Effect of Sensory Faults within Robot Swarms

Stijn Coppens
Supervisor(s): Ranga Rao Venkatesha Prasad, Suryansh Sharma, Ashutosh Simha
EEMCS, Delft University of Technology, The Netherlands
22 June 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Robotic swarms provide a great many uses within a world increasingly relying on autonomous systems. Alas these swarms are also very vulnerable to faults, even the smallest fault can cripple the performance of a whole swarm. Such a fault could be one of several types; those that have to do with sensors, communication, ... This paper will try to discover the effect that these faults have on the performance of a robotic swarm attempting to complete a specified task, e.g. a multiple destination routing problem or a variant of a foraging task. With the experiments concluded and the data gathered it proved possible to observe correlations between fault and performance, but this dependency heavily changes based on the algorithm used and fault introduced.

***Keywords*—** Robotic Swarm, Fault Tolerance, Sensory Faults

1 Introduction

The increasing prevalence of electric and smart vehicles on the road as well as other sentient systems that rely on autonomy and self-driving capabilities leads to an increase in amount of possible incidents with these vehicles. With even more of these on the way as well as a growing number of autonomous heavy vehicles within the transport industry there is a clear need for robust algorithms that avoid accidents as much as possible. A well-coordinated Intelligent Transportation Systems (ITS) is still an open area for research [1] [2].

This problem can be solved through robust robot swarms with a high amount of fault tolerance. As this is a very broad and complex set of circumstances and problems it is beneficial to split up the issue and tackle them individually. A very crude split can be found by distinguishing between sensory and mechanical problems. This paper will focus on the former. Sensory issues can include anything from e.g. a distance sensor failing all the way to malicious interference by any number of ill-intending actors.

While the topic of robotic swarms has been researched thoroughly, this has mostly focused on the ways in which these swarms can help humans. Should fault tolerance be discussed this is almost always with regards to mechanical failures rather than sensory ones. This paper shall try to formalise and characterise these failures so that these sensory issues can become easier to recognise and prevent.

The paper will be structured with the research methodology following a more formal problem description. Previous work by fellow colleagues will be discussed together with the main contributions from this paper. Following this the experiments and their results will be discussed as well as the ethical implications and restrictions of this research. Finally some general conclusions will be drawn from the gathered data to discuss the best course of action for solving these real life swarming problems.

2 Swarming Formalised

The goal of a robot swarm is to accomplish a certain task through the use of collaboration with other robots within the swarm. In the purest form this should happen without any overhead or centralisation, the robots should communicate with each other and thus gain information about the progress of the given task solely through this process and the sensors these robots possess. This creates one of the bigger vulnerabilities within these systems as any fault introduced into this vital process will cause the whole swarm to either misbehave or completely stop functioning. As outlined by A. Winfield and J. Nembrini in *Safety in numbers: Fault tolerance in robot swarms* [3] the failure of an avoidance sensor (in this case, the front facing distance sensor of the robot) causes the robot to collide with obstacles in its way and thus losing it's ability to do an obstacle avoidance task (see Table 3: Summary of Failure Modes and Effects). From this same table it can be seen that a breakdown in communication can also cause this ability to be lost depending on the swarming algorithm used. A more clear example of communication breakdown causing the swarm to falter can be seen in a foraging task. Intuitively communication between agents within the swarm is paramount to the effectiveness of the swarm as a whole.

This paper will try to explore the effect of these sensory errors on the performance of a robot swarm that performs an obstacle avoidance/routing problem [4], a collaborative foraging task or attempting to maintain a delta formation. This shall be done through 2 sub questions:

- What is the effect of sensory faults on the performance of a robot swarm acting out a routing or foraging problem?
- How do we measure the effect of these faults accurately, taking into account multiple types of failures?

3 Related Works

As with everything to do with swarms this particular subject has seen an increase of research in the last decade. These contributions, while very useful and interesting in their own right, tend to focus on the detection of failures (both mechanical and sensory in nature) and their prevention rather than quantifying the effect on the performance of the swarm as a whole in its execution of a task when a fault is introduced. A. Winfield and J. Nembrini [3] propose several interesting ideas with regard to the different types of failures that can occur within a robot swarm trying to perform several tasks and the effect this might have on the swarm as a whole but alas there is no formal quantification of performance loss caused by these faults.

An attempt at a more formal definition of the effect of a purely mechanical failure on a robotic swarm is made by J. D. Bjerknes and A. F. T. Winfield [5] but once again this fails to properly discuss the effect of sensory failures. With no apparent attempts at any formal definitions of the effect of sensory faults, this paper will strive to fill that void. To that end the existence of already created swarming simulations is very useful. A MatLab based simulation, SwarmLab [6] attempts to accurately simulate two boid-based swarming algorithms [7]. The two algorithms in question are those proposed by Vasarhelyi [8] and Olfati-Saber [9].

4 Methodology

To carry out a fair and unbiased test it is paramount to devise a proper method, because of the many sided nature of the problem under in-

vestigation, it is both beneficial and necessary to create separate simulations that handle all errors independently.

- Front facing sensory error:
  - Path Finding: The goal of this experiment is for a swarm to get through a "maze" of zones where they are not allowed to come. One of the agents within the swarm will have a failing front distance sensor and will as such misjudge its distance to the forbidden zones. The effect of this on the performance will be measured. The experiment shall be run through SwarmLab [6], a point-mass based simulation written in MatLab where for one agent of the swarm a defect will be introduced [10].
  - Delta Formation: The experiment consists of a swarm trying to maintain a delta shaped formation while one of the swarm members has a failing distance sensor. A simple solution as well as one in a damped environment shall be discussed. The whole simulation will be implemented from scratch in Python.
- Breakdown of communication between agents: In this experiment a swarm must complete a foraging task while there is a malicious agent spreading false information. The effect of the malicous actor on the performance of the swarm is once again the goal. This will be simulated through BW4T [11] with code and ideas by Z. Chen, S. Coppens, M. Stroia and K. Zhu outlined in appendix A. A solution will also be discussed in the form of the ABI [12] model and the changes of the swarm's performance will be discussed.
- Delay in sensor results: Utilising the same delta swarm from before the incoming distance readings are delayed for a few cycles for one agent. The goal is to see the effect on the swarm and whether it can stay in formation.

5 Experimental Setup

As mentioned earlier in section 4 the experiment will be ran in several distinct phases.

All three simulations allow for the changing of plenty of parameters and for the sake of clarity and repeatability these shall be included in appendix B together with links to the code repositories.

5.1 SwarmLab Based Simulation

The first experiment with regards to the effect of a sensory error will be ran in MatLab. An already implemented boid-esque swarm attempting a path finding task was used as a base for this particular question [6]. The swarming algorithms used are those proposed by Vasarhelyi [8] and Olfati-Saber [9]. While both algorithms are heavily based on the flocking rules outlined by Reynolds and Craig they do both contain some key differences [7].

In Vasarhelyi's algorithm there is a force repelling the point-masses from each other should they be within a certain range of each other. This is done to avoid collisions while still holding as short a distance to its neighbours as possible without actually colliding. The velocity of agent i is described by

$$v_i^d = \tilde{v}_i^{trg} + \sum_{j \neq i} (v_{ij}^{rep}) \sigma(r_c - |d_{ij}|)$$

In short, the velocity of agent i is calculated based on the magnitude of the target tracking velocity (the end goal) and the sum of all neighbours within a specified range (the d parameter). Within this first term there are the repulsion from the arena walls, blocking objects within the arena and any agents within the neighbourhood.

The method proposed by Olfati-Saber behaves very similarly besides the fact that the different velocities are calculated per agent and not all at once. This causes some side effects with regards to speed and flocking behaviour that is discussed in detail in *Flocking for multi-agent dynamic systems: algorithms and theory* [9].

As both of these algorithms use the positions of the 'forbidden zones' obtained directly through the map without any properly simulated distance sensors introducing a fault can get slightly more tricky. When the next velocity vector of the point-mass is calculated in the $compute\_vel\_(alg\_name)\_no\_front$ method the read distance can be manipulated while still being left as the actual distance for the purpose of detecting whether the point-mass is currently within a restricted zone and thus scoring. Because of so called "flock centering" following the swarms boid like nature in both of the used algorithms [7] the changing of one point-mass' perceived distance will lead to a performance change throughout the whole swarm and as such this error will only be introduced for one specific agent. This not only makes the readings slightly more interesting it also reflects real life as the likelihood of the same sensory error occurring for more than one robot within a swarm at roughly the same time is extremely small.

