university of groningen / faculty of science and engineering

# An agent-based model of the game "The Mind"

Bachelor's Project Thesis

Daan Windt, s3486427, d.windt@student.rug.nl
Supervisor: Dr H.A. de Weerd

**Abstract:** People cooperate in many ways, often without communicating beforehand. When we pass each other on a sidewalk, we do not have to talk to do so. This is an example of a game theory concept called a coordination game, where two or more agents have to coordinate their choices to cooperate on a shared task. "The Mind", a game released by White Goblin Games, transfers this concept into a game that a group of two to four people can play. Unique numbered cards are divided between the players, those that are not dealt are discarded. The players have to put down the cards in order from lowest to highest as a team, without communicating. They can put down a card at any time. A model was created of a society of agents forming groups to play "The Mind", where each agent has an initially random "response time" variable that determines how long they wait to play a card, based on the card's value. It is adjusted when the wrong card is played. The society as a whole becomes better at cooperating and wins more the longer the model is run, and the response times converge, though not completely.

## 1 Introduction

Societies all over the world can exist thanks to agreed-upon rules. One familiar kind of societal rule are laws, which are imposed upon society by a select group of individuals and enforced by certain assigned individuals. Another kind of societal rule are social norms, which are set and enforced by society at large, without the need for a central authority (Axelrod (1986)). We all learn that it is not polite to stare at people, that we should not skip the line when we are in a queue, and that we should stay on the right when passing each other on the sidewalk.

In the last example, there is no practical difference between everyone passing each other on the left or the right: when countries change the side of the road they drive on, economic concerns and the way vehicles are built weigh heavier than the inherent safety of one side over the other (Bari (2014)). What matters is that everyone follows the same rule. This is so important when it comes to motor vehicles, that governments pass laws that mandate driving on a certain side of the road, even though they disagree on which side of the road it should be. This is an example of a coordination game, which has a payoff matrix like table 1.1. Coordination games have been used before to identify where social norms lie (Krupka and Weber (2013)).

**Table 1.1: Payoff matrix for which side of the road to drive on**

| | r | l |
|---|---|---|
| r | 1, 1 | 0, 0 |
| l | 0, 0 | 1, 1 |

However, there are also coordination games where not all Nash equilibria are Pareto efficient: on a busy road, any speed that drivers can agree upon would technically be a Nash equilibrium, but the Pareto efficient speed would be the fastest speed that the drivers can safely drive. This situation is also more complicated because the drivers' responses are continuous variables, and the payoff for every driver is continuous as well: if one driver goes 79 kph while everyone else drives 80 kph, the effect is minimal and increases the more different drivers' speeds are.

The Mind is a card game published by White Goblin Games, which has all the aforementioned characteristics. In a simplified form, it consists of a group of players being dealt cards from a shuffled deck of the numbers 1-100 all appearing once. Each player gets the same amount of unique cards, and the cards that are not dealt are set aside. The players then have to put down their cards in order from lowest to highest, without being allowed to

communicate or see each other's cards. Players do not take turns, therefore they have to be able to estimate whether they have the lowest card in the game. To do this, they have to synchronise the time they take to play a card with a specific number, by adjusting to their teammates' response times. If a player puts their card down too quickly, everyone discards all cards lower than the one played and the game continues. If this happens during a game, for the purposes of this paper the game is considered lost, while the game is won if everyone manages to put down their cards.

This paper presents a model of a community of agents playing this simplified version of The Mind. Agents form teams that play the game, adjusting to their teammates' response times and afterwards forming different teams where they adjust to those teammates' response times, and so on. The players need to adjust their response times, a continuous variable, to be as similar to each other as possible, just like drivers need to coordinate their speed on a road. The amount of players can vary like in the other example. The payoff of a single game is either a win or a loss, but over multiple games a particular response time will result in a continuous win percentage, just like the payoff of the driving speed on a road. However, since The Mind has very simple rules, any results of studies are clearer than in a real world scenario, thus being on a convenient level of complexity between a payoff matrix and real life situations.

# 2 System description

## 2.1 General model description

The model was made in Netlogo, using the x64 OpenJDK binaries bundled with the program on Windows 8.1 and Windows 10 2004. Versions 6.1.1, 6.2.0 and 6.2.1 were used for building the model and version 6.2.1 for testing, which was done on Windows 10 2004. Netlogo allows for an interface of the model consisting of parameters that can be adjusted with sliders and buttons, and graphs that are automatically generated to show the values of useful variables.

