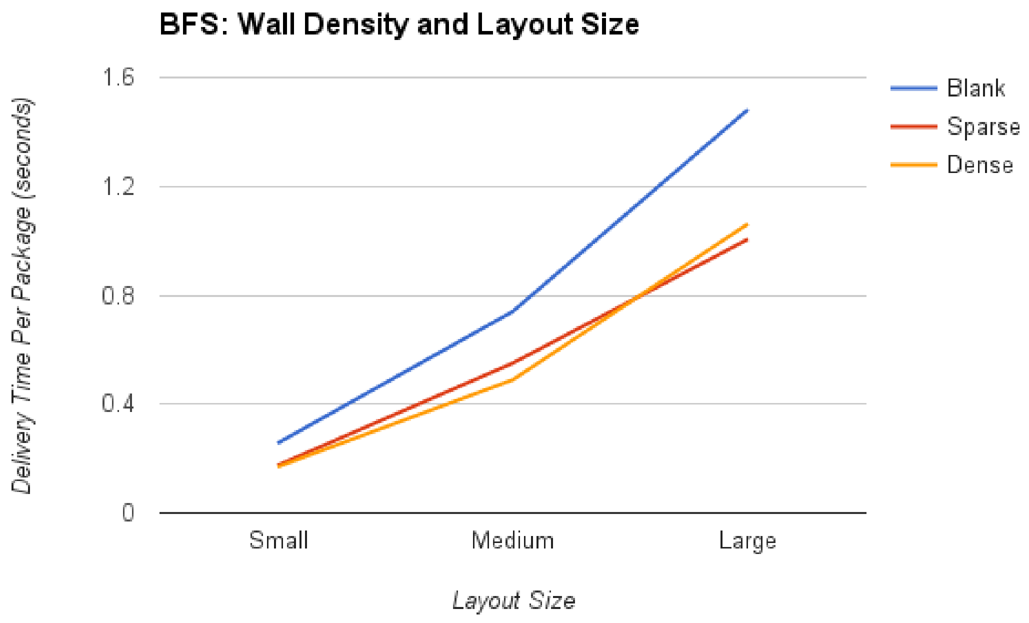

Figure 7: *Delivery time per package for BFS test case for all layout configurations*

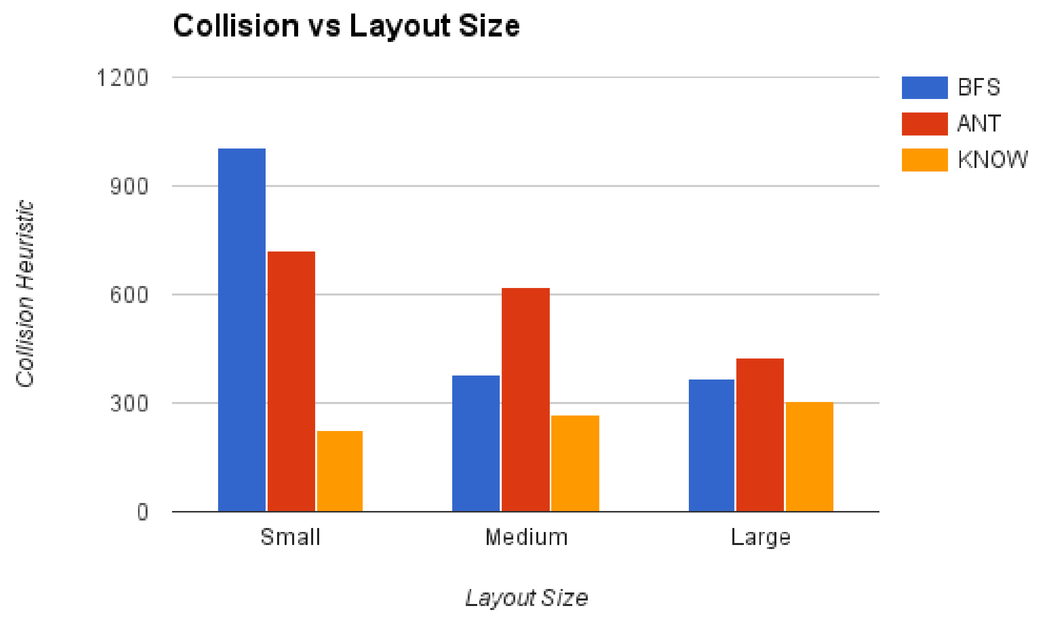

Figure 8: *Collision heuristic compared across all test cases and varying layout size.*

