Winter 12-25-2013

Understanding Human Learning Using a Multiagent Based Unified Learning Model Simulation

Vlad T. Chiriacescu
University of Nebraska-Lincoln, vlad_chiriacescu@yahoo.com

UNDERSTANDING HUMAN LEARNING USING A MULTIAGENT BASED UNIFIED LEARNING MODEL SIMULATION

by

Vlad T. Chiriacescu

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Leen-Kiat Soh

Lincoln, Nebraska

December, 2013

UNDERSTANDING HUMAN LEARNING USING A MULTIAGENT BASED UNIFIED LEARNING MODEL SIMULATION

Vlad Teodor Chiriacescu, M.S.

University of Nebraska, 2013

Adviser: Leen-Kiat Soh

Within cognitive science, computational modeling based on cognitive architectures has been an important approach to addressing questions of human cognition and learning. Modeling issues such as limited expressivity in representing knowledge and lack of appropriate selection of model structure represent a challenge for existing architectures. Furthermore, latest research shows that the concepts of long-term memory, motivation and working memory are critical cognitive aspects but a unifying cognitive paradigm integrating those concepts hasn't been previously achieved. Derived from a synthesis of neuroscience, cognitive science, psychology, and education, the Unified Learning Model (ULM) provides this integration by merging a statistical learning mechanism with a general learning architecture. Based on the ULM cognitive principles, this thesis presents a novel computational architecture called C-ULM that addresses the modeling issues outlined above and introduces a novel computational integration of long-term memory, motivation and working memory. C-ULM is implemented as a multi-agent simulation where the agent communication is grounded on the actions of teaching and learning. Both communication actions consist of two main phases: allocating working memory for teaching or learning and using the working memory content in order to update the agent's long-term memory. From a cognitive perspective, C-ULM provides a test of the viability of the learning mechanisms proposed in the ULM. In addition, as showcased by C-ULM

experiments, it offers insights that lead to a better understanding of the human learning mechanisms especially in the cases of long-term learning and problem solving where data from human subjects is generally not available. From a multi-agent perspective, it advances the literature by providing the first multi-agent based simulation that incorporates long-term memory, working memory, motivation and the relationships among them into an effective modeling framework. Furthermore, it offers the foundation for novel agent reasoning models and insights into modeling agent-to-agent knowledge transfer based on the principles of human learning and teaching processes.

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

1.1. Problem

One of the most important methods to address questions of human cognition and learning is the use of computational modeling based on cognitive architectures. Although there are several architectures mentioned in both cognitive science and computational modeling literatures, issues such as limited expressivity in representing knowledge and lack of appropriate selection of model structure represent a challenge for existing approaches. Furthermore, an integrative computational paradigm that puts together in a single model the concepts of long-term memory, motivation and working memory is needed. Latest research in cognitive related fields such as neuroscience and psychology suggests that these concepts are key components of human cognition (Shell et al. 2010). A unified computational model of long-term memory, motivation and working memory can thus greatly extend the type of cognitive research questions that can be addressed using computational simulations.

1.2. Related work

One of the most widely known computational models used for understanding human cognition is SOAR (Lehman et al. 2006). SOAR is a production based system that is geared towards problem-solving by the use of states and operators that make transitions among those states. Long-term memory, one of the fundamental components of SOAR, can store procedural, semantic and episodic knowledge. Working memory is responsible for triggering retrievals from long-term memory and consists of a hierarchy of states and their associated operators. One of the issues with SOAR is the fact that by representing knowledge as sets of rules, the system loses expressivity power in describing knowledge

that cannot be represented as rules. This issue drives the need for a new, connectionist based model of knowledge that is more expressive and thus can describe more types of information.

Computational modeling inspired from cognitive science is an active topic in the field of artificial intelligence or AI. A reference model within the AI field of multiagent systems (or MAS) is the cognitively inspired Belief-Desire-Intention architecture (Rao and Georgeff 1995). This architecture is based on beliefs that represent information such as the likely state of the environment, desires that specify the possible agent objectives and intentions that represent actions or a sequence of actions taken in order to accomplish a certain objective. One of the problems that can arise in BDI agents happens when the number of beliefs, intentions and desires is very large and the issue of selecting the most important subset for the subsequent steps arises. This problem creates the need for a meta-reasoning model that has the power to filter a large set of beliefs, desires and intentions. Such a feat can be realized with an integrative model that uses working memory and motivation as a guided filtering mechanism.

A recent multiagent model incorporates the idea of motivation and motivation profiles in agents (Merrick 2011; Shafi et al. 2012; Hardhienata et al. 2012). In those works, the authors develop three motivation profiles inspired from the Atkinson's Risk Taking Model (RTM). Those profiles indicate how much an agent is inclined to pursue high-risk tasks carrying high reward upon completion. The main issue with these types of models is that they only take into account the risk involved in attempting a task and do not relate to what an agent knows about the environment and existing tasks. Thus, there is a need for a more comprehensive motivation model that comprises of both intrinsic motivation

given by the agent long-term knowledge and extrinsic motivation given by the risks involved in attempting tasks. Such a comprehensive model of motivation can be achieved through the use of an integrative paradigm that models together motivation and knowledge stored in long-term memory by laying out the interactions taking place between them. One of the intersections between cognitive science and MAS fields is given by multiagent-based classroom simulations (Sklar and Davies 2005; Spoelstra and Sklar 2008). These works present a model of teaching and learning and as compared to SOAR and CYC they use graphs for knowledge representation. Those models outline a sequence of teaching and learning stages that have to be completed in order for student agents to perform the act of learning. However, in those works, the concept of motivation is a model parameter and it is not strongly grounded in cognitive science principles. This issue leads to the need for a comprehensive model that integrates motivation with long-term knowledge and working memory so that the resulting computational model is deeply rooted in cognitive science.

1.3. Proposed solution

Our work attempts to resolve the issues mentioned above by creating a connectionist model (called C-ULM) based on the principles of the Unified Learning Model or ULM (Shell et al. 2010). The ULM is a comprehensive learning theory that was developed from a synthesis of research in cognitive science, psychology and education. ULM has begun to influence thinking and practice in fields such as scholarship of teaching and learning (Wilson-Doenges and Gurung 2013), situated cognition (Durning and Artino 2011), pedagogy (Nebesniak 2013), cognitive function (Wasserman 2012), and computer simulation (Khandaker and Soh 2011).

Learning in ULM results from the interaction of three cognitive components: long-term memory, working memory, and motivation. Specifically, as derived from the principles of neuron synaptic connection strengthening and weakening, learning results from attention, repetition, and connection. Attention to information in working memory is necessary as a precondition to learning. Unique to the ULM is the premise that attention in working memory is a process dependent on motivation. That is, we attend to information when we are motivated to attend. While automatic parallel processing is always occurring, ULM argues that learning requires motivated attention. In ULM, knowledge is built when distinct pieces of information that are held simultaneously in working memory are connected and stored as chunks in long-term memory. The connections in these chunks continue to strengthen or decay depending on repetition due to knowledge retrieval via pattern matching and spreading activation throughout the chunk. As with findings in neural studies (Turk-Browne et al. 2008), this repetition causes knowledge chunks to ultimately reflect statistical regularities present in the information.

We have developed a MAS simulation in which each single agent is based on the C-ULM model. Specifically, the architecture of C-ULM can be summarized as follows:

- Each single agent has a cognitive architecture that consists of the three main ULM components: long-term memory, motivation and working memory
- Long-term memory (or LTM) is represented as an undirected, weighted graph where nodes indicate knowledge concepts and weighted edges—with a certainty measure on each weight—indicate a quantified connection between two concepts.

- Motivation is computed for each concept and is a function of the certainty that an agent has towards the weights for connections involving the analyzed concept.
- Working memory (or WM) is the buffer that is filled with units of information. We look into two types of units: singleton concepts and concept chunks (i.e., groups of connected concepts)
- Agent communication is grounded on the actions of teaching and learning and has at its core algorithms that perform the processes of (1) allocating working memory for teaching and learning and (2) using the working memory content to update the long-term knowledge of a learner or a teacher.
- A feature of the learning process is represented by the spread activation factor, which guides how the certainty for the weights of all connections reachable from a starting connection is to be updated. The amount of change in certainty for a connection is inversely proportional to the distance between this connection and the starting connection.
- In our simulation, knowledge decay (or, simply put, forgetting) is triggered when connections do not enter working memory for a given number of simulation time steps. The decay consists in increasing the uncertainty for the involved LTM connection weights.
- A task is represented similarly as long-term memory but without a certainty measure on the weights of the connections between its concepts. In each simulation time step, every agent attempts to solve one of the available tasks.

1.4. Contributions

Our contributions can be considered from two perspectives. Theoretically, the C-ULM provides a test of the viability of the learning mechanisms proposed in the ULM. While an operative computational simulation cannot prove that the underlying theory is correct, the ability to derive an operative computational simulation provides evidence for the plausibility of the theory. From the cognitive modeling perspective, C-ULM advances the literature by providing the first multi-agent based simulation that incorporates long-term memory, working memory, motivation and the relationships among them into an operative modeling framework. Additionally, from the agent perspective, C-ULM could benefit agent research at two levels. First, the modeling of individual agent reasoning can potentially be improved by the functions and relationships between long-term memory, motivation and working memory represented in C-ULM. Second, C-ULM can potentially improve the modeling of agent-to-agent knowledge transfer based on the principles of human teaching and learning processes.

From a technical point of view, our contributions can be summarized into the following aspects. First, we designed C-ULM based on the architecture and principles of the ULM model. Second, we implemented C-ULM in a multiagent simulation by using the Repast framework (North et al. 2006). Another design and implementation contribution has been made by the addition of the chunking mechanism to the already existing framework. Lastly, an important contribution has been made by designing, running and analyzing several experiments. We specifically analyzed the impact that various system parameters and the chunking mechanism have on the learning behavior of the multiagent system and also their effect on the system performance at solving tasks.

We would like to acknowledge the important contributions made by Derrick Lam and Ziyang Lin to the model design, implementation and initial experiments. Specifically, we would like to thank Derrick Lam for his contribution to model design, for starting the Java class design, for implementation contributions to the classes for motivation, concept and task representation and also for his contributions to implementing the main simulation class. Furthermore, we would like to thank Ziyang Lin for his contribution to model design and for his implementation contributions to the class for long-term memory representation.

1.5. Overview

In chapter 2 we present the Unified Learning Model and related works from cognitive science and multiagent fields. In chapter 3, we describe the architecture of a C-ULM based agent and the agent communication protocol. Furthermore, we present the most important algorithms pertaining to agent learning, agent teaching, task representation and agent task attempt. In chapter 4 we present the UML class diagram describing the class architecture of the implemented simulation. Furthermore, we present the general functionality and the most important methods of each class. The results obtained through a variety of experiments are presented in Chapter 5. Finally, we conclude our work in Chapter 6 and offer directions for future research.

Chapter 2. BACKGROUND AND RELATED WORK

In this chapter, we start by describing the main features of the ULM model (section 2.1) and in the subsequent sections we present the models and architectures mentioned in the first chapter and draw some comparisons with the C-ULM model.

2.1. ULM Background

In the first part of this section we describe the 3 main components of the ULM model – working memory, long-term memory, and motivation (section 2.1.1). In the second part we present the three ULM learning principles and how they provide the interactions among the 3 ULM components (section 2.1.2). We conclude this section by presenting the main reasons for choosing ULM as the basic modeling architecture (section 2.1.3). Further, we describe the motivation for creating and designing the C-ULM model (section 2.1.4).

2.1.1. ULM Components

Central to the Unified Learning Model (ULM) is the idea that all learning takes place in three primary components: (1) **working memory** (WM) which receives and processes sensory information, (2) **long-term memory** which stores long-term knowledge and (3) **motivation** which guides the agent's attention. These components encompass the basic cognitive architecture of ULM and they are also the main components in C-ULM.

2.1.1.1. Working Memory

The first ULM component is working memory. It is the component that realizes learning and thus expands our existing knowledge. It contains a storage area for temporarily holding sensory input elements and knowledge retrieved from long-term memory. Furthermore, it contains a processing system that uses attention and other cognitive actions in order to operate on and change the content of temporarily stored elements. Working memory has a limited storage capacity (around 4 units or slots) and because of this is considered the bottleneck of the learning process.

2.1.1.2. Long-term memory

The second component, long-term memory (or LTM), refers to the type of memory that stores long-term knowledge. LTM results from groups of interconnected neurons that repeatedly fire among each other over the synapses that connect them. Neural synapses strengthen proportionally to the amount of firings of the involved neurons. The strengthening and weakening of neural synapses shapes the evolution of our knowledge.

There are two main types of long-term knowledge. The first type is episodic knowledge and it contains the information we have about our personal experiences. This type of knowledge is usually highly detailed with sensory information and it can also be strongly related to emotional content. Whenever it is associated to strong emotions, its content can remain almost unaltered for years or even decades. The second main type of knowledge is called semantic knowledge. This is the knowledge that doesn't relate to us and is taught at all educational levels. It can be classified into categories or domains such as mathematics, computer science, sociology and so on. It is also highly hierarchical since those broad domains can be further split into subdomains that can in turn be split into other subdomains (for example, artificial intelligence is a subdomain of computer science and machine learning is a subdomain of artificial intelligence). An important difference between semantic and episodic knowledge is that learning episodic knowledge doesn't require learning effort as compared to semantic knowledge that requires sustained effort. Semantic knowledge has two main types: procedural and declarative knowledge. Declarative knowledge can be further split into objective knowledge such as knowledge of objects and actions and symbolic knowledge such as the meaning of words, mathematical knowledge, explanation of natural phenomena and physics laws.

Procedural knowledge is knowledge that we use in order to take a sequence of actions that lead to a desired goal. It is usually represented as a sequence (or procedural chain) of conditional statements such as “If X happens, then I will do Y.”.

2.1.1.3. Motivation

The third component, motivation, is the psychological construct that determines humans to put effort into achieving desired outcomes. While those outcomes can be anything that relates to an individual that tries to attain them, the ULM model focuses on learning semantic knowledge as an outcome. Thus, the model focuses on what drives people to put effort to attend new information and use their working memory capacity and processing system in order to expand and refine their existing semantic knowledge.

There are various types of motivators that lead people to behave in a certain way but only the cognitive and emotional motivators drive them to achieve learning goals. One of the basic cognitive motivators is goals. Goals are what drives working memory processing. Without a purpose seen as having a value for the individual, putting forth mental effort doesn’t make sense. The value of a goal is based on knowledge. For example, we learned that being sociable and friendly helps us in relating to other people. Thus, we might have a high value on the goal of becoming more sociable and friendly. Emotions are also an important motivator for learning and they can direct working memory to thoughts that trigger new emotions. According to the emotional state of the individual, those thoughts can trigger positive, negative or a mix of new emotional outcome.

2.1.2. ULM learning principles

The interactions taking place between working memory, long-term memory and motivation lead to the three ULM principles of learning.

2.1.2.1. Learning principle 1: Working Memory Allocation

The first principle states that **learning is a product of working memory allocation**. There are two required conditions that need to be met so that working memory is allocated to a given task. The first condition is to have enough available working memory in order to temporarily store elements such as sensory inputs or retrieved memories. The second condition is to direct processing toward the temporarily stored elements. In ULM, there are three learning rules that explain how working memory allocation leads to learning.

The first rule states that new learning requires attention. Studies showed that when working memory attends and processes temporarily stored elements the chance of triggering the process of long-term potentiation (or LTP) increases. This is a process that realizes the transfer of the processed information into long-term memory (or LTM).

The second rule states that learning requires repetition. Without repetition of certain information, a single act of attending to it is unlikely to lead to a change in long-term memory and consequently no learning occurs. However, repetitively focusing attention on the same information increases the chances of effective learning. Furthermore, if this repetition is done over an extended time period, information retrieval from long-term memory will become faster and more consistent.

The third rule states that learning is about connections. During working memory processing new connections between various pieces of information can be created and

existing connections between knowledge elements can be broken down. The ability of working memory processing to connect in a flexible manner various pieces of knowledge leads to integrated knowledge structures such as concepts, propositional networks or production systems. One important aspect of this connecting ability is that it expands working memory capacity by creating memory chunks. A memory chunk is an interconnected knowledge unit that occupies only one slot of working memory capacity.

2.1.2.2. Learning principle 2: The Prior Knowledge Effect

The second principle states that **working memory's capacity for allocation is affected by prior knowledge.** This influence of existing long-term knowledge on working memory allocation is due to the interaction between LTM and working memory during the learning process.

When no memory chunks containing knowledge related to the new input exist in LTM, the input has to be attended and processed in working memory for multiple times. Furthermore, repeated retrieval from LTM will eventually result into transforming the attended input into usable knowledge. Without such repeated retrieval, knowledge in LTM will weaken and decay. This entire process requires a considerable amount of effort.

However, if at the time of attending the new input, there is already a long-term knowledge chunk stored into LTM, pattern match retrieval of the chunk will be triggered. If the entire input is already in the chunk, then the corresponding neural synapses are further strengthened. If only a part of the input is contained in the chunk, the new part will be appended to the chunk. In this case, working memory allocation requires much

less effort and is more efficient because storage doesn't depend on immediate working memory processing or long-term potentiation.

2.1.2.3. Learning principle 3: Working Memory and Motivation

Finally, the third principle is: **Working memory allocation is directed by motivation**. A large array of sensory inputs, knowledge chunks and procedural chains can be potential working memory elements at any given time. Due to the working memory capacity limits, there is a need to reduce this large collection of information items to a manageable number. Furthermore, this selection process is optimized with respect to our goals. Thus, goals restrict allocation only to those knowledge chunks and sensory inputs that are relevant for the goal. Furthermore, goal values differentiate between competing goals so that the higher valued goals are more frequently selected than the lower valued goals. Finally, positive emotions and interest in learning sustain the prolonged and repeated allocation of working memory towards the knowledge units perceived to lead to the selected goals.

2.1.3. Motivation for using ULM as the basic model

We decided to use ULM as the basic model for our simulation since it is a recent model that provides a synthesis of the latest findings in cognitive science and psychology related fields. Furthermore, ULM is appealing from a computational standpoint since it argues that all the complex learning phenomena happen due to the relatively simple interactions taking place between the 3 key components, that is working memory, long-term memory and motivation. In addition, to the best of our knowledge, this is the first comprehensive model that specifies how motivation directly influences the underlying cognitive processes of working memory and learning (Shell et al. 2010).

2.1.4. Motivation for creating C-ULM

From a cognitive science perspective, we created the C-ULM model in order to address the type of cognitive science related questions that need a computational simulation. For example, we have investigated the issue of working memory capacity and learning efficiency in the case of learning by chunking and learning without chunking. Another reason for creating C-ULM was to provide to the cognitive research community the first model that integrates LTM, motivation and working memory into a single operative framework.

From a multi-agent point of view, we created C-ULM in order to provide to the AI community a novel way of agent knowledge sharing through the ULM-inspired processes of learning and teaching. As guided by the ULM theoretical framework, those processes emerge as complex mental phenomena that result from the interactions taking place between the 3 main components. Furthermore, we believe that individual agent reasoning can also be improved by modeling together long-term memory, motivation and working memory and using them as an agent reasoning framework. Finally, we were driven to create a comprehensive model for agent motivation that combines internal motivation based on agent knowledge and external motivation based on expected rewards for solving tasks.

2.2. Relation to Cognitive Informatics

One of the most relevant related fields to C-ULM is the field of cognitive informatics. Below we present some works done in this field and how they relate to the C-ULM model. Of note, the paragraphs below have been included in a C-ULM research paper

that was submitted for publication to the International Journal of Cognitive Informatics and Natural Intelligence (IJCINI).

