Group performance and division of labour in an environment with different personalities

Lecture with Computer Exercises: Modelling and Simulating Social Systems

Project Report

João Dinis Sanches Ferreira, Miguel Pérez Sanchis
(*Iberia*)

ETH - Zürich
Department of Humanities, Social and Political Sciences

Agreement for free-download

We hereby agree to make our source code for this project freely available for download from the web pages of COSS. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

João Dinis Sanches Ferreira *Miguel Pérez Sanchis*

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

GROUP PERFORMANCE AND DIVISION OF LABOUR IN AN ENVIRONMENT WITH DIFFERENT PERSONALITIES

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Sanches Ferreira	João Dinis
Pérez Sanchis	Miguel

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 9/12/2018	João Ferreira
	~~Miguel PS~~

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Contents

1 Abstract

The potential of a group of people is often much more than the sum of its parts. A better understanding of human interactions can therefore have a great impact in the productivity and motivation of people.

In this project, we propose a multi-agent simulation in order to model task allocation and measure group performance in an environment where agents have different personalities and thus pairwise interactions depend on the agents involved. It is first of all a re-design of an existing model to a more modern, adaptable, and higher level code. The project also consists of an expansion of this model, primarily including personalities as a new dimension in the definition and interaction of agents. Lastly, it links the results obtained regarding task allocation with personality-classification theory.

2 Individual contributions

Both team members were active and contributed equally to the project. Throughout the semester there was an intention of dedicating a similar amount of hours from both parties.

João focused mainly on the adaptation of the simulator to python, the design of the workplace, and the interaction of the agents. He included interactive libraries and the use of a notebook to help with the understanding of the simulator and the reproducibility of the code.

Miguel, on the other hand, worked on the modelling, the work splitting, and the high-level decisions that had to be made in order to have a complex but understandable model. He also worked on the dynamic plots and helped with debugging, as well as with understanding the existing literature.

3 Introduction and Motivations

Human interactions is a very complicated topic to model and address. Not only because of the complexity of its nature, but also because of it being a topic that lays between very different research areas such as psychology and computer science. Many approaches to this have been attempted, but the balance between complexity and comprehensibility is subtle. Initially, we were interested in modeling division of labour, as we had read very intriguing results published regarding the behaviour of ants. However, articles gave little detail regarding simulation, and thus many were discarded.

We were lucky to find a very well-balanced example in Dr. Zoethout's research ([6] and [7]): the model was complex enough, understandable and details regarding the simulator were given. He focused on task allocation and *rotation* (i.e., how people switch tasks), as well as group performance (how much time tasks need to get done). We will refer to his simulator from here onward by its name, WORKMATE-I.

In order to improve and expand this previous research, many alternatives were explored. We decided to include the personality dimension to it after coming across *16Personalities* [2]. Many personality tests such as this are based on Carl Gustav Jung's theory of psychological types, which was in the 1920s refined and used to patent a personality indicator. The Myers-Briggs Type Indicator (MBTI), named after her co-authors Katharine Cook Briggs and Isabel Briggs Myers, is still used nowadays in areas such as hiring people, making teams, or giving career advice. We modeled the interaction of agents based on the matching coefficient between personality types that can be found in the literature (for an example with a lot of information, see [5]).

3.1 Fundamental Questions

- Our primary focus is to determine the evolution of a group's performance over time depending on relationships of individuals in the group based on MBTI and task variety (a concept developed in [6] which refers to how many different skills are needed in order to carry out a given set of tasks).
- In addition, we want to see the evolution of *mood* or *frustration*, which is the additional dimension we incorporated to WORKMATE-I, as agents interact. This variable indicates how comfortable a person is regarding the work environment and inter-personal relationships.
- As a secondary goal, we would like to tweak some of the existing parameters of the model- mainly expertise and motivation- to see how this affects the new simulator.

3.2 Expected Results

- In an environment with high task variety, we are expecting similar results both when there are "good" and "bad" inter-personal relationships. We only expect a different order of magnitude in the performance time, but with similar variations. This is because agents will have to adapt to new tasks with different skills and their allocation

time will take longer due to discussions caused by the bad relationship (for more details see section 4).

- When task variety is low, however, we expect the performance time curve to decrease faster when agents have a good relationship. The reasoning behind this is that a better relationship will lead to quickest decisions made regarding task allocation and they will be in a better mood so their performance will be better.
- Despite not mentioning it in section 3.1, we were also interested in replicating all the results found in [6] in order to verify we were following the same path as the previous researchers.

These hypotheses, however, were made before simulating and thus have to be validated by the results in the following sections of this document.

4 Description of the Model

Modeling complex interactions has to be done always taking into account Bonini's paradox: *"As a model of a complex system becomes more complete, it becomes less understandable. Alternatively, as a model grows more realistic, it also becomes just as difficult to understand as the real-world processes it represents"* [1]. As explained before, we chose WORKMATE-I because of its comprehensibility, but adding new layers inevitably increases its complexity. As a note to the reader, some of the basic definitions and formulas that we will list in this section can be found in [6] and [7] partially[1]. However, for completeness and in order to allow for a fully understanding of the project to people that don't have previous knowledge, we decided to give a medium-to-high level of detail in this section.

The following subsections aim to introduce the reader, as mentioned, to all the concepts needed to understand the simulator.

4.1 Tasks and Actions

A **Task** is defined as the main unit of work that has to be done by the agents. It consists of **actions**, which are the smallest units of work. This is, only one agent can be working on each action. Each task consists of ≥ 1 action, and different tasks can consist of the same or different actions. This is, if the task T consists of several actions a_i, we can write $T = \{a_1, ..., a_k\}$. What we state here is that we can have $T_i \cup T_j \neq \emptyset$.

[1]Of course not everything can be found in these references due to the changes we made and the new characteristics we included

Tasks can also consist of several **cycles**. These are the number of times the same set of actions is repeated. This is done to allow repetitions, although one could say one task consisting of two cycles is equivalent to two identical sequential tasks.

Since actions cannot be sub-divided, only one **skill** is associated to every action. A skill is an ability needed for an agent to perform an action.

The concept of **task variety** is also important in this project. It can also be found as *dynamic complexity* in other articles like [4]. It refers to the amount of new actions (and thus, skills) that a new task has with respect to the previous one. We found the example in [6] to be easy to understand: *"at t1,the task consists of action 2, 3, and 4, which activate skills 2, 3, and 4. In case of a high task variety at t2, the task consists of actions 4, 5, and 6, whereas in case of a low task variety,the task consists of actions 3, 4, and 5"*.

4.2 Agents

An **agent** models a person with the basic properties needed to carry out the actions the tasks consist of. Skills are divided into two types:

1. Active skills: are the ones in the short-term memory (STM). When agents begin to perform a task, the set of skills associated to the actions the task is made of are "activated" and sent to the STM.

2. Passive skills: are the skills that are not being used. They therefore belong to the long-term memory (LTM).

For implementation details, read section 5.

Each agent has three dimensions associated to it. Two of these were already present in WORKMATE-I, and are defined for each of the skills of the agent:

1. **Expertise (e)**: how well the agent performs a certain skill.

2. **Motivation (m)**: how willing is the agent to perform a certain skill.

A third characteristic of the agent is:

3. **Frustration (f)**: how uncomfortable an agent is regarding the work environment and the interaction with its peers. We will also talk about **mood** as the opposite concept of this[2].

Note that this consists only of one value per agent.

[2]refer to appendix A.1 for a more detailed explanation if it seems confusing.

4.3 Interactions

Agents have two nodes each, I and You, whose values are in $[0, 1]$. Each agent's I-node represents its willingness to do the task, and its You-node represents how willing the agent is to let another agent do that task. If an agent has a high value in the I-node (excited node) and a low value in the You-node (inhibited), it means that the agent *really* wants to carry out a certain task, and vice versa.

The **first step** in the allocation process corresponds to the initial value of the I and You nodes. Expertise and motivation are compared to thresholds θ_i that determine if a certain action is associated to a skill that is "easy enough" ($e \lesseqgtr \theta_e$) or "interesting enough" ($m \lesseqgtr \theta_m$) to be done. Based on this, we can distinguish three cases:

1. If $e < \theta_e$ the agent is unable to carry out the action, so:

$$I = 0 \tag{1a}$$
$$You = 1 \tag{1b}$$

2. If $e > \theta_e$ and $m > \theta_m$ the agent is willing to do the action as follows:

$$I = \lambda \frac{e - \theta_e}{e_{max} - \theta_e} + (1 - \lambda) \frac{m - \theta_m}{m_{max} - \theta_m} \tag{2a}$$
$$You = 0 \tag{2b}$$

3. If $e > \theta_e$ but $m < \theta_m$:

$$I = \lambda \frac{e - \theta_e}{e_{max} - \theta_e} \tag{3a}$$
$$You = \frac{m - \theta_m}{m_{max} - \theta_m} \tag{3b}$$

Note that $\lambda \in [0, 1]$ is a parameter that decides the balance between e and m. Refer to [6] for all the details on why these formulas are chosen.

