Trusting triangles in a world of squares

Neil Van Acoleyen[1] , **Jeroen Bechtold**[1] , **Bob Brockbernd**[1] , **Jan Willem Eriks**[1]
n.h.vanacoleyen@student.tudelft.nl, j.w.j.bechtold@student.tudelft.nl, b.j.brockbernd@student.tudelft.nl, j.w.a.eriks@student.tudelft.nl

Abstract

In the world of collaboration not every party can be trusted on their competence and willingness. That is why a collaboration framework which creates trust between the parties is required. This paper describes the collaboration framework which was developed during this project. Wherefore, four different agents were developed with different skills and weaknesses. To overcome the weaknesses and combine strengths the agents were developed using coactive design principles. The agents communicate their findings and actions to finish the task in a coordinated way, but the agents also validate received messages by their own experiences. If an agent concludes that a message is false, it will communicate this as well. Based on these messages the agents build a reputation and can alter their behavior if they do or do not believe what another agent has messaged. The results of this project show that the more agents the faster the task is completed. From this it can be concluded that the agents do collaborate efficiently even with crippled or untrustworthy agents.

1 Introduction

This paper describes our collaborative agents developed for the Collaborative Artificial Intelligence course of the CSE bachelors programme at TU Delft. The goal is to implement 4 agents with specific constraints and design a collaboration framework based on trust and reputation to overcome the agents limitations. The agents have to complete a common task in the joint-activity environment MATRX Blocks[Team.,] World for Teams.[Pasman and Murukannaia,] This task is to explore a world of rooms and move specific blocks to a drop off location.

The four agents and their unique properties are: Strong, this agent can carry two blocks at a time. Colour blind, this agent is not able to see the colours of the objective blocks. Liar sends false messages bout 80% of the time. And lazy quits an announced task bout 50% of the time.

In section 2 the coactive design process is described, section 3 shows the implementation of each of the before mentioned agents, the framework for communication and coordination is discussed in section 4, the trust and reputation mechanism is described in section 5, in section 6 the results are discussed and finally in section 7 we will conclude this paper.

2 Coactive Design

Mapping out the inter-dependencies between agents is a key part of coactive design. [Johnson *et al.*, 2014] For this an IA table was built. It can be found in the project submission as an excel file. We designed our agents to be independently capable of completing the goal as the configuration of agents would vary. However some agents possess limitations making it difficult or impossible for them to operate independently. The colorblind agent is not able to find correct goal blocks with certainty. Lacking information, they can use messages sent by other agents to fill in the blanks. We decided to have the colorblind agent bring possible goal blocks next to the drop zone to have them checked by other agents with color vision. This way when agents would visit the drop zone they could "Find" these blocks and move them to the correct slot if needed. This design choice kept the colorblind agents as active as possible and not relying too much on findings made by other agents. Another detail that can be noticed from the IA table is the soft inter-dependency between all other agents for finding the goal blocks. Since all found goal blocks would be broadcast to the other agents, efficiency would be increased as this way blocks will be discovered faster.

3 Agents Description

The agents created for this paper were designed as state machines. At the start we created two base agents. The Liar and the Lazy agent. The base which was build for the Liar agent was then later used to develop the colorblind and the strong agent. So firstly this basis will be discussed and then in their respective subsections the agents specifics will be explained.

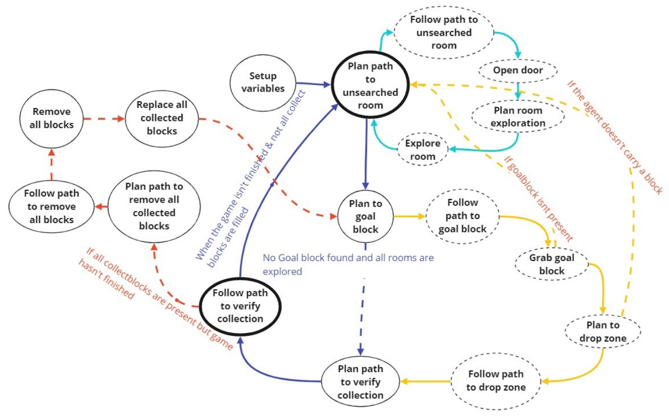
Follow path to unsearched room
Setup variables
Plan path to unsearched room
Open door
Remove all blocks
Replace all collected blocks
Plan room exploration
If the agent doesn't carry a block
Explore room
Follow path to remove all blocks
Plan path to remove all collected blocks
When the game isn't finished & not all collect blocks are filled
If goalblock isnt present
Plan to goal block
Follow path to goal block
No Goal block found and all rooms are explored
Grab goal block
If all collectblocks are present but game hasn't finished
Follow path to verify collection
Plan to drop zone
Plan path to verify collection
Follow path to drop zone