The model world itself consists entirely of agents, of which there are two types: patches, which are stationary agents that form the background and have no impact on this model other that visibility for the user, and turtles, which are agents that can move around freely. For the rest of this paper, the word "agent" is used to refer to turtles specifically. The model is updated every tick: the tick speed of the model can be adjusted as a standard feature of Netlogo.

The agents in the model world form groups to play a game. This is more similar to real life, where people do not always play in the same group. There is a set amount of cards each player can get, which is a parameter that can be adjusted, because there could be differences in the behavior of the model based on the amount of cards that are dealt: the more cards are divided, the longer the game will go on for, and the more chances there will be for players to adjust their response times to each other. This may lead to more regional variation and more extreme fluctuations around a stable response time when it is reached. Groups have to be small enough to give every player that amount of cards, which is enforced by the model.

Once a group is formed and the cards are dealt, each player starts a countdown of a whole number of ticks, based on their response time and the value of their lowest card. If multiple agents' countdowns end at the same time, the tie is broken randomly. The agents could have also just compared each others response times and have the player with the lowest response time play their card. While this would have sped up the model considerably, this would eliminate the factor that games with more cards in play, due to a different amount of players, may last a different amount of time, which would impact the formation of groups that play the next game, etc. This would also be less realistic, since this would technically require the agents to communicate, even though this would not change the outcome of one single game. The current model implements a strategy that a group of human players might implement.

When one of the players' countdown reaches zero, it plays its lowest card, according to the rules of The Mind. All other players discard any cards that are of a lower value: in case this happens, the game is considered lost, but is still played to the end. This is a simplification of how the real game handles losses, but the simulated version does not need to be fun to play like the original. A more simple fail condition is easier to compute and still preserves the behavior

of players adjusting to each others' response times.

All players that should have played their card earlier lower their response times, while the player that played its card too early raises its response time. The amounts by which these values are adjusted are parameters. These parameters need careful adjustment: the model might have a tendency for response times to move upwards or downwards if one of those events happens more frequently. The more these parameters are adjusted by though, the more the players' response times will fluctuate.

Then all players reset their timer, again based on their response time and their lowest card, subtracted by the value of the card that was played last. The game continues until all players have played their cards. This is to keep the length of games more consistent, so no one game has an outsized influence on the agents' behavior.

When the game is over, the players set a randomized wait timer, during which they cannot start a new group, but can still join one being formed by another agent. This wait timer can be disabled, in which case the agents have a tendency to immediately form the same group again, since all players around them will still be playing their own game. Once this timer runs out, the agents can form groups on their own again, and a new game starts if they have not joined another group.

This concludes a general overview of the model. The following subparagraphs will discuss all aspects of the model in more detail.

## 2.2 Interface description

### 2.2.1 Left

The interface of the model consists of a left, middle and right part. The left part consists of buttons that control functions, and sliders and an input field that control parameters: see figure 2.1. Clicking the "randomize" button sets *current-seed* to a new randomly generated seed. Clicking the "setup" button executes the function SETUP once. Clicking the "go" button executes the function GO every tick until the button is pressed again.

The parameter *lower-when-too-late* determines how much an agent lowers its response time when it is too late to prevent another agent playing a card higher than its lowest card. It can be any integer between and including 0 to 10. Negative values would cause agents to double down when they make a mistake, and higher values than 10 were not practically necessary for studying the agents' behaviour, though there is no theoretical reason why it could not be higher.

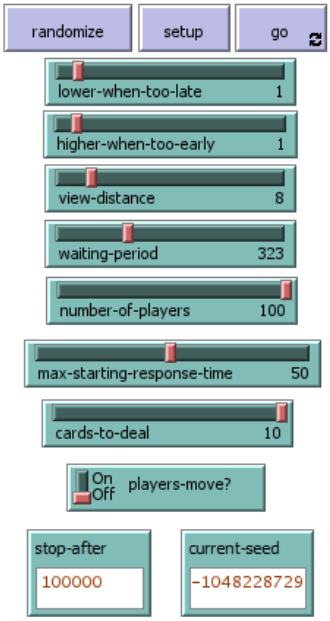


**Figure 2.1: The left side of the model interface, containing the function and parameter controls.**

*higher-when-too-early* determines how much the agents raise their response times when they play their card too early, and is also limited between and including 0 and 10 for the same reason.

*view-distance* determines from how far a game leader can see other agents to form its group with. It can be between 0 and 100, since any number below 0 would be the same as 0. The maximum value of 50 was chosen because the model world has a size of 32 by 32 patches. The maximum distance an agent could see would be $\sqrt{(32^2 + 32^2)} \approx 45.25$ patches, so 50 is a convenient round number to use.