2.2.1. Formal Knowledge Representation System (FKRS)

One particularly relevant work in cognitive informatics is that by Tian, Wang, Gavrilova, and Ruhe (2011). They describe and propose a formal knowledge representation system (FKRS) based on the object-attribute-relation (OAR) model and its concept algebra (Wang, Tian & Hu, 2011). It uses as a linguistic base the well-known WordNet and is comprised of three main components: concept formation, conceptual knowledge representation and knowledge visualization. FKRS and OAR are examples of semantic level symbolic models (McClelland, 2009). They model knowledge in linguistic and language terms. The C-ULM operates at a level more similar to a connectionist model. The learning processes of the ULM that are modeled in C-ULM are not language or symbol based. They reflect statistical Hebbian neural learning process. These are more elemental than symbolic language. As discussed by McClelland (2009), these approaches differ but are complementary rather than antagonistic.

The FKRS can prove helpful in obtaining a more structured representation of the knowledge that is being learned. The ULM argues that knowledge in the brain comes to reflect statistical regularities in the information being learned. FKRS provides a rigorous description of the properties of concepts. This could provide guidance as to what statistical regularities exist in the knowledge by describing specific attributes and objects pertaining to a given concept. An important connection can be established between the OAR model and the C-ULM knowledge representation. In the OAR model, there are networks of objects, attributes and relation that connect objects and attributes forming

networks of objects and attributes. Of note, those objects and attributes are seen as partially connected (and not fully connected) in a similar fashion as knowledge is represented in C-ULM. Thus, the C-ULM concepts could correspond to OAR's objects and the relations between them represented by C-ULM's connections. Furthermore, C-ULM allows for a large variety of relations given the relative connection strength indicated by the connection weight value. As future work, attributes can be incorporated within C-ULM concepts or as an alternative, concepts can represent attributes that form specific chunks that in turn represent corresponding OAR objects.

2.2.2. Layered Reference Model of the Brain (LRMB)

Another important cognitive informatics connection can be made between the C-ULM architecture and the layered reference model of the brain (LRMB) (Wang & Chiew, 2010; Wang, Wang, Patel, & Patel, 2006). The LRMB is a formal, layered model of cognitive processes in the brain. In this model, the brain has 7 seven abstraction layers of processes with primitive processes operating at the sub-conscious level and higher cognitive functions such as learning, problem solving and decision making operating at the conscious level and relying on the mechanisms of previous levels. The distinctions between sub-conscious and conscious levels mirror other recent formulations such as Kahneman's (2011) System 1 and System 2. The LRMB is a process oriented model. The ULM (Shell et al., 2010) is a knowledge oriented model. In the ULM, all process distinctions are seen as distinctions in knowledge with knowledge including all forms of data contained in the brain from sensory information to higher-order skills. Although the ULM recognizes that different brain areas, such as sensory memory modules or the motor cortex, have different outputs similar to the abstraction layers of the LRMB, the ULM

holds that within the range of what that particular area is capable of outputting, its outputs are the results of neural plasticity learned via the ULM principles. From the perspective of the ULM, the distinctions represented in the LRMB reflect differences in the types of knowledge that different parts of the brain/cognitive system are encoding. Sensory memory modules are encoding statistical regularities in low level data associated with the sense. Language modules are encoding statistical regularities in the language. The functional model of the LRMB reflects a general information processing approach to cognition. The ULM shares this approach. However, the ULM merges the LRMB functions of short-term memory and natural intelligence (NI-OS and NI-APP) into a single working memory consistent with much recent thinking (Saults & Cowan, 2007). The ULM also merges all sensory, motor, and general cognitive functions into a single long-term memory. This makes the C-ULM a much simpler computational model than LRMB. It may be that the observable outputs of the natural intelligence of the brain are better modeled by something like the LRMB and the acquisition of the knowledge that produces that intelligence is better modeled by something like C-ULM. Whether this is a fruitful approach needs to be established in future research.

Because the C-ULM architecture reflects these ULM consolidations of knowledge and working memory, many LRMB levels and processes are represented within the C-ULM. For example, Layer 1, Sensation, is represented by concepts received by a learning agent in C-ULM. Those stimuli enter the second layer through the short-term memory (STM), which is akin to the working memory in C-ULM. Layer 4, Perception, has two important modules: attention and emotions. The first module, attention is modeled within C-ULM by the use of the awareness threshold that filters what enters into

short term-memory. The second module, emotions, is modeled to a certain degree in C-ULM by the motivation concept and motivation scores for concepts. Furthermore, as meta-cognition processes, we model the search module of Layer 5 (Meta-Cognition) when we do breadth-first search to find the appropriate concepts that will be retrieved for teaching or updated for learning. The memorize module of Layer 5 is further characteristically represented by the acquisition of new connections and also by the update of connection weights in C-ULM. Furthermore, the C-ULM's chunking process—an important process in ULM—leads to an ever increasing efficientization of the way STM is being used in the learning process. A chunk represents a network of concepts that are more related to each other than to other concepts. From a knowledge representation point of view, the chunk is a higher, more abstract level of knowledge that is a synthesis of individual concepts. Thus the C-ULM's concept of chunking can be related to the LRMB's modules of Abstraction and Synthesis found at Layer 5 (Meta-cognition) and Layer 6 (Meta-inference). C-ULM also models the interaction happening at the top LRMB layer, between the learning and the problem solving processes. Thus, more learning steps enhance problem solving and in turn, solved problems lead to new learning experiences (coming from the knowledge obtained by solving the task).

There are additional parallels between C-ULM and the LRMB based problem solving model proposed by Wang and Chiew (2010). Within C-ULM, problem solving happens through the process of attempting and solving a task. Just as in Wang and Chiew (2010), solving a problem requires a set of representation and search operations. Within C-ULM, the representation operations are those operations that alter the long-term memory (LTM) structure of an agent (acquiring new connections and in the latest version, also pruning

extremely unused connections). On the other hand, the search operations are those operations that, taking into account agent knowledge but also task feedback update both the LTM structure and connection weight values. These series of structure and weight updates are essentially searching through the problem space in order to find the suitable configuration of connections and weights that leads to solving the task.

2.2.3. Wang's memorization model

In relation to the cognitive informatics model of memorization proposed by Wang (2009b), the C-ULM shares a focus on repetition and connection or relation as the primary learning processes. As noted previously, the OAR model that Wang uses operates at a symbolic level and the C-ULM is a statistical based model. Also, the C-ULM in merging short-term memory into a more general working memory and merging various Sensory Buffer Memory (SBM), Conscious-Status Memory (CSM), Long-Term Memory (LTM), and Action-Buffer Memory (ABM) from Wang into a single Long-Term Memory. Wang's memorization model is intended to apply to one specific type of cognitive process from the LRMB model. The C-ULM is meant to apply to all learning of all of the knowledge included in the LRMB model, making C-ULM a more general statement of how knowledge is acquired across all brain and cognitive components.

2.2.4. Emotional regulation model

Recent work in cognitive informatics has focused on motivational regulators that perform roles similar to C-ULM motivators. Rosales, Jaime, and Ramos (2013) introduced an emotional regulation model having two main components, i.e., emotional response and emotional regulation. When the virtual agents respond to a risk situation, their emotions could influence the decision-making process adversely. The emotional

regulation process helps them to ignore, regulate or use their emotions. The regulation component consists of two modules—namely, a reappraisal module and a suppression module. When a virtual agent's average of perceived behavior and required behavior is the same as the expressed behavior indicating "emotional response", the suppression algorithm basically switches a virtual agent's attention and ignores the highly affective objects—where each object has an emotional memory, elicited in the agent that stored the object in the first place, for example—in the scene.

2.2.5. Moral decision making model (MDM)

Cervantes et al. (2013) introduced a moral decision making (MDM) model for agents based on ethical, moral, and religious principles as well as on individuals' beliefs of right and wrong, feelings, and emotions. The computational process of this model consists of 3 phases: (1) assessment of options including filtering using a set of moral and ethical rules based on experiences, prejudices, emotions, cost-benefit analysis and moral evaluation, (2) execution of the selected option by which it is sent to the working memory and new execution plans are generated in a planning process, and (3) outcome evaluation where the executed actions are further evaluated. This MDM model provides a potential set of additional motivational considerations that could be incorporated into C-ULM. Clearly, human teaching and learning have moral and ethical dimensions. Learning and teaching of C-ULM could consider moral and ethical rules in decisions about what to teach and what not to teach, or what to learn and what not to learn. The above 3-step computational process could potentially inform C-ULM in deciding what learning and teaching tasks to perform, evaluating the outcomes, and reinforcing the decision. C-

ULM only considers the knowledge being shared in a teaching interaction and the knowledge required for task completion.

2.2.6. C-ULM motivators

In the ULM, Shell et al. (2010) propose that all motivators impact learning via motivation and attention direction in working memory. Other processes like morals, ethics, and emotions clearly impact human behavior including learning. Currently, C-ULM only models two of these motivators: self-efficacy and expectancy/task reward. These were chosen because they have consistently been found to be among the strongest motivators in prior studies (Schunk & Zimmerman, 2008; Shell et al., 2010). Also, as discussed in Shell et al. (2010), self-efficacy and expectancy/task reward have the most clear neurological foundations of the available motivational constructs. But, future work needs to expand the scope of motivational influences to include the types of moral and emotional factors noted by Cervantes et al. (2013) and Rosales et al. (2013).

2.3. SOAR architecture

2.3.1. Overview

SOAR is a cognitive architecture that embeds a set of mechanisms and structures that process domain-based information in order to produce appropriate behavior (Lehman et al. 2006).

The main components of SOAR are:

- **States and Operators** are the basic structures supported by the architecture. The states contain all the information about the present situation. They describe what are the current goals and problem spaces of the cognitive system. Operators are used by the

system in order to traverse problem spaces. By applying an operator to a current state, the system moves to another state determined by the operator.

- **Working Memory (WM)** contains the hierarchy of states and their associated operators. Working memory contents trigger retrieval from long-term memory (LTM).
- **Long-term Memory (LTM)** is the repository for domain content that is processed by the architecture to produce behavior. SOAR supports three LTM representations: procedural knowledge encoded as rules, semantic knowledge encoded as declarative structures, and episodic knowledge encoded as episodes. Procedural memory is accessed automatically during the decision cycle, while the semantic and episodic memories are accessed deliberately through the creation of specific cues in working memory. SOAR does not access and modify the LTM content directly. Instead, the LTM is changed indirectly through the use of working memory retrievals.
- **The Perception/Motor Interface** is the mechanism used by SOAR to create a bidirectional mapping between the external world and the internal working memory representation.
- **The Decision Cycle** is the basic cognitive process of the SOAR architecture. The decision cycle comprises three phases. The first one, called the elaboration phase, involves parallel access to LTM to elaborate the state, suggests new operators, and evaluates the operators. The second phase, called the decision phase, contains the procedure that interprets the language of operator preferences. The result of this procedure is either a change to the selected operator or an impasse if the preferences are incomplete or in conflict. In the third phase, called the application phase, existing rules fire in order to modify the current state.

- **Impasses** occur when there is a lack of necessary knowledge and represent an opportunity for learning. When an impasse arises, the architecture creates a new substate whose goal is to resolve the impasse. In this manner, impasses impose a goal/substate hierarchy on the working memory contexts.
- **Learning Mechanisms:** SOAR has four architectural learning mechanisms. The main and most developed mechanism is chunking. Through this mechanism, the cognitive system creates new rules in LTM whenever results are generated from an impasse. This mode of learning speeds up performance and moves knowledge retrieved in a substate up to a state where it can be reused in the future, thus preventing impasses in similar future situations. The second type of learning mechanism, called reinforcement learning, adjusts the values of preferences for operators. The last two learning mechanisms are episodic and semantic learning. The former stores a history of experiences, while the latter captures more abstract declarative statements.

2.3.2. Comparison with C-ULM

One of the main differences between SOAR and C-ULM is how the chunking process works in the two models. Chunking in SOAR consists of the accretion of new condition-based rules. This accretion is made on the basis of existing knowledge. In contrast, in C-ULM chunking results in more concept nodes being connected. The resulting connection pattern, taken together with the information stored in the concept nodes leads to a much more general representation of information than the one based on rules. Thus, C-ULM is more general in the chunking mechanism than SOAR since C-ULM is not restricted to creating condition-based rules but any type of complex concept (chunk) using any type of connection pattern among its subconcepts.

Another difference related to the chunking mechanism is related to when chunking is triggered. In the SOAR architecture, chunking occurs only when impasses are formed in a given state/substate level. In C-ULM, the equivalent of an impasse is the failure of an agent to solve a task. In contrast to SOAR, C-ULM performs chunking independently of the outcome (success or failure) of attempting to solve a task. Instead, C-ULM chunking is dependent upon the content of the working memory. We believe that in this manner, learning by chunking is internally driven by its intricate connection to working memory processing as compared to being externally driven by the impasses that can arise due to attempting certain tasks.

One similarity between SOAR and C-ULM is the interaction between the long-term memory and working memory. Similar to SOAR, in C-ULM, the content of working memory decides upon what connections are retrieved from long-term memory and then updated as a result of the learning process.

Probably the main difference between the two systems is given by the fact that C-ULM is a connectionist model while SOAR is a production rule system. In this sense, C-ULM uses concepts, connection patterns and connection weights in order to represent information. Meanwhile, SOAR uses a hierarchy of states to represent information about the current situation. Among other types of information, those states describe rule-based information such as condition-based rules.

Regarding goals, SOAR is a goal-driven system that strives to attain concrete domain-related goals by learning and improving its behavior in order to reach those goals. In contrast, C-ULM is more focused on how to achieve the necessary knowledge that can be further used to model behavior and reach domain-related goals. In order to model the

achievement of necessary knowledge, the C-ULM model uses ‘training tasks’ that contain partial information regarding the problem domain. By attempting those training tasks, a C-ULM agent is reaching a learning-based goal and not a concrete, domain-related goal. As an example, a task in C-ULM could be learning how to throw the ball in a baseball game and a more complex task could be learning how to play baseball. In contrast, the goals of a SOAR based system would be the goal of actually displaying the appropriate baseball related behavior, e.g., throwing the ball appropriately given the external factors or other strategies that have to be used to win the game. From this point of view, the SOAR system integrates both learning and the resulting behavior while C-ULM focuses on modeling the learning mechanics.

Another difference is related to how the concept of reinforcement learning is used in the two cognitive architectures. SOAR uses reinforcement learning in order to adjust the values of preferences for operators. Those operators are used in order to ‘move’ from a system state to another or to a sub-state. By adjusting the values of preferences for operators, the SOAR system is only indirectly using reinforcement learning as a mode of learning. This is because once those operator preference values are set, learning by chunking is the learning mode that uses operators with the new values in order to create new rules. In contrast, in C-ULM, reinforcement learning is directly affecting learning by shortening the confusion intervals when a task has been solved and lengthening those intervals when an agent failed to solve a task. Taken together with the overall picture of a connectionist model, we believe that this is a slightly better inclusion of the idea of feedback based learning (or reinforcement learning) in a cognitive architecture.

Finally, episodic and semantic learning are related to two different types of knowledge that can be stored within a SOAR-based system. As compared to SOAR, C-ULM leverages the power of the connectionist model in order to create a cognitive architecture that doesn't discriminate between different types of knowledge. Simply put, different types of knowledge, such as the episodic and semantic ones, can be represented using a different set of concept nodes and connection patterns among those nodes. In this regard, due to its enhanced generality, we believe that the C-ULM approach makes one more step towards the ultimate goal of obtaining a unified theory of cognition (UTC).

2.4. Belief-Desire-Intention (BDI) architecture

2.4.1. Overview

The BDI architecture is one of the reference architectures for building multi-agent systems. Within this architecture, there is a clear distinction between the notion of belief as it is used in BDI agents and the notion of knowledge. Thus, beliefs represent only information held by the agents regarding the likely state of the environment. In contrast, knowledge is a much more general concept that can embed different types of information that were learned by an agent. In this regard, C-ULM uses the term knowledge in order to embed any type of information that can be learned by an agent.

The objectives of the BDI agents are represented by the desire component, which is seen as the motivational state of the system. In contrast, C-ULM makes a clear distinction between motivation, which is one of the 3 main components of the model, and objectives. In C-ULM an agent objectives are solving various tasks.

Within the BDI architecture, there is a balancing need between continually changing course of action in an ever-changing environment and continuing to execute a previously

selected action until its completion (Rao and Georgeff 1995). This balancing it's achieved through the use of the intention component. By representing the currently chosen course of action as the system's intentions, the action that is performed may change before the previous action has completed but the frequency of changes is reduced. This balancing problem between switching actions and continuing the same action doesn't appear in the C-ULM design. This is because in each time step, each agent performs two actions: a learning or a teaching action followed by a task attempt action. In contrast to the BDI intention component, the C-ULM motivation component is not involved in changing the action performed but rather in the process related to each of the 3 possible actions.

In BDI systems, only beliefs and intentions have explicit representation. Desires are transiently represented as a type of event. Goals are somewhat similar to desires but represent a certain level of agent commitment for achieving them. They essentially represent a partial state of the world which the agent has decided to attempt to achieve. The standard BDI architecture lacks specifications for goal representation and policies for maintaining goal consistency. However, the BDI-G architecture (Thangarajah et al. 2002) offers a framework in which goals are represented and logical rules pertaining to goals can be specified. Those rules of inference can assure that goals are consistent with each other.

2.4.2. Comparison with C-ULM

The main difference between the BDI and BDI-G architectures on one hand and the C-ULM architecture on the other hand is made by the existence of the working memory component in the C-ULM model. Supported by working memory and the chunking

process, a C-ULM based system decides how much spatial (or even time) windowing is necessary when updating and retrieving agent knowledge. For example, a simple task might require a few small chunks to be retrieved in working memory while a much more complex task might require retrieval of large knowledge chunks.

Thus, while the BDI and BDI-G models provide the foundation for agent reasoning, C-ULM provides a guided process allowing designers of agent systems to give an agent a meta-reasoning approach for focusing on a subset of beliefs, desires, intentions and goals that the agent is going to use in the next steps. In the case of long-term exposure of BDI/BDI-G agents to a given environment, their entire set of beliefs, desires, intentions and goals (BDIG) can grow very large and a filtering process becomes necessary in order for efficient operations to take place. In this sense, the working memory processing and the chunking mechanism of C-ULM provide a way of selecting the most important BDIG subset that would benefit the agent in the subsequent steps. We believe that those aspects position C-ULM as a meta-reasoning framework that balances the trade-off between exploration and exploitation, making it a key component of a complex multi-agent system.

2.5. Multiagent-based classroom simulations

2.5.1. SimEd simulation

2.5.1.1. Overview

The SimEd simulation is a multiagent-based classroom simulation. It simulates a learning environment that models behaviors and interactions of ‘teacher’ and ‘student’ agents (Sklar and Davies 2005).

The simulation comprises of several key components. The first component is the knowledge domain. This is represented using a directed weighted graph and each concept (node in the graph) has an index number associated. The notation for this graph is {C}.

Another key component is the teacher behavior model and it is presented in 3 types: the lecture model where the teacher continues to teach new concepts regardless of the student progress; the lecture-feedback model where the teacher repeats concepts based on feedback from the students and the tutorial model where the teacher reacts in a personalized manner for each student.

The student knowledge model is modeled as a weighted directed graph that is a subset of {C}.