As sensory failures tend to not be binary in nature (either always working to full capacity or never working at all) it is beneficial to look at the effect on performance as the percentage of failure changes together with the scale of the failure (the scale of the added offset changes). It could also be relevant to look at the correlation (if one should exist) between the score, the amount of agents and the percentage of failure. Because of the random nature of the failure percentage it is necessary to run these tests a certain number of times per setpoint. For both scale of failure and percentage of times a failure occurs 21 values were chosen, from 0%-100% with steps of 5%. To counteract the aforementioned randomness each different combination of setpoint was ran 20 times and their results averaged out.

Sadly this number is not quite enough to rule out the possibility of any randomness causing changes within the results. It is however impossible to run each of these permutations say 100 times due to the long time this would take. Each of these simulations takes anywhere from 30 seconds to a minute. With a total of $20 \cdot 21^2 = 8820$ runs this would, worst case scenario, lead to one single complete run taking more than 5 days. With 2 algorithms needing to be tested and 2 tests per algorithm this is not possible within the timeframe of this project. As a consequence of this only the experiment regarding both the scale and percentage of the error shall be run in full. The experiment with a changing number of agents will feature an always failing sensor.

As there is no definitive way to score the performance of a swarm a custom scoring function is required. It needs to take into account all of the different heuristics that are important in this specific situation:

- Amount of collisions with obstacles (Percentage of time in simulation) (obst_collision)
- Amount of collisions with other agents (Percentage of time in simulation) (agent_collision)
- Order of the swarm staying constant (Average) (order)
- Connectivity of the swarm (Average) (connectivity)

The final scoring formula then becomes:

$$score = (3 \cdot obst\_collision + 0.3 \cdot agent\_collision + 0.2 \cdot order + 0.2 \cdot connectivity)^{4.3}$$

As to penalise the collisions with obstacles more its weight is multiplied by a scalar of 10. To achieve a better distribution the calculated score is then scaled according to $score\_final = score^{4.3}$. This function achieves the goal of a low score for high amount of errors and high score for low amount of errors for both algorithms.

5.2 Python Based Delta Formation

The simulation used for this experiment is point-mass based and simple in nature. The swarm starts at velocity v_0 in the correct delta shape at which point one of the agents can have a failure in its distance sensing with regard to the other robots within the swarm. As such it will lose its position within the delta. A simple solution can be to increase the agents velocity if it's too far away and vice versa. As before the error is not boolean in nature. To make the experiment more representative of the system can be seen as a group of swarm members with springs between each other trying to keep each other in the correct range while also going through some sort of air resistance, thus causing damping. The nonlinear differential equation can be formalised as

$$x_i = \begin{cases} \dot{x}^i = v^i \\ \dot{v}^i = a^i = \sum_{j \in neighbours^i} F_{ij} - k_{damp} v^i \end{cases}$$

$$F_{ij} = \frac{(x^i - x^j)}{|x^i - x^j|} - k_{spr}(d_{ij} - d_0)$$

k_{damp}	damping constant of the system
k_{spr}	spring constant of the system
d_0	spring length at rest

As the system is two dimensional in nature its representation is simplified

$$x_i = \begin{bmatrix} x_1^i \\ x_2^i \end{bmatrix}, v_i = \begin{bmatrix} v_1^i \\ v_2^i \end{bmatrix}, a_i = \begin{bmatrix} a_1^i \\ a_2^i \end{bmatrix}$$

A failed front facing distance sensor can be introduced in the calculation of F_{ij}. The distance can be manipulated for a specific agent along the y-axis. The full state of the system with all the different swarm members can then be described by

$$Z = \begin{bmatrix} x^1 \\ v^1 \\ x^2 \\ v^2 \\ . \\ . \\ . \\ x^n \\ v^n \end{bmatrix}, \dot{Z} = \begin{bmatrix} v^1 \\ a^1 \\ v^2 \\ a^2 \\ . \\ . \\ . \\ v^n \\ a^n \end{bmatrix} = f(Z)$$

The next state of the system can be found through the equation

$$Z_{t+1} = Z_t + \Delta t \cdot f(Z_t)$$

This system can be further characterised as being underdamped, critically damped or overdamped. The critical damping coefficient in this system with point-masses can be characterised as

$$c_c = 2m\sqrt{\frac{k_{spr}}{m}} = 2 \cdot 1\sqrt{\frac{k_{spr}}{1}} = 2\sqrt{k_{spr}}$$

The distinction of the system type can then be made as follows

$k_{damp} > c_c$	overdamped system
$k_{damp} == c_c$	critically damped system
$k_{damp} < c_c$	underdamped system

This is an oversimplification of the physics behind this process and as such the reader is referred to *Engineering Mechanics. Dynamics* by R. C. Hibbeler and K. B. Yap [13]. With both methods formalised there is now the need to score them. The performance of the swarm is measured by the mean distance between the agents. It is then scaled according to a normal distribution with (multiplied by 3 for scaling purposes)

$$\mu = \sqrt{2}$$

$$\sigma = 0.0997356$$

$$score_i = 3|\frac{1}{\mu\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}|$$

The experiment shall be run over different permutations of failure scale and probability for the basic implementation as well as an underdamped system (as close as possible to being critically damped, e.g. $k_{spr} = 101$ and $k_{damp} = 20$), a critically damped system (e.g. $k_{spr} = 100$ and $k_{damp} = 20$) and an optimal system gotten through hypertuning of the spring and damping constants with a constant failure.

Another experiment shall be ran with no distance error present within the system. However all of the swarm members have delayed measurements coming in. They start out by receiving the starting positions of their fellow swarm members but at some delayed point 'new' measurements will start coming in. This amount of time shall be changed to several setpoints to measure the effect of this fault. The system will be in both critically damped and underdamped states.

5.3 BW4T Based Simulation

The experiments pertaining to the effect of a communication breakdown were ran through a MTRX based Python simulation. MTRX is a tool that allows for the creation of human-agent collaborative tasks and was used by researchers all over (including a few at the TU Delft) to create BW4T [11]. This is a framework that allows for easy creation of some collaborative task and differing types of agents with an already created communication protocol between these agents. The task that the swarm must complete will be a foraging task in this case. The level will be made up of a set of rooms each containing several blocks. There is a goal zone where the agents must deliver the goal blocks. Only 3 of the blocks are required, they are characterised by both their shape and colour and duplicates are possible. To add another layer of complexity the blocks must be delivered in a certain order as otherwise the task will not be seen as completed correctly and will require reordering. The swarm will consist of 3 separate agents, 2 'normal' agents and 1 'lying' agent.

The normal agents are capable of picking up 1 block at the time and will always relay relevant information as they discover it to ease the process for the other agents. The agents will send messages to other agents in the following cases:

- A goal block with colour *a*, shape *b* was found at location *(x, y)*
- A goal block with colour *a*, shape *b* was picked up at location *(x, y)*
- A goal block with colour *a*, shape *b* was dropped of at goal location *(x, y)*
- A door was opened at *(x, y)*
- A room with name *a* was searched
- The agent is moving to a room with name *a*

In short, the swarm works together by opening and searching rooms for goal blocks. Once a goal block is found it can be picked up and delivered next to the goal zone. Then finally one agent will drop off the blocks in the correct order at the drop zone and the task will be completed. The reader is referred to appendix A to gain a better understanding of the whole algorithm and the different phases involved. As the performance of the swarm is very much dependent on the quality of the communication and exchanged information it is this that shall be tested. The lying agent has a certain probability of providing faulty information to its fellow swarm-mates. When the agent lies it will provide a faulty location, colour, shape, name or any other permutation that might be possible within that specific message. The agent specifically avoids using the correct information and

can thus very much be seen as a malicious actor trying to drag the performance of the whole swarm down. To avoid the process devolving into a 2 agent foraging task a basic trust model is implemented following the Ability, Benevolence and Integrity system as proposed by Mayer, Davis and Schoorman [12]. As this experiment will only deal with the effects of a lying agent the only relevant metric is that of Integrity. Explained in a very intuitive manner this comes down to an agent losing Integrity through the eyes of another agent that finds out that given information is demonstrably false (important to note that these trust values are not centralised and every agent stores their own trust dictionary). Based on that Integrity score an agent can decide to ignore information received from another agent with a score below a certain threshold.