*waiting-period* determines the amount of ticks that agents wait after a game is finished until they try to start another one. This is done to prevent the same group of agents playing over and over again because all agents around them are already in a different game. The minimum value is 0 because values below 0 would not make a difference, the maximum value of 1000 has no specific reason.

*number-of-players* determines the number of players that is created by the SETUP function. It has a minimum of 0 and a maximum of 100, since with that many agents the model world is already covered by them.

*max-starting-response-time* determines agents' maximum response time when they are created. It ranges between and including 1 and 100. For the model to work, larger values than 100 are unnecessary, but there is no reason that they could not be used. Values lower than 1 cause the model to crash, so 1 is the minimum.

*cards-to-deal* determines the amount of cards dealt to each player in a game. The minimum is 1, which is always needed to play the game, and the maximum is 10. A larger maximum would theoretically be possible.

*players-move?* is a Boolean that determines whether the players always move forward.

*current-seed* is a built-in variable of Netlogo that contains the seed for the random number generator.

*stop-after* is the number of ticks after which the model stops automatically.

### 2.2.2 Middle

The middle part of the interface only contains the model world and the output field, see figure 2.2. The model world consists of a background made out of patches and agents that can have various shapes and colors. The agents can have any default Netlogo shape, which indicates which group they are playing in. Shape 0 in the built-in list *shapes* is only used by agents that are not part of a group.

Agents' color is determined by the function SETCOLOR. The higher an agent's response time is compared to *max-starting-response-time*, the brighter the agent's color is.

As a standard Netlogo feature, the world can be set to wrap around, so agents that exit on the left enter again on the right, etc. This also results in distance functions returning the smallest possible travel distance between two agents, accounting of the ability for agents to wrap around the world. This setting is recommended to be turned on for both horizontal and vertical movement when *players-move* is set to *true*, since this causes the agents to move in a straight line and would cause agents to get stuck on the borders of the world otherwise.

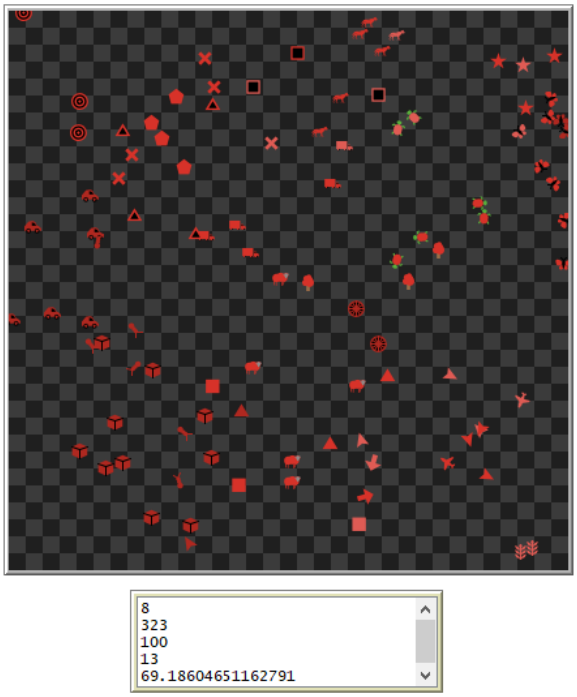


**Figure 2.2: The middle of the interface, containing the model world and output field.**

The output field outputs the view distance, waiting period, number of players, the delta between the highest and lowest response times, and the win percentage in a way that is easy to copy.

### 2.2.3 Right

The right side of the model interface contains monitors for various variables. The "player response times" windows displays the range between the highest and lowest values of the *response-time* variable of all players. Each tick the *response-time* variable of a random agent is plotted, and since there are many ticks per second, all agents end up having their *response-time* variable plotted eventually, resulting in this graph. The axes are initially set to 0-10 for the x-axis and 0-5 for the y-axis, but these default values are only seen before the start of the model since the axes are automatically scaled.