The student behavior model comprises of 3 components. The first is aptitude – a value between 0 and 1 that is related to the concepts difficulty (for example, if s.aptitude < c.difficulty, then concept ‘c’ is considered hard by student s). The second is motivation – the motivational level is determined by the teacher’s choice of question (if question is hard – student feels challenged and motivated). The third student behavior component is emotion – the emotional level increases if student answers correctly a question and decreases if answers incorrectly.

The simulation also includes an assessment component with three attributes: attribute ‘progress’ indicates what concept number a student is learning at a certain time; attribute ‘question’ indicates what concept number a teacher is teaching at a certain time; attribute ‘done’ indicates whether a student has learned all the concepts or not (is 1 or 0).

A typical learning scene comprises of 6 steps. First, the teacher perceives. This involves the teacher setting the challenge based on its own motivational level. Second,

the teacher acts. This involves setting the value of question and presenting the concept associated to that question (concept C_q). Third, the student perceives the question and asks itself if the concept is hard or easy. If it is hard, it is more motivated to try to answer.

In the fourth step, the student acts. If both motivational and emotional levels are high then the student attempts to answer the question; if both are low, it doesn't attempt to answer. In the fifth step, the student reacts – his emotional level increases if question was answered correctly; otherwise, it decreases. Lastly, in the sixth step the teacher reacts and his emotional level increases or decreases according to the number of students that answered correctly their current questions.

2.5.1.2. Comparison with C-ULM

The first similarity between the two systems is given by the simulation environment, ie. both simulations model a classroom-based environment with two types of agents: students and teachers. Another similarity is given by the representation of the knowledge domain, ie. both simulations use weighted graphs for representing long-term knowledge (simEd) and long-term memory (C-ULM).

However, in the case of simEd, the knowledge is represented using directed weighted graphs. In the case of computational ULM, the knowledge is represented using undirected weighted graphs.

Another difference regarding long-term knowledge/memory representation is related to the use in C-ULM of a certainty measure associated with each existing connection between any two concepts. This certainty measure is the confusion interval associated to that connection. The confusion interval allows for a more finely tuned modeling of how much an agent knows about the relationship between two concepts. Thus, if an agent

knows little about a particular connection, then the corresponding confusion interval length will be rather large (closer to a value of 1) and if the agent knows more about the connection, then the confusion interval length will be smaller (closer to a value of 0).

Motivation is interpreted and used in different manner in the two models. In the simEd simulation, motivation depends on the difficulty of the question posed by the teacher. Thus, motivation depends only on factors external to the agent. In contrast, computational ULM has two factors that influence motivation: one is internal and depends on how much the agent knows about the connections incident to a given concept. The other is external and depends on what rewards the agent could obtain if it successfully solves tasks that contain the given concept. We believe this is a more accurate representation of the motivation component because we use the internal component that relies on agent knowledge (Shell et al. 2010). In turn, this is made possible by the type of modeling we use for representing knowledge – ie. the use of the confusion interval to fine tune how much the agent knows about a given connection.

In the simEd simulation working memory is not part of the proposed cognitive model, while in computational ULM, working memory is intricately linked to motivation since motivation is the one that drives working memory allocation (Shell et al. 2010). In turn, what is allocated in working memory dictates how the long-term memory (knowledge) is affected after each learning experience.

Furthermore, in the simEd simulation, the teacher teaches only one concept at a time for each individual student agent. According to the teacher behavior model used (lecture, lecture-feedback or tutorial model) the teacher goes on to teach the next concept. Concepts are thus taught in a serial manner. In contrast, in computational ULM, a teacher

agent can teach multiple concepts in each single time step. How many concepts or chunks a teacher can teach is governed by the working memory capacity of the teacher. The ability to teach multiple concepts in the same time step allows for many more learning scenarios and outcomes as compared to the simEd simulation.

2.5.2. Group learning simulation

2.5.2.1. Overview

Another multi-agent based classroom simulation is geared towards understanding group learning and the interactions between various parameters that describe the human learning process (Spoelstra and Sklar 2008). In this regard, knowledge domain is represented with directed weighted graphs just as in the SimEd simulation.

The process of learning is viewed as having 3 main stages: the ‘early’ stage when most of the new concept acquisition is done; the ‘intermediate’ stage when further associations are made between concepts learned in the first stage and errors in understanding from the first stage are overcome. The last stage is called the ‘autonomous’ stage where no new learning takes place but a deeper understanding of the already acquired concepts is developed.

Reviewing pedagogical literature, the authors mention the ‘trilogy of the mind’, the 3 components that are mostly remarked as influencing human learning. These are cognition, motivation and emotion. The cognitive component is defined by what Vygotsky called the ‘zone of proximal development’. The other two components are dependent upon the characteristics of the learning environment and the interactions a learner has with others.

Motivation is modeled as a value that indicates how much a learner tries to acquire new knowledge. Emotion is also modeled as a value. However, this value is affected by the outcome of a learning experience. Thus if a student was successful in acquiring a skill, his emotion value increases; otherwise, it decreases.

The paper presents ‘goal structures’ as being a key characteristic of the learning environment. The focus of those goal structures can be individual, competitive or cooperative depending upon how the teacher engages the students in the act of learning. A teaching methodology that implements all the goal structures is called the STAD learning method. This essentially is comprised of 5 steps: teacher presentations, student teamwork or individual work, quizzes, individual improvement and team recognition. The group learning simulation uses the STAD learning method as the basis for the learning process.

The simulation environment consists of some environmental and individual parameters. The environmental parameters are **group composition** (heteregenous teams with different proportions of high and low ability learners); **group size**, **team rewards**, **concept difficulty** and the **amount of time available to master each concept** (measured in ‘ticks’). The individual parameters are the paper’s focus and consist of:

- **Learner ability** – 2, 3 and 4 different levels of learning abilities are mentioned in pedagogical studies but in this paper the authors use two levels – ability of 1 and ability of 2
- **Improvement** – increase in knowledge throughout the learning of a new concept; improvement is not only contingent upon one’s ability but is also dependent on motivation, emotion, zone of proximal development and concept difficulty.

- **Motivation** – it is initialized randomly in the interval [0.1, 1] and has a normal distribution with a mean set at 0.5; it depends on whether the difficulty of the concept that has to be learned is within the learner's zone of proximal development; it also depends on whether the learner passed the quiz at the end of a presented concept
- **Emotion** - it is initialized randomly in the interval [0, 1] and has a normal distribution with a mean set at 0.5. It depends on the following: (1) emotion of the teammates: if the emotion of a teammate is higher than the emotion of the learner, the emotion of the teammate is decreased by 0.01; otherwise the emotion of the teammate is increased by 0.01; and (2) the rank of the team of the student after the quiz; if the team scores relatively well, the team members become happy (resulting in an increase of emotion); otherwise, they become sad
- Other mentioned parameters include: '**zone'** (the center of the zone of proximal development), the **likeliness to help** other learners, and **competitiveness**

2.5.2.2. Comparison with C-ULM

An interesting similarity between the two simulations is related to the perspective on the learning process. Thus, in the group learning simulation, learning is seen (and implemented) as having 3 stages – first is the 'early' stage when most of the concept acquisition is done; the second step is the 'intermediate' stage where new associations are being made between concepts mostly learned in the first phase; lastly, in the 'autonomous' stage the agent deepens its understanding of the already acquired concepts.

In C-ULM most of the concepts are learned in the very beginning of the simulation. Thus, this short period can be considered as the 'early' stage. In the 'intermediate' stage more connections between the already acquired concepts are acquired. The longest phase

is deepening the understanding on the existing concepts and connections by shortening the confusion interval lengths of acquired connections and getting agent weights closer to the task required weights. However, in C-ULM, learning was not implemented to take place in those 3 stages. Instead, those stages were identified as an emergent behavior resulting from the simulation. Furthermore, in C-ULM, just as in real life, the stages are not completely separated one from another and there is considerable overlap. For example, in the 'early' stage we also acquire connections between already acquired concepts and the process of shortening confusion interval lengths takes place in all three stages.

The group learning simulation relies its agent structure on Vygotsky's trilogy of mind, where the 3 key components of learning are cognition, motivation and emotion. In comparison, the Unified Learning Model and C-ULM consider that the 3 key components of learning are long-term memory, motivation and working memory. C-ULM doesn't explicitly model emotion. This is because ULM views emotion as a motivator along with cognitive motivators such as goals and it is integrated in the more general ULM concept of motivation. The cognition component of Vygotsky's trilogy of mind is considered as a more general term in ULM that accounts for all learning. In turn, ULM breaks down the cognition term (as it relates to learning) into the three ULM components of long-term memory, motivation and working memory.

In the group learning simulation motivation influences how much a learner tries to learn new knowledge and the outcome of the learning experience influences emotion. In comparison, in computational ULM, the knowledge strength for given concepts influences motivation which in turn affects working memory allocation; after learning is

performed some knowledge is further strengthened and thus the motivation for the involved concepts increases. The linkage between the cognitive and the motivational aspects is stronger in C-ULM. This is because knowledge strength (confusion interval length) affects motivation, which in turn affects what is being learned. Further, what is being learned affects the knowledge strength and consequently the motivation for further learning.

We can compare the 'ability' characteristic in the paper's model with the working memory capacity in C-ULM. One difference is that the paper presents cases of heterogenous groups where the agents have different ability values (1 or 2). Meanwhile, C-ULM allows too for agents with different working memory capacities but the experiments done so far use the same working memory capacity for each agent.

This group learning simulation uses the term of concept difficulty. In contrast, in C-ULM, all concepts are seen as equal in difficulty however the tasks that they form can be harder or easier to solve depending on the number of connections that form those tasks. Thus, the concept of difficulty can be found at the task level in C-ULM.

2.6. Agent motivation profiles

2.6.1. Overview

Three agent motivation profiles based on the Atkinson's Risk Taking Model (RTM) are achievement motivation, affiliation motivation and power motivation profiles (Merrick 2011; Shafi et al. 2012; Hardhienata et al. 2012). Those models influence risk-taking behavior depending on the obtained incentives. Agents endowed with achievement motivation manifest a preference for tasks of intermediate difficulty. The affiliation motivation model is characterized by a preference of avoiding conflicts through risk

minimization. Finally, power motivated agents choose to take extreme risks in order to obtain higher incentive goals. Six variables are needed to construct achievement, affiliation, and power-motivated agents (Shafi et al. 2012). These are: turning points of approach ($M+$), turning points of avoidance ($M-$) of a goal, gradients of approach ($\rho +$) and gradients of avoidance ($\rho -$) of a goal, relative motivation strength (S), and an incentive value for success (IS).

2.6.2. Comparison with C-ULM

In C-ULM, agents are more motivated for more difficult tasks because these are seen as carrying a higher learning reward. By making the analogy between agent incentives in the above mentioned papers and C-ULM's learning rewards (the amount of knowledge obtained due to solving a task) we can observe that from a goal/task-oriented point of view, the current C-ULM agents are all power motivated agents. Those type of agents were shown to exhibit better leadership roles, making them suitable for coordinating a team of agents having the other two motive profiles, ie. achievement and affiliation motivated agents (Hardhienata et al. 2012). However, as mentioned above, the three motive profiles presented by Merrick make use of no less than 6 parameters.

In contrast, a power motivated agent in ULM is simply created by taking into account the learning reward of a task when computing motivational scores. This can be easily changed for part or all of the agents so that the new motivational scores are computed using the inverse of the task reward. In this manner, we achieve agents that seek easier, more solvable tasks that also carry lower learning rewards. Such agents can be seen as similar to affiliation motivated agents. The idea is that the generality of the C-ULM approach allows for an easier design or redesign of goal-oriented motivation types.

The motivation score formula for creating the current ULM motivation profile (similar to the power motivated agent) is given below in Eq. (1):

$$m_X^{A(t)} = \sum_{Y \in SC_X} \left(\frac{1}{l_{XY}^{A(t)}} \right) \cdot \sum_{k \in T_X} (R_k) \quad (2.1)$$

where X is a concept in agent A's knowledge; $m_X^{A(t)}$ is the agent A's motivational score for concept X at time step t; SC_X is the set of concepts connected to concept X; XY is the edge connecting concepts X and Y; $l_{XY}^{A(t)}$ is the length of agent A's confusion interval for edge XY at time step t; T_X is the subset of tasks that require concept X; and R_k is the reward for task k.

Simply by taking the inverse of the second sum we can obtain a motivation profile that is similar to the affiliation motivated agents. Thus, we would have the following motivation score equation:

$$m_X^{A(t)} = \sum_{Y \in SC_X} \left(\frac{1}{l_{XY}^{A(t)}} \right) \cdot \frac{1}{\sum_{k \in T_X} (R_k)} \quad (2.2)$$

Furthermore, as compared to the goal-oriented motivation types presented by Merrick, the motivation of C-ULM agents also incorporates the long-term knowledge that agents have in relation to existing tasks. Thus, motivation is connected to the LTM component and enables the expression of a wider range of motivation profiles without the explicit use of additional modeling parameters.

For example, we can have two power motivated agents but one has stronger knowledge than the other (is more certain about the relationships among concepts), thus having a motivation profile based on completely different motivation scores:

$$m_X^{A(t)} = \sum_{Y \in SC_X} \left(\frac{1}{l_{XY}^{A(t)}} \right) \cdot \sum_{k \in T_X} (R_k) \quad (2.3)$$

$$m_X^{B(t)} = \sum_{Y \in SC_X} \left(\frac{1}{l_{XY}^{B(t)}} \right) \cdot \sum_{k \in T_X} (R_k) \quad (2.4)$$

where

$$\sum_{Y \in SC_X} \left(\frac{1}{l_{XY}^{A(t)}} \right) > \sum_{Y \in SC_X} \left(\frac{1}{l_{XY}^{B(t)}} \right) \quad (2.5)$$

Agents A and B are both power motivated agents and have connections among the same concepts (Eq. (2.3) and (2.4)) but the confusion intervals for agent A are overall smaller than those for agent B (Eq. (2.5)) making its motivation for learning about concept X higher than the one of agent B.

Chapter 3. METHODOLOGY

3.1. Overview

In this chapter we start by presenting the architecture of the C-ULM model and the C-ULM components of long-term memory, motivation and working memory (Section 3.2). In Section 3.3 we present the communication protocol taking place between teacher and learner agents. In the remaining sections we present the knowledge decay process (Section 3.4), the structure of C-ULM tasks and the task attempt and feedback processes (Section 3.5) and utility methods that set up the task required knowledge and the initial agent knowledge (Section 3.6). Finally, we emphasize how the C-ULM model implements the ULM learning principles (Section 3.7).

3.2. Single-Agent Model

In this section we present in detail the C-ULM model for long-term memory, motivation and working memory, from the single agent perspective. We introduce new

concepts (such as the confusion interval and motivational scores), we present the equations that describe the functionality of each of the 3 components and present some examples to further clarify the functionality induced by the model.

3.2.1. Long-term memory

Long-term memory (or LTM) is being modeled as an undirected, weighted graph where nodes represent knowledge concepts and weighted edges represent a quantified connection between two concepts. Initially, agents do not have the necessary knowledge to solve a task but in some cases they might have a 'vague idea' of how to solve the problem. Key to our modeling of the LTM component is measuring the vagueness for each particular edge weight. We realize this by assigning a certainty measure called confusion interval to each edge weight. This interval is bounded and its length indicates how certain is the agent regarding the associated weight. For example, if the length is very small, the agent is quite certain about the weight of the edge and it has a solid knowledge about it. When an agent has to solve a task or teach another agent about a given connection weight, the agent will use a weight randomly generated from the associated confusion interval. The center of this confusion interval is also the edge weight.

Figure 3.1 presents an example of an agent's long-term memory. Next to each LTM connection is the confusion interval corresponding to that connection. The second value (bolded in Figure 1) in the confusion interval represents the interval center (or midpoint) and the edge weight. The other two values represent the minimum and the maximum values of the confusion interval. The lower bound on the minimum value is 0 and the upper bound on the maximum value is 1. As discussed later in this section, both the edge

weight and the length of this interval are updated during the learning process (Eqs. (3.3), (3.5) and (3.10)). Specifically, the edge weight can move in both directions, towards 0 or 1. The length of the confusion interval is shortened by the learning process (Eq. (3.5)) and it is increased by the process of knowledge decay (Eq. (3.10)). The confusion interval instantiates the statistical learning inherent in the ULM learning process of repetition. Similar to neuronal synapses that get strengthened through repetitive stimulus exposure, knowledge connections in C-ULM strengthen with repetition and weaken (decay) with disuse.

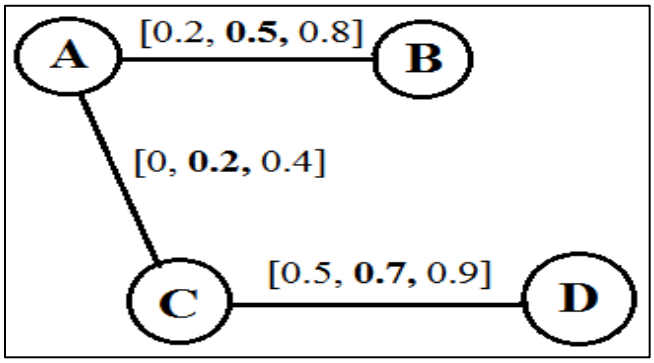

Fig. 3.1 LTM with concepts A, B, C, D.

On each edge is outlined the associated confusion interval.

3.2.2. Motivation

We use the notion of motivational scores to model the motivational component of the architecture. Each concept found in agent knowledge has a motivational score associated with it. A higher score reflects a higher motivation for teaching or learning about the associated concept while a lower score indicates a lower motivation related to that concept. This score is a function of: (1) the underlying confusion intervals for the connections that contain the concept, and (2) the expected rewards for the tasks that use the concept, as shown in Eq. (3.1):

$$m_X^{A(t)} = \sum_{Y \in SC_X} \left(\frac{1}{l_{XY}^{A(t)}} \right) . \sum_{k \in T_X} (R_k) \qquad (3.1)$$

where X is a concept in agent's A knowledge; $m_X^{A(t)}$ is the agent's A motivational score for concept X at time step t; SC_X is the set of concepts connected to concept X; XY is the edge connecting concepts X and Y; $l_{XY}^{A(t)}$ is the length of agent's A confusion interval for edge XY at time step t; T_X is the subset of tasks that require concept X; and R_k is the reward for task k. The rationale behind this formula is to allow two types of motivators that exist at the architectural level of ULM (Shell et al. 2010): the intrinsic one that captures the notion of self-efficacy, i.e., length of confusion intervals, and the extrinsic one that assesses the expectancy of possible rewards when using the concept for solving tasks.

Below we present an example of computing the motivational score for a given concept in an agent's LTM.