With this basic trust model now implemented 2 different experiments shall be run; one where there is no inter-round trust buildup and one where there is. In the first case, that of only intra-round trust buildup, the experiment is ran 440 times (to try and achieve a Monte Carlo simulation [14] and remove all randomness) and the trust values reset between every round. This shall be done for 21 different lying probability setpoints between 0 (never lying) and 1 (always lying). These values are then averaged out to get a representative result despite the inherent randomness of the lying probabilities. To see the effect of inter-round learning and the possibility of tolerance against this type of attack the experiment will once again be run but this time the trust values will not be reset between every round, only between setpoints of the lying probability. Once again, values are averaged out.

It might be noted here that this is not entirely the correct manner as during the learning the trust values continue to develop and as such the score of the runs will increase until at some point they plateau because the agents no longer learn anything. It was decided to still take this course of action as opposed to pre-training the agents before running the experiment as this would skew values and would make any conclusions with regards to building fault-tolerance null and void.

For this experiment there is an easier score available and there is no need for heuristics; the amount of time (measured in ticks) that it takes for the swarm to complete the assigned foraging task.

6 Results

Due to several different experiments being ran over three simulators it is beneficial to discuss them all separately. Experiments ran with two changing variables are displayed in a scatter plot. These plots were made using the graphing software Datylon [15].

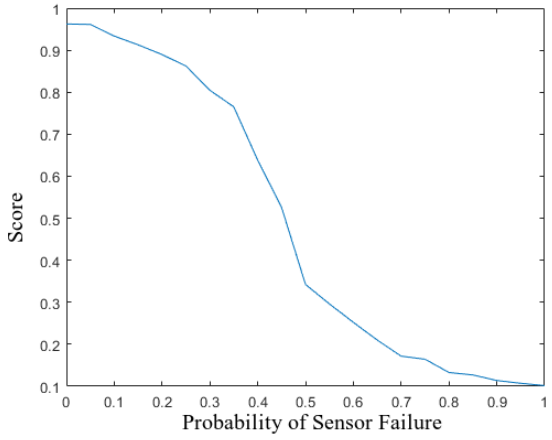

Figure 1: Effect of the fault probability on the score for the Vasarhelyi algorithm.

6.1 SwarmLab: Vasarhelyi

Fault Probability - Score Relation

Observable in figure 1 is the nearly linear fashion in which the score drops as the probability of fault occurrence increases. There does appear to be some squared error or sigmoid present in the data.

Fault Probability - Fault Scale - Score Relation

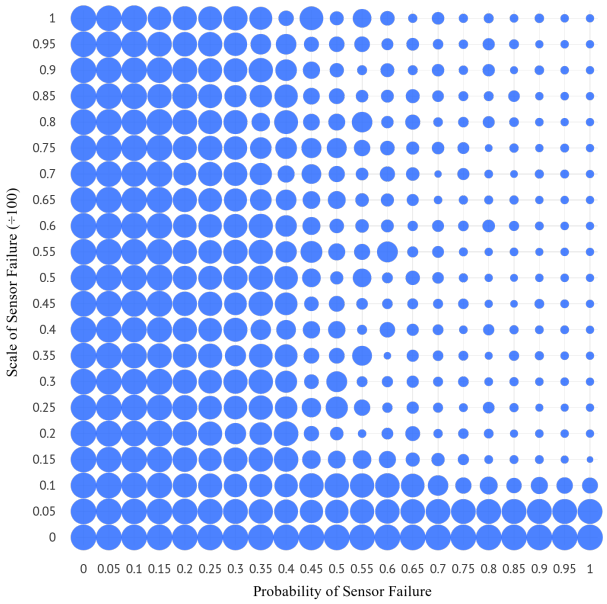

Figure 2: Effect of the scale and probability of a sensor failure on the score for the Vasarhelyi algorithm. Score shown as size of marker.

It is apparent in graph 2 that the scale of the fault is not as influential on the probability of the occurrence of a sensor failure. It can also be noted that there seems to be some inconsistency or randomness in the data. This might be due to the aforementioned long running time and thus the lack of performing a proper Monte Carlo simulation. Noticeably the effect of the sensor failure probability is not linear; the score seems to drop of a 'cliff' at around 50% probability regardless of failure scale.

Swarm Size - Fault Effect

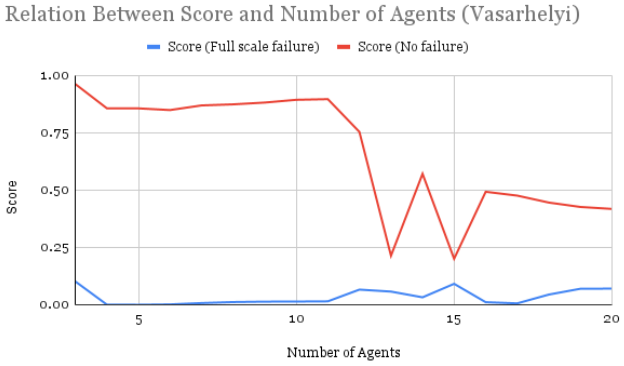

Figure 3: Effect of the swarm size on score for the Vasarhelyi algorithm.

Apparent in plot 3 is that there is lot of noise and that the results do not allow for any conclusions to be drawn. The only useful, notice-

able trend is that the score decreases as the number of agents within the swarm increases.

6.2 SwarmLab: Olfati-Saber

Fault Probability - Score Relation

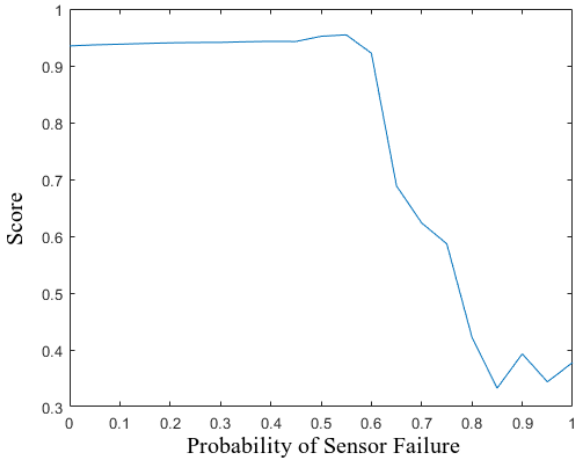

Figure 4: Effect of the fault probability on the score for the Olfati-Saber algorithm.

Contrary to the Vasarhelyi algorithm the score does not go down linearly as the probability of a sensor failure increases as seen in figure 4. Rather the score goes up a bit before going on a steep decline.

Fault Probability - Fault Scale - Score Relation

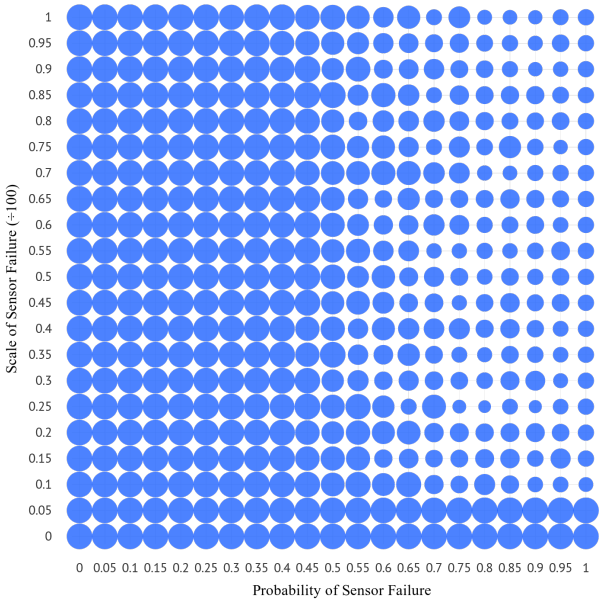

Figure 5: Effect of the scale and probability of a sensor failure on the score for the Olfati-Saber algorithm. Score shown as size of marker.

From a first glance at plot 5 it becomes clear that the score loss for this algorithm is less dramatic than that for Vasarhelyi. Once again the effect of the failure scale seems to be less impactful than that of the probability of an error occurring. The point at which the 'cliff' occurs also appears to be slightly later on in the experiment.