The "win%" plot tracks the percentage of games that has been won during this run of the model. The formula used to calculate it is $wins/(wins + losses) * 100$, where *wins* and losses are both variables of the model. The y-axis is always between

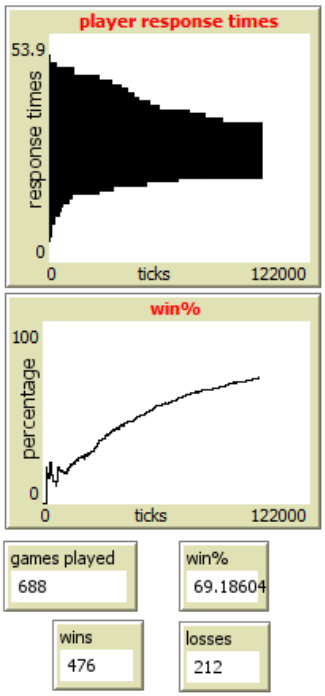


**Figure 2.3: The right side of the model interface, displaying the values of various variables.**

0 and 100 and the x-axis starts between 0 and 10. Both scale automatically, but since the win percentage is never above 100 the y-axis does not change.

The "games played" monitor displays the value of $wins + losses$, the "win%" monitor presents the value used in the "win%" plot numerically and the "wins" and "losses" monitors show the values of their respective variables.

## 2.3 Variable descriptions

This section will describe all variables that are used in this model, starting with the global variables:

*wins* contains a count of all games that have been played by any agents in the model that resulted in a win.

*losses* contains a count of all games that have been played by any agents in the model that resulted in a loss.

*current-shape* contains the index in the built-in *shapes* list of the shape that the next group of agents that will play a game will take.

*players* is a list of all agents in the model.

The following variables are agent-specific:

*cards* is a list of all cards that this agent is currently holding.

*response-time* contains this agent's current response time, which is multiplied by its lowest card subtracted by the highest card played to result in the time this player will wait to play its lowest card.

*time-left* contains the amount of ticks that this agent will still wait to play its current lowest card.

*is-winning?* is a Boolean that contains whether the team playing the game is currently winning, meaning that no agent has played a card it should not have played.

*card-played* contains the last card that has been played by an agent in this group. This variable is only used by the game leader.

*game-leader* contains the current game leader for this group.

*in-game?* is a Boolean that contains whether this agent is currently in a game.

*team* is a list that contains all members of the team this agent is in, including the agent itself.

*time-to-wait* contains the amount of ticks left for this agent to wait to try to form a new team, when the agent is not currently in a game.

All other variables are local to the function they are declared in.

## 2.4 Function descriptions

To run the model, the first step is to set the random seed. This can be done either by filling in the "current-seed" field or by clicking the "randomize" button, which uses the NEW-SEED function built into Netlogo. The next step is to click the "setup" button, which executes the function with the same name. After this function has set up the game, the "go" button can be clicked, which executes the GO function every tick. For all loops are used for sets of agents. In this case variables mentioned in that loop are presumed to belong to agents from the set, except when the variable has been mentioned before in that function. Cases where this is ambiguous are clarified.

The pseudocode for SETUP is contained in algorithm 2.1. Lines 1 and 2 ensure that remnants of previous runs of the model are removed. Lines 8-14 make the background, which is the only purpose patches are used for in this model. Lines 15-24 create the agents: these lines would be combined in

**Algorithm 2.1** SETUP

**Require:** *current-seed* is set.
**Ensure:** Model is set up for GO to run.
1: Reset the model.
2: Reset the tick counter.
3: SET-SEED
4: $wins \Leftarrow 0$
5: $losses \Leftarrow 0$
6: *current-shape*$\Leftarrow 1$
7: $wins \Leftarrow 0$
8: **for all** *patches* **do**
9: **if** $pxcor\%2 = pycor\%2$ **then**
10: Color the patch dark grey.
11: **else**
12: Color the patch light grey.
13: **end if**
14: **end for**
15: Create *number-of-players* players.
16: **for all** *players* **do**
17: Randomize position.
18: *shape* $\Leftarrow$ 0th item of *shapes*
19: *response-time* $\Leftarrow$ Random number between 5 and *max-starting-response-time*
20: *in-game?* $\Leftarrow false$
21: SET-COLOR
22: *cards* $\Leftarrow \emptyset$
23: *time-to-wait* $\Leftarrow 0$
24: **end for**
25: **for all** *players* **do**
26: CREATE-GAME
27: **end for**

one function call in Netlogo. Lines 25-27 make all players attempt to start a game with their neighbors. *pxcor* and *pycor* are the x and y coordinates of a patch.

GO is performed every tick if the "go" button is pressed. It updates all agents, then ends the current tick. The next tick starts based on the model speed. See algorithm 2.2.

TICK-PLAYER checks if the agent is currently in a game. If not, it moves if it is supposed to, then tries to form a team if it is allowed to. See algorithm 2.3.