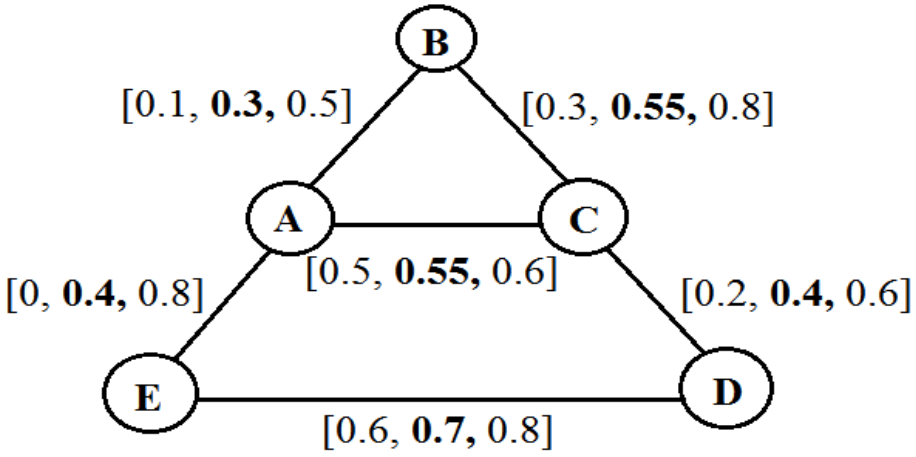

Fig. 3.2 LTM representation with concepts A, B, C, D, and E

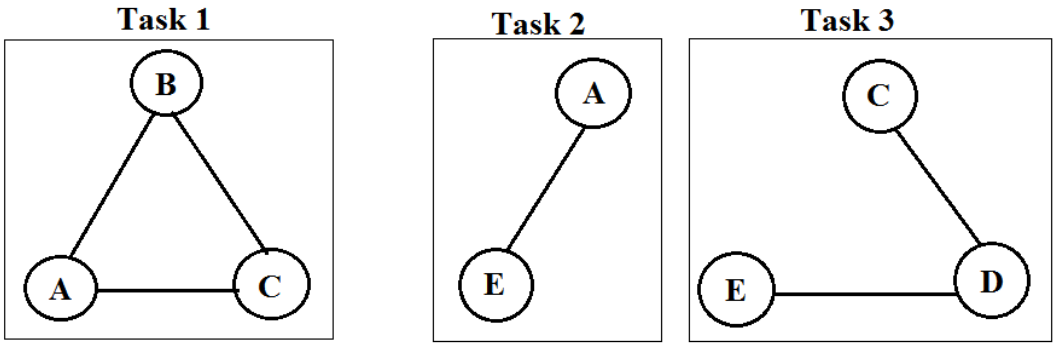

Fig. 3.3 Three tasks with their required concepts.

The reward for each task is equal to the number of connections a task has. Thus, we have that reward for task T_1 is $R_{T_1} = 3$, reward for task T_2 is $R_{T_2} = 1$ and reward for task T_3 is $R_{T_3} = 2$.

In this example, the motivational score for concept A becomes:

$$m_A = \left(\frac{1}{l_{AB}} + \frac{1}{l_{AC}} + \frac{1}{l_{AE}}\right) \cdot (R_{T_1} + R_{T_2}) = \left(\frac{1}{0.4} + \frac{1}{0.1} + \frac{1}{0.8}\right) \cdot (3 + 1) = 13.75 \cdot 4 = 55 \quad (3.2)$$

where l_{AB}, l_{AC} and l_{AE} represent the confusion interval lengths for connections AB, AC and AE.

3.2.3. Working Memory

Just as the LTM component, working memory (or WM) is also represented using weighted graphs. The key differences are that (1) the graphs do not have a confusion interval associated and (2) the working memory capacity indicates the maximum number of concepts or knowledge chunks allowed in the WM graph. C-ULM follows the ULM architecture in modeling the WM functionality. Thus, there are two main steps. In the first step, WM is allocated; in the second step, WM is processed and agent long-term knowledge is updated.

3.2.3.1. Working Memory Allocation

In order to realize WM allocation, we introduce the concept of awareness threshold (AT). This threshold indicates how aware the agent is of external and internal stimuli. If a stimulus has an intensity that is higher than this threshold, the agent becomes aware of that stimulus and consequently it allocates a WM slot for that stimulus. In our modeling, the concepts are the stimuli, and the motivational scores represent the stimulus intensity for the associated concept. Thus, the awareness threshold dictates what is attended, within the general architectural principle that motivation directs WM allocation.

Another important concept that relates to WM allocation is the process of WM chunking. As mentioned in Section 2.1.2.1, the ULM model refers to the chunking mechanism that helps humans make use of more existing knowledge during the learning process. According to the ULM model, a memory chunk is an interconnected knowledge unit that occupies only one slot of working memory capacity. In the C-ULM model we have created two versions of working memory allocation. In the first version, we do not use WM chunking such that each concept in WM occupies one WM slot. In the second version, we have implemented WM chunking so that multiple concepts organized or linked up as a memory chunk can occupy one WM slot. In the next two subsections we present those two versions.

3.2.3.1.1. Allocation without chunking

When chunking is not used, the concepts with motivation scores higher than the awareness threshold AT are allowed to enter the working memory. If the number of concepts with motivation scores higher than AT exceeds the number of working memory slots, then the concepts with the highest motivation scores will enter working memory.

AT varies uniformly across agents, but it is constant for each agent during the whole simulation.

The generic expression for whether concept X can enter working memory allocation is as follows:

$$X \in WM, \text{if } mscore_X \geq AT \quad (3.3)$$

$$X \notin WM, \text{if } mscore_X < AT \quad (3.4)$$

Where:

- X is the concept being evaluated for working memory candidacy
- WM is the agent's working memory
- $mscore_X$ is the agent's motivation score for concept X as defined by Equation 3.1
- AT is the learner's awareness threshold

Example

Below, we present an example covering working memory allocation without chunking for both a teaching agent and a learning agent. This happens at an arbitrary time step t, and the LTM representations of the teaching and learning agents at the start of the time step are illustrated in Figure 3.4.

In the first part of the time step, motivation scores for the teacher agent are computed. Because there are no isolated concepts in the teacher's LTM, the motivation scores for all concepts are computed (concepts A, B, C, D, E, F, G and Q). If there were any isolated concepts present they would have no chance of entering working memory because their motivation scores are not calculated.

Below is presented working memory allocation for the teacher agent:

- m-$score_A$=0.6 ≥ AT=0.5 → concept A enters teacher's working memory

- m-score$_B$=0.7 ≥ AT=0.5 → concept B enters teacher's working memory
- m-score$_C$=0.9 ≥ AT=0.5 → concept C enters teacher's working memory
- m-score$_D$=0.65 ≥ AT=0.5 → concept D enters teacher's working memory
- m-score$_E$=0.86 ≥ AT=0.5 → concept E enters teacher's working memory
- m-score$_F$=0.1 < AT=0.5 → concept F doesn't enter teacher's working memory
- m-score$_G$=0.4 < AT=0.5 → concept G doesn't enter teacher's working memory
- m-score$_Q$=0.2 < AT=0.5 → concept Q doesn't enter teacher's working memory

Only concepts A, B, C, D and E have a motivation score higher than the teacher's awareness threshold. Therefore, the sub-graph taught by the teacher is formed with concepts A, B, C, D and E.

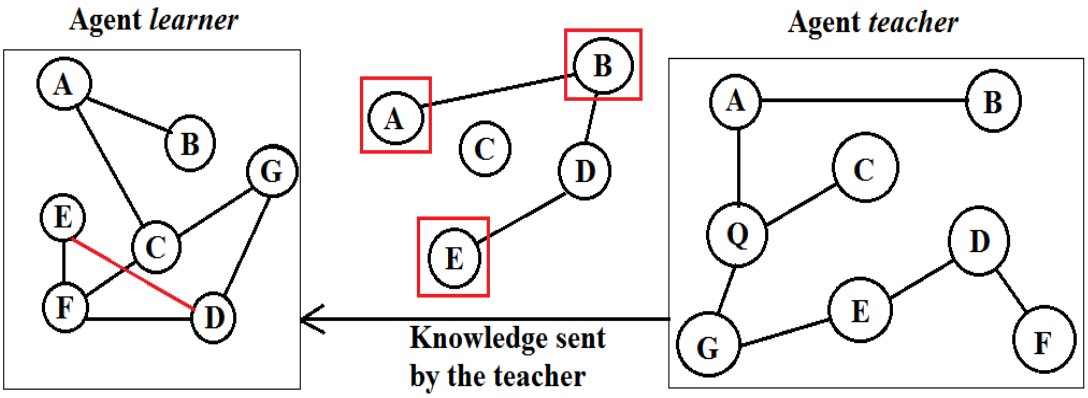

Fig. 3.4 Knowledge transfer at time step t

After the teacher decides upon the concepts to teach, the sub-graph (i.e. knowledge) to be presented to the learner agent is formed from the concepts present in the teacher's working memory and the edges that connect them. These edges have an instantiated weight that is randomly selected from a uniform distribution. Edges unrelated to those concepts are not present in the sub-graph created. Similarly, concepts with motivation scores below the teacher's awareness threshold are not a part of the shared knowledge.

Below, we present the working memory allocation for a learner with $AT = 0.4$, given some arbitrary motivation scores. Of note, the learner's motivation scores for those concepts are different from the teacher's motivation scores for the same concepts.

- m-score$_A$=0.9 $\geq$ AT=0.4 $\rightarrow$ concept A enters learner working memory
- m-score$_B$=0.7 $\geq$ AT=0.4 $\rightarrow$ concept B enters learner working memory
- m-score$_C$=0.1 $<$ AT=0.4 $\rightarrow$ concept C doesn't enter learner working memory
- m-score$_D$=0.2 $<$ AT=0.4 $\rightarrow$ concept D doesn't enter learner working memory
- m-score$_E$=0.8 $\geq$ AT=0.4 $\rightarrow$ concept E enters learner working memory

Therefore, at time step t, the learner allocates its working memory with concepts A, B and E. These are marked in red squares in the taught knowledge graph. The red connection on the learner side (connection DE) is a new connection added to the learner's LTM during the WM processing stage. This happens since concept E entered the learner's working memory and it is connected to concept D in the taught knowledge graph sent by the teacher.

3.2.3.1.2. Allocation with chunking

When chunking is used for a learner agent, we first identify the LTM chunks that contain the connections taught by the teacher. If the number of those LTM chunks is greater than the number of WM slots, then allocation occurs by discarding the LTM chunks containing the concepts with the lowest motivation scores. Furthermore, within the remaining LTM chunks, we select only those concepts with a motivation score higher than the awareness threshold AT. Similar to the allocation process without chunking, AT varies uniformly across agents, but it is constant for each agent during the whole simulation.

When chunking is used for a teacher agent, we first identify the concepts within the teacher agent's LTM that have a motivation score above the awareness threshold AT. We identify the LTM chunks that contain these selected concepts. If the number of LTM chunks is greater than the number of WM slots, then we discard those chunks containing the concepts with the lowest motivation scores. The remaining concepts are included in the LTM chunks that enter the teacher's working memory.

Example

Below, we present an example covering working memory allocation with chunking for both a teacher agent and a learner agent. This happens at an arbitrary time step t, and the LTM representations of the teacher and learner agents at the start of the time step are illustrated in Figure 3.5.

Below is presented working memory allocation for a teacher with AT = 0.5, WM capacity = 3 and arbitrary motivation scores:

- m-$score_A$=0.58 $>$ AT=0.5 → concept A is selected and its LTM chunk is identified
- m-$score_B$=0.8 $>$ AT=0.5 → concept B is selected and its LTM chunk is identified
- m-$score_C$=0.62 $>$ AT=0.5 → concept C is selected and its LTM chunk is identified
- m-$score_D$=0.08 $<$ AT=0.5 → concept D is not selected
- m-$score_E$=0.68 $>$ AT=0.5 → concept E is selected and its LTM chunk is identified

- m-$score_F$=0.82 > AT=0.5 → concept F is selected and its LTM chunk is identified
- m-$score_G$=0.1 < AT=0.5 → concept G is not selected
- m-$score_H$=0.4 < AT=0.5 → concept H is not selected
- m-$score_I$=0.22 < AT=0.5 → concept I is not selected
- m-$score_J$=0.46 < AT=0.5 → concept J is not selected
- m-$score_K$=0.52 > AT=0.5 → concept K is selected and its LTM chunk is identified
- m-$score_L$=0.16 < AT=0.5 → concept L is not selected
- m-$score_M$=0.68 > AT=0.5 → concept M is selected and its LTM chunk is identified
- m-$score_N$=0.56 > AT=0.5 → concept N is selected and its LTM chunk is identified
- m-$score_O$=0.72 > AT=0.5 → concept O is selected and its LTM chunk is identified
- m-$score_P$=0.88 > AT=0.5 → concept P is selected and its LTM chunk is identified

Since we selected 4 LTM chunks for the teacher agent, only 3 will enter working memory because the WM capacity is 3. Thus, the LTM chunk containing concept K (the concept with the lowest motivation score that is still above AT) doesn't enter the teacher's working memory and the sub-graph taught by the teacher is formed from 3 LTM chunks: the chunk of concept B, the chunk of concept E and F and the chunk of

concept M. The selected concepts that enter teacher's WM and lead to the taught sub-graph are: A, B, C, E, F, M, N, O and P.

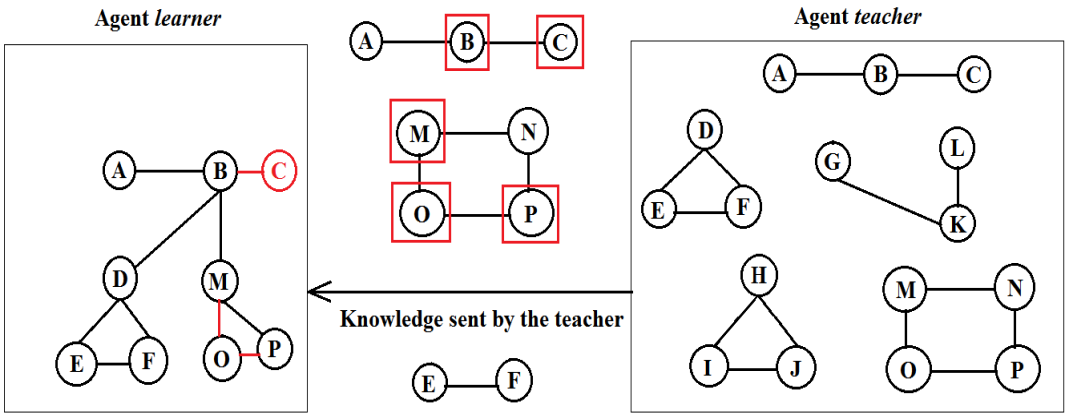

Fig. 3.5 Knowledge transfer at time step t

After the teacher decides upon the concepts to teach, the sub-graph (i.e., knowledge) to be presented to the learner agent is formed from the concepts present in the teacher's working memory and the edges that connect them. These edges have an instantiated weight that is randomly selected from a uniform distribution. Edges unrelated to those concepts are not present in the sub-graph created. Similarly, concepts with motivation scores below the teacher's awareness threshold are not a part of the shared knowledge. In short, the knowledge is only transferred if the teacher has the motivation to teach it, or the motivation to teach knowledge concepts that are related to it.

Below, we present the working memory allocation for a learner with AT = 0.6, WM capacity = 2 and arbitrary motivation scores:

- m-$score_A$=0.5 < AT=0.6 → concept A is not selected
- m-$score_B$=0.74 > AT=0.6 → concept B is selected and its LTM chunk is identified

- m-$score_C$=0.94 > AT=0.6 → concept C is selected and its LTM chunk is identified
- m-$score_E$=0.62 > AT=0.6 → concept E is selected and its LTM chunk is identified
- m-$score_F$=0.64 > AT=0.6 → concept F is selected and its LTM chunk is identified
- m-$score_M$=0.9 > AT=0.6 → concept M is selected and its LTM chunk is identified
- m-$score_N$=0.2 < AT=0.6 → concept N is not selected
- m-$score_O$=0.8 > AT=0.6 → concept O is selected and its LTM chunk is identified
- m-$score_P$=0.74 > AT=0.6 → concept P is selected and its LTM chunk is identified

Since we selected 3 LTM chunks for the learner agent, only 2 will enter working memory because the WM capacity is 2. Thus, the LTM chunk containing concepts E and F (the concepts with the lowest motivation score that are still above AT) doesn't enter the learner's working memory. Thus, the following chunks enter the learner's WM: the chunk of concept B (containing only the connection BC), and the chunk of concepts M, O and P. The selected concepts that enter learner's WM are marked in red squares in Figure 3.5. These are the concepts B, C, M, O and P. On the learner agent side, the red edges and nodes represent the new connections and the new concepts added to the learner's LTM during the WM processing stage.

3.2.3.2. Working Memory Processing

After working memory is allocated, its content is processed and agent's LTM is updated based on the statistical learning principles embodied in the ULM learning process of repetition. Specifically, working memory processing in C-ULM is performed by updating the confusion interval centers and lengths of LTM connections corresponding to working memory connections. In the case of a learner agent, the processing step updates both the confusion interval centers and lengths. In the case of a teacher agent, only the confusion interval length is updated since a teacher agent only reinforces its existing long-term knowledge without receiving new information about the task weights.

3.2.3.2.1. Updating the confusion interval center

The mechanism for **updating a learning agent's confusion interval center** is given by Eq. (3.5):

$$w_{XY}^{L(t)} = \frac{cic \cdot \left[f(X,WM_L) \cdot m_X^{L(t)} + f(Y,WM_L) \cdot m_Y^{L(t)} \right] \cdot w_{XY}^{T(t)} + w_{XY}^{L(t-1)}}{cic \cdot \left[f(X,WM_L) \cdot m_X^{L(t)} + f(Y,WM_L) \cdot m_Y^{L(t)} \right] + 1} \qquad (3.5)$$

Where:

- $w_{XY}^{L(t)}$ and $w_{XY}^{L(t-1)}$ are the learner agent confusion interval centers for edge XY during simulation time steps t and t-1, respectively
- $m_X^{L(t)}$ and $m_Y^{L(t)}$ are the learner agent's motivational scores for concepts X and Y at time step t
- $w_{XY}^{T(t)}$ is the instantiated weight value for edge XY communicated by the teacher via a weighted sub-graph at time step t

- cic is a learning coefficient that influences how much the confusion interval's center moves towards the weight communicated by the teacher ($w_{XY}^{T(t)}$); it is a simulation constant with a uniformly distributed value in the interval (0.8; 1.2); values between 0.8 and 1 indicate a learner agent that moves slower towards the weights taught by teacher agents (as compared to the reference value of 1); values between 1 and 1.2 indicate a learner agent that moves faster towards the weights taught by teacher agents (as compared to the reference value of 1).
- f is a function that returns 0 or 1 based on whether the given concept is currently present in the given WM.

Function f is described by Eq. (3.6) below:

$$f(Z, WM) = \begin{cases} 0, & Z \notin WM \\ 1, & Z \in WM \end{cases} \qquad (3.6)$$

Of note, Eq. 3.5 is a weighted average between the taught weight $w_{XY}^{T(t)}$ and learner agent's previous weight $w_{XY}^{\mathrm{L}(t-1)}$. Due to this weighted average structure, the agent's L weight could converge to agent's T weight but only after repeated updates. Motivation, represented by the weight of term $w_{XY}^{T(t)}$, controls how many updates are necessary for this convergence to occur. The rationale behind Equation (3.5) is to allow the learner agent to adjust towards the taught weights by repeated weight updates, thus incorporating the ULM learning process of repetition (Shell et al. 2010).