The **second step** in the allocation process is the excitation and inhibition of the nodes. Agents interact with each other modifying the initial values we have just explained. The equations are as follows:

$$I_2 := I_2 - iI_1I_2|I_1 - I_2| \tag{4a}$$
$$You_2 := You_2 - iYou_1You_2|You_1 - You_2| \tag{4b}$$
$$You_2 := You_2 + \varepsilon(1 - You_2)I_1|I_1 - I_2| \tag{4c}$$
$$I_2 := I_2 + \varepsilon You_1(1 - I_2)|You_1 - You_2| \tag{4d}$$

Parameters ε and i set the excitation and inhibition of agents when they interact. They can be modified by the user, as will be explained in section 5. At every time-step of the allocation process, agents interact with each other based on these equations. The reasoning behind this is that since each agent's I-node represents its willingness to do the task, it excites the other agent's You-node and inhibits the other agent's I-node. The You-node excites and inhibits analogously.

The following figure helped us understand the interaction between the agents that leads to the task allocation:

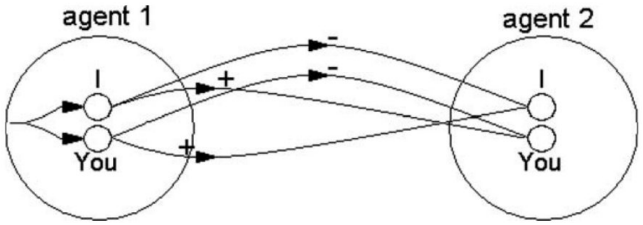


Figure 1: Excitation (+) and inhibition (-) of two agents as found in [6]

This second step finishes when agents have reached an agreement. This is, for agent 1 $I > You$ and for agent 2 $You > I$.

We considered that not only allocation time should affect frustration (as seen in equation 6), but also the other way around. This helps to capture out the complexity of interactions in a way that is closer to reality. After allocation between two agents, the time is updated according to the following equation:

$$T := T(1 + 0.5[\frac{1}{3}\frac{f_1 + \frac{1}{3}\frac{f_{max}}{2}}{\frac{f_{max}}{2}} - \frac{1}{3}\frac{f_2 - \frac{f_{max}}{2}}{\frac{f_{max}}{2}} - \frac{1}{3}\frac{r_{ij} - 0.5}{0.5}]) \tag{5}$$

where:

- r_{ij} is the "relationship factor" between agents i and j involved in the interaction. It is a value in $(0, 1)$ based on the *normalized interpersonal relationship scaling matrix* that can be found in Table 2 from [5], but slightly modified so that it doesn't reach its extreme values.
- f_1 and f_2 are the frustrations that agents have based on previous interactions.
- It is worth noticing that this is just $T := T(1 + \alpha)$ making α dependant on frustrations and relationship between the agents.

4.4 Frustration update

Frustration (f) is updated once the tasks were allocated by the agent, and just before they carry out the actions:

$$f_{t+1} := 0.8 f_t + 0.2 I(T, r_{ij}) \tag{6}$$

where $I(T, r_{ij})$ can be thought as the "immediate frustration": it is an indicator of how satisfied an agent is with its interaction with another agent. It is defined as follows:.

$$I(T, r_{ij}) = I_{max}(1 - e^{-\beta \frac{1-r_{ij}}{r_{ij}} \frac{\mathfrak{T}/\mathfrak{T}_{max}}{1-\mathfrak{T}/\mathfrak{T}_{max}}}) \tag{7}$$

where:

- T is the allocation time. This is, the time-steps it took the agents to reach an agreement for all the actions in the cycle.
- β is a normalization parameter to set the balance between relationship and discussion time.
- I_{max} is a design scale parameter.
- $\mathfrak{T}_{max} = \frac{T_{max}}{10}$ and $\mathfrak{T} = min\{T, \mathfrak{T}_{max} - 1\}$. These auxiliary variables are added in order for the allocation time to have a meaningful effect on the immediate frustration. It is a small detail that was mentioned in [6]: allocation time is modeled in a way that can take an arbitrarily high time, and it has to be stopped.

Note that if the relationship is good ($r \to 1$) or the allocation time is brief ($T \to 0$) then $I \to 0$ which makes sense, since we want frustration to be low if the agents have a good relationship or if they don't discuss much. The opposite happens if the relationship is bad or the discussion time is too high: $I \to I_{max}$.

Note too that this process has memory, as indicated by equation 6. Frustration at time t depends on immediate frustration I but also on past frustration at time $t - 1$.

4.5 Performance

Once the tasks are allocated for each of the cycles, agents carry out the actions. The performance time for each agent a is calculated using the

following equation:

$$t_{perf\_a} = \sum_{i=1}^{N_{actions}} \frac{t_{action\_i}}{\alpha_e \frac{e_i}{e_{max}} \alpha_m \frac{m_i}{m_{max}} \alpha_f \frac{1-f}{f_{max}}} \tag{8}$$

where:

- $\alpha_e + \alpha_m + \alpha_f = 1$. These are scale parameters to set how expertise, motivation and frustration relate to each other when performing a task
- $1 - f \equiv \mu$ is the mood of the agent. We need this and not simply f because a better mood has to decrease the performance time and not the opposite.
- t_{action_i} is the minimal time to complete action i.

The time taken by the group to complete the whole task depends on the allocation time that was done previously, and also on the slowest of the agents. The formula is the following:

$$T_{perf} = max\{t_{perf_1}, ..., t_{perf_n}\} + T_{coordination} \tag{9}$$

4.6 Expertise and Motivation update

After a task is performed, expertise and motivation (which we will both name p for 'parameter' in the formula) are updated for all the skills. If the skill was in the STM, then its associated expertise will increase, but its motivation will decrease due to boredom. The opposite happens if the skill is in the LTM. The formula for "learning" or "getting motivated" is:

$$p_{t+1} = p_t + \lambda_p \frac{p_{max} - p_t}{p_{max}} \tag{10}$$

and the update formula for "forgetting" or "getting bored" is:

$$p_{t+1} = \frac{(p_t - \mu_p) p_{max}}{p_{max} - \mu_p} \tag{11}$$

In both formulas, p can be substituted by e or m. Note that the learning/-motivation rate (λ) and the forgetting/boredom rate (μ) change depending if we are computing expertise or motivation.

We include the following sketch in order to summarize what happens in the simulator from step 4.3 to 4.6:

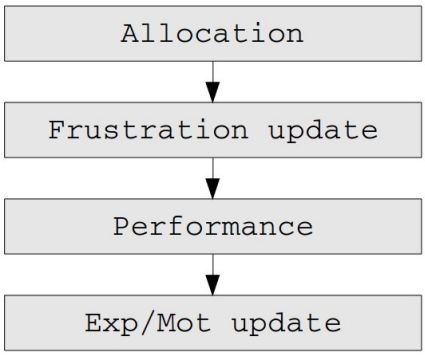


Figure 2: Steps of the simulator for each action

5 Implementation

Our simulator shares the properties of being agent-based and deterministic with the WORKMATE-I simulator which it expands.

We were unfortunately forced to abandon the addition of random components to our simulaor due to time constraints. This is certainly one of the most pressing expansions to consider as future work.

Our simulator was developed using Python 3 and uses object-oriented programming as the programming paradigm. This contributes significantly to improving code clarity and readability.

In the following subsections we will give two points of view of the simulator. First, it is described as a *black box*, by identifying its expected outputs, when provided a particular set of inputs. Its internal functioning is then described in greater detail, by detailing the classes and functions that make it work.

5.1 Input/Output

In order to generate the results that can be found in section 6, our simulator takes as input a JSON[3] file which features three main entities as **inputs**:

1. **Parameters** which can be tuned to adjust the behaviour of the model.
2. The **list of agents** which integrate the workplace.
3. The **list of tasks** to be carried out sequentially. One task consists of a list of actions, and each action requires one single skill.

Several things can be plotted with our simulator:

[3] `https://www.json.org/`

1. **Evolution of frustration:** Each agent keeps track of their own level of frustration over time. This enables the qualitative assessment of the differences in interactions between different personality types. The horizontal axis represents *negotiation* rounds.

2. **Evolution of expertise / motivation:** Expertise and motivation can be individually plotted for multiple agents, with the horizontal axis representing the flow of time, in cycles. This is analogous to the plots created by WORKMATE-I in [6].

3. **Group performance:** equation 9 is calculated for every action, when agents discuss in order to allocate the tasks and decide who does what. The horizontal axis is cycles as before.

5.2 Classes

We include the following UML-like class diagram to help the reader understand the relationships between the classes in the simulator:

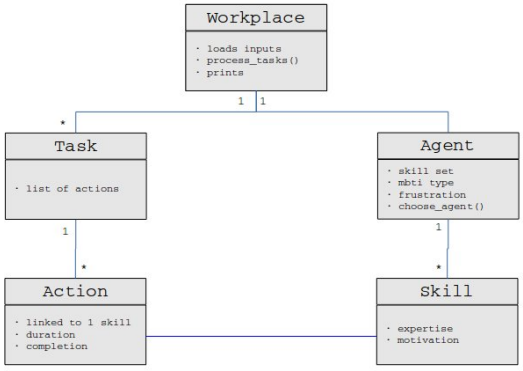


Figure 3: Steps of the simulator

We included the most basic traits of each class inside the boxes. The numbers on top of the links indicate how many objects of each class contain or interact with the objects of the class linked to it. The asterisk means "many" or ≥ 1. For example, *1 agent* contains *many skills*.