When CREATE-GAME is called, the agent looks for nearby agents who are not already in a game, then starts a game if teammates are available. See algorithm 2.4.

When FIND-TEAMMATES is called, the agent

**Algorithm 2.2** GO

**Require:** SETUP has been executed at least once.
**Ensure:** Update the model each tick.
1: **if** *ticks* = *stop-after* **then**
2: Clear the output window.
3: Print *view-distance*.
4: Print *waiting-period*.
5: Print *number-of-players*.
6: Print difference between maximum and minimum *response-time* of agents.
7: Print $wins/(wins + losses) * 100$
8: Stop the model.
9: **end if**
10: **for all** *players* **do**
11: TICK-PLAYER
12: **end for**
13: End the tick.

**Algorithm 2.3** TICK-PLAYER

**Require:** SETUP has been executed at least once.
**Ensure:** Agent takes appropriate action based on its current status.
1: **if** *in-game?* **then**
2: PLAY-GAME
3: **else**
4: *time-to-wait* $\Leftarrow$ *time-to-wait*$-1$
5: **if** *players-move?* **then**
6: Move forward one patch in the agent's current direction.
7: **end if**
8: **if** *time-to-wait*$<= 0$ **then**
9: CREATE-GAME
10: **end if**
11: **end if**

**Algorithm 2.4** CREATE-GAME

**Require:** SETUP has been executed at least once.
**Ensure:** Agent starts a game if it is able to.
1: **if** !*in-game?* **then**
2: FIND-TEAMMATES
3: **if** FORM-TEAM **then**
4: SETUP-GAME
5: PLAY-GAME
6: **end if**
7: **end if**

**Algorithm 2.5** FIND-TEAMMATES

**Require:** SETUP has been executed at least once. Agent is not in a game.
**Ensure:** Agent finds potential teammates if available.

```
1: team ⇐ all players within view-distance that are not in a game
2: team-size ⇐ size of team
3: maximum-team-size ⇐ ⌊100/cards-to-deal⌋
4: team ⇐ maximum-team-size members of team
```

**Algorithm 2.6** FORM-TEAM

**Require:** SETUP has been executed at least once. Agent is not in a game.
**Ensure:** Agent forms a team if it has found potential teammates. Return whether or not it succeeded.

```
1: if size of team > 1 then
2:    temp-shape ⇐ current-shape
3:    current-shape ⇐ current-shape+1
4:    if current-shape = size of shapes then
5:       current-shape ⇐ 1
6:    end if
7:    for all agents in team do
8:       in-game? ⇐ true
9:       game-leader ⇐ this agent
10:      team ⇐ team of this agent
11:      shape ⇐ temp-shapeth item of shapes
12:   end for
13:   return true
14: end if
15: return false
```

looks for all agents within *view-distance* that are not in a game, then creates as big a team as it can. The maximum team size is determined by the amount of cards each player is dealt, since each player needs exactly that amount. See algorithm 2.5.

When FORM-TEAM is called, the agent picks a shape for the team, then makes sure all members of the team share the variables they need for the game to start. See algorithm 2.6

SETUP-GAME shuffles the virtual deck of cards and deals *cards-to-deal* cards to each agent. It also initializes the agent's timer and allows it to keep track of whether the team is winning. See algorithm 2.7.

**Algorithm 2.7** SETUP-GAME

**Require:** This agent is a team leader.
**Ensure:** The game is ready to be played.

```
1: card-played ⇐ 0
2: shuffled-list ⇐ shuffled list of cards between and including 1-100
3: for all agents in team do
4:    cards ⇐ sorted list of cards-to-deal cards from shuffled-list
5:    shuffled-list ⇐ shuffled-list\cards
6:    SET-TIME-LEFT
7:    is-winning? ⇐ true
8: end for
```

**Algorithm 2.8** SET-TIME-LEFT

**Require:** This agent is in a game.
**Ensure:** The agent's timer is set correctly.

```
1: if length of cards = 0 then
2:    time-left ⇐ −1
3: else
4:    time-left ⇐ (lowest card value - card-played of game-leader) * response-time)
5: end if
```

SET-TIME-LEFT sets the timer of this agent to -1 if it has no cards and to the correct value based on its response time and the lowest card it has otherwise. See algorithm 2.8.

When PLAY-GAME is called, the agent first determines whether its timer for playing its lowest card is equal to 0. If so, it plays its card, on line 3. On lines 4-5, it checks if any other agents had to discard any cards, in which case it adjusts its response time appropriately. On lines 9-13, the agent checks if it should stop the game. On lines 14-16, if the game has not been stopped, the agent asks all team members to reset their timers based on the new highest card played and their new lowest cards. See algorithm 2.9.