3.2.3.2.2. Updating the confusion interval length

The mechanism for **updating a learning or teaching agent's confusion interval length** for a given connection c' is given by Eqs. (3.7), (3.8) and (3.9):

$$l_{c\prime}^{A(t)} = l_{c\prime}^{A(t-1)} - sf \cdot mf \cdot cil \qquad (3.7)$$

$$sf = 1 - \frac{d(c,c\prime)}{D} \quad (3.8)$$

$$mf = f(X, WM) \cdot (m_X - AT) + f(Y, WM) \cdot (m_Y - AT) \quad (3.9)$$

Where:

- $l_{c\prime}^{A(t)}$ and $l_{c\prime}^{A(t-1)}$ are the confusion interval lengths for agent's A connection c' (connected by a graph path to connection c) at time steps t and t-1 respectively
- sf is the spread factor (defined by Eq. (3.8))
- mf is the motivation factor (defined by Eq. (3.9))
- cil is a learning coefficient that dictates the magnitude of the change in the confusion interval length during a simulation time step; it is a simulation constant that has a positive random value below 0.01; the reason for cil having such a small value is given by the rather large range of the other two update factors (sf and mf) as compared to the maximum length of the confusion interval; without having cil so small, the confusion interval would get very small in very few time steps. As a consequence, we wouldn't have enough learning repetitions, the ULM model wouldn't be correctly followed and the overall system performance would be rather low. Furthermore, the cil coefficient makes sense only operationally; from a conceptual point of view, the coefficient's small value could also be incorporated in other factors. In other words, we can say that cil is a modeling coefficient so that the entire behavioral output of the equation makes sense conceptually.
- $d(c, c')$ is the graph distance from connection c existent in WM and agent knowledge to a connection c' existent only in the agent knowledge

- D is a normalization factor considered to be the upper-bound on the distance between a pair of connections in the knowledge graph — that is, any distance greater than this value is set to D
- m_X and m_Y are the motivational scores for concepts X and Y, respectively
- f is the WM presence function defined by Eq. (3.6)
- AT is the awareness threshold for the learner

These equations implement a statistical learning algorithm where both the connection center and confusion interval are repeatedly updated. As noted in the ULM, by virtue of the law of large numbers, this repetitive update process should lead to convergence on the actual weights of task connections.

Additionally, we instantiate spread activation which is an architectural component that results from the associative nature of human knowledge (Anderson 1983). Spread activation says that if a concept is activated, then this activation spreads to any connected concept. Furthermore, the activation of all connected concepts is smaller and it decreases with the distance from the initial concept. In C-ULM (Eq. (3.7) and (3.8)), the update made to the confusion interval length of connection c' reachable from connection c decreases as the updated connection c' is farther from connection c.

3.2.4. Knowledge Decay

The ULM learning process of repetition says that repeated connections are strengthened but that non-repeated connections weaken. To accomplish this, we use a statistical learning algorithm that weakens knowledge through decay. If a concept does not enter WM for a specified number of time steps, the concept is considered unused and the associated confusion intervals of all connections involving that concept are increased.

The **knowledge decay mechanism for updating an agent's confusion interval length** for a connection involving an unused concept is given by Eq. (3.10):

$$l_{XY}^{A(t)} = \begin{cases} l_{XY}^{A(t-1)} \cdot e^{r_{dec}}, u_X < u_X^{(t)} \leq DF \cdot u_X \\ l_{XY}^{A(t-1)}, u_X^{(t)} \leq u_X \; or \; u_X^{(t)} > DF \cdot u_X \end{cases} \quad (3.10)$$

where X is the unused concept, Y is a concept (used or unused) connected to concept X, $l_{XY}^{A(t)}$ and $l_{XY}^{A(t-1)}$ are the confusion interval lengths for agent's A connection XY at time steps t and $t-1$, respectively; e is the natural number; r_{dec} is the knowledge decay rate (i.e. the rate at which the confusion interval grows) and is an experimental parameter set to a constant value (between 0 and 1); u_X indicates how many time steps concept X can remain unused without triggering knowledge decay for connections involving X; $u_X^{(t)}$ is the number of time steps that concept X has been unused for at time t; $DF \cdot u_X$ is an upper-bound on the number of time steps for which knowledge decay is applied to connections involving concept X; and DF is a decay multiplication factor. The idea that unused knowledge eventually decays over time is inspired by the Knowledge Decay Theory (Harris 1952). While an exponential function for the forgetting process is a common assumption among memory models, latest research shows that a power law better fits the observed data (Kahana and Adler 2002). We plan on using power law functions for memory decay in our future work.

3.3. Multiagent Framework

In this section we present the agent communication and interaction protocol consisting of the actions of teaching and learning as illustrated in Fig. 3.5. In this protocol, first, the teacher agent selects the concepts to be taught and allocates its WM for them. The concept selection process is done by the algorithm **TeachAllocate**. Then, the teacher

agent produces the knowledge TK to be taught using **TeachProcess**. This has two effects. First, the teacher agent itself learns from the teaching as well. Thus, this leads to a shortening of confusion intervals for the connections in teacher's knowledge that correspond to the connections found in TK. Second, correspondingly, the learner agent performs the algorithm **LearnAllocate** in order to filter taught knowledge TK. The "filtered" TK (or FTK) resides in the WM of the learner agent. The learner agent then proceeds to perform **LearnProcess**, which duly updates the confusion interval lengths and centers according to the knowledge update process described earlier (Section 3.2.3.2, Working Memory Processing).

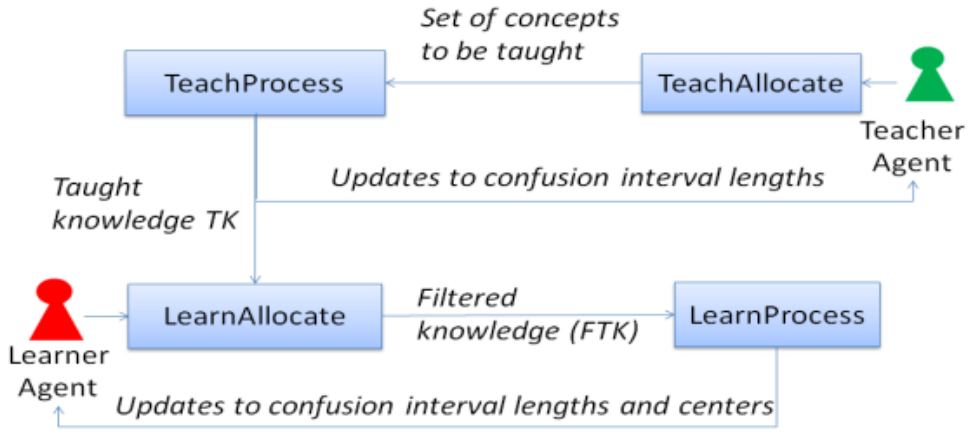

Fig. 3.6 Communication protocol between a teacher and a learner agent.

In Table 3.1 presented below, we summarize the main purpose of the learning and teaching algorithms described above.

Table 3.1 Main purpose of the learning and teaching algorithms

Algorithm	Main Purpose
learnAllocate_basic	Allocates learner agent's WM and uses one WM unit per knowledge concept

learnAllocate_chunking	Allocates learner agent's WM and uses one WM unit per knowledge chunk
learnProcess	Updates learner agent's knowledge given the WM content
updateWeight	Updates the weights of agent knowledge connections
updateConfusionInterval	Updates the confusion interval lengths of agent knowledge connections
teachAllocate_basic	Allocates teacher agent's WM and uses one WM unit per knowledge concept
teachAllocate_chunking	Allocates teacher agent's WM and uses one WM unit per knowledge chunk
teachProcess	Updates teacher agent's knowledge given the WM content

3.3.1. Learning

Learning is comprised of two stages: allocating working memory and processing the content of working memory. Allocating working memory for the learner has two versions.

The first one is learnAllocate_basic, where we count how many of the taught concepts have a motivation score higher than the awareness threshold. This number is then compared to the working memory capacity in order to ensure that the capacity is not exceeded by allocating too many concepts.

The second version is learnAllocate_chunking, where we compute the number of LTM knowledge chunks that taken together contain all the connections taught by the teacher. In the case of this version, this number is then compared to the working memory capacity

in order to ensure that the capacity is not exceeded by allocating too many concept chunks.

Working memory processing for the learner agent is done by the algorithm learnProcess and it consists in updating the learner agent's confusion interval lengths and centers.

3.3.1.1. Working memory allocation for learning without chunking

The algorithm that performs allocation without chunking is called learnAllocate_basic (allocating working memory for the learner agent). This algorithm ensures that taught concepts with a motivation score higher than the awareness threshold enter the working memory of the learning agent. In line 1, we sort all connections in the sub-graph taught by the teacher. We perform the sort in a descending order by the maximum motivation score between the two concepts that form each connection. In lines 5-25, we loop through all connections in the sorted order and increase the number of concepts in working memory (wm_concepts) if the motivation score of the concept is higher than the awareness threshold AT. We denote the two concepts that make up a connection by using the attributes $concept_1$ and $concept_2$. The motivation score is denoted by the attribute m_{score} for each concept. In line 20, we check if the number of concepts added to working memory is greater than the number of working memory slots ($L.WM.wmSlots$). If it is, we break out of the loop and the method terminates. Otherwise, we check that at least one concept of the current connection has a motivation score greater than AT. If this condition is met (line 22), we add the current connection to the constructed graph $L.WM$ in line 23 (the working memory graph of the learning agent), and continue with the loop until all connections are examined.

Algorithm learnAllocate_basic

Input: L – the learning agent

TK – taught knowledge

Returns void

1. **Sort** all connections e_i in TK by $max(\ e_i.concept_1.m_{score},\ e_i.concept_2.m_{score})$
2. //(in descending order)
3. Set $wm\_concepts$ to 0
4. Set $L.WM$ to nil
5. **Loop** through all connections e_i in TK
6. Set isAboveAT to false
7. **If** $(e_i.concept_1.m_{score} > L.AT)$ **then**
8. $wm\_concepts \leftarrow wm\_concepts + 1$
9. Set isAboveAT to true
10. **End If**
11. **If** $(e_i.concept_2.m_{score} > L.AT)$ **then**
12. $wm\_concepts \leftarrow wm\_concepts + 1$
13. Set isAboveAT to true
14. **End If**
15.
16. //We break if the number of concepts is greater than the working memory capacity
17. //or if the motivation score for both concepts was below AT; consequent connections

18. //will have a lower motivation score since connections are sorted in descending order
19. //by their motivation scores
20. **If** ($wm\_concepts > L.WM.wmSlots$) **then**
21. break // abort as there are too many concepts
22. **Else If** (isAboveAT status is true) **then**
23. add e_i to $L.WM$ // insert the edge into the working memory
24. **End If**
25. **End Loop**

End Algorithm

3.3.1.2. Working memory allocation for learning with chunking

The algorithm that performs allocation with chunking is called learnAllocate_chunking (allocating working memory for the learner agent by using LTM knowledge chunks). This algorithm is very similar to learnAllocate_basic with the exception that we do not allocate one concept in each working memory slot but instead allocate an entire chunk. We find and count the chunks in the learner LTM knowledge that taken together contain all of the taught connections (lines 10-18). Each chunk is identified in line 16 by calling the method bfsVisit_Structure. In line 23, we check if the number of LTM knowledge chunks (instead of number of concepts as in learnAllocate_basic) is greater than the number of working memory slots ($L.WM.wmSlots$). If it is, just as in learnAllocate_basic, we break out of the loop and the method terminates. Otherwise, we check that at least one concept of the current connection has a motivation score greater than AT. If this condition is met (line 25), we

add in line 26 the current connection to the constructed graph $L.WM$ (the working memory graph of the learning agent), and then continue with the loop until all connections are examined.

The rationale for designing learnAllocate_chunking method is to leverage the chunking mechanism in order to obtain a higher agent efficiency at low, human-like working memory capacities.

Algorithm learnAllocate_chunking

Input: L – the learner agent

TK – taught knowledge

CIT – threshold for the confusion interval for a connection (used when the connection is visited by method bfsVisit_Structure)

Returns void

1. **Sort** all connections e_i in TK by $max(e_i.concept_1.m_{score},\ e_i.concept_2.m_{score})$
2. //(in descending order)
3. Set $L.WM$ to nil
4. Initialize the Visited status of all connections e_j in $L.G_A$ to false
5. **Loop** through all connections e_i in TK
6. Set isAboveAT to false
7. **If** ($e_i.concept_1.m_{score} > L.AT$ or $e_i.concept_2.m_{score} > L.AT$) **then**
8. Set isAboveAT to true
9. **End If**
10. Locate in $L.G_A$ the corresponding e_j for e_i
11. **If** (e_j is **not** found **or** $e_j.CI >$ CIT) **then** // that means the agent A does not know

12. // much about e_j
13. Increment connectedComponents by 1
14. **Else If** (e_j's Visited status is false) **then**
15. Increment connectedComponents by 1
16. Call **bfsVisit_Structure** ($L.G_A$, e_j, CIT) // change the Visited status of all edges
17. // in $L.G_A$ that are "reachable" from e_j

to true.

18. **End If**
19. // We break if the number of knowledge chunks is greater than the working
20. // memory capacity or if the motivation score for both concepts was below AT;
21. // consequent connections will have a lower motivation score since connections are
22. // sorted in descending order by their motivation scores
23. **If** ($connectedComponents > L.WM.wmSlots$) **then**
24. break // abort as there are too many chunks
25. **Else If** (isAboveAT status is true) **then**
26. add e_i to $L.WM$ // insert the edge into the working memory
27. **End If**
28. **End Loop**

End Algorithm

3.3.1.3. Working memory processing for learning

The algorithm that performs working memory processing is called learnProcess. This algorithm uses the concepts found in working memory (added by the learnAllocate method) in order to update the long-term knowledge of the learner agent. It calls the

updateConfusionInterval method (line 8) to ensure that all LTM connections reachable from a connection found in working memory will get updated based on the distance-based spread factor described in Equation 1. It also fully connects the concepts in learner's LTM that correspond to the concepts found in its working memory. In order to do this, we loop through all pairs of concepts in working memory that are not connected and connect them in the learner's LTM graph (in case they were not already connected in that graph). The loop is performed in lines 10-17 and creating the edge in the LTM graph is done in line 14. In contrast to connections that entered working memory because they were taught by a teacher, the confusion interval of connections created in line 14 is set to the maximum value of 1 (line 13). Thus, we emphasize that those connections start to exist by being "weaker" than all the other connections that were taught by a teacher agent.

Algorithm learnProcess

Input: L – the learner agent

TK – taught knowledge

Returns void

1. **Loop** through all connections e_i in $L.WM$ // working memory of L
2. M-score_X = $e_i.concept_1.m_{score}$
3. M-score_Y = $e_i.concept_2.m_{score}$
4. // $L.K$ = knowledge graph for agent L
5. // $L.AT$ = awareness threshold for agent L
6. // $L.CF$ = confusion interval for agent L
7. // $TK.e_{\mathrm{i}}.weight$ = weight of connection e_i in taught knowledge

8. **Call** updateWeight (L, $TK.e_i.weight$)

9. **Call** updateConfusionInterval ($L.K,\ e_i, M-score\_X\ , M-score\_Y, L.AT, L.CF$)

10. **End Loop**

11. **Loop** through all concepts c_i in $L.WM$

12. **Loop** through all concepts c_j in $L.WM$

13. **If (**$i \neq j$ and $edge(c_i, c_j) \notin L.WM$ and $edge(c_i, c_j) \notin L.K$) **then**

14. Set $edge(c_i, c_j).CI$ to 1

15. Add $edge(c_i, c_j)$ to $L.K$

16. **End If**

17. **End Loop**

18. **End Loop**

End Algorithm

3.3.1.3.1. Method updateWeight

The method that updates the connection weights (the centers of confusion intervals) is called updateWeight and is used in line 8 of the algorithm learnProcess. According to Equation (3.5) (section 3.2.3.2.1), this method updates the weight of a given connection (e_{XY}) by taking into account two main factors: the existing weight in the learner agent's LTM and the weight taught by the teacher. A short form of Equation (3.5) is given by Equation (3.11) presented below:

$$e_{XY}.W = \frac{e_{XY}.W + Adjustment \cdot TW}{1 + Adjustment} \tag{3.11}$$

Where:

- e_{XY} = connection between concepts X and Y in learner agent's knowledge
- $e_{XY}.W$= weight of connection e_{XY}

- TW = the weight taught by the teacher
- $Adjustment$ = weight for TW (in the sense of a weighted average between $e_{XY}.W\ and\ TW$)

The weighted average is computed in line 5 of the method updateWeight. The weight for learner's weight (KW) is 1 and the weight for TW (Adjustment) is computed in line 1 by taking into account the motivation scores of the two concepts X and Y and the learner agent's knowledge factor L.KF. In lines 7-8 we bound KW between 0 and 1.

Algorithm updateWeight

Input: L – learner agent

TW – weight taught by the teacher

Returns void

1. $Adjustment = (L.e_{XY}.m_{score\_X} + L.e_{XY}.m_{score\_Y}) * L.KF$
2. // $L.e_{XY}.m_{score\_X}$ = the motivation score for concept X
3. // $L.e_{XY}.m_{score\_Y}$ = the motivation score for concept Y
4. // L.KF = the learner agent's knowledge factor
5. L.KW = (L.KW + Adjustment * TW) / (1+Adjustment)
6. // L.KW = learner agent's knowledge weight
7. $KW \leftarrow max(KW, 0)$ // lower-bound the learner's knowledge weight
8. $KW \leftarrow min(KW, 1)$ // upper-bound the learner's knowledge weight

End Algorithm

3.3.1.3.2. Method updateConfusionInterval

The method that updates the confusion interval is called updateConfusionInterval and is used in line 10 of the algorithm learnProcess. The confusion interval is updated during

the learning process according to Equation (3.7) (Section 3.2.3.2.2). The update amount is comprised of three factors: spread factor (defined by Equation (3.8), Section 3.2.3.2.2), motivation factor (defined by Equation (3.9), Section 3.2.3.2.2), and a learning coefficient defined as cil in Equation (3.7). Of note, this method is also called in the algorithm for teacher agent working memory processing (algorithm teachProcess, Section 3.3.2.3). Thus, as it can be seen in the algorithm description below, it treats both types of agents: a learner and a teacher agent.

First, we compute the motivation factor. This factor depends on the differences between the motivation scores for the concepts involved in the analyzed connection and the awareness threshold. Note that we update the confusion interval differently for an agent given its role: a learner or a teacher. If the agent is a learner, then we first check whether the input edge exists in the learner's knowledge. If it is, then we call bfsConfusionUpdate (line 10) which involves not only updating the interval of the input edge but also propagating the impact of the update to other edges (using the spread function defined in Eq. (3.8)).

Note that we call bfsConfusionUpdate with four input parameters: the original knowledge graph (G_A), the edge or connection (e), upper-bound distance in the knowledge graph (factor D used in Equation (3.8)) and an update factor, which is the product of the motivation factor and the learning coefficient cil (Equation (3.7)). Now, if the edge is not in the learner's knowledge, then the edge is added to the learner's knowledge (line 14) and its confusion interval is computed by calling the method newEdgeConfusionInterval (line 15). On the other hand, if the agent is a teacher, then we know for sure that the edge must be already in the agent's knowledge. Thus, we

immediately call updateEdgeConfusionInterval (line 18). This method is actually part of the bfsConfusionUpdate subroutine and thus it is consistent with how a learner agent is updated.