Swarm Size - Fault Effect

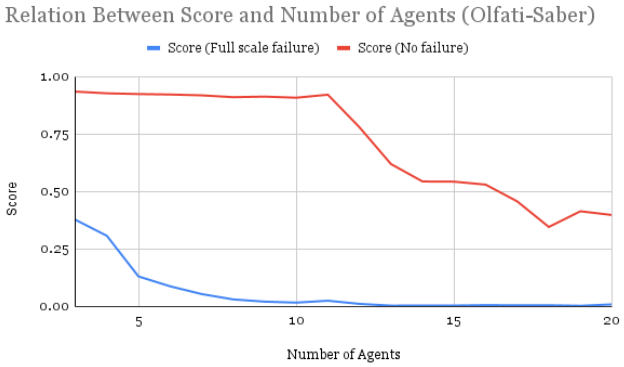

Figure 6: Effect of the swarm size on score for the Olfati-Saber algorithm.

Unlike the resulting data from this same experiment on Vasarhelyi the data in graph 6 seems to be more useful and representative. For the full scale failure the data once again drops quite linearly but the more interesting result comes from the no failure experiment. There the relation between score and number of agent follows the score and fault probability; staying constant or even slightly increasing until the score hits the 'cliff' at around 12 agents within the swarm.

6.3 Delta Formation: Basic Implementation

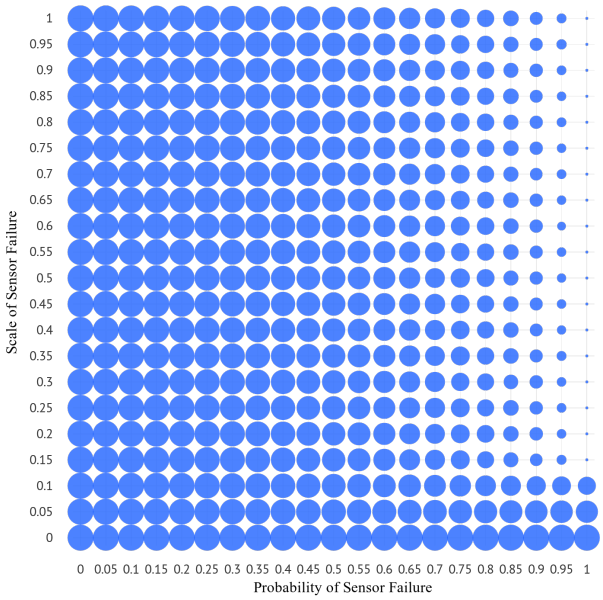

Figure 7: Effect of the probability of sensory failures and failure scale on the score within a delta formation swarm. Score shown as size of marker.

Clear in plot 7 is that this simple algorithm keeps the score up as long as possible as the probability of a failure increases. More interesting is the fact that the score slightly increases as the scale of the error increases for the same probability of a failure. Also noticeable is the dramatically low score on the very highest of failure probabilities.

6.4 Delta Formation: Damped System

Critically Damped System

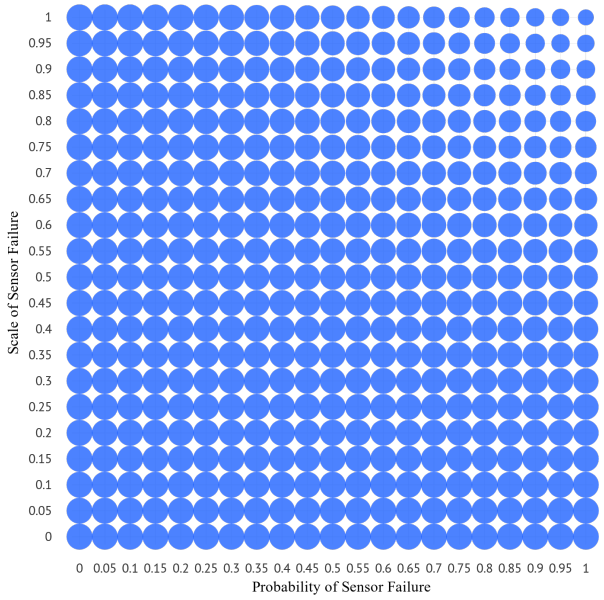

Figure 8: Effect of the probability of sensory failures and failure scale on the score within a delta formation swarm in a critically damped system. Score shown as size of marker.

Plot 8 shows a clear linear decrease in score for the higher combinations of both failure probability and scale. It is however apparent that the failure probability has slightly more effect on the score but this can be considered negligible.

Underdamped System

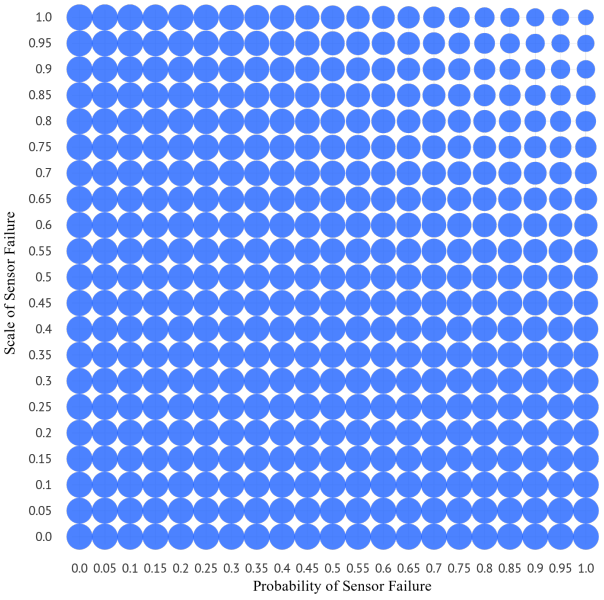

Figure 9: Effect of the probability of sensory failures and failure scale on the score within a delta formation swarm in an underdamped system. Score shown as size of marker.

The data of graph 9 follows a very similar pattern as that in plot 8. The overall scores are lower but no new conclusions can be drawn from this data.

Optimally Tuned System

System came into its optimal state at a very low spring constant with a very high damping constant. This leads to a severely overdamped system that has barely any performance drop-off regardless of fault probability or fault scale; the only loss of performance is in the highest echelons of both fault probability and scale and even then it only loses less than a percent of performance.

6.5 Delta Formation: Information Delay

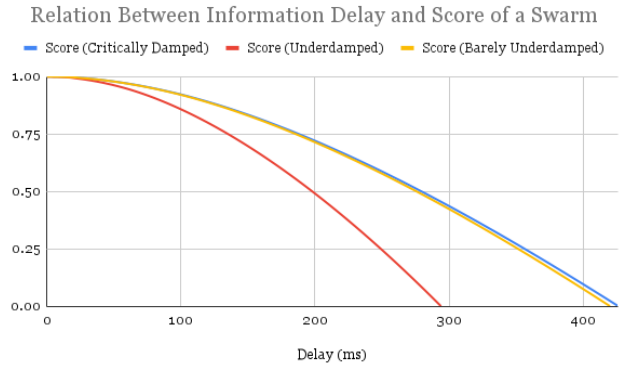

Figure 10: Effect of the delay of information gathering on the score of a swarm attempting to maintain a delta formation.

Graph 10 shows a clear parabolic relation. Depending on the damping coefficient (discussed in section 5.2) the gradient of the function also changed where a more underdamped system leads to a lower score.

6.6 BW4T: Intra-Round Learning

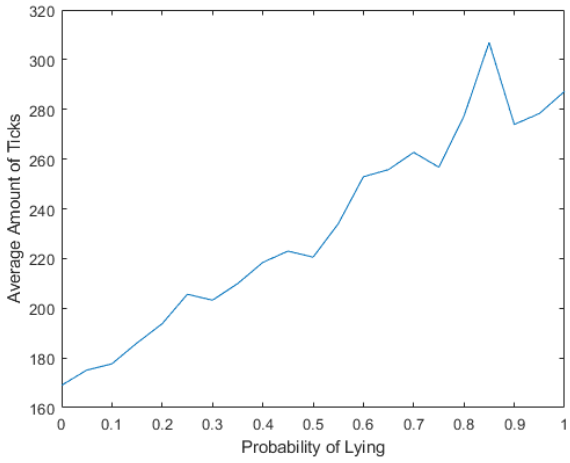

Figure 11: Effect of the probability of lying within a foraging task without inter-round learning.

Apparent in figure 11 is the near linear manner in which the amount of ticks the task takes goes up as the probability of lying increases. There are however several outliers along the path and it is clear that these outliers occur more as the probability of lying increases.