COUNT-DOWN decrements an agent's timer and returns whether it has reached 0. See algorithm 2.10.

When DISCARD-CARDS is executed, the agent checks whether any other agents in this game have cards that they should have played earlier. If they do, they throw away those cards and adjust their response time. The function returns whether the card the agent played was the lowest one, so whether

**Algorithm 2.9** PLAY-GAME

**Require:** The game is set up completely.
**Ensure:** The agent plays the game.

```
1: if COUNT-DOWN and agent has cards left then
2:    card ⇐ lowest card in cards
3:    card-played of game-leader ⇐ card
4:    if DISCARD-CARDS then
5:       response-time ⇐ response-time + higher-
         when-too-early
6:       SET-COLOR
7:    end if
8:    remove card from cards
9:    if cards = ∅ then
10:      if STOP-GAME then
11:         return
12:      end if
13:   end if
14:   for all agents in team do
15:      SET-TIME-LEFT
16:   end for
17: end if
```

**Algorithm 2.10** COUNT-DOWN

**Require:** Agent is playing a game.
**Ensure:** Agent's timer is decremented and its status returned.

```
1: if time-left < 0 then
2:    time-left ⇐ time-left−1
3: end if
4: return time-left = 0
```

**Algorithm 2.11** DISCARD-CARDS

**Require:** Agent is playing a game.
**Ensure:** Agent discards its card and updates its response time if necessary.

```
1: is-lowest? ⇐ true
2: for all agents in team do
3:    lowered? ⇐ false
4:    while agent has cards left and agent has a
      lower card than the one played do
5:       for all agents in team do
6:          is-winning? ⇐ false
7:       end for
8:       is-lowest? ⇐ false
9:       remove lowest card from cards
10:      if !lowered? then
11:         response-time ⇐ max(response-time -
            lower-when-too-late)
12:         SET-COLOR
13:         lowered? ⇐ true
14:      end if
15:   end while
16: end for
17: return !is-lowest?
```

none of the other agents threw away any cards. See algorithm 2.11

STOP-GAME checks if the game should be stopped and does so if required, making sure that all agents are no longer in this game. The variable *should-stop?* is not strictly necessary in pseudocode form, since returns can be used to exit the for loop, but this is not possible in Netlogo. See algorithm 2.12.

SET-COLOR sets the color of an agent to a shade of red, with black being color 10 in Netlogo and bright red color 19. See algorithm 2.13.

SET-SEED only sets the seed used by the random number generator to *current-seed*. Since the specifics depend on the programming language for the only line in this function, pseudocode is omitted.

**Algorithm 2.12** STOP-GAME

**Require:** Agent is playing a game.
**Ensure:** Agent stops the game if appropriate.

```
1:  should-stop? ⇐ true
2:  for all agents in team do
3:    if agent has cards left then
4:      should-stop? ⇐ false
5:    end if
6:  end for
7:  if should-stop? then
8:    if is-winning? then
9:      wins ⇐ wins + 1
10:   else
11:     losses ⇐ losses + 1
12:   end if
13:   for all agents in team do
14:     in-game? ⇐ false
15:     time-to-wait ⇐ waiting-period−5+random number between 0 and 10
16:     shape ⇐ 0th item of shapes
17:   end for
18:   return true
19: end if
20: return false
```

**Algorithm 2.13** SET-COLOR

**Ensure:** Agent has the color appropriate for its response time.

```
1:  color ⇐ ⌊10+response-time/max-starting-response-time∗10⌋
```