5.2.1 Workplace

The *workplace* is the main class of the simulator. It orchestrates the interaction between all instances of other objects. Multiple workplace objects may coexist in a given file. The lifetime of a workplace object consists of three

stages: 1. the creation of the workplace; 2. the processing of tasks; 3. the exporting of results.

The workplace object is able to plot the evolution for expertise, motivation, frustration, task allocation times and system/agent performance. The default parameters may be overriden by the input JSON file. The loaded parameters may be printed at any time by calling `print_parameters()`.

5.2.2 Agent

An *agent* is one who undertakes the execution of actions, upon allocation. The agent's MBTI type is constant. Their level of frustration is internally logged over time. The levels of expertise and motivation for each skill are stored as properties of each *skill* in the agent's *skillset*.

The functions for task allocation and negotiation are also defined here.

5.2.3 Task

Task objects consist of an identifier and a list of actions.

5.2.4 Action

Each action is linked to a skill and stores two additional parameters: the number of cycles that have been dedicated to it, and the total number of cycles required for it to be considered completed.

5.2.5 Skill

A skill is uniquely identified by its ID. Skill objects store the history of the corresponding expertise and motivation values of the agent that holds them.

6 Results and discussion

6.1 Replication of previous results

6.1.1 Group performance

As briefly mentioned in section 3.2, our first goal was to replicate previous results of WORKMATE-I. This might seem trivial at first, but it was without any doubt the most difficult part of the project. We would like to highlight the fact that our main reference (articles [6] and [7]) do not contain a single line of code. Dr Zoethout provided us with part of the code, but it was not everything and it was programmed using "Delphi6" in what we think is

Pascal. Furthermore, our approach to the code was to have a more general and expandable project, so the structure of it is also different. All of this resulted in having to reverse-engineer many parts and took many hours of work.

Plots that prove that we achieved our goal can be found in appendix B. Original plots as found in [6] can also be found in appendix B as there was no more space here.

6.2 Generation of new results

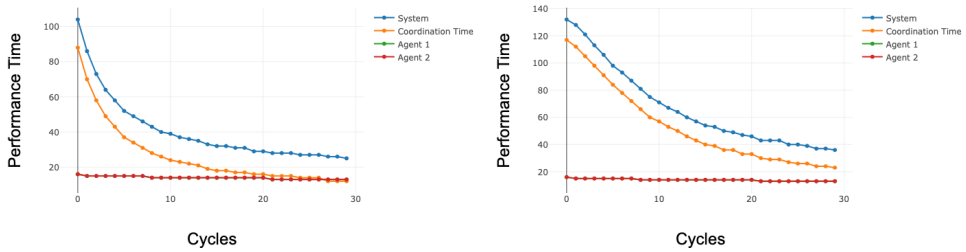
As mentioned in section 3.1, we wanted to determine how both task variety and relationships affect **performance**. We simplified the experiments by having Low (L) and High (H) task variety (using the same definitions as the ones used in [6]), and having Good (G) and Bad (B) relationships. We used $r_{ij} = 0.83$ for good relationship and $r_{ij} = 0.17$ for bad relationship. These values correspond to agents being 'ISFP' and 'ENTP' in the former case and 'ISTP' and 'ESTJ' in the latter. Consult [5] for more information about these numbers.

In order to avoid modifying too many variables at a time, we used the same parameters for low task variety as the ones used in Experiment 1 in [6], and the same parameters[4] for high task variety as the ones used in Experiment 2. Results are show in figure 4.

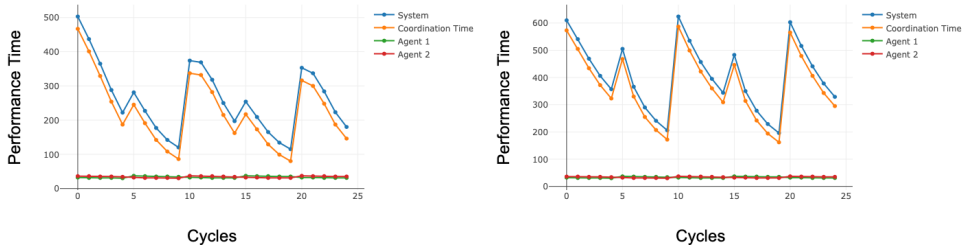
Results show that when task variety is low (L, top row), the behaviour is similar but the decrease in performance time is faster when relationship is good (G). This matches our predictions perfectly, and it makes sense: in a team were people get specialized and their relationship is good, performance time can only get better with time.

In the case of High task variety, one of the hypotheses was correct: the performance in case G is always lower than in B after the first few cycles. In G, performance is always lower than 400 time units, whereas it is always higher than 400 in B. Our other hypothesis was that the shape of the plots was going to be the same in G and B. This is not completely correct. Due to the fact that frustration and inter-personal relationship both affect allocation time (as seen in equation 5), it takes less to allocate the tasks and therefore performance time decreases. This explains the first of our hypotheses. However, since we made allocation time affect frustration, this circular effect makes a positive feedback stopping the allocation time to go

[4] this was done with the parameters we "inherited" from WORKMATE-I. Our simulator, being an expansion, has more parameters.



(a) LG **(b)** LB



(c) HG **(d)** HB

Figure 4: Replication of previous experiments

further, and thus it stops and the plot changes its shape.

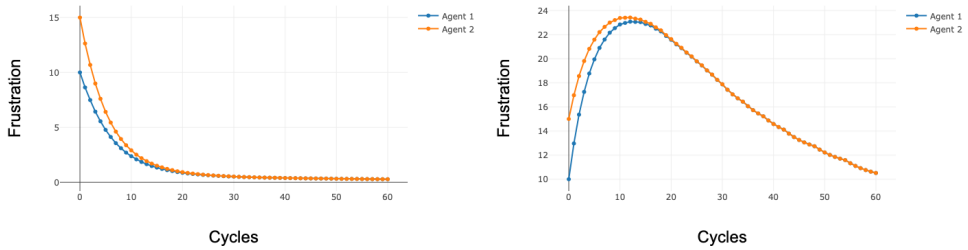
We can draw the following conclusion from these plots: as long as people are highly motivated, in an environment with low task variety people can specialize and minimize their interactions with people they don't like, and performance will be good. If there is high task variety, however, people have to allocate new tasks and discussions are unavoidable.

6.2.1 Frustration

We also wanted to see the evolution of mood/frustration. Figure 5 shows this. In both cases, task variety is low and thus agents can specialize on their own tasks, reducing allocation time at every cycle.

Results show that when relationship is good, then frustration goes quickly to zero.

When relationship is bad, however, the result was a little more difficult to predict. Nevertheless, it makes perfect sense: frustration goes up due to the bad relationship of the agents during the first 10 to 15 cycles. In the meantime, however, the agents are becoming more and more specialized



(a) Good relationship **(b)** Bad relationship

Figure 5: Frustration evolving over time (cycles)

in their own tasks. When they reach a certain level of specialization, their interaction is reduced and therefore they do not get frustrated because they do not discuss. This proves that in an environment where people are highly motivated, conflicting personalities can work together as long as they do not have to interact too much, and performance will be good.

7 Summary and Outlook

7.1 Main Contributions

An important outcome of this project has been bringing back a simulator that had been unused for some time. We reverse-engineered its internal mechanisms, polished some parts and adapted it for more uses in the future. It is now implemented in Python, a language for which thousands of libraries have been coded, so it is easier to modify and include new features.

Our next step was to expand WORKMATE-I. We did so by including the frustration/mood of agents and their inter-personal relationships. It adds complexity to the model, but in a limited and controlled way.

Another contribution of our group was proposing a way of modeling how people get frustrated, and measuring their happiness regarding their peers in a work environment. All the ideas regarding this were developed from scratch, and required a lot of creativity but also testing.

Lastly, we linked two areas of research with this work: task allocation and personality types. More work could be done in the future in order to strengthen this interesting bond.

7.2 Limitations

Despite the contributions made, we are aware of the limitations of this project. There are many of them, especially if we consider the full complexity of human interactions.

To begin with, we would like to mention the absence of randomness. We considered it, but there was no time for adding new features to the simulator. Having deterministic results also helped us enormously to debug the simulator, as well as to be closer to WORKMATE-I. This means that results obtained with this simulator can only partially describe human interaction, as randomness and unpredictability are human traits.

Another aspect of the simulator we considered and did not fully develop in the end was the possibility of having more complex tasks. There are situations where tasks depend on each other in very complicated ways. We had part of the project working with that, but it turned out to be too complex (refer to Bonini's paradox in section 4).

Lastly, we want to highlight the absence of real data to tune the parameters the simulator consists of. It would have been a great improvement to the model to tune those λs and βs according to scientific results. Some literature review was done, but it was difficult to find results that related to those parameters in a reasonably-enough direct way.