6.7 BW4T: Inter-Round Learning

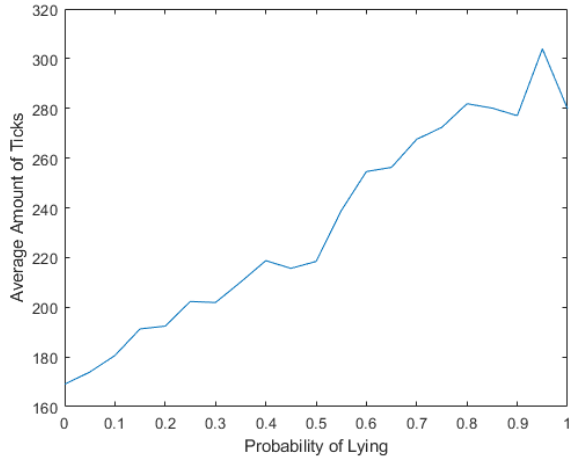

Figure 12: Effect of the probability of lying within a foraging task with inter-round learning.

Although figure 12 is similar in the linear fashion as figure 11, without learning, there does seem to be a bit of a change in slope around 0.4 probability. There are again still some outliers caused by noise.

7 Responsible Research

As there is no private or real world data involved in this research there is no need for anonymising. The notion of robot swarms used in real life situations does bring with it the repercussions of possible use cases of divided swarms. There are a plethora of possibly bad scenario's such as extensive surveillance of individuals which would be eased through the use of a swarming system. As this is very much out of scope for this article the reader is encouraged to read *The upside and downside of swarming drones* by Lachow [16].

More pressing concerns might be those regarding the repeatability of the performed experiments. Due to the innate randomness of some of the experiments performed it was paramount that they were repeated enough times to reduce the susceptibility of the data to outliers within the dataset. Sadly due to time constraints it is not always feasible or possible to run as many simulations as would be strictly necessary to achieve a Monte Carlo simulation [14] but with a relatively low standard deviation it is presumed that this gathered data is sufficient to draw some first conclusions.

8 Discussion

The data collected during the experiments show mixed results. Some show apparent trends, but others are more difficult to interpret and as such will not be further considered to draw conclusions from.

8.1 SwarmLab: Vasarhelyi Results

Fault Probability - Score Relation

Clear nearly linear relation between the probability of a fault occurring and the resulting score of the run in graph 1. This can be explained by the nature of the Vasarhelyi algorithm and how its velocity vectors are computed. With the agents calculating said accelerations/velocities all at once utilising last rounds' velocities and positions the algorithm is less sensitive to outliers. This causes the more linear drop off in performance that can be seen.

Fault Probability - Fault Scale - Score Relation

As in the graph 1 above the same result can be concluded from the x-axis of figure 2. Noticeable however is the seemingly ineffectiveness of the scale of the error. This can be explained by the fact that if the scale of the error is small, the error might fall inside of the range of the point-mass and as such the result will be ignored in a sense. At some point the probability becomes great enough that almost regardless of the scale the score drops off the aforementioned 'cliff'. Also noticeable is that a failure scale smaller than 0.1 is almost completely inconsequential to the score of the run.

Swarm Size - Fault Effect

As noted in the section "Vasarhelyi Swarm Size - Fault Effect", the results seen in plot 3 seem rather noisy. This can possibly be caused by an oversight within the design of the scoring function and how it handles outliers. It is unclear why this does not happen for the Olfati-Saber algorithm. Due to the amount of noise within the graph the only conclusion that can be drawn is the performance loss with more agents within the swarm due to less swarm coherence.

8.2 SwarmLab: Olfati-Saber Results

Fault Probability - Score Relation

Apparent in figure 4 is the non-linear relation between the fault probability and the score. As the algorithms work in contrary fashions (calculating velocities using info gathered agent by agent/all agents at once, Olfati-Saber using the former) it stands to reason that the gathered results are contrarian as well. The slight increase in score as the probability of a fault increases can be explained by this. The higher susceptibility to outliers can cause the agent to follow a nearby agent that has had an error and as such is going a lot faster because it is not stopping for an obstacle ahead of it. When this failure doesn't occur often the agent with the failure will not collide with the obstacle and as such will not drop the score. Due to this fact the other agents will also speed up to try and follow the faulty agent and thus keep higher swarm cohesion and order. However once the faulty agent starts colliding with obstacles (due to a higher probability of a failure) this effect starts to work in the opposite manner; the agents try to follow an agent with a failure and will thus also collide with the obstacle.

Fault Probability - Fault Scale - Score Relation

It is immediately clear from plot 5 that for the Olfati-Saber algorithm the fault scale appears to have almost no impact on the score. Other than that the results are very similar to the fault probability - score relation graph found above.

Swarm Size - Fault Effect

The immediately clear trend in graph 6 is that of the score dropping as the number of agents within the swarm increases. Interestingly the score almost follows the same trend as for the fault probability - score relation when there is no fault within the algorithm; staying constant and even increasing for some time before finally dropping drastically.

8.3 Delta Formation: Basic Implementation

The slight increase of score as the fault scale increases seen in graph 7 can be explained by the thresholds chosen. At some point the distance between robots within the swarm becomes large enough that there shall be some acceleration/deceleration for the lagging/encroaching agent. As the failure scale increases the possibility of the agent getting into these thresholds so does the probability of a correction in the speed of the agent. The noticeably low scores for the upper echelons of sensor failure probability can be explained by the normal distribution used for scoring.

8.4 Delta Formation: Damped System

Critically Damped System

The overall higher scores seen in figure 8 compared to that of graph 7 can be explained by the added dampening in the system; when

a sensor failure occurs the "springs" between the swarm members will cause an acceleration to compensate for the perceived change in distance. As the dampening effect by very definition counteracts any acceleration, the effect of the sensory fault will therefore be lessened if not completely counteracted.

Underdamped System

The slight decrease in minimum score seen in graph 9 can be explained the same way as in the section above, 8.4; as the system is in an underdamped state the damping constant has decreased and as such its dampening effect is less effective. This then causes the accelerations caused by the "springs" to have more effect and as such reduce swarm coherence.

Optimally Tuned System

The near perfect score seen here is due to the fact that both parameters were tuned together. If the spring constant decreases its effect will lessen and vice versa, same for the damping constant; a low spring constant then means that there is nearly no force keeping the agents together once a small distance forms up. This coupled together with a high damping constant (effect explained in section 8.4) causes the system to be immune to outliers as they don't cause any reaction in their neighbours nor can the accelerate quickly due to the strong dampening effect. This coupled together leads to basically neutralising the sensory fault. A more real-world representative experiment would be to tune these parameters independently.

8.5 Delta Formation: Information Delay

From the parabolic functions seen in graph 10 the conclusion can be gathered that the swarm does still function up until some delay from which it cannot recover. This seems to be between 50-100 ms depending on the system configuration. The lesser score depending on the system state has been explained earlier in section 8.4 and it stands to reason that seen relation also follows from this.

8.6 BW4T: Intra-Round Learning Results

These outliers seen in the data in figure 10 can be explained by the failure of the agents to complete the task should enough misinformation (or in another few, rare set of circumstances) be present. Should the swarm fail to complete the task this will be seen as a timeout and the run will return a result of a 1000 ticks. The large number of runs serves to smooth out this value to try and avoid large spikes such as those appearing at a lying probability of 0.85. Why this particular spike still appears is not immediately apparent but it can be presumed that at this setpoint of lying probability the trust system fails to discover quickly enough whether some agents are completely unreliable in the information that they share. This causes some agents to still assume the correctness of false information and as such act in an incorrect manner.

8.7 BW4T: Inter-Round Learning Results

The expectation would be that this experiment performs better for the lower probabilities of lying with trust buildup over multiple rounds the lying nature of agents is uncovered/known earlier and as such the swarm will not lose performance by trusting said agents. The nearly equal results in the higher lying probability reaches can be explained due to fact that with such a high probability these agents will be caught soon and as such not be included in the process regardless. However this cannot be accurately seen in the data in figure 11. There is a slight hint of a sigmoid (as described above) but it is not visible enough to be distinguished from a linear function affected by noise. This result could be specified more by running more epochs per setpoint of lying probability and thus filter out outliers. Another possibility could be to make the agents within the swarm recognise when they are in a hard lock and handle this better or terminate sooner.

9 Conclusions and Future Work

This paper endeavored to discover the relation between introduced sensory faults within a robotic swarm and whether or not this relation can be quantified for different types of errors. Multiple experiments were conducted on multiple types of sensory faults within different custom written simulations. It was discovered that while usually quantifiable, it is hard to create a definitive function that describes the effect of any given type of sensory error.
While multiple different types of sensory failures were discussed it is by no means a comprehensive list and in the future this could be a good subject for further research into this topic. It could also be beneficial to run tests with more epochs; this was not doable in the scope of this research paper, due to time constraints.