Algorithm updateConfusionInterval

Input: G_A – the knowledge graph (containing concepts) specific to a particular agent A

e – edge that exists in working memory (connects concepts X and Y)

M-Score_X – motivational score for concept X

M-Score_Y – motivational score for concept Y

D – upper-bound distance between a pair of connections in G_A graph

AT – attention threshold

CF – confusion factor

Returns void

1. Set MF to 0 //MF – motivation factor
2. **If** M-Score_X >= AT **then**
3. Increment MF by M-Score_X – AT
4. **End If**
5. **If** M-Score_Y >= AT **then**
6. Increment MF by M-Score_Y – AT
7. **End If**
8. **If** (A *is learner*) **then**
9. **If** $e \in G_A$ **then** // edge e is in the agent's knowledge
10. **Call** bfsConfusionUpdate (G_A, e, D, MF*CF)
11. **Else**

12. //If edge e doesn't exist in agent's knowledge, it is created there and
13. //its initial confusion interval is computed by calling the method
14. //newEdgeConfusionInterval
15. add e to G_A
16. $e.CI \leftarrow$ newEdgeConfusionInterval (G_A, e)
17. **End If**
18. **Else** // A is teacher
19. $e.CI \leftarrow updateEdgeConfusionInterval\ (e.CI, MF * CF)$
20. **End If**
21. **End Algorithm**

3.3.1.3.3. Method bfsConfusionUpdate

The bfsConfusionUpdate method realizes a Breadth-First Search (bfs) traversal of the knowledge graph and updates the length of the confusion interval for all edges reachable from the starting edge according to Equation (3.7). In bfsConfusionUpdate, a connection c is considered reachable if there is a path between the starting edge and c. The underlying graph traversal algorithm used is Breadth-First Search. In this type of search we visit all the neighbors of a given edge and then visit the neighbors of those neighboring edges. The process repeats until all edges reachable from the initial edge were visited. The algorithm is perfectly suited for computing the distance from the initial edge. For example, the initial edge (the edge where the traversal starts) has a distance of 0, its direct neighbors have a distance of 1 and the neighbors of those neighbors (excluding the initial edge) have a distance of 2. The process of computing the distance is done in line 13. Based on the computed distance, in line 7 we compute the spread factor

according to Equation (3.8). Line 8 adjusts the confusion interval length of the analyzed edge according to Equation (3.7). This is done by calling the updateEdgeConfusionInterval method.

Algorithm bfsConfusionUpdate

Input: G_A – the knowledge graph (containing concepts) specific to a particular agent A

e – edge that exists in working memory and in agent knowledge

(connects concepts X and Y)

D – upper-bound distance between a pair of connections in graph G_A

UF – confusion interval update factor

Returns void

1. Set e's Visited status to true.
2. Set $e.dist$ to 0.
3. // e is the edge in working memory and the bfs traversal starts from it.
4. // dist = distance of current edge from the edge where the bfs traversal started
5. Initialize a FIFO queue Q
6. Q.enqueue(e)
7. **While** Q is **not** empty // start the visits
8. s ← Q.dequeue() // retrieve the front edge/connection from the queue
9. SF ← 1 – s.dist / D
10. $s.CI \leftarrow updateEdgeConfusionInterval\ (s.CI, SF * UF)$ //Eq. (3.7)
11. **Loop** through all neighboring edges $n_{s,i}$ of s
12. // we check the edge was not visited by a bfs traversal that started with
13. // another edge in the working memory than e

14. **If** ($n_{s,i}'s$ Visited status is false) **then**

15. $n_{s,i}.dist \leftarrow min(s.dist + 1, D)$ // add 1 to the distance until it is D

16. Set $n_{s,i}'s$ Visited status to true

17. Q.enqueue ($n_{s,i}$)

18. **End If**

19. **End Loop**

20. **End While**

End Algorithm

3.3.1.3.4. Method updateEdgeConfusionInterval

The algorithm updateEdgeConfusionInterval returns the updated confusion interval length for a given connection (using Equation 3.7). Note also that the confusion interval length is bound between 0.05 and 1, as imposed in the updateEdgeConfusionInterval method.

Algorithm updateEdgeConfusionInterval

Input: CI – the confusion interval of a knowledge edge

Δ – confusion interval change amount

Returns double (the adjusted CI value)

1. $CI \leftarrow min(CI, 1)$ // upper-bound the CI

2. $CI \leftarrow CI - \Delta$ // update the CI

3. CI $\leftarrow max(CI, 0.05)$ // lower-bound the CI

4. **Return** CI

End Algorithm

3.3.1.3.5. Method newEdgeConfusionInterval

The algorithm newEdgeConfusionInterval returns the initial confusion interval length for a newly created connection in a learner agent's knowledge. The value of the length depends on the confusion interval length values of connections neighboring the new connection. Each neighboring connection shares one (and only one) concept with the new connection. The equation used for computing the confusion interval length of the new connection is presented below:

$$e_{XY}.CI = \frac{\sum_{i \in SC_X} e_{Xi}.CI + \sum_{j \in SC_Y} e_{Yj}.CI}{|SC_X| + |SC_Y|} \quad (3.12)$$

Where:

- e_{XY} = connection between concepts X and Y
- SC_X = set of concepts connected to concept X (except concept Y)
- SC_Y = set of concepts connected to concept Y (except concept X)
- $|SC_X|$ and $|SC_Y|$ are the cardinalities of the sets of concepts related to concepts X and Y, respectively

Equation (3.12) has been designed with the intuition that the initial confusion interval length for a newly created connection should resemble the interval length of its neighborhood in the learner agent's knowledge. This is supported by cognitive and psychological research showing that self-efficacy (which is modeled by the confusion interval) generalizes across related knowledge such as reading and writing (Shell et al. 1989, 1995).

Below we present the pseudo-code of the algorithm newEdgeConfusionInterval. In lines 7-20 we loop through all concepts in the learner agent's knowledge and look for connections involving either concept X (lines 10-13) or concept Y (lines 16-19). We

count those connections and add up their confusion interval lengths. Afterwards, in line 22 we compute the average confusion interval length of connections involving either concept X or concept Y and return this value. Otherwise, since it has no neighbor connections, it should start out by being the weakest connection possible. Thus, we simply assign the largest confusion interval length to the connection (line 24).

Algorithm newEdgeConfusionInterval

Input: G_A – the knowledge graph (containing concepts) specific to a particular learner agent A

e – a newly created edge in learner agent's knowledge

(connects concepts X and Y)

Returns double (the CI value for the newly created connection)

1. $X = e.concept_1$
2. $Y = e.concept_2$
3. Set SC_X to 0
4. Set SC_Y to 0
5. Set sum_CI_X to 0
6. Set sum_CI_Y to 0
7. **Loop** through concepts c_i in G_A
8. // loop through all concepts in G_A that are connected to X (except Y)
9. // and add their confusion intervals and count them
10. **If** $(c_i \neq Y \; and \; edge(X, c_i) \in G_A)$ **then**
11. $SC_X = SC_X + 1$ // compute the cardinality of the set
12. $sum\_CI\_X = sum\_CI\_X \; + edge(X, c_i).CI$

13. **End If**

14. // loop through all concepts in G_A that are connected to Y (except X)

15. // and add their confusion intervals and count them

16. **If** $(c_i \neq X \; and \; edge(Y, c_i) \in G_A)$ **then**

17. $SC_Y = SC_Y + 1$ // compute the cardinality of the set

18. $sum\_CI\_Y = sum\_CI\_Y \; + edge(Y, c_i).CI$

19. **End If**

20. **End Loop**

21. **If** $(SC_X + SC_Y > 0)$ **then**

22. **Return** $(sum\_CI\_X + sum\_CI\_Y)/(SC_X + SC_Y)$

23. **Else**

24. **Return** 1 // we assign to it the largest confusion interval length

25. **End If**

End Algorithm

3.3.2. Teaching

Similar to the learning process, teaching is comprised of two stages: allocating working memory and processing the content of working memory for the teacher agent. Just as in the case of a learner agent, the process of allocating working memory for the teacher has two versions.

The first one is teachAllocate_basic, where one working memory slot is occupied with one concept. We compute how many concepts with a high motivation are not isolated and then compare this number with the working memory capacity in order to ensure that the capacity is not exceeded by allocating too many concepts.

The second version is teachAllocate_chunking, where we compute the number of LTM chunks that taken together contain all the connections with high motivation that are not isolated. In the case of this version, this number is then compared to the working memory capacity in order to ensure that the capacity is not exceeded by allocating too many chunks.

Working memory processing for the teacher agent is done by the algorithm teachProcess and it consists in updating the teacher agent's confusion interval lengths.

3.3.2.1. Working memory allocation for teaching without chunking

The method teachAllocate_basic ensures that the concepts with the highest motivation scores for the teacher will be the ones that are being taught. We exclude from this list the concepts that are isolated in teacher's LTM. The rationale for this exclusion is the fact that C-ULM knowledge, similar to general human knowledge, is contained in the connections existent between various concepts (Shell et al. 2010). Thus, according to the ULM model, the C-ULM isolated concepts do not have a human counterpart and are just the by-product of the computer simulation.

In line 2 we sort all the concepts in teacher agent's LTM by their motivation scores (in descending order). In lines 3-10 we loop through the sorted concepts and add all connected concepts to a concept list. The process of adding concepts to this list stops when the size of the list reaches teacher agent's working memory capacity (line 6). The concept list will be provided as input to the teachProcess method.

Algorithm teachAllocate_basic

Input: T – the teacher agent

Returns void

1. // T.K = the knowledge graph of the teacher agent
2. Set $T.concept\_list$ to nil
3. **Sort** all concepts c_i in T.K by $c_i.m_{score}$ (in descending order)
4. **Loop** through sorted concepts c_i in $T.K$
5. **If** c_i is not isolated **then**
6. Add c_i to $T.concept\_list$
7. **If** $T.concept\_list.size \geq T.WM.wmSlots$ **then**
8. break // abort as there are too many concepts
9. **End If**
10. **End If**
11. **End Loop**

End Algorithm

3.3.2.2. Working memory allocation for teaching with chunking

The method teachAllocate_chunking of a teaching agent corresponds to the learnAllocate_chunking method of a learner agent. Similar to the teachAllocate_Basic method, we first create the concept list T.concept_list. We add to this list only those concepts that have a motivation score higher than the awareness threshold T.AT (lines 7 – 15). We compute how many long-term knowledge chunks can be entered into the working memory in lines 16-23. This is done by calling bfsVisit_Structure to find all other concepts connected to the concept at hand. This network, or a connected component, is seen as a chunk. The concept list is then used in the teachProcess method in order to determine the knowledge graph that is being taught.

Algorithm teachAllocate_chunking

Input: T – the teacher agent

CIT – threshold for the confusion interval of a connection (used when the connection is visited by method bfsVisit_Structure)

Returns void

1. **Sort** all connections e_i in $T.G_A$ by $max(e_i.concept_1.m_{score},\ e_i.concept_2.m_{score})$
2. //(in descending order)
3. Initialize the Visited status of all connections e_i in $T.G_A$ to false
4. Set $T.concept\_list$ to nil
5. Set connectedComponents to 0
6. **Loop** through all connections e_i in $T.G_A$
7. Set isAboveAT to false
8. **If** ($e_i.concept_1.m_{score} > T.AT$) **then**
9. Add $e_i.concept_1$ to $T.concept\_list$
10. Set isAboveAT to true
11. **End If**
12. **If** ($e_i.concept_2.m_{score} > T.AT$) **then**
13. Add $e_i.concept_2$ to $T.concept\_list$
14. Set isAboveAT to true
15. **End If**
16. **If** (e_i's Visited status is false) **then**
17. Increment connectedComponents by 1
18. Call **bfsVisit_Structure** ($T.G_A$, e_i, CIT) // change the Visited status of all edges
19. // in $L.G_A$ that are "reachable" from e_j to true.

20. **End If**

21. // We break if the number of knowledge chunks is greater than the working

22. // memory capacity

23. **If** ($connectedComponents > T.WM.wmSlots$) **then**

24. break // abort as there are too many chunks

25. **End If**

26. **End Loop**

End Algorithm

3.3.2.3. Working memory processing for teaching

The method teachProcess updates the confusion intervals of connections that are being taught and creates the knowledge sub-graph that is the product of teaching. This sub-graph is 'sent' to the learner and it will fill the learner's working memory. We use a double loop (lines 2-19) in order to exhaustively check every pair of concepts in the concept list filled by the method teachAllocate. If the two concepts are connected in teacher agent's LTM, we create the corresponding edge in the taught sub-graph TK (line 15). Furthermore, we update the confusion interval in the teacher agent's LTM (line 16). This update is done by the method updateConfusionInterval presented in section 3.3.1.3.2. In order to compute the weight of edges that make up the taught sub-graph TK, we pick up a random value—generated uniformly—in an interval centered around the weight of the corresponding edge in teacher agent's LTM (line 12).

Algorithm teachProcess

Input: T – the teacher agent

Returns TK – the knowledge graph that is being taught (the learner's WM will be filled with TK)

1. Set TK to nil //initialization
2. **Loop** through concepts c_i in $T.concept\_list$
3. **Loop** through concepts c_j in $T.concept\_list$
4. // $T.G_A$ – the knowledge graph of the teacher agent
5. **If** ($edge\ (c_i, c_j) \in T.G_A$) **then**
6. // We compute the motivation scores for concepts c_i and c_j
7. M_X = $c_i.m_{score}$
8. M_Y = $c_j.m_{score}$
9. center = $edge\ (c_i, c_j).mean$
10. Δ = $edge\ (c_i, c_j).CI$
11. // We pick up a random value in the interval $[center - \Delta/2, center + \Delta/2]$
12. taught_weight = $random\ (center - \Delta/2, center + \Delta/2)$
13. $e_{ij}.mean$ = taught_weight
14. // We add a connection with the taught weight to taught knowledge graph TK
15. add e_{ij} to TK
16. **Call** updateConfusionInterval ($T.G_A, edge\ (c_i, c_j), M\_X, M\_Y, T.AT, T.CF$)
17. **End If**
18. **End Loop**
19. **End Loop**
20. **Return** TK

End Algorithm

3.4. Knowledge Decay

The process of knowledge decay (or simply put, forgetting) is triggered whenever a connection hasn't been used for a specified number of time steps. In its basic form, if a given connection hasn't entered working memory for a certain number of time steps, knowledge decay will be triggered for that connection and as a consequence its confusion interval is enlarged. In the chunking version, if a connection hasn't entered working memory for a number of steps that would normally trigger decay in the basic version and this connection is connected to a connection that entered working memory, then decay will not start and the confusion interval will instead be shortened. If however, a connection is not used for a number of time steps and it is not connected to a connection that enters working memory, then knowledge decay is triggered. Thus, LTM connections experience less decay in the chunking version of the decay process.

Below we present the algorithm for realizing knowledge decay in the context of the chunking mechanism. As mentioned earlier, if a connection is reachable from a connection found in the working memory, then that connection will not decay (the confusion interval will instead be shortened). We count the number of steps that passed since a connection entered working memory for the last time. This is done by incrementing the disuse attribute in line 11. In lines 2-6, we visit all connections that are reachable from the connections currently found in working memory and reset their disuse attribute to 0. In lines 7-16, we apply decay (increase the confusion interval length) for all connections that were not visited in lines 2-6 and haven't entered working memory for at least START_DECAY steps and at most END_DECAY steps. START_DECAY represents the maximum number of consecutive steps without triggering decay for a

connection that didn't enter working memory and hasn't been connected to a connection inside working memory. END_DECAY indicates how many steps the decay process is allowed to continue. Thus, after END_DECAY steps, the decay process is stopped even if the connection or any connection reachable from it hasn't yet entered working memory. In this manner, decay won't be triggered indefinitely for a connection that never enters working memory or does so in a very infrequent manner.

Algorithm decayKnowledgeGraph

Input: G_A – the knowledge graph (containing concepts) specific to a particular agent A

WM_A – the list of connections (edges) found in the working memory of agent A

DR – decay rate

START_DECAY – number of consecutive steps without triggering decay for a connection that didn't enter working memory and hasn't been connected to a connection inside working memory

END_DECAY – number of consecutive steps until decay is stopped (the connection still hasn't entered working memory)

Returns void

1. Initialize the Visited status of all edges e_i in G_A to false
2. **If** (agent is not idle) //agent learned or taught during the time step
3. **Loop** through all edges e_i in WM_A
4. Call **bfsDecayVisit**(G_A, e_i) // visit all graph edges that are reachable from e_i
5. **End Loop**
6. **End If**
7. **Loop** through all edges e_i in G_A

8. **If** e_i's Visited status is false

9. // e_i is not reachable from any connection found in WM_A; therefore it is not

10. // 'used' in the current step and the disuse attribute increases

11. $e_i.disuse \leftarrow e_i.disuse + 1$

12. **If** $e_i.disuse \in (START\_DECAY, END\_DECAY)$ then

13. $e_i.CI \leftarrow decayKnowledgeConnection(e_i.CI, DR)$

14. **End If**

15. **End If**

16. **End Loop**

End Algorithm

The actual decay algorithm for a single connection is named decayKnowledgeConnection and is called in line 13 of the decayKnowledgeGraph algorithm. The decayKnowledgeConnection algorithm simply implements the exponential growth model embedded in Equation (3.10).

Algorithm decayKnowledgeConnection

Input: CI – the confusion interval of a knowledge connection

DR – decay rate

Returns double (the adjusted CI value)

1. $CI \leftarrow CI \cdot e^{DR}$ // CI is lengthened by multiplying it with a value greater than 1

2. $CI \leftarrow min(CI, 1)$ // upperbound the CI

3. **Return** CI

End Algorithm

The algorithm bfsDecayVisit (called in line 4 of the decayKnowledgeGraph algorithm) performs a Breadth-First-Search of all connections that are reachable from a given connection and sets their disuse attribute to 0 (lines 7-11 in bfsDecayVisit).

Algorithm bfsDecayVisit

Input: G – a graph of concepts

e – an edge (a connection)

Returns void

1. Set e's Visited status to true.
2. Set e.disuse to 0.
3. Initialize a FIFO queue Q
4. Q.enqueue(e)
5. **While** Q is **not** empty
6. s ← Q.dequeue() // retrieve the front node/concept from the queue
7. **Loop** through all neighbors $n_{s,i}$ of s
8. Set $n_{s,i}$'s Visited to true
9. Set $n_{s,i}.disuse$ to 0.
10. Q.enqueue($n_{s,i}$)
11. **End Loop**
12. **End While**

End Algorithm

3.5. Agent Tasks

Similar to an agent's LTM, a task is represented by a weighted graph consisting of nodes that represent knowledge concepts and edges that represent the connections

between those concepts. In contrast to LTM, however, these connections do not have an associated "confusion interval". Each connection weight of a given task has to be matched within a certain margin of error by an agent's weight for that connection in its LTM in order for the agent to successfully solve the task. The process of attempting a task (Section 3.4.1) uses its own chunking mechanism in order to retrieve the necessary LTM chunks. The agent uses the retrieved chunks in order to match the task connection weights and eventually solve the task. After a task is attempted, an agent obtains feedback from the task. This process is called the task feedback process and is described in Section 3.4.2.

3.5.1. Task Attempt

In order to realize a chunking mechanism while an agent attempts a task, we have to start with the concepts required by the attempted task. These concepts are connected in a certain pattern that creates the task graph. Chunking can be easily realized by counting how many separate task sub-graphs are in a given task graph and then assume that each sub-graph is contained by one working memory slot.

For example, a task with 10 connections but only 3 separate sub-graphs containing those connections could be solved by an agent with a working memory of 3 or more slots. If the working memory capacity is below the number of separate sub-graphs then the task is abandoned. This is because there are not enough memory slots to fit every chunk found in the task graph.