7.3 Future work and improvements

A first change that should be made to this simulator is slightly modify it to work with several agents. The logic does not change much, but complexity with two agents was already very high due to the other variables in the program. It is something that we intentionally left undone, but that should not take very long to adapt.

Another idea that we had in mind was to use this simulator to analyze group formation. Letting agents go in and out of the group depending on their mood history could be an interesting experiment. This shows that the simulator can be used in the future and expanded for other research areas.

We wanted to add a last idea for future work, which is the use of different inter-relationships. We explored the idea of working with Belbin Team Roles (BTR), which has been related to MBTI in several studies like [3]. It was one of the many things we could not do due to lack of time.

The number of ways in which the simulator can be modified and adapted is almost only limited by imagination. We came up with many during our brainstorm sessions, but only the imperative ones were listed here.

References

[1] J.M. Dutton and W.H. Starbuck. *Computer simulation of human behavior*. Wiley series in management and administration. Wiley, 1971. ISBN: 9780471228509.

[2] NERIS Analytics Limited. *16personalities.com - Personality Test*. `https://www.16personalities.com/`.

[3] Radu Ogarcă, Liviu Crăciun, and Laurențiu Mihai. "The influence of the behavioral profile upon the management team's performance." In: *Annals of the University of Craiova, Economic Sciences Series* 1 (2015).

[4] Robert E Wood. "Task complexity: Definition of the construct". In: *Organizational behavior and human decision processes* 37.1 (1986), pp. 60–82.

[5] Lianying Zhang and Xiang Zhang. "Multi-objective team formation optimization for new product development". In: *Computers & Industrial Engineering* 64.3 (2013), pp. 804–811.

[6] Kees Zoethout, Wander Jager, and Eric Molleman. "Formalizing self-organizing processes of task allocation". In: *Simulation Modelling Practice and Theory* 14.4 (2006), pp. 342–359.

[7] Kees Zoethout, Wander Jager, and Eric Molleman. "Task dynamics in self-organising task groups: expertise, motivational, and performance differences of specialists and generalists". In: *Autonomous Agents and Multi-Agent Systems* 16.1 (2008), pp. 75–94.

Appendix A Comments

A.1 About frustration

We did not come up with a good word in English to define *frustration* or *mood*. We apologize if the use of these words led to confusion to the reader, because it also confused us too. With this comment we want to make sure that the reader understands what we meant in every moment. The concept being defined (if referred to as frustration) is the uneasiness or not-comfortable feeling due to working or interacting with people with which one has a bad relationship. We can think of it as a value $f \in [0, 1]$ or $f \in [0, f_{max}]$. Conversely, mood can be thought of as the opposite (mathematically, $\mu \in [0, 1]$ or $\mu \in [0, \mu_{max}]$ where $\mu_{max} = f_{max}$.

A.2 Looking Back

At the end of the project, we wanted to ask ourselves: *Have we accomplished the goals of the course?* Professors Aguilar and Antulov-Fantulin mentioned on the first day they were seven of them, which we list below:

1. Acquiring understanding of the basics of MATLAB and Python.
2. Attaining practical knowledge necessary to run computer simulations.
3. Understanding basic properties of complex social systems.
4. Acquaintance with main quantitative modeling approaches for social systems.
5. Implementing (simple) models of social systems, replicating and extending established models.
6. Learning to pose a scientific research question
7. Becoming confident in communicating scientific results in an academic context.
8. Making your research reproducible

Goals 1 and 2 have been clearly fulfilled. Many hours were spent working on the simulator. The team had to make many decisions that are not even mentioned here, regarding aspects such as what libraries to choose or how to organize the code. Goals 3 and 4 were also satisfied, although we are aware of the limitations of the course. The complexity of social systems

leads to an almost infinite number of ways of approaching it. The main goal we worked on was 5, although we also often discussed many aspects of 6. The freedom we had to choose and develop a project sometimes played against us. Lastly, we can only hope that 7 and 8 are good enough for our peers and professors evaluating our work.

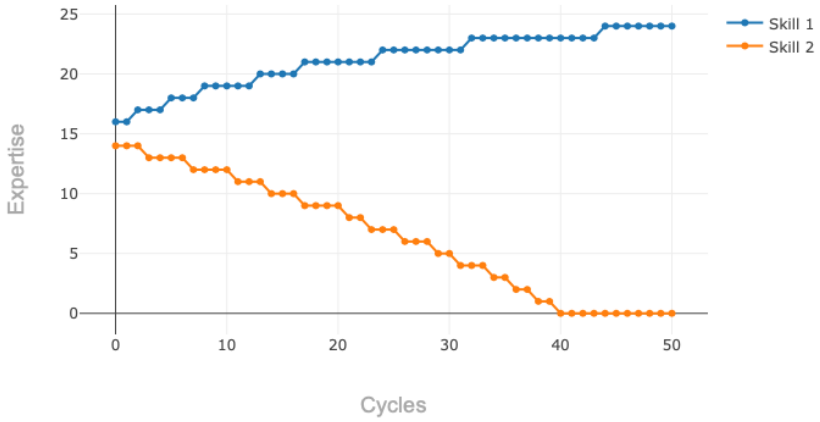
A.3 Other comments

- We would like to thank Dr. Zoethout for his quick response when we contacted him, and for having released almost all the code for free access of anyone interested. This helped with some details when understanding WORKMATE-I.
- Teams could be of up to 4 people. If we had known the other people in class, we could have formed a group twice as big and generated a more complex model. We conclude that a project as ambitious as the one we carried out should be done by a group formed by 3-4 people.
- The 15-page limit forced us to reduce the size of images and move some of them to the appendix. We apologize for that, but we had no alternative. All images have a high enough quality so that in a computer the reader can zoom in and see all the axis and legends.

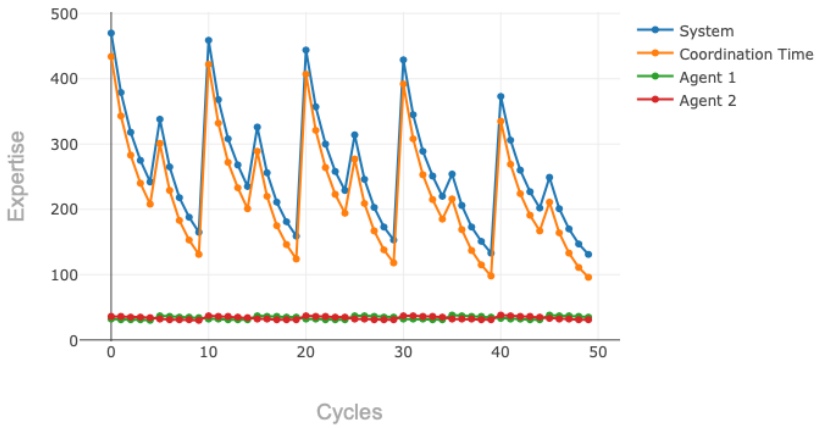
Appendix B Figures

B.1 Original experiments with WORKMATE-I

The following plots prove that we achieved our goal:



(a) Experiment 1 in [6]



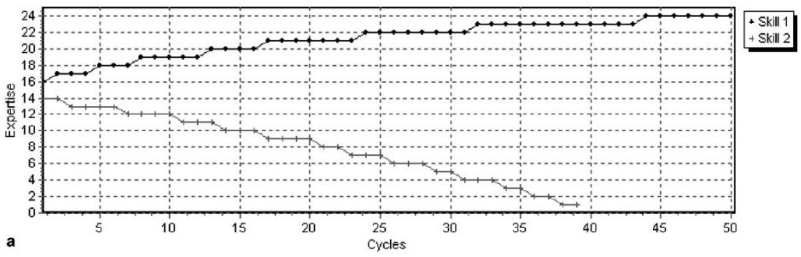
(b) Experiment 2b in [6]

Figure 6: Replication of previous experiments

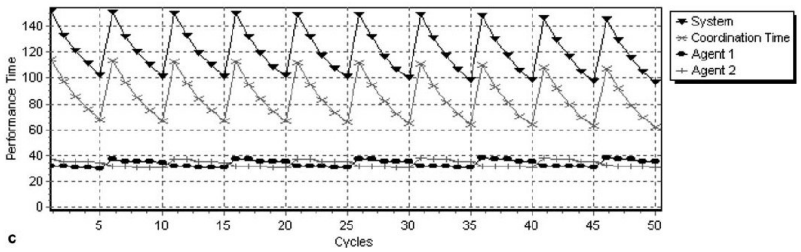
Results in figure 6a show the evolution of two skills s_1 and s_2 that an agent has. The context is two agents performing one task that consists of two

actions and 50 cycles. There is expertise, but no boredom. Initially, agent a_1 is more predisposed to do s_1 and the other to do s_2. That is why, after the first interaction, a_1 picks s_1 and vice versa. After completing the first cycle, a_1 is better at performing s_1, and that affects the next allocation making him more willing to do it again. As time goes by, it becomes an expert at s_1 but forgets its skill related to action 2.