Figure 1: The basic state diagram of the Liar, Colorblind and Strong agent

The main state machine of the code decides which action to undertake. This state machine is located in *decide_on_bw4t_action* As can be seen in figure 1 the state diagram is quite complex. The main flow of the state diagram is shown with solid arrows. If something unusual happens the stripped transitions are used.

- The agent is initialized and the bookkeeping is setup.
- The main state `Plan path to unsearched room` decides whether a new room needs to be explore or a previously detected goal block needs to be picked up.
- When a room needs to be searched the light blue track is taken. Here the agent tries to efficiently move through an arbitrary size room. (As long as the *state.get_room_objects()* returns the correct room tiles.)
- When a goal block is found the yellow track is followed. In this track the agent goes to the goalblock. Which it gets by calling the function: *getBlockToGrab()*. Then it will make sure it is still present. Then pick it up and if it is actually picked up it will be returned to the collect location. Only the relevant goal blocks which are needed for the correct order are delivered and picked up. The other blocks are kept in memory for in the future.
- After successfully completing a block drop the agent will check the other collect points (*updateCollect()*) to check if there are already blocks present. If they are not all present yet the agent goes back to the main state to see which track it should take next.
- However if all blocks are present and the the game is not finished yet then someone dropped a wrong block. The red track is then followed. In this stage the agent goes to the last collect point in the sequence and stands left to it. It will then move all the blocks from the collect point to the left side of the point. Afterwards it will move to the next in the sequence and do the same. When all the blocks are returned then the agent will place back the goal blocks in the correct order. To make sure that when multiple agents are in this state that they do not interfere with each other there is a 90% chance to delay starting this sequence. This way there is a higher chance that the agents are more spaced out in their actions and that therefore one agent will finish this task successfully.

3.1 Color Blind agent

The color blind agent could be considered the most limited agent discussed in this paper. For it to complete the task on its own would be purely based on luck. This is because it can only distinguish the blocks based on their shape. But this agent can still be help full to other agents. It could move blocks that are the correct shape and bring them close to the collection point. Then the other agents can quickly pick up the required blocks from this point and deliver them to their collection zone. Also the colorblind agent can still try to find lies. Every lie that is not based on color (location, doors, etc.) is still detectable. The major shortcoming of the colorblind agent is that it can not complete a room by itself. Since it wont be able to align the goal blocks with the collect points. We decided to not try and brute force this since the course staff guaranteed that all rooms will be solvable. Brute forcing the goal blocks while other agents are more sufficient in aligning the correct blocks would result in messing with the correctly collected blocks and would therefore only be of negative impact.

3.2 Strong agent

The strong agent is unique in the sense that it can carry two different blocks. This can be helpful to be more efficient in delivering blocks. However for the rest this agent is nothing special. Except for in a few situations it just follows the basic principles of the base algorithm. One of these situations is when the strong agent detects the first following collect block after detecting the second one in the order. We decided to not instantly pickup the second block since that would stop others from dropping it of which could block the hole process in case that was the only available block. However when the first block is detected it will also pick up this second block and deliver both of them. To drop off both blocks a new state is introduced in the *decide_on_bw4t_action()* this state is called the *DOUBLE_DROPPOFF* state. It will only go to this state when to goalblocks are carried. So when a second one can not be found it will just deliver the first block.

3.3 Lazy agent

The lazy agent has been designed with a predetermined constraint. By chance it can quit performing the through messages announced tasks. First the difference in design between lazy and other agents will be described than we will go into more detail of the states of this agent.

The lazy agent is as well a state machine based agent. Where the design of this agent differs is that the state doesn't solely depend on the phase but on it's mode as well. Here the mode is either exploring or goal. And the phases are moving, opening doors, etc. The mode changes the actual behavior of the states corresponding to the phases. Also the low level implementation of the state mechanics is different. Where the other agents use a while loop, the lazy agent can chain states by using the `next` method and passing it the next phase. This call chain will than return an action once one of the states decides to make an action.