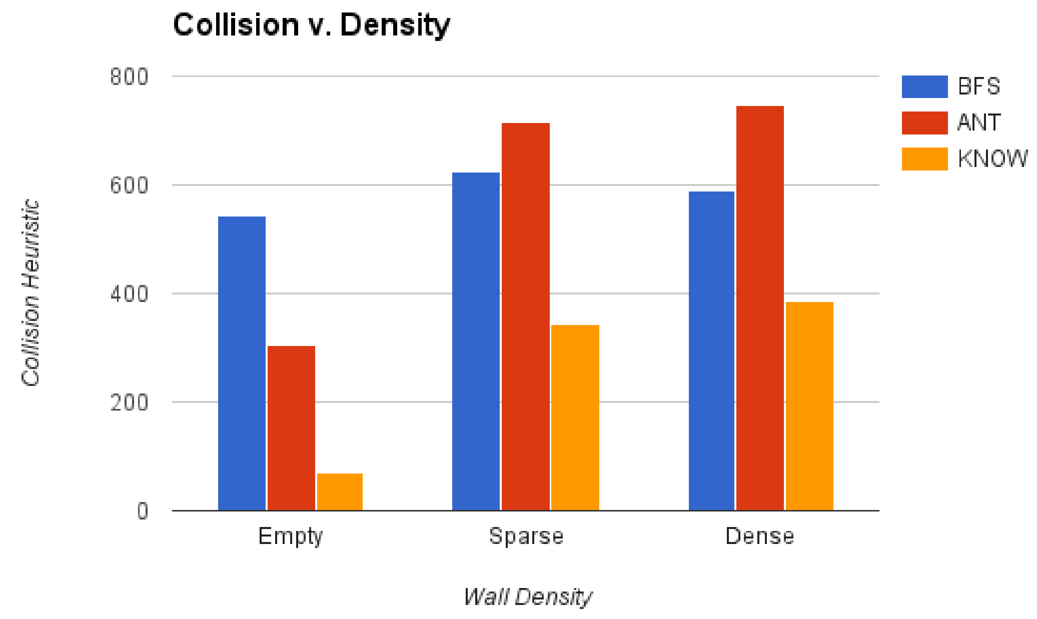

Figure 9: *Collision heuristic compared across all test cases and varying wall density.*

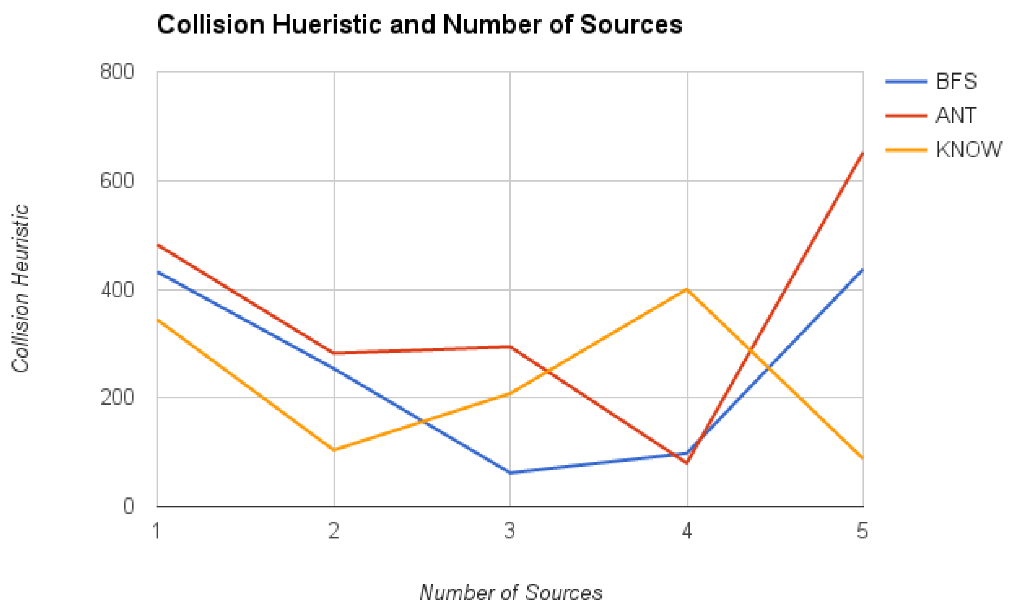

Figure 10: *Collision heuristic compared across all test cases and varying number of sources.*

V. Discussion

While our tests were not overly exhaustive, there are still some interesting observations to be made.