Below we present three variants of this algorithm: taskAttempt_Basic, taskAttempt_Structure and taskAttempt_Structure_Weight. In the basic version, a chunk is at least one isolated concept. In the structure version, a chunk is at least one edge with

a confusion interval length below a specified threshold. If the number of chunks found in the agent's LTM is greater than the number of working memory slots, the task is abandoned as it means that the amount of LTM "units" that the agent has to retrieve and store—in order to solve the task—in its working memory is more than what its working memory can hold. Otherwise, the task is completed. In the following, taskAttempt_Basic invokes another function bfsVisit, while taskAttempt_Structure invokes bfsVisit_Structure, correspondingly. Conceptually, bfsVisit and bfsVisit_Structure are very similar. Both are used to visit neighboring, reachable concepts (nodes in bfsVisit) or connections (edges in bfsVisit_Structure) in a graph. In bfsVisit, a node n is considered reachable if there is a connection between the current node and n and that connection has a confusion interval smaller than the threshold CIT. Likewise, in bfsVisit_Structure, an edge c is reachable from a current edge e if that edge c has a confusion interval smaller than the threshold CIT.

CIT is a simulation constant and its range is between 0 and 1 (the maximum confusion interval length). The rationale for using CIT is to allow the simulation user to specify how strong the connections within a chunk should be (strong connections have small confusion intervals). If they are very strong, CIT can be set to a value close to 0 such as 0.1 and chunks will be formed only by connections having confusion interval lengths less than 0.1. During our experiments, we opted for the most explorative (less selective) process and thus considered chunks formed with connections of any strength. Therefore, we set CIT to the maximum value of 1.

Algorithm taskAttempt_Basic

Input: G_T – task graph, containing the concepts needed to complete a task,

W – the number of working memory slots

G_A – the knowledge graph (containing concepts) specific to a particular agent

CIT – threshold for the confusion interval for a connection for it to be visited

Returns a Boolean (true if task is accomplished, false otherwise)

1. Initialize the Visited status of all concepts c_i in G_A to false
2. Set connectedComponents to 0
3. **Loop** through all concepts c_i in G_T
4. Locate in G_A the corresponding c_j for c_i
5. **If** (c_j's Visited status is false) **then**
6. Increment connectedComponents by 1
7. Call **bfsVisit**(G_A, c_j, CIT) // change the Visited status of all concept nodes
8. // in G_A that are "reachable" from c_j to true.
9. **End If**
10. **End Loop**
11. **If** (connectedComponents > W) **then** // task too difficult for memory
12. **Return** false
13. **Else**
14. **Return** true
15. **End If**

End Algorithm

Algorithm bfsVisit

Input: G – a graph of concepts

c – a node (a concept)

CIT – required threshold for the confusion interval of a connection in order to visit it

Returns void

1. Set c's Visited status to true.
2. Initialize a FIFO queue Q
3. Q.enqueue(c)
4. **While** Q is **not** empty
5. s ← Q.dequeue() // retrieve the front node/concept from the queue
6. **Loop** through all neighbors $n_{s,i}$ of s (directly connected)
7. **If** (CI($n_{s,i}$, s) < CIT) **then** // CI(a,b) is the confusion interval of the
8. // connection a-b
9. Set $n_{s,i}$'s Visited to true
10. Q.enqueue($n_{s,i}$)
11. **End If**
12. **End Loop**
13. **End While**

End Algorithm

The second variant (taskAttempt_Structure) checks that the agent has the structure (set of connections) of the task in its knowledge (lines 3 – 11). In lines 12 – 16, it checks if the task is too complex for the agent's working memory. If it is too complex, then there are too many LTM chunks that have to enter working memory. In this case, there are not enough working memory slots to accommodate the task and the agent fails to solve the task (lines 12-13). If the task is not so complex, then the number of retrieved LTM

chunks is less than the working memory capacity and the agent consequently readily solves the task (lines 14-15).

Algorithm taskAttempt_Structure

Input: G_T – task graph, containing the concepts needed to complete a task,

W – the number of working memory slots

G_A – the knowledge graph (containing concepts) specific to a particular agent

CIT – threshold for the confusion interval for a connection for it to be visited

Returns a Boolean (true if task is accomplished, false otherwise)

1. Initialize the Visited status of all connections e_i in G_A to false
2. Set connectedComponents to 0
3. **Loop** through all connections e_i in G_T
4. Locate in G_A the corresponding e_j for e_i
5. **If** (e_j is **not** found **or** $e_j.CI >$ CIT) **then**
6. **Return** false
7. **Else If** (e_j's Visited status is false) **then**
8. Increment connectedComponents by 1
9. Call bfsVisit_Structure(G_A, e_j, CIT) // change the Visited status of all edges // in G_A that are "reachable" from e_j to true.
10. **End If**
11. **End Loop**
12. **If** (connectedComponents > W) **then** // task too difficult for memory
13. **Return** false
14. **Else**

15. **Return** true

16. **End If**

End Algorithm

Algorithm bfsVisit_Structure

Input: G – a graph of concepts

e – an edge (a connection of two concepts with a confusion interval)

CIT – threshold for the confusion interval for a connection for it to be visited

Returns void

1. Set e's Visited status to true.
2. Initialize a FIFO queue Q
3. Q.enqueue(e)
4. **While** Q is **not** empty
5. s ← Q.dequeue() // retrieve the front edge/connection from the queue
6. **Loop** through all neighboring edges $n_{s,i}$ of s (directly connected)
7. **If** ($n_{s,i}.CI < CIT$) **then** // (a.CI is the confusion interval of the edge a)
8. Set $n_{s,i}$'s Visited status to true
9. Q.enqueue($n_{s,i}$)
10. **End If**
11. **End Loop**
12. **End While**

End Algorithm

The third variant, taskAttempt_Structure_Weight is similar to taskAttempt_Structure. It checks to see that the structure of the task completely exists in the agent's knowledge

(lines 4-14). In lines 9-13 we count the connected components. We do this in order to see if the working memory capacity is large enough for attempting the task (lines 15-16). If it is, we proceed to the last step, namely checking to see if the weights used by the agent to attempt the task match the task weights. This is done in lines 17-23 by calling checkDistances method. We emphasize that this check for weight match is done only if the entire task structure has been found in the agent's knowledge (line 17). This makes sense since otherwise, we would check for weights without having the necessary connection in agent's knowledge or the confusion interval of such a connection would be too large (maximum value of 1). Thus, the only difference between taskAttempt_Structure_Weight and taskAttempt_Structure is that in the former one we make a more complex task attempt. This task attempt is mainly composed of two parts: the first one, identical to taskAttempt_Structure is to check whether the task structure is found in the agent's knowledge. The second part makes the difference between the two: we check whether the agent's weights used for attempting a task are close enough to the task weights. That is, this third variant is a more stringent version to make sure that an agent's knowledge has to match both structurally as well as in terms of weight in order to be able to attempt to solve a task.

Algorithm taskAttempt_Structure_Weight

Input: G_T – task graph, containing the concepts needed to complete a task,

W – the number of working memory slots

G_A – the knowledge graph (containing concepts) specific to a particular agent

CIT – threshold for the confusion interval for a connection for it to be visited

Returns a Boolean (true if task is accomplished, false otherwise)

1. Initialize the Visited status of all connections e_i in G_A to false
2. Set connectedComponents to 0
3. Set matchTaskStructure to true
4. **Loop** through all connections e_i in G_T
5. Locate in G_A the corresponding e_j for e_i
6. **If** (e_j is **not** found **or** $e_j.CI$ > CIT) **then** // that means the agent A does
7. // not know much about e_j
8. matchTaskStructure = false
9. **Else If** (e_j's Visited status is false) **then**
10. Increment connectedComponents by 1
11. Call **bfsVisit_Structure** (G_A, e_j, CIT) // change the Visited status of
12. // all edges in G_A that are "reachable" from e_j to true.
13. **End If**
14. **End Loop**
15. **If** (connectedComponents > W) **then** // task too difficult for memory
16. **Return** false
17. **Else If** ($matchTaskStructure\ is\ true$) **then**
18. Set matchTaskWeights to true
19. matchTaskWeights ← checkDistances (G_A, G_T)
20. **Return** matchTaskWeights
21. **Else**
22. **Return** false
23. **End If**

End Algorithm

Called in line 19 of the taskAttempt_Structure_Weight algorithm, the checkDistances method checks whether all the agent's weights used to attempt the task are close enough to the corresponding task weights. In order to do this, we loop through all connections in the task graph, locate the corresponding connection in agent's knowledge and pick up a uniformly distributed random value from the interval centered around the agent's knowledge connection weight (lines 3-5). If at least one random value is not close enough to the task weight (an error margin of 0.05), then the algorithm returns false (lines 6-10). Otherwise, it returns true in line 12. The rationale for the "close enough" design is to allow for "approximate matching" so that tasks can be attempted without having to exactly match the task connection weight values.

Algorithm checkDistances

Input: G_A – the knowledge graph (containing concepts) specific to a particular agent

G_T – task graph, containing the concepts needed to complete a task

Returns a Boolean (true if task is accomplished, false otherwise)

1. Set matchTaskWeights to true
2. Initialize the Matched status of all connections e_i in G_A to true
3. **Loop** through all connections e_i in G_T
4. Locate in G_A the corresponding e_j for e_i
5. agent_attempt_weight = $random\ (e_j.mean - e_j.CI/2, e_j.mean + e_j.CI/2)$
6. **If** $(e_i.mean - 0.05 \leq agent\_attempt\_weight \leq e_i.mean + 0.05)$ **then**
7. Set matchTaskWeights to false
8. Set $e_j's$ Matched status to false

9. **Return** matchTaskWeights
10. **End If**
11. **End Loop**
12. **Return** matchTaskWeights

End Algorithm

3.5.2. Task Feedback

The task feedback process is a reinforcement learning type of feature that occurs immediately after the task attempt process. If an agent solved a task, the weight centers for the agent's LTM connections corresponding to the task connections are set to the weight values randomly picked from the associated confusion intervals and such that all interval lengths are set to smaller values. This signifies that the agent has reached a higher level of confidence in its long-term knowledge about the connections involved in the solved task. In a similar fashion, humans also learn from accomplishing specific tasks, not only from what they are being taught by others (Shell et al. 2010). Correspondingly, if an agent failed to solve a task, the confusion interval lengths of the involved connections are increased. Similarly, after failing to accomplish a specific task, a person might explore other options of solving it (Shell et al. 2010). In C-ULM, this exploration for solutions is increased by the increase of confusion interval lengths. Thus, in a way, the "rewards" for solving or failing tasks are integrated into an agent's reasoning process as "self-efficacy"—confidence in what the agent knows, as in the shortening or lengthening of confusion intervals.

Below we present the algorithm taskFeedback that accomplishes the task feedback process. In case the agent failed to solve a task, we enlarge the confusion intervals for all

connections that were not matched during the task attempt process (lines 2 – 7). The rationale behind this design is to increase the capability to explore other values for those connections that were not matched. By increasing the confusion interval lengths for the unmatched connections, the range of potential values at the next time step is increased. We make the general assumption that when a task connection weight is not matched, the current agent's confusion interval doesn't contain the task weight. Following this assumption, we increase the agent's confusion intervals for the unmatched connections so that the chance for its new intervals to contain the task connection weights to increase. This also corresponds to decay: if an agent's connection is not matched at this time, its knowledge of the connection starts to decay, leading to the agent's confusion interval being increased. This confusion interval increase is given by the value of the input constant FF. FF is a simulation constant and its value (0.15) was chosen through various experiments testing for system performance.

If the agent successfully solved a task, we set its connection weights to the task weights and its confusion interval is set to a very small value given by the input constant SF (lines 8 – 11). SF is another simulation constant and the reason for it being very small (0.005) is to minimize confusion for the given agent connection involved in a solved task. By minimizing confusion, the motivation of the two concepts connected by this connection is sharply increased. Thus, as a teacher in subsequent time steps, the agent will feel strongly motivated to teach about the weight of this connection. As a learner, it will also feel strongly motivated to learn even more about this connection and there are chances he will learn about incorrect weights for this connection. As a potential direction for future work, the cic coefficient (Section 3.2.3.2.1) for connections involved in solved

tasks could be sharply decreased so that a successful agent becomes less prone to learning of incorrect weights after solving the task. Given this aspect and the fact that the teaching process changes only confusion intervals and not connection weights, a smaller cic value would enable the agent to teach the correct weights for a longer period of time. Another important reason for such a small SF value is to slow down—as a counterweight, sort of speak—potential confusion interval increases resulting from the knowledge decay process. Thus, the knowledge about the correct weights obtained by solving the task is kept for a longer period of time as compared to a scenario with a larger SF value.

Algorithm taskFeedback

Input: G_A – the knowledge graph (containing concepts) specific to a particular agent

G_T – task graph, containing the concepts needed to complete a task

FF – confusion interval update amount for failed task attempt feedback

SF - confusion interval for successful task attempt feedback

Returns a Boolean (true if task is accomplished, false otherwise)

1. Set matchTaskWeights to true
2. **Loop** through all connections e_i in G_A
3. Locate in G_T the corresponding e_j for e_i
4. **If** (e_i's Matched status is false) **then**
5. $e_i.CI \leftarrow enlargeConfusion(e_i.CI, FF)$
6. **End If**
7. **End Loop**
8. **If** ($matchTaskWeights\ is\ true$) **then**
9. $e_i.mean \leftarrow e_j.mean$

10. $e_i.CI \leftarrow SF$

11. **End If**

End Algorithm

3.6. Agent and task knowledge initialization

In this section we present the algorithms used for creating the initial agent and task knowledge graphs. During a simulation, the task knowledge graphs remain unchanged and the agent knowledge graphs evolve through the processes of learning, teaching, knowledge decay, and task attempts and completions.

To create agent and task knowledge, we follow a 2-step process: first, we create an initial knowledge graph for each agent and task. This step is performed by the method initKnowledgeTopology.

In contrast to the first step, the second step differs for the agent and task knowledge creation.

For agent knowledge creation, we make sure that all the connections present in any task knowledge graph are also present in at least one arbitrary agent knowledge graph. The rationale for this design is to have an agent connection space that completely includes the task connection space when the simulation is started. In this manner, any connection present in the tasks can be taught by at least one agent that contains that connection from the beginning of the simulation. The method that ensures this characteristic of the agent knowledge for our simulation is called agentKnowledgeCreation.

The method agentKnowledgeCreation follows the 2-step process described above in order to create the agent knowledge. The first step is performed by calling the method

initKnowledgeTopology. The second step is performed by using two help methods: findUnmatchedTaskConnection finds task connections that do not exist in any agent knowledge created by the method initKnowledgeTopology and getUCL_Agent_List finds a random list of agents for which we add connections found by the method findUnmatchedTaskConnection.

The first step in agent and task knowledge creation is represented by the initKnowledgeTopology method. This method creates the initial knowledge graph for an agent or task knowledge. This method was already used in the previous designs and is further used in the new algorithms agentKnowledgeCreation and taskKnowledgeCreation.

We use a double loop in lines 1-13 in order to access all pairs of available concepts. Since we use undirected graphs, we access each pair of concepts only once. We obtain a random value (dubbed connectionValue) in line 3 and compare it to a connectivity threshold in line 4 (connectionThreshold). If connectionValue is lower than connectionThreshold , then we create a new connection with a confusion interval and a weight center (lines 6-9) and add it to the initial knowledge graph (line 10). The underlying strategy of this initialization is to generate random values for the confusion interval and weight center of the connection between a pair of concepts c_i and c_j. The idea behind comparing the values of connectionValue and connectionThreshold is to ensure that not all possible connections are created in the agent knowledge. All values of connectionValue that are below connectionThreshold will lead to a new connection and all values of connectionValue that are above connectionThreshold will *not* lead to a new connection (lines 3 and 4). In this manner, the probability to generate a connection

between any two concepts is given by the value of connectionThreshold. Furthermore, this probability indicates the level of graph connectivity. Thus, if connectionThreshold has a low value then the graph connectivity is low (sparse graph); however, if connectionThreshold has a high value (close to 1) then the graph connectivity is high (dense graph). If connectionThreshold is 1 then we obtain a mesh topology for the initial agent knowledge.

Algorithm initKnowledgeTopology

Input: CL – list of available concepts

connectionThreshold – threshold that indicates the connectivity of the knowledge graph

Returns G_A – the initial knowledge graph

1. **Loop** through all concepts c_i in CL
2. **Loop** through all concepts c_j in CL
3. connectionValue = getDouble() // obtain a random value between 0 and 1
4. **If** (connectionValue < connectionThreshold)
5. //numConcepts = number of available concepts
6. maxConfusion = getDouble()
7. minConfusion = getDouble()
8. $e(c_i, c_j).\mathrm{CI}$ = maxConfusion – minConfusion
9. $e(c_i, c_j).\mathrm{weightCenter}$ = (maxConfusion + minConfusion) / 2
10. add $e(c_i, c_j)$ to G_A
11. **End If**
12. **End Loop**

13. **End Loop**
14. **Return** G_A

End Algorithm

One of the help methods used in the second step in agent knowledge creation is the getAgentSubList method. This method creates an arbitrarily sized and arbitrarily ordered set of agents. In order to create this set, we perform three steps: (1) we generate a random integer (in line 1) in order to determine the size of the agent set, (2) we randomize the order of the agents (in line 3) by using the shuffle method, and (3) we extract the first maxAgentIndex agents from the input list AL. We perform the third step in line 4 by using the minimum index of 1 and maximum index of maxAgentIndex. As its name implies, the rationale for creating this help method is to create a subset of the list of agents given as an input.

Algorithm getAgentSubList

Input: AL – agent list

Returns agentSubList – a subset of the list of agents given as input

1. maxAgentIndex = getInteger (1, AL.size)
2. // obtain a random value between 1 and AL.size
3. AL = Shuffle (AL) // we change the order of agents in the list in a randomized manner
4. agentSubList = AL [1, maxAgentIndex]
5. // we store in agentSubList a subset of the agent list AL; this subset starts with the
6. // first agent in AL and ends with the agent having the index maxAgentIndex
7. **Return** agentSubList

End Algorithm

Another help method used in the second step of agent knowledge creation is the insertTaskInformation method. This method updates an existent edge between two concepts in the agent knowledge or adds a new edge if there exists a task that connects the two concepts. In line 2 we ensure that each agent in the input list AL has a confusion interval between 0.1 and 1 for the edge that is being updated. In line 3 the weight of agent connection e is changed to a random value chosen in an interval centered on the task connection weight. In lines 4-6 we add connection e to the agent knowledge if this connection doesn't already exist in this knowledge graph. The rationale for this method is to insert the information related to a given task connection t_e (the connection itself and its required weight) in a set of agents given as input (AL). Of note is that we do not provide the agents with the exact information (the value for the assigned weight is not exactly the task weight value) and the confusion interval is at a minimum of 0.1 (line 2). In this manner, we show that the system is robust enough to solve complex tasks even if it is not given the exact information related to the existent task connections.

Algorithm insertTaskInformation

Input: AL – agent list

e – edge to be added to agent graph knowledge or only changed in terms of confusion interval and weight

t_e – task edge that relates the same two concepts as edge e

Returns void

1. **Loop** through all agents a_i in AL
2. $e.\mathrm{CI} = \mathrm{random}\ (0.1, 1)$

3. $e.\text{weight} = \text{random}\,(t\_e.\text{weight} - 0.05, t\_e.\text{weight} + 0.05)$
4. **If** $(e \notin a_i.G_A)$ **then**
5. add e to $a_i.G_A$
6. **End If**
7. **End Loop**

End Algorithm

Now, having described the help methods initKnowledgeTopology, getAgentSubList and insertTaskInformation, we are now ready to present our agentKnowledgeCreation and taskKnowledgeCreation methods as follows.