The initial states of this agent are:

- `init_vars`: This state initializes its datastructure based on the first provided game state.
- `what_to_do`: This state determines the mode of the agent. If there is a known deliverable goal block it will try to deliver that block to the drop off location. Otherwise it will explore. It sets the destination for the next states.

The order of the exploration states are:

- `calculate`: Calculates the path to destination set by `what_to_do`. In this case a room door.
- `moving`: Moves to destination set by `what_to_do`. This state also always opens a door when it passes a door since it makes state management easier and the state of doors are not used in our strategy.
- `plan_room_explore`: Plans a path to cover all squares inside a room.
- `explore_room`: Follows the planned path. And check blocks in its neighborhood. It also keeps track of the squares where it has been, to be able to validate findings of other agents. Once the room has been fully explored the next state will be `what_to_do`.

The order of the goal states are (calculate and move states are reused):

- `calculate`: calculates path to goal block.
- `moving`: moves to goal block.
- `pickup_block`: picks up block.
- `calculate`: calculates path to drop off zone.
- `moving`: moves to drop off zone.
- `drop_block`: drops block at drop off zone. The next state is `what_to_do`.

Every tick the `what_am_i_doing` method is called which. This method validates its current state based on the latest known game state (messages and observations). When the agent is moving to or carrying a goal block but another trusted agent drops a similar goal block in the mean time this method will force the agent to stop and reevaluate its next move. This method also contains a chance to quit the exploration mode by chance, which simulates lazy behavior.

3.4 Liar agent

The liar agent is a fully functional agent. However it has the tendency to to lie about its actions about 80% of the time. This chance is calculated by the function *toLieOrNotToLieZetsTheKwestion()* The lying of the agent is made to be as believable as possible. For example when the liar finds a goal block it needs to send a message that it found one. But if it then sends a message containing a block which is not even possibly a goal block then the lies are picked up fast. Therefore the liar tries to generate lies as realistic as possible. For the Previous example it will send a message with a different visualization. If there is none available then it will lie about the location of the goal block. All messages are generated in such a way so that all lies could be the truth. One way to pick out the liar is to check on whether the located goal blocks are in the same room as the liar said that it would search. Since the lies do not propagate (so a lie of going to room 1 will not result in a lie of opening the door of room one). We decided to do this since otherwise it would be Even though the Liar can not be trusted on its word it will still contribute to the objective by still doing the best possible action for the team. It will however inform its teammates incorrectly.

4 Communication and Coordination

4.1 Communication

For the communication system we decided to go as barebones as possible. This choice was made to not be to dependent on optional messages which would not be supported by external agents. Secondly lying would be harder and depending on the custom message even impossible. Take the last reason for example. A possibility with these new messages would have been to assign the tasks better, for example by having a leader and followers. However since there is no guarantee that all agents will be present during the evaluating nor are there any guarantees for how many agents are available. Which as a result would still entail that all agents must be able to complete the room without other help. And therefore since they all can function on their own a leader system is not needed. Due to the previous reasons we decided to not create any new/unnecessary messages. We did however add three messages to the base protocol.

1. ”Found block by **agent-id** approved”
2. ”Found block by **agent-id** denied”
3. ”Trust belief of **agent-id** : **trust-score-of-agent**”

These first two messages are send as a response to an agent declaring that they found a block. When the receiving agent confirms that this is indeed true then they send the approved message. If they know the block is a lie they send the denied message. And lastly an agent can declare their trust believe over another agent by sending the third message.

4.2 Coordination

As mentioned before, the agents were designed to be able to complete the goal on their own. Nonetheless, this does not restrict them of having coordination. With the help of the ”Found goal block” messages agents can form a global layout of the map and by sending ”Picking up goal block” messages agents know that that goal block is taking care of. Our agents can thus perform the goal independently but also when in a group of competent agents, coordinate with others to not do the same task twice.

5 Trust and Reputation

Trust and Reputation mechanisms allow agents to evaluate how trustworthy another agent is. This trust belief can be derived from 3 different sources: 1. Direct Experiences, 2. Indirect Experiences and 3. Reputation. These trust sources will be discussed in the following subsections in the same order followed by an explanation on how we compute the trust belief based on the sources.