References

[1] J. Zhao, H. Xu, H. Liu, J. Wu, Y. Zheng, and D. Wu, "Detection and tracking of pedestrians and vehicles using roadside LiDAR sensors. Transportation research part C: emerging technologies." March 2019, Accessed on: 5-May-2022. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0968090X19300282?via%3Dihub

[2] R. Abbasi, M., K. M., and M. et al., "An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems." *Journal of Cloud Comp*, vol. 9, no. 1, pp. 1–14, December 2020.

[3] A. Winfield and J. Nembrini, "Safety in numbers: Fault tolerance in robot swarms," *International Journal of Modelling Identification and Control*, vol. 1, pp. 30–37, 01 2006.

[4] Y. Leung, G. Li, and Z.-B. Xu, "A genetic algorithm for the multiple destination routing problems," *IEEE Transactions on Evolutionary Computation*, vol. 2, no. 4, pp. 150–161, 1998.

[5] J. D. Bjerknes and A. F. T. Winfield, *On Fault Tolerance and Scalability of Swarm Robotic Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 431–444. [Online]. Available: https://doi.org/10.1007/978-3-642-32723-0_31

[6] E. Soria, F. Schiano, and D. Floreano, "Swarmlab: a matlab drone swarm simulator," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 8005–8011.

[7] C. Reynolds, "Flocks, herds, and schools: A distributed behavioral model," *ACM SIGGRAPH Computer Graphics*, vol. 21, pp. 25–34, 07 1987.

[8] C. Virá gh, G. Vásárhelyi, N. Tarcai, T. Szörényi, G. Somorjai, T. Nepusz, and T. Vicsek, "Flocking algorithm for autonomous flying robots," *Bioinspiration &amp Biomimetics*, vol. 9, no. 2, p. 025012, may 2014. [Online]. Available: https://doi.org/10.1088%2F1748-3182%2F9%2F2%2F025012

[9] R. Olfati-Saber, "Flocking for multi-agent dynamic systems: algorithms and theory," *IEEE Transactions on Automatic Control*, vol. 51, no. 3, pp. 401–420, 2006.

[10] MATLAB, *version 9.12.0.1884302 (R2022a)*. Natick, Massachusetts: The MathWorks Inc., 2022.

[11] M. Johnson, C. Jonker, M. Riemsdijk, P. J. Feltovich, and J. Bradshaw, "Joint activity testbed: Blocks world for teams (bw4t)," 11 2009, pp. 254–256.

[12] R. C. Mayer, J. H. Davis, and F. D. Schoorman, "An integrative model of organizational trust," *The Academy of Management Review*, vol. 20, no. 3, pp. 709–734, 1995. [Online]. Available: http://www.jstor.org/stable/258792

[13] R. C. Hibbeler and K. B. Yap, *Engineering mechanics. Dynamics*. Pearson, 2017, ch. 22, pp. 631–670.

[14] P. Bonate, "A brief introduction to monte carlo simulation," *Clinical pharmacokinetics*, vol. 40, pp. 15–22, 02 2001.

[15] Datylon, *version R49.2*. Antwerp, Belgium: Datylon, 2022.

[16] I. Lachow, "The upside and downside of swarming drones," *Bulletin of the Atomic Scientists*, vol. 73, no. 2, pp. 96–101, 2017. [Online]. Available: https://doi.org/10.1080/00963402.2017.1290879

A BW4T Collaborative Agents

Collaborative AI BW4T Practical Assignment

Zhiyi Chen , **Stijn Coppens** , **Marco Nicola Stroia** and **Kevin Zhu**
TU Delft

Abstract

Automated agents that can operate by themselves or in tandem with humans are an important domain of collaborative AI development. These agents can facilitate human tasks or replace part of human workload, potentially improving efficiency and day to day life of the users.

In this paper we are tackling an introductory approach to developing collaborative agents. These agents will perform simple tasks in a game-like environment named BW4T (Blocks World for Teams), based on the Collaborative Artificial Intelligence course curriculum given at TU Delft.

1 Introduction

Interaction between intelligent agents and between humans and such agents is an important field of research in collaborative AI, and it paves the way for easing the workload of humans in environments where tasks can be partially or even fully automated by AI. Studying the interaction between these automated agents is important since it provides insight on how to further optimize their behaviour when cooperating with different (and sometimes uncooperative) agents.

This paper will focus on the interaction between different kinds of automated agents, and offers a general description of how they collaborate in a specific environment, what their purpose is, what are the strategies and algorithms that they each implement, and how we tested and evaluated them in order to achieve the best results.

The environment in question is named BW4T (Blocks World for Teams), a game-like world where the objective is to cooperate with the other agents (either AI or human) and deliver a sequence of blocks in a particular order as fast as possible. Our implementation extended the MTRX framework for BW4T.

2 High level description of the agent and the code structure

This section will include a high-level description of the overall architecture of our agents, and the steps they take in order to complete the given task.

For this assignment, we were tasked with developing four agents that have to cooperate in order to deliver a certain set of blocks to a destination, and order them in a predefined order. Generally, agents have the same goal of delivering the blocks to the destination. Each of the four agents had to be implemented with a specific characteristic: one agent has to be lazy, one agent has lie about their actions and observations, one agent can not distinguish colors and one agent has the ability to carry two blocks at once. The general algorithm and the particularities of each agent will be described in the following subsections.

2.1 General Algorithm

All of our agents were developed on top of a common base implementation, which we called the "Normal Agent". This agent performs the tasks at hand correctly and efficiently, and provides an idea of an ideal way to solve the problem. All the implemented agents change certain aspects of the normal agent in order to fit their defining characteristics.

Most of the logic of our agents is performed in two methods "filter_observations" and "decide_on_bw4t_action". "filter_observations" is used to process information received from the other agents in the team, updates the understanding agents have about the task at hand and the world state, and change the level of trust each agent has in the other agents.

The "decide_on_bw4t_action" method is used to decide the behaviour of an agent for the current turn. This behaviour depends on the phase the agent is currently in. These phases are:

- FOLLOW_PATH_TO_CLOSED_DOOR: in this phase agents will look for the closest unexplored room, send a message, and then move towards that location. The process will continue even after all rooms have been explored. This phase also possesses a bidding aspect to prevent multiple agents from going to the same room. The bidding aspect will be explained in more detail in section 2.3.
- OPEN_DOOR: the agents will open the door (if closed) and send a message.
- SEARCH_ROOM: agents will explore the room they visited and send the corresponding message. While exploring they will search for blocks matching their objective. If such a block is found, another message will be

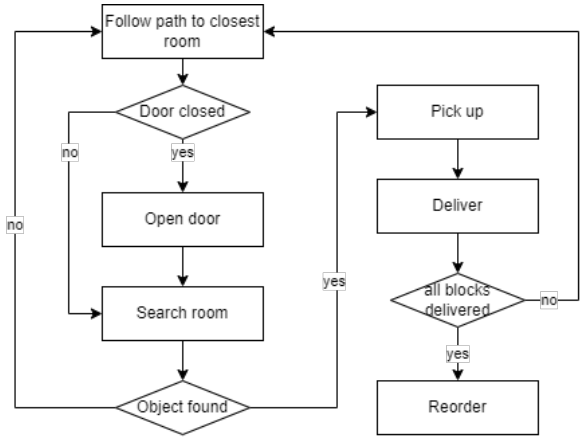

Figure 1: Simplified flow of actions assuming a single, normal agent.

sent and the agent will proceed to pick up the block.

- PICK_UP: the agents will pick up the target block and send the corresponding message.
- DELIVER_OBJ: in this phase, agents will move towards the drop location of the block they picked up. After reaching the destination, they will drop the block and send the corresponding message.
- REORDER_OBJ: when all blocks have been delivered but in the wrong order, agents will attempt to reorder the object. If a missing (or wrong) block is found in the drop-zone, a message is sent and agents will continue exploring. This phase possesses a bidding aspect to prevent multiple agents from reordering at the same time.
- DEAD: agents reach this phase when all blocks have been delivered, and will not perform any further action. It is possible to get out of this phase if any agent notices

As seen in Figure 1, these phases flow into each other, but this flow can be altered by an agent characteristic or information received from other agents. For example, the lazy agent can decide to change it's phase without finishing it, or an agent could go straight to the pick up phase if it receives a message stating that a block has been dropped outside the drop-zone.