A second experiment was carried out in order to be more certain that we could replicate the behaviour correctly. The next context is 10 tasks consisting of 4 actions and 5 cycles. There are two agents, each of which has 40 skills. There is again no boredom. Results (figure 6b) show that high task variety implies having peaks in the performance time due to an increase in the allocation time caused by the agents not being experts in the new tasks. Note that the Y-axis has a different scale because of the way we compute allocation time. It is not a problem, however, because there are no units for time. The simulator explains what things happen, and can be used to compare times, but not for calculating time in *real-life* scenarios.



(a) Experiment 1



(b) Experiment 2b

Figure 7: Original experiments in [6]

Appendix C Code

Parameters

```
#########################################################################
#########################################################################
# This is just the file where we have all the parameters of the model#
#########################################################################
#########################################################################
# DEFAULT PARAMETERS
TASK_UNIT_DURATION = 10
ALPHA_E = 0.5
ALPHA_M = 0.5
ALPHA_F = 0
BETA = 1
LAM_LEARN = 1
LAM_MOTIV = 0
MU_LEARN = .5
MU_MOTIV = 0
TH_E = 10
TH_M = 10
MAX_E = 25
MAX_M = 25
MAX_H = 25
EXCITE = .1
INHIBIT = .1

MAX_COORD_STEPS = 1000

MBTI = [[0.67, 0.33, 0.83, 0.5, 0.83, 0.5, 1, 0.67, 0.5, 0.17, 0.67, 0.33,
     0.67, 0.33, 0.83, 0.5],
        [0.33, 0.67, 0.5, 0.83, 0.5, 0.83, 0.67, 1, 0.17, 0.5, 0.33, 0.67,
     0.33, 0.67, 0.5, 0.83],
        [0.83, 0.5, 0.67, 0.33, 1, 0.67, 0.83, 0.5, 0.67, 0.33, 0.5, 0.17,
     0.83, 0.5, 0.67, 0.33],
        [0.5, 0.83, 0.33, 0.67, 0.67, 1, 0.5, 0.83, 0.33, 0.67, 0.17, 0.5,
     0.5, 0.83, 0.33, 0.67],
        [0.83, 0.5, 1, 0.67, 0.67, 0.33, 0.83, 0.5, 0.67, 0.33, 0.83, 0.5,
     0.5, 0.17, 0.67, 0.33],
        [0.5, 0.83, 0.67, 1, 0.33, 0.67, 0.5, 0.83, 0.33, 0.67, 0.5, 0.83,
     0.17, 0.5, 0.33, 0.67],
        [1, 0.67, 0.83, 0.5, 0.83, 0.5, 0.67, 0.33, 0.83, 0.5, 0.67, 0.33,
     0.67, 0.33, 0.5, 0.17],
        [0.67, 1, 0.5, 0.83, 0.5, 0.83, 0.33, 0.67, 0.5, 0.83, 0.33, 0.67,
     0.33, 0.67, 0.17, 0.5],
        [0.5, 0.17, 0.67, 0.33, 0.67, 0.33, 0.83, 0.5, 0.33, 0, 0.5, 0.17,
     0.5, 0.17, 0.67, 0.33],
        [0.17, 0.5, 0.33, 0.67, 0.33, 0.67, 0.5, 0.83, 0, 0.33, 0.17, 0.5,
     0.17, 0.5, 0.33, 0.67],
```

```
        [0.67, 0.33, 0.5, 0.17, 0.83, 0.5, 0.67, 0.33, 0.5, 0.17, 0.33, 0,
     0.67, 0.33, 0.5, 0.17],
        [0.33, 0.67, 0.17, 0.5, 0.5, 0.83, 0.33, 0.67, 0.17, 0.5, 0, 0.33,
     0.33, 0.67, 0.17, 0.5],
        [0.67, 0.33, 0.83, 0.5, 0.5, 0.17, 0.67, 0.33, 0.5, 0.17, 0.67,
    0.33, 0.33, 0, 0.5, 0.17],
        [0.33, 0.67, 0.5, 0.83, 0.17, 0.5, 0.33, 0.67, 0.17, 0.5, 0.33,
    0.67, 0, 0.33, 0.17, 0.5],
        [0.83, 0.5, 0.67, 0.33, 0.67, 0.33, 0.5, 0.17, 0.67, 0.33, 0.5,
    0.17, 0.5, 0.17, 0.33, 0],
        [0.5, 0.83, 0.33, 0.67, 0.33, 0.67, 0.17, 0.5, 0.33, 0.67, 0.17,
    0.5, 0.17, 0.5, 0, 0.33]]
```

Frustration movie

```
# FOR MORE INFO VISIT:
# https://matplotlib.org/api/_as_gen/matplotlib.animation.FuncAnimation.
    html#matplotlib.animation.FuncAnimation

import matplotlib.pyplot as plt
from matplotlib import animation
import numpy as np
import os


################################################
#                  AUX. PARAMS                 #
################################################
__BASE_DIR = os.path.dirname(os.path.abspath(__file__))
moods_file = os.path.join(__BASE_DIR, 'moods.data')
mood_data = []

NAGENTS = 2
ymin, ymax = 0, 0 # initial values used to plot


################################################
#                AUX. FUNCTIONS                #
################################################
def read_moods():
  ''' Reads moods from file '''
  with open(moods_file) as fr:
    for line in fr:
      mood_data.append(np.fromstring( line, dtype=np.float,
                       sep=',' ))

def animate(i):
  '''
    Modifies the height of the bars because it is called for each
```

```
    frame of the mp4
  '''
  y=mood_data[i+1]
  for i, b in enumerate(barcollection):
    b.set_height(y[i])


##################################################
#                    MAIN CODE                   #
##################################################
if __name__ == "__main__":
  # STEP 1: read the input file
  read_moods()
  nframes= len(mood_data) - 1

  # STEP 2: prepare the plot
  fig = plt.figure()
  plt.ylim([np.min(mood_data),np.max(mood_data)])
  x_coord=range(1,NAGENTS+1)
  barcollection = plt.bar(x_coord,mood_data[0])

  # STEP 3: call the animator. It will update the plot.
  anim=animation.FuncAnimation(fig,animate,repeat=False,blit=False,
                  frames=nframes,interval=500)

  # STEP 4: save the movie and plot
  anim.save('mymovie.mp4',writer=animation.FFMpegWriter(fps=10))

  plt.show()
```

Skill

```
class Skill:
  def __init__(self, _id = -1, exp = 0, mot = 0):
    self._id = _id
    self.expertise = [exp]
    self.motivation = [mot]

  def __str__(self):
    return '< skill_id: ' + str(self._id) + \
          ', expertise: ' + str(self.expertise) + \
          ', motivation: ' + str(self.motivation) + ' >'
```