5.1 Direct Experiences

A direct experience is a piece of trust information that an agent can gather on their own. In our agents this takes 2 different forms. The first one is verifying if a received message from an agent follows logically from the previous received messages from that same agent. For example, if an agent claims to be searching a specific room the trust mechanism checks if that agent had sent a message stating that it was moving to that specific room in the first place. This logical succession of messages can be applied to the majority of the messages sent. This method was designed to spot out the liars among the agents. Since these agents were designed to lie 80% of the time, they would not present a logical succession of actions in their messages.
The second form of direct experience our agents possess is the ability to verify if found goal blocks by others correspond with the goal blocks that the agent found itself. Our agents keep track of every goal block they found and whenever another agent sends a message stating that it found a goal block, they check their history to see if this matches with their findings. It is possible that an agent has not explored that goal block yet, in that case nothing can be said about the reliability of the information source.

5.2 Indirect Experiences

Indirect experiences are when other agents share their direct experiences. Taking into account the 2 types of direct experiences the agents possess, it is useless to broadcast the results of the verification of the logical succession of messages. Every agent has the same trust mechanism and thus already has computed this information for itself. However the other type of direct experience is useful to broadcast. While one agent could not be able to confirm the found goal block by some random agent, another agent could. Whenever an agent can approve or deny a found goal block, it sends that information to all the other agents to inform them of this finding. This allows agents to also get some sense of the reliability of the message even if they cannot confirm this by themselves.

5.3 Reputation

Reputation is information on how other agents perceive their team members. This essentially comes down to sending a message containing their trust belief in a specific agent. However a decision had to be made for when this reputation broadcast would happen. We opted to use as trigger every time that an agent were to send a "Picking up goal block" message. At that point, all other agents would send their trust belief about that agent to form a reputation on it and determine if it can be trusted with that goal block. This way, reputation information would not be sent too often but would still be there when it was needed.

5.4 Trust Belief

Using the information gathered via direct experience, indirect experience and reputation, a trust belief in other agents can be formed. In order to achieve this, metrics had to be used to quantify the information gathered. All trust processing can be found in _trustBelief(). For direct and indirect experience we set the default for both at 0, then a number would be subtracted or added depending on if the message received was trustworthy or not. We opted to subtract 0.4 in case of an untrustworthy message and add 0.1 in case of a trustworthy message. The reason for these values is that we considered a lie or inconsistency to have a bigger impact on trust. If an agent were to be trustworthy it should not make many lies or inconsistencies and would be able to recuperate from one by given trustworthy information afterwards. On the other hand, agents that lie often or that are generally inconsistent would dive deep into the negative values marking them as untrustworthy. Reputation works in a slightly different way. The default is also 0 but when new reputation information comes in, this incoming trust belief is then averaged with the current reputation value. These three values are combined to form the final trust belief. Direct experience is the most reliable source of information since it is computed by the agent itself, in contrast to indirect experience and reputation which come from other agents. Therefore we weighted the values as follows: $trustBelief = (3 \cdot directExperience + indirectExperience + reputation)/5$ (performed by _computeTrustBelief())

Trust based decisions can now be performed on the basis of this trust belief. If the value is positive, the agent can be trusted and if it is negative the agent is considered untrustworthy. This boolean trust decision is used by our agents when another agent sends a message that it has picked up a goal block. If this other agent is trusted, our agents trust that the goal block will be brought to the collect points. However if it is not trusted, then our agents do not mark that goal block as picked up and might come back later if need be to pick it up if it is still there.

6 Result Analysis

6.1 Initial testing

During the creation of our agents we first started by implementing a base agent and made sure that they are able to complete the whole world by themselves. For this we tested on the basic world with the basic settings. (Seed = 1, room-size = (6,4) , nr-rooms = 9, rooms-per-row = 3, block-per-room = 3) We kept on developing until the worlds could be finished. Later when we started developing specialized agents we made sure that they also could complete this basic room themselves and in combination with some other agents. After the agent was able to finish the basic world themselves we started to experiment with the world settings to make sure that the agents would also function in different worlds. Most bugs were ironed out in this phase.