I. Observations

I.1 Factors Impacting Delivery Time

As expected, the delivery time per package increases for bigger layouts. In Figure 6, the overall increase for the ANT test cases shows that the time increase is close to exponential. This is intuitive because larger layouts have longer distances that the agents have to travel through. There does not appear to be any trends with the density of walls of the layout. The "sparse" wall layout does better than both "blank" and "dense". This may be attributed to a couple of factors:

- Since there are no walls in the empty layout, Agents will take the path that is simply the change in x and y to get to their goal (this will always be the shortest path in a layout with no walls). This would create more collisions because agents are more likely to cross paths when there are no walls to guide their paths.
- The location of walls in our sparse test cases may have caused agents to collide less often as they may have allowed packages to be delivered more efficiently.
- The dense test case may have performed worse than sparse for the same reason that sparse may have done well, which is that the location of its walls may have caused more collisions to occur.

Similarly, BFS has the same increasing trend for delivery time per package over an increase in layout size as ANT (Figure 7). BFS also displays a decreasing trend for delivery time per package as the wall layouts become more dense. This can be explained by expanding on our previous reasons, which is that the paths that BFS finds performed better in the test layouts we provided.

Another interesting result is that when there are four sources and the distribution of agents varies, the overall trend is that the delivery time per package decreases, as observed in Figure 5. This is most clear in the KNOW case, where the distribution of Agents amongst all four sources allows them to deliver packages more efficiently. This makes sense because distributing Agents at different sources allows them to be farther away from each other, avoiding collision, and allowing packages to be delivered sooner. This is also the case with increasing the number of sources. As seen in Figure 4, the overall delivery time decreases significantly for ANT. BFS and KNOW are slightly more noisy, but they do not completely disregard the trend.

I.2 Collision Heuristic

As expected, the number of collisions decreases significantly with an increase in layout size (Figure 8). The number of collisions for ANT is also significantly less in the small layout compared to BFS. We hypothesize that this is due to the use of the routing table to improve paths based on collisions. Because collisions occur so often in the small layout, the routing table optimizes very quickly to choose better routes.

As previously stated, it is intuitive that the number of collisions increases as the density of walls does, which is the case in Figure 9. In Figure 10, we also see that the collision decreases significantly as the number of sources is increased, allowing Agents to be spread out. However, not all of the data supports this, as both ANT and BFS increase in the number of collisions for layouts with five sources. We hypothesize that the location of the fifth source may have caused an increase in the number of collisions between

I.3 Search Algorithm

One of the most important aspects of our system was the search algorithm used. For our ANT test case, we used UCS, which expands all nodes that lead to the goal in order to find the one with the least weight. However, BFS returns the first path found because it will have the smallest cost, returning a path by expanding less nodes. Thus, BFS performs much better than ANT in terms of delivery time because it spends less time exploring paths. However, as stated earlier, it handles collisions poorly in small spaces, as demonstrated in Figure 9.

VI. Future Work

The nature of this application-based project begets two main areas for future work: algorithm and simulation. We were largely constrained by the Pacman simulation and time limitations to this project. While we will be unable to continue working on this past graduation, we hope our research highlights how different extensions can be made on the original Pacman AI simulator.

I. Algorithm

In terms of the algorithm we would like to develop a more complex system of cooperation. Currently, our algorithm only seeks to move agents out of the way of other agents as they are making deliveries but considerations on maybe exchanging packages or not picking up the strictly next highest priority item at a source could all enable better cooperation between agents. At times when agents are close to colliding with each other, currently their routes are recalculated relative to each other without concern for the priority of package they are carrying. In a more optimal cooperative state, agents with less priority will go around agents that greater priority or urgency in delivering their package. In addition, while our current agents are homogeneous, there could be a heterogeneous set of agents in which some only seek to clear traffic while others strictly deliver. This would differentiate tasks with regards to building collective knowledge of the environment.

Furthermore, another area for development is on the pre-processing. In this algorithm we assume that packages are given to the us in the order they want to be delivered. However, it might be more optimal for distribution centers to sort packages by location irrespective of their shipping status. In addition, right now all agents are able to go anywhere throughout the grid to deliver a package but with preprocessing, they may be assigned to a specific quadrant and then be able to focus on making short, repeated trips rather than covering the entire floor. Packages might also be preprocessed with information in addition to their priority such as their size, weight, and fragility. These factors could all contribute to an agent being able to carry multiple packages or needing some kind of special agent or human to handle a particular package.