2.2 Communication protocol and coordination

The actions the agents perform are affected by the actions of their team-members, and this information is shared through messages they send to each other. The default messages defined by the assignment itself are: communicating the room they are currently moving to, which door they open, which room they are searching, when they find a goal block, what block they have picked up and where they have dropped it (with the last three messages containing information about the block's visuals and location).

Besides these default messages, our bidding algorithm makes use of additional custom messages, used to assign and prioritize the best agent for the job. All of these messages are important for planning the actions that are required to complete the objective, and they contribute greatly in regards to the overall efficiency and building trust images of each other agents.

When messages parsed at the beginning of each turn, they update the knowledge an agent has about the state of the world and what tasks have been accomplished by other agents. This allows agents to infer the next best course of action, better distribute the tasks and cooperate with each other. The parsing of messages is also affected by trust, which will be explained in section 3.3.

2.3 Bidding

Our bidding algorithm is used to enforce better collaboration between agents by preventing multiple people from performing the same task. It is used in different phases of the program, such as deciding who is going to explore which room and who is going to perform the reordering action. This bidding applies when two separate agents declare the same action during the same turn: in such a scenario a bid will be created and agents will append additional messages containing their distance to the objective, a random roll (between 1 and 100) and their trust image (more details in section 3.1. In the following turn, each agent will compare their own bid to the opponents, with the bidder closest to the objective being the winner (with the random roll being used to break up ties). The winner will get to perform the declared action, while the losers will move on to a different unexplored door (in the case of room exploration) or go into the dead phase (in the case of reordering).

The bidding algorithm is also influenced by the trust values, which will be explained in section 3.3

2.4 Colorblind Agent

The colorblind agent follows the standard implementation of the normal agent but differs in one key aspect; it's inability to see colors. For our implementation, we interpreted this characteristic as the agent not having direct access to the color related fields in the visuals of both the blocks and the drop-zone. As a result, he is unable to complete the objective by himself. He will still attempt to explore rooms and send messages when finding a block that match any of the goal blocks on all characteristics but the color, but these messages will be usually ignored by himself and other agents (although it will decrease how much he is trusted). When possible, the colorblind agent will try to be helpful by delivering blocks found by other agents (if the color field is present in the message) but will be unable to confirm whether said block is correct, making him a bad match for liar agent.

2.5 Strong Agent

The strong agent is a version of the normal agent that can carry two blocks at the same time, making him a direct improvement in terms of efficiency. It performs all of its actions correctly and will never lie in its messages. After picking up a block, if there are still goal blocks left to be delivered and he is able to carry more blocks, he will continue exploring until the last missing block is picked up (by him or others). Due to his high efficiency, he will be easily trusted by other agents,

and said trust will tend to rise in a faster manner compared to others, giving him better odds at receiving jobs through bidding.

2.6 Liar Agent

The liar agent is characterised by it's dishonesty about shared information. It will still complete all tasks as the normal agent would but any messages it sends out with information regarding his actions or observation will have a 80% chance of containing false information. It will not lie with regards to the type of action it's taking (for example if the agent says that they're picking up a block, they are actually picking up a block). It can however lie about the location and/or appearance of the block it's picking up. Thus any messages pertaining to blocks (finding a block, picking up a block and dropping a block) will have an 80% chance of containing a lie with regards to the location where this happened or what the exact block was.

Note that the false locations and visual information about the blocks that will be used in these lies will always be within the confines of the map (excluding walls) or actual goal blocks. The other messages with information regarding doors/rooms (moving to room, opening door of room and searching through room) will lie about which room the agent is interacting with, but will still use an existing door/room in the map. In short, the liar agent will always complete it's declared action but the information it gives out while doing said action has an 80% chance of being a lie.

2.7 Lazy Agent

The main trait of the lazy agent is its lack of willingness to complete tasks that it started. This is implemented in a way that allows the agent to give up on the task it is currently performing 50% of the time. This is done by calculating the number of turns required to perform the task, for example the number of steps needed to move from A to B, and giving each step an equal chance of being lazy and changing task. The sum of the probabilities of being lazy in each step will sum up to 50%, meaning that while the agent will complete the task half the time, in the other half it could be interrupted in any stage of the process.

The tasks are prioritized in such a way that the agent will always give up on tasks in order to perform easier or just as hard tasks, if they are available and necessary. For example, the lazy agent could stop searching a room and look for another room to search, if there are unsearched rooms left, instead of going to pick up and deliver a block. The lazy agent can finish the task by itself given enough time, but does so in an very inefficient manner. Despite this, it always provides truthful information and will also declare it's change of action. For example, if it is in the middle of delivering a block and decides to switch task, it will first drop the block that is currently being carried and notify the other agents of its location.

3 Trust

Trust is an important aspect in collaboration between humans, but it also applies to human-agent and agent-agent collaboration. Because of this, our agents implement a trust mechanism that relies on the information from received messages and knowledge about the world in order to decide how trustworthy other agents are.

3.1 Trust Model

In order to include trust mechanisms into our agents, we decided to implement an algorithm following the ABI model, described in [Mayer *et al.*, 1995]. This model is built upon three main "factors of perceived trustworthiness": Ability, Benevolence and Integrity. These three factors are influenced by the perception of each agents in regards to events that happen in the world, and the declared actions and observations of the other agents. Together they contribute towards deciding on a level of trustworthiness to be attributed to these agents. Additionally, this trustworthiness can be used to influence which actions will be designated to each agent in a team, which also affects the amount of contribution that an agent can provide in a round.

Trust levels of an agent towards others can be influenced by different factors, such as direct experiences of the agent, or indirect knowledge learned from others. According to [Weiss, 2013] direct experiences are considered the most reliable since "they come form a direct perception without intermediaries". The book also mentions that "almost every multi-agent trust model uses the agent's direct experiences as a source for image calculation". Because of these aspects, we based a majority of our trust calculations on direct experiences of the agent, which in the case of this project were represented by the messages agents sent to a public channel of communication.

Another important aspect of the way agents build trust, is the reputation the other agents build for themselves. A simple way to tackle this problem is having agents communicate their trust images of all the other agents in the team. This way, every agent has access to the perception each team member has, and these perceptions can influence the decisions made while performing tasks. An example of such an implementation is our bidding algorithm, explained in section 2.3.

3.2 Trust Implementation

This subsection will cover the ways in which the trust images of agents are used while solving the given tasks and how their usage optimizes the problem solving process.

On an individual level, agents build trust by interpreting messages sent by the other team member and using this information to form a trust image on them. The main method we used to derive this information is by reading the messages received from each agent, and looking for different patterns in order to draw conclusion in regards to their activity and trustworthiness. This way, different patterns target different aspects of trust defined in the ABI model. For example, if an agents notices that another agent picked up and successfully delivers two of the goal blocks, it will increase the perceived level of Ability for that agent. Conversely, if the agents notices someone declaring they picked up a block and then dropped it at the wrong location, it will decrease the Benevolence and Integrity of that agent.

These trust parameters are also affected by the quality of the received information, as they provide insight on the characteristics of the sender. This is done for example by detecting and punishing misleading or impossible observations by decreasing Integrity, punishing lacking or useless information by decreasing Ability, but also rewarding correct behaviour and truthful messages.

These trust values persist between rounds played, so team members can start working more efficiently with each other after more rounds in the same team. This information is stored in a memory file that each agent has. At the beginning of a round, it gets imported from the file and is further modified throughout the round. In the case in which the files are not there, the program will initialize the trust image to default values and create the file at the end, in order to be used in the next iteration.

3.3 Influence of Trust on Agent Actions

In our implementation there are two aspects that are directly affected by the trust images that are built up by our agents: the bidding algorithm and the communication protocol.

The bidding algorithm uses the trust value akin to a reputation system. The more trusted you are by your team members, the more positively your bid will be considered by the algorithm. Different tasks will take into consideration different trust parameters (integrity, ability, benevolence), and the amount of trust that each participating agent has towards a particular member determines the member's trustworthiness value. This value is then subtracted from their distance value. As such, an agent that is highly trusted by its team members will have a positive value that will effectively decrease the distance taken into consideration by the bidding algorithm, making them more likely to win the bid.