6.2 Combining agents

After all the agents were created we tried to run the simulations of 100 different worlds. At first some worlds would not result in a success, we then check whether the world was even possible to succeed. If this was the case we ironed out the issues that arose during the testing. However some rooms were uncompletable due to the lack of goal blocks. Therefore we tried to optimize the world generation settings to find some general settings which almost always generates completable rooms. We ended up with the following settings: *(Seed = between 1 and 100 , block-per-room = 2 or 3, room-size = (6,4), nr-rooms = 40/block-per-room up to 70/block-per-room, rooms-per-row = 9/block-per-room up to 20/block-per-room)* With a 100 runs over these worlds with 2 agents of each type we had a 100% success rate. The average ticks needed to complete the world was 319. We did more of such tests and the result can be found in the table below. Note that for all these tests the previously described world settings were used.

Agents	success/total	avg. tick
2x (Color,Liar,lazy,Strong)	100/100	319
1x (Color,Liar,lazy,Strong)	30/30	478
Liar, Lazy, Strong	30/30	508
Color, Lazy, Strong	30/30	796
Color, Liar, Strong	30/30	353
Color, Liar, Lazy	30/30	559
Color, Liar	30/30	568
Color, Lazy	25/30	1147
Color, Strong	30/30	817
Liar, Lazy	30/30	534
Liar, Strong	30/30	520
Lazy, Strong	30/30	798
Color	0/30	3000
Liar	30/30	583
Lazy	21/30	1361
Strong	30/30	926

As can be seen in the table above the average tick completion time goes down the more agents are used to complete the room. This shows that the agents do help each other and make the whole system more efficient. There is one other difference which can be found. This is for when the liar is not in the game the average tick will go up. This happens because the liar will search the rooms closest to the collect point first while the other agents pick random rooms.

An other observation to be made is that teams with the lazy agent are on average slower than teams without this agent. This happens because the lazy agent is less efficient in itself than other agents. Because it sometimes stops during an action. However he still did some actions towards the previously stated goal which were gone to waste. Due to this inefficient nature the lazy agent sometimes wont even be able to complete the world in time. This can be clearly seen in the success rate of the solo Lazy agent and the Lazy agent combined with the color agent. However the success rate does increase when the lazy agent is accompanied by the color agent. Which shows that combining agents result in a more efficient solving of the world. And that helping agents and with their inefficiencies results in a better overall result.

One of the shortcomings of these tests is that since the worlds are random the test were not identical. To get a better average more tests should be run. However due to time constraints this could not be done.

6.3 Shortcomings

One of the shortcomings of our agents is that even though they are programmed to work with larger rooms the standard function *state.get_room_objects(room_name)* provided by the framework and used by the agent does not support rooms of different sizes that (6,4) Therefore our agent may not be able to finish a world where the room sizes differ from the usual size. If this *state.get_room_objects(room_name)* is adapted to support different sizes than our agent should be able to handle them as well. Another possible improvement would be to start at a random location and the explore all the rooms close by and then deliver the goal blocks found. This could majorly improve the average tick since less time would be wasted by traveling. On top of that all liar agents currently follow the same path. Which would make having multiple liars not that beneficial. If there were guarantees on which agents would always be present then a master slave construction would be able to greatly improve the efficiency of the world completion. However we had no such guarantees so therefor we did not implement this. But it could be a great and challenging improvement to make.

7 Conclusion

The agents implemented in this report have shown the ability to work together. Even when they are not sure who to trust and who to distrust. Through a process of direct and indirect experiences they build an internal trust model. With the ability to cooperate they can strengthen and complement each others weaknesses. Both the Colorblind and Lazy agent have success score that is relatively low (0/30 and 21/30). But combined they achieve a score of 25/30. Although many agents are able to successfully complete the task withing the required amount of ticks, they will perform these tasks in less ticks when coupled with other agents. This means they can successfully work together on achieving a common goal and help each other.

References

[Johnson *et al.*, 2014] Matthew Johnson, Jeffrey M. Bradshaw, Paul J. Feltovich, Catholijn M. Jonker, M. Birna van Riemsdijk, and Maarten Sierhuis. Coactive design: Designing support for interdependence in joint activity. *J. Hum.-Robot Interact.*, 3(1):43–69, feb 2014.

[Pasman and Murukannaia,] Wouter Pasman and Pradeep K. Murukannaia. Collaborative ai (cse3210): Bw4t-matrx wik.

[Team.,] MATRX Team. Matrx: Accelerating human-agent teaming research together.