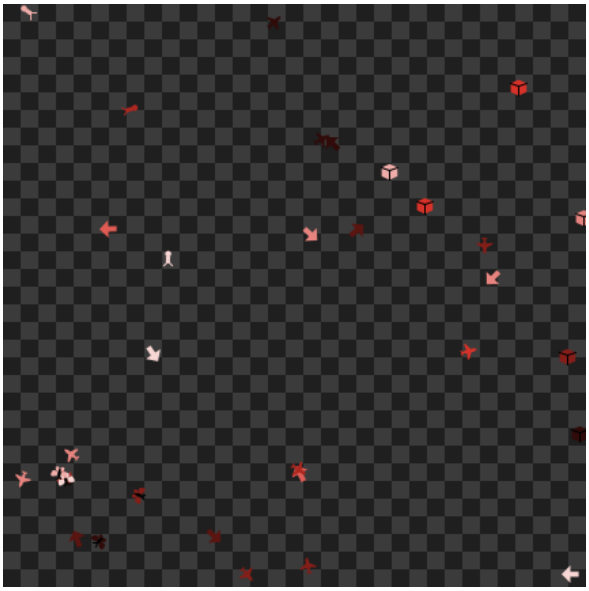

**Figure 3.1: The model world with 33 agents.**

# 3 Results

To show the behavior of the model under different circumstances, the view distance, waiting period and number of players were manipulated as independent variables, to show their effect on the delta between the highest and lowest response times and the win percentage. The view distance was set to 8, 25 or 42, the latter being so high that agents can see almost all other agents. The waiting period was 0, 150 or 300. The number of players was 33, 66 or 99. The model was run once with each configuration, totaling 27 runs of 200 000 ticks. The seed was kept the same at 1564099703, so the starting position of the agents would be the same. *lower-when-too-late* and *higher-when-too-early* were set to 1, the maximum starting response time to 50 ticks, 10 cards were dealt to each agent every game and no agents were able to move.

All experiments were run with the same seed, so the starting position of agents was the same as long as the amount of agents stayed the same: see figures 3.1, 3.2 and 3.3. The lower the amount of agents, the more the agents are clustered and isolated.

The graphs show the impact of the distance agents can view, the waiting period after each game for agents to try to start a new group and the number of agents.

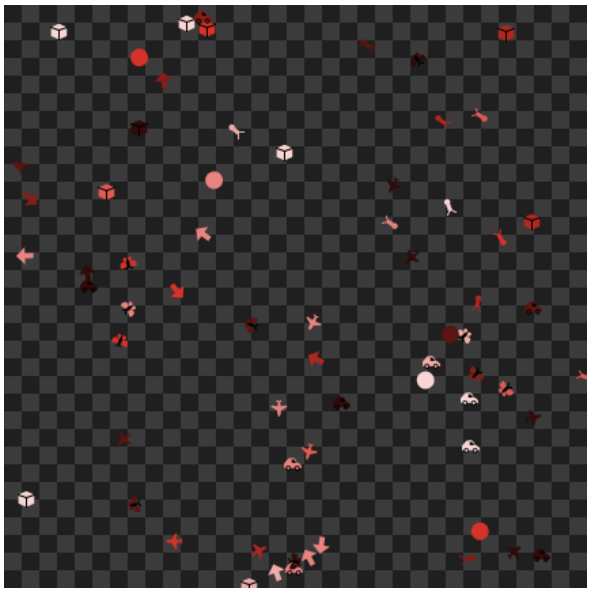

**Figure 3.2: The model world with 66 agents.**

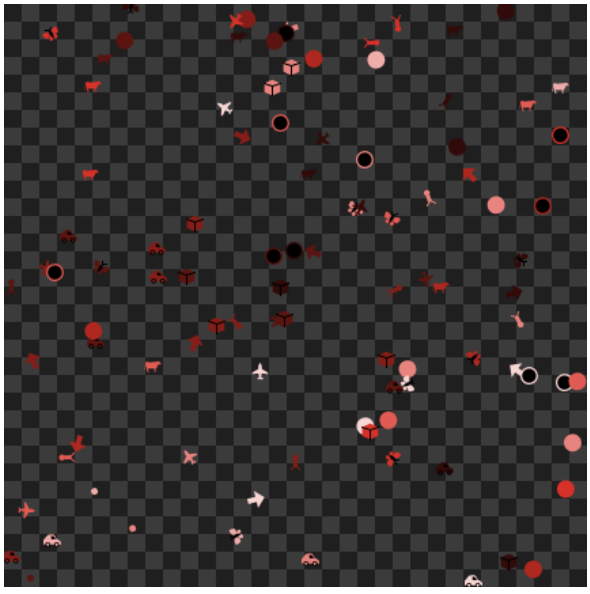

**Figure 3.3: The model world with 99 agents.**

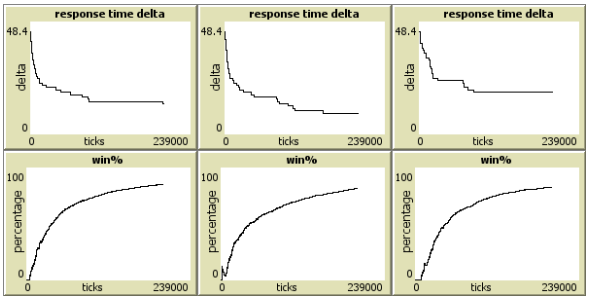


**Figure 3.4: Graphs showing the win percentage and delta between the lowest and highest response times of agents. The left pair corresponds to a viewing distance of 8 tiles, the middle 25, the right 42.**