Agent

```
import math
import numpy as np
import my_parameters as P
```

```
class Agent:
    def __init__(self, _id, mbti = None, initial_frustration = None,
    skillset = [], verbose = False):
        self._id = _id
        self.mbti = '' if mbti == None else mbti

        self.skillset = sorted(skillset, key=lambda s: s._id)  # Ensure
    skills are stored in order
        self.frustration = [] if initial_frustration == None else [
    initial_frustration]

        self.allocation_times = []
        self.performance_times = {}

        # Agent Memory
        self.stm = []                       # Should be updated when task is
    allocated
        self.ltm = self.skillset.copy()  # Initially, all skills are part
    of the long-term memory

        self.current_action = []
        self.action_history = []

        self.verbose = verbose
        self.validate_internals()

    ###################################################################
    # ---------- INTERNAL VALIDATION ----------
    ###################################################################
    # Used to check if values provided are correct when initializing agent
    # This applies to all the following functions "validate_something"

    def validate_internals(self):
        return \
        self.validate_frustration() and \
        self.validate_skillset() and \
        self.validate_mbti()

    def validate_frustration(self):
        if self.frustration == []:
            if self.verbose:
                print("Warning: no initial frustration provided!")
            return False
        return True

    # Sanity check - skills should be consecutive integers starting at
    zero
    def validate_skillset(self):
        try:
```

```
            assert [skill._id for skill in self.skillset] == list(range(
len(self.skillset)))
        except AssertionError:
            print('Invalid skillset provided! Please verify the
correctness of your input.')
            exit(1)

    def validate_mbti(self):
        if self.mbti == '':
            if self.verbose:
                print("Warning: empty MBTI provided!")
            return False

        valid = True

        if len(self.mbti) != 4 or not isinstance(self.mbti, str):
            valid = False

        if self.mbti[0] not in ['E', 'I']:
            valid = False
        if self.mbti[1] not in ['N', 'S']:
            valid = False
        if self.mbti[2] not in ['T', 'F']:
            valid = False
        if self.mbti[3] not in ['J', 'P']:
            valid = False

        try:
            assert valid == True
        except:
            print('Invalid/incomplete MBTI provided!')
            exit(1)

        return valid

    ###################################################################
    # ---------- INTERNAL UPDATES ----------
    ###################################################################

    def calculate_performance_time(self, skill_ids, assignments, time):
        '''
        Calculates the time it takes to performe the tasks that can be
        found in vector 'assignments'.
        '''
        self.performance_times[time] = sum(
            [
                P.TASK_UNIT_DURATION / ((P.ALPHA_E * self.
get_latest_expertise(skill_ids[ix]) / P.MAX_E) +
                                        (P.ALPHA_M * self.
get_latest_motivation(skill_ids[ix]) / P.MAX_M) +
```

```
                                        (P.ALPHA_F * self.get_frustration
() / P.MAX_H))
                for ix, assignment in enumerate(assignments) if assignment
 == self._id
            ]
        )

        return self.performance_times[time]

    def update_frustration(self, immediate_frustration):
        '''Frustration is updated with memory using a moving average.
Immediate
           frustration is weighted 0.2 and previous one 0.8
        '''
        if self.frustration == []:
            return

        MOV_AVG_FACTOR = 0.8
        self.frustration.append(MOV_AVG_FACTOR * self.frustration[-1] +
(1-MOV_AVG_FACTOR) * immediate_frustration)

    def update_memory(self):
        ''' Learning and forgetting depending on skills being in stm or
ltm'''
        # Learn
        for skill in self.stm:
            new_exp = skill.expertise[-1] + P.LAM_LEARN * ( (P.MAX_E -
skill.expertise[-1]) / P.MAX_E)
            new_mot = ((skill.motivation[-1] - P.MU_MOTIV) * P.MAX_M) / (P
.MAX_M - P.MU_MOTIV)
            skill.expertise.append(new_exp)
            skill.motivation.append(new_mot)

        # Forget
        for skill in self.ltm:
            new_exp = ((skill.expertise[-1] - P.MU_LEARN) * P.MAX_E) / (P.
MAX_E - P.MU_LEARN)
            new_mot = skill.motivation[-1] + P.LAM_MOTIV * ( (P.MAX_M -
skill.motivation[-1]) / P.MAX_M)
            skill.expertise.append(new_exp)
            skill.motivation.append(new_mot)

    def insert_alloc_time(self, coord_time):
        self.allocation_times.append(coord_time)

    def flush_prev_act(self, assignments, skill_ids):
        # Clear current_action, update action_history
        self.action_history.extend(self.current_action)
        self.current_action = []
```

```
        # Clear short-term memory, restore these skills to long-term
    memory
        self.ltm = self.skillset.copy()

        promote_to_stm = list(set([skill_ids[i] for i, a in enumerate(
    assignments) if a == self._id]))
        self.stm = [self.ltm[i] for i in promote_to_stm]

        for i in promote_to_stm[::-1]:
            del self.ltm[i]

    # ---------- GETTERS ----------
    def get_latest_expertise(self, skill_id):
        return self.skillset[skill_id].expertise[-1]

    def get_latest_motivation(self, skill_id):
        return self.skillset[skill_id].motivation[-1]

    def get_frustration(self):
        return self.frustration[-1] if len(self.frustration) > 0 else -1

    def get_initial_i_you(self, wp, skill_id):
        ''' Returns initial value of I and YOU nodes before agents discuss
     '''
        # Determine current expertise and motivation for this skill
        exp = self.get_latest_expertise(skill_id)
        mot = self.get_latest_motivation(skill_id)

        # Should scale P.ALPHA_E, P.ALPHA_M locally
        # Normalise alphas
        factor = 1 / (P.ALPHA_E + P.ALPHA_M)

        ALPHA_E = P.ALPHA_E * factor
        ALPHA_M = P.ALPHA_M * factor

        # Compute i, you
        # Situation 1 - insufficient expertise
        if exp < P.TH_E:
            i = 0
            you = 1
        # Situation 2 - Sufficient expertise, sufficient motivation
        elif mot >= P.TH_M:
            i = ALPHA_E * (exp - P.TH_E) / (P.MAX_E - P.TH_E) + \
                ALPHA_M * (mot - P.TH_M) / (P.MAX_M - P.TH_M)
            you = 0
        # Situation 3 - Sufficient expertise, insufficient motivation
        else:
            i = ALPHA_E * (exp - P.TH_E) / (P.MAX_E - P.TH_E)
            you = ALPHA_M * (P.TH_M - mot) / P.TH_M

        return i, you
```

```
    def get_mbti_ix(self):
        ''' MBTI is coded with binary (I=1, E=0 and so forth)
        INFP is 1111 and ESTP is 0000
        '''
        ix = 0

        if self.mbti[0] == 'I':
            ix += 8

        if self.mbti[1] == 'N':
            ix += 4

        if self.mbti[2] == 'F':
            ix += 2

        if self.mbti[3] == 'P':
            ix += 1

        return ix

    # ---------- OUTPUT ----------
    # TODO - Update with MBTI / Frustration
    def __str__(self):
        stm_str = 'Short-term memory:\n' + '\n'.join(list(map(str, self.
    stm))) + '\n'
        ltm_str = 'Long-term memory:\n' + '\n'.join(list(map(str, self.ltm
    ))) + '\n'

        curr_act_str = 'Current action:\n' + str(self.current_action) + '\
    n'
        act_hist_str = 'Action history:\n' + str(self.action_history) + '\
    n'

        return '--- AGENT ' + str(self._id) + ' ---\n' + stm_str + ltm_str
     + curr_act_str + act_hist_str

# Returns tuple containing:
# (assignments, allocation_times, skill_ids, action_ids)
def choose_agent(wp, action):
    ''' This function belongs to the whole class AGENT
        It takes two agents and makes them negotiate the task allocation
    '''
    # We are working with only two agents for now
    i0, you0 = wp.agents[0].get_initial_i_you(wp, action.skill_id)
    i1, you1 = wp.agents[1].get_initial_i_you(wp, action.skill_id)

    # frustration should be updated before their interaction
    f0, f1 = wp.agents[0].get_frustration(), wp.agents[1].get_frustration
    ()
```

```
    # Begin negotiation process
    agent, allocation_time = negotiate(i0, you0, i1, you1,
                                       r_ij = get_relationship(wp.agents
    [0], wp.agents[1]),
                                       f0 = f0, f1 = f1)

    # Update each agent's internal tracking of allocation time
    wp.agents[0].insert_alloc_time(allocation_time)
    wp.agents[1].insert_alloc_time(allocation_time)

    # Calculate immediate frustration with information from latest
    interaction
    f0, f1 = calculate_immediate_frustration(wp.agents[0], wp.agents[1])
    wp.agents[0].update_frustration(f0)
    wp.agents[1].update_frustration(f1)

    # Update action progress by one cycle
    action.completion += 1

    return (agent, allocation_time, action.skill_id, action._id)

def negotiate(i0, you0, i1, you1, inhibit = P.INHIBIT, excite = P.EXCITE,
    r_ij = -1, f0 = -1, f1 = -1):
    # If parameters are not specified, they also hold no effect over the
    system
    # if r_ij == -1 or f0 == -1 or f1 == -1:
    #     r_ij = 0.5
    #     f0 = P.MAX_H/2
    #     f1 = P.MAX_H/2

    MAX_DELTA = 0 if (r_ij == -1 or f0 == -1 or f1 == -1) else 0.5

    allocation_time = 0

    while (i0 > you0 and i1 > you1) or \
          (you0 > i0 and you1 > i1):

        diff_i = abs(i0 - i1)
        diff_you = abs(you0 - you1)

        prev_i0, prev_i1, prev_you0, prev_you1 = i0, i1, you0, you1

        # Update agent 0
        i0 -= inhibit * prev_i0 * prev_i1 * diff_i
        you0 -= inhibit * prev_you0 * prev_you1 * diff_you
        you0 += excite * (1 - prev_you0) * prev_i1 * diff_i
        i0 += excite * prev_you1 * (1 - prev_i0) * diff_you

        # Update agent 1
        i1 -= inhibit * prev_i0 * prev_i1 * diff_i
        you1 -= inhibit * prev_you0 * prev_you1 * diff_you
```

```
        you1 += excite * (1 - prev_you1) * prev_i0 * diff_i
        i1 += excite * prev_you0 * (1 - prev_i1) * diff_you

        allocation_time += 1

        if allocation_time >= P.MAX_COORD_STEPS:
            if np.random.randint(0, 2):
                i0, you0, i1, you1 = 1, 0, 0, 1
            else:
                i0, you0, i1, you1 = 0, 1, 1, 0
            break

    # The agent that will perform this action has been determined
    agent = 0 if i0 > you0 else 1

    # DEBUG
    # print([agent, allocation_time])

    # Adjust allocation_time with a factor based on r_ij, f0, f1
    allocation_time *= (1 + MAX_DELTA * ((-(r_ij - 0.5)/0.5 + (f0 - P.
    MAX_H/2)/(P.MAX_H/2) + (f1 - P.MAX_H/2)/(P.MAX_H/2)) / 3))

    return agent, allocation_time

def get_relationship(agent0, agent1):
    ''' Returns the r_ij (relationship between agents as a number in (0,1)
    )'''
    r_ij = P.MBTI[agent0.get_mbti_ix()][agent1.get_mbti_ix()] \
           if agent0.validate_mbti() and agent1.validate_mbti() \
           else -1
    return r_ij if r_ij != 0 else np.finfo(float).eps

def calculate_immediate_frustration(agent0, agent1):
    ''' Calculates I(T,r_ij) with the formula that can be found in the
    report'''
    immediate_frustrations = []

    r_ij = get_relationship(agent0, agent1)
    personality = (1 - r_ij) / r_ij

    for agent in [agent0, agent1]:
        HARD_LIMITER = 0.1
        alloc_time = agent.allocation_times[-1] if agent.allocation_times
    [-1] < round(P.MAX_COORD_STEPS * HARD_LIMITER) else (round(P.
    MAX_COORD_STEPS * HARD_LIMITER) - 1)
        coord_penalty = (alloc_time / (P.MAX_COORD_STEPS / 10)) / \
                        (1 - (alloc_time / (P.MAX_COORD_STEPS / 10)))

        immediate_frustrations.append(P.MAX_H * (1 - math.exp(-P.BETA *
    personality * coord_penalty)))
```

```
    return immediate_frustrations[0], immediate_frustrations[1]
```