An additional area for extension is on having more robust sources set up. Currently the model assumes that the number of packages still at the source does not affect the physical space occupation of the source which may need to be customized based on the specific warehouse need. Extensions could include having sources decrease in size as packages are delivered and introducing wait time when agents try to pick up a package. Instead of having some number of preset sources that remain in the same location over the course of delivery. It is feasible to imagine a warehouse needing to create new sources of packages in order to introduce more delivery traffic into the environment for throughput. Also, the package allocation system at sources may fail and the model should be able to handle spinning up new sources and moving packages to the new location.

II. Simulation

The Pacman simulation afforded our algorithm many advantages in that many parts of the graphics, agents, and game play interaction had already been set up in various parts of the simulation. However, with more time, there are several areas in which we would optimize the simulation. First, we would focus on a better graphical interface to testing. We realized during the testing the phase that the high powered graphics were affecting our total delivery times. We then had to redo the tests and close the graphics simulator. In addition, we had trouble coordinating the visual display of package destinations to each agent which would have made it clearer to see the travel patterns.

Next, another part of the simulation that can be worked on is information encapsulation. In order to finish the algorithm in time, we also stored many aspects about the game state in the agent and vice versa. To promote modularity, we would like to practice better information encapsulation in which agents only know what they need to know in order to accomplish the delivery. The agents should contain decision functions and take in game state as inputs rather than storing the complete states themselves. This would produce agents that are lighter wait and more appropriate for mobile hardware.

In addition, we should investigate resource and energy allocation with regards to the simulation. In real robots, there are limited amounts of computational power and a lot of sensory noise that could disrupt agent perceptions. Our model could introduce a limited power supply to agents or slow down the agent with each additional search that must be calculated. Since we are recomputing the route and doing a full search at every step, this seems like an unrealistic demand on the hardware of swarm robots.

VII. Contributions

JN and Fatma worked together as partners throughout this project. JN was responsible for setting up the search agents and package assignment process. She owned the code for setting up the multiple queues and having agents compute routes to a specific destination as opposed to normal GhostAgent behavior which is random movement or semi-directional. With regards to testing, JN created testing layouts and did the testing on the breadth first search algorithm and the know-all case.

Fatma worked on the package transport aspect of the algorithm and she owned the collision calculations and agent rerouting based on nearest packages. She also owned changing the core game play classes to receive our custom layouts. With regards to testing, Fatma set up the metrics to be measured and tested the UCS algorithm with cooperative traffic rerouting.

VIII. Code and Running Tests

All of the code used in this project can be found at: `https://www.dropbox.com/s/av11v5xrbxxiqyb/cs289_akcay_fang.zip?dl=0`. Currently, the submitted code supports running test layouts that are size medium with four sources. This includes test layout: 4, 6, 7, 8, 12, 13, 14. These tests can be called by specifying the layout flag to be "test" + str(layout number). An example would be: ‘python pacman.py -l test8 -g DirectedGhost‘

References

[1] J. DeNero, D. Klein, P. Abbeel (2014). `http://ai.berkeley.edu/project_overview.html`

[2] Ghazy, Ayman M., Fatma El-Licy, and Hesham A. Hefny. "Threshold Based AntNet Algorithm for Dynamic Traffic Routing of Road Networks." Egyptian Informatics Journal 13.2 (2012): 111-21. Web.

[3] Caro, G. Di, and M. Dorigo. "Mobile Agents for Adaptive Routing." Proceedings of the Thirty-First Hawaii International Conference on System Sciences. Web.

[4] Dantu, Karthik, Bryan Kate, Jason Waterman, Peter Bailis, and Matt Welsh. "Programming Micro-aerial Vehicle Swarms with Karma." Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems - SenSys ’11 (2011). Web.

[5] Wurman, Peter R. , Raffaello D’Andrea, and Mick Mountz. "Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses." Association for the Advancement of Artificial Intelligence Magazine 8. (2008): 9-19. Web.

A. Test Cases

Table 1: *My caption*

Test	Num Sources	Sources with Ghosts	Layout Size	Wall Density
1	1	1	Medium	Blank
2	2	2	Medium	Blank
3	3	3	Medium	Blank
4	4	4	Medium	Blank
5	5	4	Medium	Blank
6	4	1	Medium	Blank
7	4	2	Medium	Blank
8	4	3	Medium	Blank
9	4	4	Small	Blank
10	4	4	Small	Medium
11	4	4	Small	Dense
12	4	4	Medium	Blank
13	4	4	Medium	Medium
14	4	4	Medium	Dense
15	4	4	Large	Blank
16	4	4	Large	Medium
17	4	4	Large	Dense