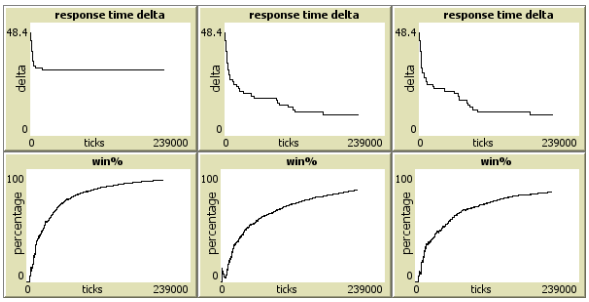


**Figure 3.5: Graphs showing the win percentage and delta between the lowest and highest response times of agents. The left pair corresponds to a waiting period of 0 ticks, the middle 150, the right 300.**

The distance agents can view did not seem to affect the percentage of games won. There was an increase in the difference between the lowest and highest response time for a high viewing distance. See figure 3.4.

The waiting period seems like it decreased both the percentage of games won and the difference between the lowest and highest response time, with the difference between 0 and 150 being especially big. See figure 3.5.

The amount of agents did not seem to affect the difference between the lowest and highest response time, but did increase the percentage of games won. See figure 3.6.

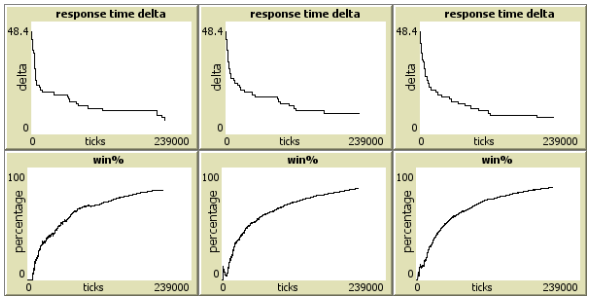


**Figure 3.6: Graphs showing the win percentage and delta between the lowest and highest response times of agents. The left pair corresponds to 33 agents in the model, the middle 66, the right 99.**

# 4 Conclusion

The agents in the model described in this paper are able to cooperate playing "The Mind" without being able to communicate with each other. However, the agents' response times never became the same across the board in any scenario, described in this paper or during development. This probably happened because agents with very similar, but not quite equal response times would almost always complete the game fully, so the pressure to change it was very low. The main reason however is that when one agent increases its response time, other agents decrease it et cetera. In a situation where both agents are one tick apart, this would lead them to still be one apart afterwards. This does not take away from the general trend in all runs of the model, which is that agents' response times get closer the longer the model is run.

An effect of a low number of agent can be that there are clusters of agents with no way of communicating between each other. This can cause a situation where the difference between the highest and lowest response time is high, but the percentage of games won is also high. As more agents are added, some can exist in between clusters, acting as a bridge. In figure 3.5 the top left graph shows that when the waiting period does not exist, the difference in response time between the slowest and fastest agents is large. This happens because as soon as a group is done playing the game, they instantly start playing the same game again with the same agents, because all other agents around are still in a different game. This means that bridging agents between clusters only play with one cluster. As seen in figure 3.2, there are no agents without a group, so there are hardly any opportunities for a different group to form. When a waiting period is instated, the difference between the slowest and fastest agent decreases, as seen in the middle and right graph in figure 3.5.

There are some parameters that could have been added to the model that reflect the way humans play this game: for example, agents' change to their reaction times could be randomized. These could be explored in a more detailed version of this model. The current model runs relatively slowly, so a change to a different programming language or optimizations to the code could help with collecting data, as well as a more compact way of storing information from the graphs, which now store one data point for all 200 000 ticks. These changes would allow a more thorough study of the behavior of this model, using many random seeds executed in batch processes to gather data. This model could also be applied to coordination games other than The Mind, so a remade, more efficient model could serve as a framework for studying thees games more effectively as well.

# References

Robert Axelrod. An evolutionary approach to norms. *The American Political Science Review*, 80(4):1095–1111, 1986.

Md Mahabubul Bari. The study of the possibility of switching driving side in rwanda. *European Transport Research Review*, 6(4):439–453, Dec 2014.

Erin L. Krupka and Roberto A. Weber. Identifying Social Norms Using Coordination Games: Why Does Dictator Game Sharing Vary? *Journal of the European Economic Association*, 11(3):495–524, Jan 2013.