Task

```
# "A task consists of actions in such a way that for every action \
# exactly one skill is required to perform this action."

class Action:
    def __init__(self, _id, skill_id, duration, completion):
        self._id = _id
        self.skill_id = skill_id
        self.duration = duration
        self.completion = completion

class Task:
    def __init__(self, _id = -1, json_task = None):
    # json_task is the task as loaded directly from the input json file
        self._id = _id

        self.actions = [Action(action['id'], action['skill_id'], action['
    duration'], 0)
                        for action in json_task['actions']] if json_task
    != None else []

    def __str__(self):
        actions_str = ''
        for action in self.actions:
            actions_str += '\tAction ' + str(action._id) + \
                           ' | Skill ID ' + str(action.skill_id) + \
                           ' | Duration ' + str(action.duration) + \
                           ' | Completion ' + str(action.completion) + '\
    n'

        return '--- TASK ' + str(self._id) + ' ---\n' + actions_str
```

Workplace

```
import numpy as np
import json
import matplotlib.pyplot as plt
from plotly.offline import download_plotlyjs, init_notebook_mode, plot,
    iplot
import plotly.graph_objs as go

from skill import Skill
from agent import Agent, choose_agent
from task import Task
from timeline import Timeline, Event
```

```
import my_parameters as P

# Useful if you need to print JSON:
# from pprint import pprint

class Workplace:
    # ---------- INITIALISATION ----------

    def __init__(self, file=None, verbose=False):
        # Create an empty workplace
        self.agents = []
        self.completed_tasks = []
        self.current_task = None
        self.tasks_todo = []
        self.time = 0
        self.timeline = Timeline() # list of TimePoints
        self.coordination_times = {}
        self.Tperf = {}

        self.verbose = verbose

        if file:
            print("Reading from input file " + file + "...\n")
            self.parse_json(file, verbose)



    def parse_json(self, filename, verbose = False):
        ''' reads json file and loads agents, tasks and parameters'''
        with open(filename) as f:
            data = json.load(f)
        for idx, agent in enumerate(data['agents'], verbose):
            self.add_agent(idx, agent, verbose = verbose)
        for idx, task in enumerate(data['tasks']):
            self.add_task(idx, task)

        self.import_parameters(data['parameters'])

    def add_agent(self, idx, agent, verbose = False):
        skills = [Skill(_id = skill['id'],
                        exp = skill['exp'],
                        mot = skill['mot'])
                  for skill in agent['skillset']]

        mbti = agent['mbti'] if 'mbti' in agent else None
        initial_frustration = agent['initial_frustration'] if '
    initial_frustration' in agent else None

        self.agents.append(Agent(_id = idx, mbti = mbti,
                                 initial_frustration = initial_frustration
    ,
```

```
                                skillset = skills,
                                verbose=verbose))

    def add_task(self, idx, task):
        self.tasks_todo.append(Task(_id = idx, json_task = task))

    def import_parameters(self, params):
        ''' Loads all parameters from dictionary that comes from json file
    '''
        if 'task_unit_duration' in params:
            P.TASK_UNIT_DURATION = params['task_unit_duration']
        if 'alpha_e' in params:
            P.ALPHA_E = params['alpha_e']
        if 'alpha_m' in params:
            P.ALPHA_M = params['alpha_m']
        if 'alpha_f' in params:
            P.ALPHA_F = params['alpha_f']
        if 'beta' in params:
            P.BETA = params['beta']
        if 'lam_learn' in params:
            P.LAM_LEARN = params['lam_learn']
        if 'lam_motiv' in params:
            P.LAM_MOTIV = params['lam_motiv']
        if 'mu_learn' in params:
            P.MU_LEARN = params['mu_learn']
        if 'mu_motiv' in params:
            P.MU_MOTIV = params['mu_motiv']
        if 'th_e' in params:
            P.TH_E = params['th_e']
        if 'th_m' in params:
            P.TH_M = params['th_m']
        if 'max_e' in params:
            P.MAX_E = params['max_e']
        if 'max_m' in params:
            P.MAX_M = params['max_m']
        if 'max_h' in params:
            P.MAX_H = params['max_h']
        if 'excite' in params:
            P.EXCITE = params['excite']
        if 'inhibit' in params:
            P.INHIBIT = params['inhibit']

        # Normalise alphas
        if P.ALPHA_E + P.ALPHA_F + P.ALPHA_M != 1:
            factor = 1 / (P.ALPHA_E + P.ALPHA_F + P.ALPHA_M)
            P.ALPHA_E *= factor
            P.ALPHA_F *= factor
            P.ALPHA_M *= factor

    # ---------- TASK PROCESSING ----------
```

```
def process_tasks(self):
    ''' Just a while loop that processes all the tasks in another
function'''
    # While there is work to do...
    while len(self.tasks_todo) > 0:
        # Tasks are handled one at a time
        self.current_task = self.tasks_todo.pop(0)

        self.process_current_task()

        if self.verbose:
            print('Processed task:\n' + str(self.current_task) + '\n')

        self.completed_tasks.append(self.current_task)
        self.current_task = None

def process_current_task(self):
    ''' Processes current tasks one by one. Called by process_tasks()
'''

    # Repeat action assignment until all actions have been completed
    while True:
        # Assign agents to each of the actions
        actions_to_process = [choose_agent(self, action) for action in
 self.current_task.actions \
                              if action.completion < action.duration]

        if len(actions_to_process) == 0:
            break

        assignments, allocation_times, skill_ids, action_ids = zip(*
actions_to_process)
        self.coordination_times[self.time] = sum(allocation_times)

        t_perfs = [agent.calculate_performance_time(skill_ids,
assignments, self.time)
                   for agent in self.agents]
        self.Tperf[self.time] = max(t_perfs) + self.coordination_times
[self.time]

        # ~ HOUSEKEEPING ~
        for agent in self.agents:
            agent.flush_prev_act(assignments, skill_ids)            #
Clear internal variables related to previous task
            agent.update_memory()                                   #
Update expertise and motivation

        # Update current actions for all agents
        for i, assignment in enumerate(assignments):
            self.agents[assignment].current_action.append(
                {
```

```
                    'task': self.current_task._id,
                    'action': action_ids[i],
                    'start_time': self.time
                }
            )

            self.timeline.add_event(Event(start_time = self.time, \
                                          duration = 1,
          # Constant, for now
                                          task_id = self.
current_task._id, \
                                          action_id = action_ids[i],
 \
                                          agent_id = assignment, \
                                          ))

        # ~ END HOUSEKEEPING ~

        self.time += 1

# ———————— GETTERS ————————

def get_sum_perf_time(self):
    return sum(self.Tperf.values())

# ———————— PRINTING ————————

def plot_skills(self, agent):
    ''' Plots the expertise of two agents as a function of number of
cycles'''
    y1 = np.round(np.array(agent.skillset[0].expertise))
    y2 = np.round(np.array(agent.skillset[1].expertise))
    x = np.round(np.array(list(range(len(y1)))))

    y1 = [y if y > 0 else 0 for y in y1]
    y2 = [y if y > 0 else 0 for y in y2]

    trace1 = go.Scatter(
        x = x,
        y = y1,
        mode = 'lines+markers',
        name = 'Skill 1'
    )

    trace2 = go.Scatter(
        x = x,
        y = y2,
        mode = 'lines+markers',
        name = 'Skill 2'
    )
```

```
        layout = go.Layout(
            xaxis=dict(
                title='Cycles',
                titlefont=dict(
                    family='Arial, sans-serif',
                    size=24,
                    color='black'
                )
            ),
            yaxis=dict(
                title='Expertise',
                titlefont=dict(
                    family='Arial, sans-serif',
                    size=24,
                    color='black'
                )
            )
        )

        data = [trace1, trace2]

        fig = go.Figure(data=data, layout=layout)

        iplot(fig)

    def plot_skills_matplotlib(self, agent):
        ''' Plots the expertise of two agents as a function of number of
    cycles'''
        y1 = np.round(np.array(agent.skillset[0].expertise))
        y2 = np.round(np.array(agent.skillset[1].expertise))
        x = np.round(np.array(list(range(len(y1)))))

        y1 = [y if y > 0 else 0 for y in y1]
        y2 = [y if y > 0 else 0 for y in y2]

        fig = plt.figure()

        plt.plot(x, y1, '.-', x, y2, '.-')
        plt.xlabel('Cycles')
        plt.ylabel('Expertise')
        plt.title('Evolution of expertise: Agent ' + str(agent._id))