The addition of trust in the algorithm provides us a way to optimize the task solving process by designating tasks to agents which are more likely to complete said task in an optimal manner, either more efficiently or without the need of other agents to fix their mistakes.

The communication protocol on the other hand, mainly uses the trust image to determine how reliable the information received are, and avoid trusting messages that have a high chance of being misleading or wrong. This does not only take into account Integrity, but also ability and benevolence: for example if an agent with very low ability explores a room, the truthfulness of his action won't be questioned, but his capability of exploring the room correctly will be. As such his room will still be considered as an option for exploration.

4 Testing the agent and the analysis of results

In the following section, the performance of our implementation will be discussed in several different configurations in regards to the agents participating in the world and the number or type of goal blocks that need to be delivered. The performance was be measured through the amount of ticks it takes to complete a round on average, if completion is possible in the given configuration. Finally, to analyse the agents in their role as collaborative with human agents, an interdependence analysis was conducted.

4.1 Strong Points

In this section we will present some of the aspects that we believe are the strong points of the implementation of our agents.

Trust Algorithm
A wide variety of cases and scenarios have been taken into consideration to increase and decrease the trust value in our three categories of trust (integrity, ability and benevolence). This allows the trust images to be built accurately and efficiently, providing insight on the behaviour of the other agents in a relatively quick manner, and showing in what areas they can or cannot be trusted.

Utilization of trust images
We incorporated the trust images in our implementation in different places to improve the robustness and efficiency of our agents. This can be seen in our bidding algorithm, which uses reputation to assign the most optimal agent to different tasks, and in our parsing algorithm, to offset the negative influence that some agents can have based on which category of trust they're considered not trustworthy.

Fidelity to the given specifications
While the implementation of the characteristic of some of the agents were open to interpretation (namely the liar and lazy agent), the way we decided to implement these features was heavily focused on how accurately we believed their behaviour would match their given title. This meant not taking shortcuts such as making our agents conduct in a manner that would too easily expose their identity or that could be easily taken advantage of for better efficiency.

Examples of this are: our liar agent being able to only make plausible lies that could not be directly detected without context, our lazy agent being able to interrupt a task at any point during the course of said task with equal chances, and, if interrupted, to only switch to a task that is considered easier (or with comparable difficulty).

Collaboration between less performant agents
While working on our implementation, we also considered how to make the agents make up and cover for each other's deficiencies, even without the presence of a strong agent. A prime example of this is the behaviour of our blind and lazy agents, which individually are unable to complete the given objective efficiently, but together can collaborate and finish the round with a relatively quick pace.

4.2 Weak Points

Our algorithm performs well in many metrics but there is still room for improvement. Below we shall discuss three different fields in which we feel performance would benefit from further optimization the most.

Lack of robustness in certain edge cases
While in general we believe our implementation to be quite robust, this becomes less true when the liar agent comes into play. Due to the highly random nature of its behaviour, a few relatively rare edge cases are present and still unaccounted, partly due to a lack of testing and difficulty in replicating the issues.

For example when there are duplicate goal blocks together with a lazy and liar agent it is possible for the agents to become confused and take much longer than strictly necessary to finish the task.

Room for further optimization of agents behaviour

Due to time restriction within the project, we had to put more focus on developing a stable, working implementation of the agents and had to forego more complex optimizations that could've potentially increased overall efficiency. This includes adding heuristic optimization in room exploration, better subdivision of tasks between agents and improving the path finding algorithms. Additionally, we could have also taken more advantage of the characteristics of individual agents through the use of trust values.

Lack of indirect experiences and reputation

Currently the trust images of our agents are mostly built up through direct experiences, by parsing messages that are shared globally and contextualizing it with the previous messages and the state of the world. This results in agents having the same direct experiences and trust images between different agents being unsurprisingly similar. As such, sharing of indirect experiences and reputation isn't quite necessary.

What we could benefit from, is a way for agents to personally infer trustworthiness, and a communication protocol that includes the sharing of this information as indirect experiences and reputation.

Inefficient code

In our implementation, a bigger emphasis was put on the efficiency of agents in completing their task, and a lesser priority to the efficiency of our algorithms. As such, part of our code is still unoptimized and run-time tends to slow down noticeably after 400 ticks. Possible reasons behind this are the way we parse messages and build trust (with run-time slowing down as the number of messages stored increase), and the way we pre-calculate distances (an issue highly relevant to the lazy agent).

4.3 Interdependence Analysis

In this selection we will discuss the interdependence between different types of agents and how they affect the OPD requirements.

Interdependence between between different types of agents

As shown in the IA tables, each agent has different capabilities and can accomplish the task of "Fill the drop zones with the correct objects in the predefined order" by itself to a different extent. The liar and the strong agents have the capacity of accomplishing all sub-tasks by themselves. In the meantime, the assistance of the supporter agents can be sometimes useful and enables the performer agents to finish the task more efficiently. For instance, for the sub-task "Locate Object", the liar and strong can explore the world and find out the correct location of the target by themselves. However, this process will take much less time with the assistance of other agents. These supporter agents can help the performer agent explore the world and locate the target blocks faster. The same applies to the sub-task "Recognize successful deliveries" (identifying which blocks have already been delivered). The supporter agents can ease this sub-task by sending correct messages to notify the performer agent that some target blocks have already been located and delivered.

On the other hand, the lazy agent also has the capacity of finishing all tasks and sub-tasks, but it is as reliable. For instance, it might stop in the midway while exploring the world or delivering the block to the correct location. Under such situations, the supporter agents can assist the lazy agent to enhance its reliability. The colorblind agent is the only type of agent, which does not have enough capacity to finish every task and sub-task by itself. As it cannot see the color of the blocks, it only receives limited information regarding the world while exploring. Thus, it cannot completely verify the correctness of target blocks when passing by. This also affects its ability to check the correctness of blocks dropped in the target locations during the re-ordering phase. Therefore, for all sub-tasks involving colors, it requires the assistance of supporter agents.

OPD requirements

From the interdependence analysis of the different types of agents, we can conclude the OPD requirements. Since the colorblind agent requires assistance from other agents on the tasks "Locate object" and "Recognize successful deliveries" and all other types of agents can perform more efficiently with such assistance, there are observability requirements to make the blocks found or delivered to the drop zone by one agent known to all the other agents. As the lazy agent has a 50% chance of giving up the delivery of the picked-up block, it cannot accomplish the task reliably. Thus, it needs the directability over other agents to ask them to finish the delivery process. Meanwhile, due to its laziness, it requires a longer period of time for reordering, which can be remedied by other agents taking over this job. The predictability requirement, in this case, is to enable the other agents to know that the lazy agent will start reordering the blocks.

There are other OPD requirements that are not explicitly shown in the IA tables. Whereas, we have added them to add efficiency to the agents. In order to avoid collision of tasks between agents, we have added the predictability requirement to inform the agents which rooms will be explored by the team members. This is done to avoid, for example, two agents exploring the same room, which might lead to a waste of searching power.

References

[Mayer *et al.*, 1995] Roger C. Mayer, James H. Davis, and F. David Schoorman. An integrative model of organizational trust. *The Academy of Management Review*, 20:709, 1995.

[Weiss, 2013] Gerhard Weiss. *Multiagent systems*. Cambridge ; London The Mit Press, 2013.

B Default Parameters

- SwarmLab, source code available at GitHub.

```
{
    'inter_agent_distance': 10,
    'agent_speed': 6,
    'sim_end_time': 100
}
```

- Delta formation swarm, source code available at GitHub.

```
{
    'mu' = sqrt(2)
    'omega' = 0.0997356
    'd_0': sqrt(2)
}
```

- BW4T, source code available on request due to copyright constraints.

```
{
    'deadline': 1000,
    'tick_duration': 0.0,
    'random_seed': 5,
    'room_size': (6, 6),
    'nr_rooms': 6,
    'rooms_per_row': 3,
    'average_blocks_per_room': 4,
    'block_shapes': [0, 1, 2],
    'block_colors': ['#0008ff', '#ff1500', '#0dff00'],
    'room_colors': ['#0008ff', '#ff1500', '#0dff00'],
    'wall_color': "#8a8a8a",
    'drop_off_color': "#878787",
    'block_size': 0.5,
    'nr_drop_zones': 1,
    'nr_blocks_needed': 3,
    'hallway_space': 2,
    'agent_sense_range': 2,
    'block_sense_range': 1,
    'other_sense_range': np.inf,
    'agent_memory_decay': 5,
    'fov_occlusion': True
}
```