        plt.legend(['Skill 1', 'Skill 2'])
        plt.draw()

        return fig

    def plot_motivation(self, agent):
        ''' Plots the motivation of two agents as a function of #cycles'''
        y1 = np.round(np.array(agent.skillset[0].motivation))
        y2 = np.round(np.array(agent.skillset[1].motivation))
        x = np.round(np.array(list(range(len(y1)))))
```

```
        y1 = [y if y > 0 else 0 for y in y1]
        y2 = [y if y > 0 else 0 for y in y2]

        trace1 = go.Scatter(
            x = x,
            y = y1,
            mode = 'lines+markers',
            name = 'Skill 1'
        )

        trace2 = go.Scatter(
            x = x,
            y = y2,
            mode = 'lines+markers',
            name = 'Skill 2'
        )

        layout = go.Layout(
            xaxis=dict(
                title='Cycles',
                titlefont=dict(
                    family='Arial, sans-serif',
                    size=24,
                    color='black'
                )
            ),
            yaxis=dict(
                title='Motivation',
                titlefont=dict(
                    family='Arial, sans-serif',
                    size=24,
                    color='black'
                )
            )
        )

        data = [trace1, trace2]

        fig = go.Figure(data=data, layout=layout)

        iplot(fig)

    def plot_frustration(self):
        ''' Plots frustration of two agents as a function of #cycles'''
        y0 = np.array(self.agents[0].frustration)
        y1 = np.array(self.agents[1].frustration)
        x = np.array(list(range(len(y1))))

        trace1 = go.Scatter(
            x = x,
```

```
            y = y0,
            mode = 'lines+markers',
            name = 'Agent 1'
        )

        trace2 = go.Scatter(
            x = x,
            y = y1,
            mode = 'lines+markers',
            name = 'Agent 2'
        )

        layout = go.Layout(
            xaxis=dict(
                title='Cycles',
                titlefont=dict(
                    family='Arial, sans-serif',
                    size=24,
                    color='black'
                )
            ),
            yaxis=dict(
                title='Frustration',
                titlefont=dict(
                    family='Arial, sans-serif',
                    size=24,
                    color='black'
                )
            ),
            showlegend=True
        )

        data = [trace1, trace2]

        fig = go.Figure(data=data, layout=layout)

        iplot(fig)

    def plot_frustration_matplotlib(self):
        ''' Plots frustration of two agents as a function of #cycles'''
        y0 = np.array(self.agents[0].frustration)
        y1 = np.array(self.agents[1].frustration)
        x = np.array(list(range(len(y1))))

        fig = plt.figure()

        plt.plot(x, y0, '.-', x, y1, '.-')
        plt.xlabel('Cycles')
        plt.ylabel('Frustration')
        plt.title('Frustration')
        plt.legend(['Agent 1', 'Agent 2'])
```

```
        plt.draw()

        return fig

    def plot_allocations(self):
        ''' Plots allocation time it took for every cycle'''
        y0 = np.array(self.agents[0].allocation_times)
        y1 = np.array(self.agents[1].allocation_times)
        x = np.array(list(range(len(y1))))

        # y1 = [y if y > 0 else 0 for y in y1]
        # y2 = [y if y > 0 else 0 for y in y2]

        trace1 = go.Scatter(
            x = x,
            y = y0,
            mode = 'lines+markers',
            name = 'Agent 1'
        )

        trace2 = go.Scatter(
            x = x,
            y = y1,
            mode = 'lines+markers',
            name = 'Agent 2'
        )

        layout = go.Layout(
            xaxis=dict(
                title='Cycles',
                titlefont=dict(
                    family='Arial, sans-serif',
                    size=24,
                    color='black'
                )
            ),
            yaxis=dict(
                title='Allocation time',
                titlefont=dict(
                    family='Arial, sans-serif',
                    size=24,
                    color='black'
                )
            )
        )

        data = [trace1, trace2]

        fig = go.Figure(data=data, layout=layout)

        iplot(fig)
```

```
    def plot_performance(self):
        ''' Plots performance times of the agents and of the whole system
    '''
        y = []
        y.append(np.round(np.array(list(self.Tperf.values()))))
        y.append(np.round(np.array(list(self.coordination_times.values())))
    )
        y.append(np.round(np.array(list(self.agents[0].performance_times.
    values()))))
        y.append(np.round(np.array(list(self.agents[1].performance_times.
    values()))))

        for _y in y:
            _y = [y if y > 0 else 0 for y in _y]

        x = np.array(list(range(len(y[0]))))

        names = [
            'System',
            'Coordination Time',
            'Agent 1',
            'Agent 2'
        ]

        data = [
                go.Scatter(
                    x = x,
                    y = _y,
                    mode = 'lines+markers',
                    name = names[i]
                ) for i, _y in enumerate(y)
               ]

        layout = go.Layout(
            xaxis=dict(
                title='Cycles',
                titlefont=dict(
                    family='Arial, sans-serif',
                    size=24,
                    color='black'
                )
            ),
            yaxis=dict(
                title='Performance Time',
                titlefont=dict(
                    family='Arial, sans-serif',
                    size=24,
                    color='black'
                )
            )
```

```
        )

        fig = go.Figure(data=data, layout=layout)
        iplot(fig)

    def plot_performance_matplotlib(self):
        ''' Plots performance times of the agents and of the whole system
    '''
        y = []
        y.append(np.round(np.array(list(self.Tperf.values()))))
        y.append(np.round(np.array(list(self.coordination_times.values())))
    )
        y.append(np.round(np.array(list(self.agents[0].performance_times.
    values()))))
        y.append(np.round(np.array(list(self.agents[1].performance_times.
    values()))))

        for _y in y:
            _y = [y if y > 0 else 0 for y in _y]

        x = np.array(list(range(len(y[0]))))

        fig = plt.figure()

        plt.plot(x, y[0], '.-', x, y[1], '.-', x, y[2], '.-', x, y[3], '.-
    ')
        plt.xlabel('Cycles')
        plt.ylabel('Time')
        plt.title('Performance')
        plt.legend(['System', 'Coordination Time', 'Agent 1', 'Agent 2'])
        plt.draw()

        return fig

    def print_parameters(self):
        print('task_unit_duration: ' + str(P.TASK_UNIT_DURATION))
        print('alpha_e: ' + str(P.ALPHA_E))
        print('alpha_m: ' + str(P.ALPHA_M))
        print('alpha_f: ' + str(P.ALPHA_F))
        print('beta: ' + str(P.BETA))
        print('lam_learn: ' + str(P.LAM_LEARN))
        print('lam_motiv: ' + str(P.LAM_MOTIV))
        print('mu_learn: ' + str(P.MU_LEARN))
        print('mu_motiv: ' + str(P.MU_MOTIV))
        print('th_e: ' + str(P.TH_E))
        print('th_m: ' + str(P.TH_M))
        print('max_e: ' + str(P.MAX_E))
        print('max_m: ' + str(P.MAX_M))
        print('max_h: ' + str(P.MAX_H))
        print('excite: ' + str(P.EXCITE))
        print('inhibit: ' + str(P.INHIBIT))
```

```
        print('\n')

        print('Maximum number of steps in coordination:' + str(P.
    MAX_COORD_STEPS))

        print('\n')

        # TODO - This can be made prettier
        mbti_types = ['ESTJ', 'ESTP', 'ESFJ', 'ESFP', 'ENTJ', 'ENTP', '
    ENFJ', 'ENFP', 'ISTJ', 'ISTP', 'ISFJ', 'ISFP', 'INTJ', 'INTP', 'INFJ',
     'INFP']

        print('MBTI Matrix:')
        print('\t'.join(mbti_types))
        for i in range(len(mbti_types)):
            print(mbti_types[i] + '\t')
            for j in range(len(mbti_types)):
                print(P.MBTI[i][j], end='\t')

    def print_history(self):
        ''' Debugging information: same as Gantt diagram'''
        for i in range(len(self.timeline)):
            print('---- Time Point ' + str(i) + ' ----')
            print(self.timeline.events[i])

    def agents_string(self):
        return 'Agents:\n' + '\n'.join(list(map(str, self.agents)))

    def tasks_string(self):
        return 'TASKS:\n\n' + '\n'.join(list(map(str, self.tasks_todo)))

    def print_current_state(self):
        ''' Printing for Debugging purposes '''
        # Print time stamp
        print("Time elapsed:")
        print(self.time, end='')
        print(" time units.\n")

        # Print Tasks: Completed, Current, To-Do
        print("Completed tasks:")
        for task in self.completed_tasks:
            print(task)

        print("Current task:")
        print(self.current_task)

        print("Future tasks:")
        for task in self.tasks_todo:
            print(task)
```

```
        # Print Agents: List, Current Engagement...
        print("Currently employed agents:")
        for agent in self.agents:
            print(agent)

        # Print history of completed actions
        print("History:")
        self.timeline.plot_gantt()
```