In the agentKnowledgeCreation method, the entire two-step process of creating the agent knowledge is executed. As mentioned before, the first step is done by calling the help method initKnowledgeTopology for each agent in the input list AL (lines 1-3). In the second step (lines 4-27), we make sure that each connection existent in a task can be found in at least one agent's initial knowledge (the knowledge that an agent has at the beginning of the simulation). By ensuring this characteristic of the set of initial agent knowledge graphs, every task connection exists in at least one agent knowledge graph at the start of the simulation (the simulation starts after agentKnowledgeCreation and taskKnowledgeCreation finish execution). In this manner, any task connection can be taught by at least one agent. We loop exhaustively through all pairs of concepts (lines 4-5) and consider the first task that contains a connection between the two concepts (lines 6-7). In lines 11-15 we create a set of agents that also contain the connection between the same two concepts c_i and c_j. If there is no agent that contains this connection, we obtain a subset of the entire list of agents AL by calling the method getAgentSubList and insert

the connection in each agent of this subset by calling the method insertTaskInformation (lines 16-18). If there are agents that contain the connection between concepts c_i and c_j, then we call method insertTaskInformation only for those agents (lines 19-20). In this manner, we add the connection to a set of agents (line 5 in insertTaskInformation) only if there are no agents that already have this connection after the execution of first step in agent knowledge creation (method initKnowledgeTopology). In line 23, we break from the task loop since the other tasks that present the same connection have also the same weight. This feature is ensured by the method taskKnowledgeCreation which is called before agentKnowledgeCreation.

According to the design of teachAllocate and teachProcess methods, a teacher can teach only connections that exist in its own knowledge. By executing the second step of agent knowledge creation (lines 4-27) we ensure that the task connection and weight spaces are completely included in the agent connection and weight spaces, respectively. Thus, all task connections and weights are 'teachable' right from the start of the simulation.

Algorithm agentKnowledgeCreation

Input: AL – agent list

TL – task list

CL – list of available concepts

connectionThreshold – threshold that indicates the connectivity of the knowledge graph

Returns void

1. **Loop** through all agents a_i in AL

2. $a_i.G_A$ = initKnowledgeTopology (CL, connectionThreshold)
3. **End Loop**
4. **Loop** through all concepts c_i in CL
5. **Loop** through all concepts c_j in CL
6. **Loop** through all tasks t_k in TL
7. **If** (task_e$(c_i, c_j) \in t_k.G_A$) **then**
8. **Set** MCL_Agent_List to empty
9. **Set** UCL_Agent_List to empty
10. **Loop** through all agents a_l in AL
11. **If** (e$(c_i, c_j) \in a_l.G_A$) // agent a_l contains the connection between
12. // c_i and c_j
13. add a_l to MCL_Agent_List
14. **End If**
15. **End Loop**
16. **If** (MCL_Agent_List is empty) **then** // no agent contains connection
17. // e(c_i, c_j)
18. UCL_Agent_List = getAgentSubList (AL)
19. insertTaskInformation (UCL_Agent_List, e(c_i, c_j))
20. **Else**
21. insertTaskInformation (MCL_Agent_List, e(c_i, c_j))
22. **End If**
23. Break
24. // we break from the task loop since other tasks containing connection

25. // task_e(c_i, c_j) have the same weight for this connection
26. **End If**
27. **End Loop**
28. **End Loop**
29. **End Loop**

End Algorithm

Algorithm taskKnowledgeCreation is the one that creates the knowledge graphs for tasks. As mentioned in the beginning of this section, the first step is the same as for agent knowledge creation. That is, the initKnowledgeTopology method is called for each task in order to create the initial knowledge graphs (lines 1-3). In the second step (lines 4-17) we make sure that the weight of a task connection has the same value in any task that contains the specified connection. For example, we have task 1 with connections AB and CD and task 2 with connections CD and WQ. When the loop in lines 5-17 ends execution, connection CD has the same required weight in both task 1 and task 2 (for example a required weight of 0.4). We use a double loop in order to check each possible pair of concepts (lines 5-6) and check all tasks to see if they have a connection between the two concepts (lines 7-8). When we find the first task that contains the current connection, we store its weight in a cell of matrix weightMatrix (lines 9-10). Subsequent tasks that also contain the current connection have their weight changed to the value stored in the weightMatrix cell (lines 11-12). In line 9, we rely on the fact that the weight of a connection cannot be 0 when it is created by lines 6-9 of method initKnowledgeTopology. With this assumption, if weightMatrix has a cell with a value of 0, the current task t_k is the first task that contains the current connection $e(c_i, c_j)$.

The rationale behind the execution of the second step of task knowledge creation is related to the ability of agents to learn, teach and attempt tasks using the most stringent task attempt method (taskAttempt_Structure_Weight). By executing this step, if an agent (AG1) solves a task that contains the above mentioned connection CD, it can teach this connection with a weight that is close to the task required weight (0.4 in the above example). Thus, another agent (AG2) that attempts another task that contains the same connection will benefit by learning from agent AG1. This is because the two tasks have the weight for a connection that involves the same pair of concepts. Without executing the second step of task knowledge creation, agents that solved tasks would mislead other agents attempting other tasks with the same connections.

Algorithm taskKnowledgeCreation

Input: TL – task list

CL – list of available concepts

connectionThreshold – threshold that indicates the connectivity of the knowledge graph

Returns void

1. **Loop** through all tasks t_i in TL
2. $t_i.G_A$ = initKnowledgeTopology (CL, connectionThreshold)
3. **End Loop**
4. Initialize weightMatrix to 0 //all matrix cells are set to 0
5. **Loop** through all concepts c_i in CL
6. **Loop** through all concepts c_j in CL
7. **Loop** through all tasks t_k in TL

8. **If** $(e(c_i, c_j) \in t_k.G_A)$ **then**

9. **If** $(\text{weightMatrix}[i, j] = 0)$ **then** //t_k is the first task containing

10. // $e(c_i, c_j)$

11. $\text{weightMatrix}[i, j] = e(c_i, c_j).\text{weightCenter}$

12. **Else** //t_k is not the first task containing $e(c_i, c_j)$

13. $e(c_i, c_j).\text{weightCenter} = \text{weightMatrix}[i, j]$

14. **End If**

15. **End If**

16. **End Loop**

17. **End Loop**

18. End Loop

End Algorithm

3.7. Relationship to ULM Learning Principles

According to the first ULM principle, learning is a product of working memory allocation. The C-ULM learning design for both learning and teaching agents follows this principle since what is being allocated into working memory using the methods learnAllocate_basic (Section 3.3.1.1) or teachAllocate_basic (Section 3.3.2.1) is further processed into the working memory using learnProcess (Section 3.3.1.3) and teachProcess (Section 3.3.2.3) methods. Then, the processed WM content changes the state of long – term memory and thus learning occurs. Thus, learning is the end result of working memory just as stated by the first ULM principle.

One of the rules related to the first ULM principle is that learning requires attention. By allowing into working memory only those concepts that have a motivation score

above the awareness threshold AT we have implemented this rule (Section 3.2.3.1). Thus, if there is enough motivation to attend a concept, then the concept will enter working memory and can be learned.

The second rule is that learning requires repeated attendance to what is being taught. This rule is incorporated in the repeated update of connection weights (Section 3.2.3.2.1) and confusion intervals (3.2.3.2.2). Thus, certain knowledge that is repetitively taught by a teacher can be learned by a learner by repetitive adjustments of the learner's weights towards the taught weights and repetitive updates of the associated confusion intervals.

The third rule states that learning is about connections. We have implemented this rule in C-ULM by using the weighted graph as the basic data structure of storing knowledge (Section 3.2.1). Thus, knowledge connections are represented by the graph edges and knowledge concepts are represented by the graph nodes. Furthermore, we have used variations of the Breadth-First-Search algorithm in order to identify the connected components of those weighted graphs and thus identify the knowledge chunks. In this manner, we were able to implement the chunking mechanism.

The second principle states that working memory's capacity for allocation is affected by prior knowledge**.** This principle is incorporated in the C-ULM chunking mechanism (algorithms learnAllocate_chunking in Section 3.3.1.2 and teachAllocate_chunking in Section 3.3.2.2). Thus, if existing chunks in the agent's LTM are rather small, then the number of concepts and connections entering working memory is also small. In contrast, if the LTM chunks are larger, then the number of concepts and connections entering working memory is also larger.

Finally, according to the third principle, motivation directs working memory allocation. In C-ULM, we relate motivation directly to WM allocation by comparing the motivation scores of knowledge concepts with the awareness threshold AT (Section 3.2.3.1). If the motivation to attend a certain concept is high enough to be above this threshold, then we allocate one working memory slot for that concept (in the basic allocation version) or for the LTM chunk containing that concept (in the allocation with chunking version).

Chapter 4. IMPLEMENTATION

4.1. Simulation details

Our C-ULM simulation is written in the Java language and is using the Repast library as the agent modeling framework (North et al. 2006). Repast is an open source toolkit and is one of several agent modeling frameworks that currently exist. One of the main goals of the Repast system is to provide support for flexible modeling of social agents. Furthermore, it allows for dynamic change of agent properties, agent behaviors and model properties.

An agent-based simulation typically proceeds in two stages. The first stage is the setup stage and it prepares the simulation for running. The second stage is the actual running of the simulation. In Repast simulations, the running of the simulation is divided into time steps or "ticks." During each time step, some action occurs and it uses the results of actions done in previous steps as its input.

4.1.1. Simulation input

Each simulation run is defined by a set of parameters that consists of the

- number of agents, tasks and concepts existent in the environment,
- the agent WM capacity,
- the normalization factor D,
- the number of simulation time steps, and
- the Repast random seed value.

The possible values taken by those parameters are defined in a text file having a Repast-based syntax for defining input parameters. Specifically, since we designed the parameter file with the intention to run the simulations resulting from all possible parameter configurations, we have used an embedded mode of defining each parameter. Thus, we defined the first parameter, 'number of agents' and set its possible values (such as 10, 20 or 30 agents). Then, inside the scope of this parameter, we have defined the second one – the 'number of concepts'. Thus, each parameter except the first and the last one defined, is within the scope of the previously defined parameter and includes in its scope the next parameter.

Below we present an example of a parameter file containing 3 parameters – 'number of agents', 'number of concepts' and 'number of tasks'. Of note, 'number of concepts' is within the scope of the 'number of agents' and 'number of tasks' is within the scope of the 'number of concepts'. This parameter file defines two simulation configurations: the first one has 10 agents, 20 concepts and 10 tasks; the second one has 10 agents, 20 concepts and 20 tasks (the number of tasks is the only one that differs between the two configurations).

```
runs: 1
NumberOfAgents
{
   set_list: 10
   {
                runs: 1
         NumberOfConcepts
         {
                    set_list: 20
             {
                runs: 1
                NumberOfTasks
                {  set_list: 10, 20  }
             }
          }
   }
}
```

For parallel execution of simulations, we use a cluster-based supercomputer called Tusker. Tusker is a 40 TF cluster consisting of 106 Dell R815 nodes using AMD 6272 2.1GHz processors, connected via Mellanox Quad Data Rate Infiniband and backed by approximately 350 TB of Terascala Lustre-based parallel filesystem. In order to run multiple simulations in parallel, we divided the parameter file into multiple files each of

which containing a subset of the initial set of parameter configurations. Then we ran the simulation with a different parameter file for each Tusker node being used.

4.1.2. Simulation output

The output of the C-ULM simulation is written in 4 csv files: Avg_evolution_data, Knowledge_evolution_data, Task_solving_data and End_sim_data. The first 3 files contain information regarding each time step of the executed simulations.

Thus, the Knowledge_evolution_data file contains data such as the index of the simulation (from an executed batch of simulations), time step values, number of agents, concepts and tasks used and agent working memory capacity. It also contains the average confusion interval and the average number of agent concepts and connections computed for each time step.

Similar to the Knowledge_evolution_data file, the Task_solving_data file contains data such as the agent working memory capacity, number of agents, concepts and tasks. Furthermore, it contains the number of solved and unsolved tasks and other performance metrics described in detail in Chapter 5, Results. The End_sim_data file contains the number of agents, concepts and tasks and it also contains the number of successful and unsuccessful task attempts. While the Knowledge_evolution_data and Task_solving_data files present the data for each time step, the End_sim_data file presents the data at the end of the simulation (the last time step).

The Avg_evolution_data file contains similar data to the Knowledge_evolution_data and Task_solving_data files but it displays the average over multiple simulation runs, each having a different random seed and all the other parameter values kept constant. Thus, we obtained a good average over several random seeds.

4.2. Class architecture

In this section we present and discuss the class architecture for the C-ULM simulation.

4.2.1. Overview

The main model class is ULMSimulationModel. This class extends the Repast SimpleModel class which offers the time-stepped simulation execution and provides the basic agent framework. The most important method of this class is the step method. In this method, we call the step method for each agent that in turn calls the learning and teaching methods. After learning and teaching is performed, we call for each agent the methods taskAttemptStep and decayKnowledge. The remaining part of the step method deals with creating and displaying the lines of the Avg_evolution_data file.

There is an abstract class for each of the following: agents (ULMAgent class), concepts (Concept class), knowledge (Knowledge class), motivation (Motivation class), working memory (WorkingMemory class) and tasks (Task class). For each of those abstract classes, there is a derived class that implements the required features. Each derived class is named by adding ‘Impl’ to the abstract class name. There is also an EdgeWeight class that stores the attributes corresponding to one knowledge connection.

4.2.2. UML class diagram

Below we present a brief UML class diagram and then we describe the main features of the aforementioned classes**.**

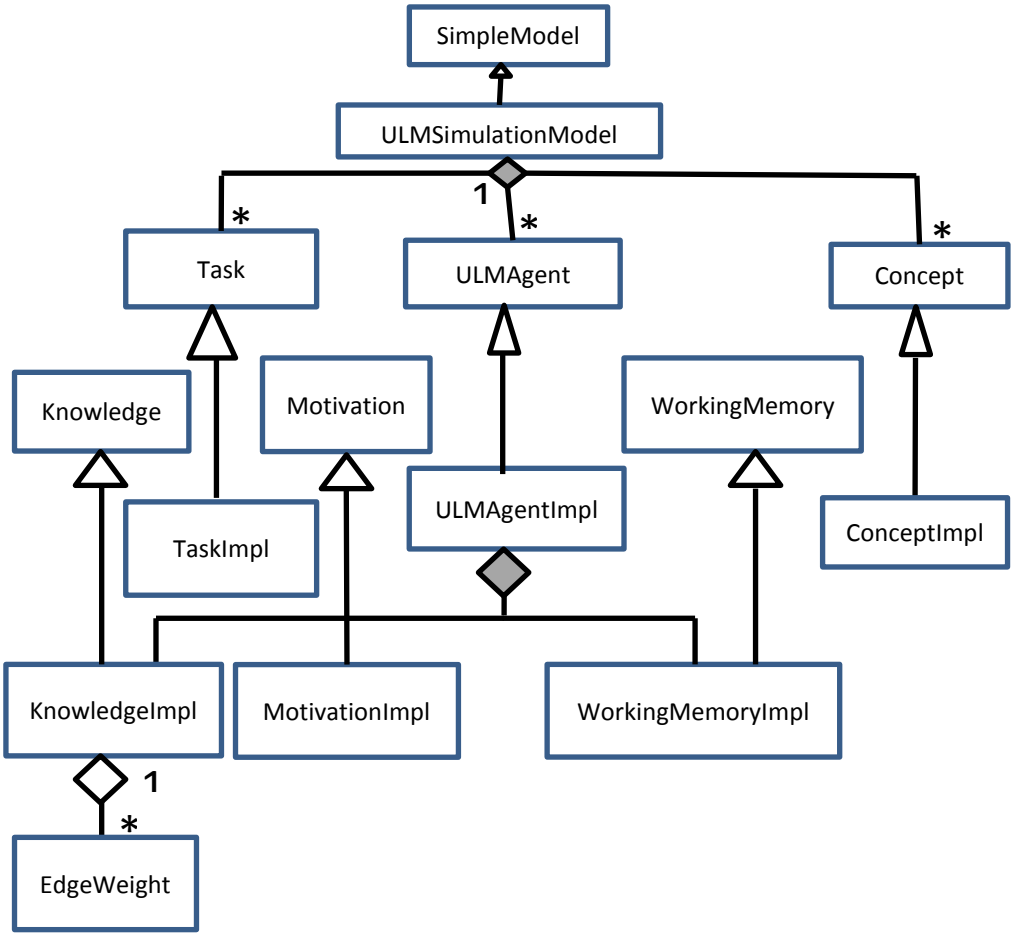

Figure 4.1 UML Class Diagram

4.2.3. Class description

4.2.3.1. ULMSimulationModel class

This is the ULM model class that inherits the simulation logic provided by the SimpleModel class. It overrides methods from the SimpleModel class such as the

buildModel and step methods. It also provides additional utility functions used by those two overridden methods and a number of getter and setter functions for various class attributes such as the number of agents, concepts and tasks, working memory capacity and the number of simulation time steps. As presented in the C-ULM class diagram (Figure 4.1), there is a one-to-many composition relationship between the ULMSimulationModel and the abstract classes for agents (the ULMAgent class), concepts (the Concept class) and tasks (the Task class).

4.2.3.1.1. buildModel method

One of the overridden methods of the SimpleModel class is the buildModel method. This method creates the C-ULM agent model and it can be summarized into 4 steps. In the first step, we set the random seed value and create a uniform distribution that will use this seed value throughout the current simulation run.

In the second step, we create the lists of concepts, tasks and agents. In the third step, we call the methods taskKnowledgeCreation and agentKnowledgeCreation methods. The first method implements the algorithm taskKnowledgeCreation and the second one implements the algorithm agentKnowledgeCreation. Both algorithms are presented in section 3.6.

Finally, the last step consists of creating 3 data recorder objects that deal with writing the performance metric data into the Knowledge_evolution_data, Task_solving_data and End_sim_data csv files. Those files are described in section 4.1.2.

4.2.3.1.2. step method

Another overridden method of the SimpleModel class is the step method. This method implements the entire behavior of the multi-agent system during each simulation time

step. The functionality of this method can be summarized into 4 stages. In the first stage, we shuffle the agent order in the list of agents so that at each subsequent simulation step, each agent can occupy a different position in the list. The agent position in the agent list is used as a priority value when performing the pairing of teacher to learner agents. Thus, shuffling the agent order ensures that each agent has approximately the same average priority over consecutive time steps.

In the second stage, we iterate through the agent list and call the step method associated to each individual agent. In the agent step method, we perform the actual teaching and learning algorithms.

In the third stage, we iterate again through the agent list and call the taskAttemptStep and the decayKnowledge methods for each agent. The first method performs the agent task attempt and the second method performs knowledge decay for the current time step.

Finally, in the fourth stage, we compute the performance metric averages and write them in the Avg_evolution_data file described in section 4.1.2.

4.2.3.2. ULMAgentImpl class

This class inherits the class attributes of the abstract ULMAgent class and it implements the agent logic. The most important methods are: step, taskAttemptStep, learn, teach, decayKnowledge and decideAction.

4.2.3.2.1. step method

This method implements the individual agent's behavior during one simulation time step. The functionality of this method can be summarized into 3 main steps. In the first step, the motivation scores for all concepts in the agent knowledge are updated by calling the method updateMotivationScores found in the MotivationImpl class.

In the second step, the agent decides whether it will teach or learn in the current time step. This action is performed by calling the method decideAction.

In the third step, an agent that decided to learn is matched with the first available teacher agent from the agent teacher queue. If there are no available teacher agents, then the learner agent is placed in the learner queue. Thus, it will be retrieved from this queue when a teacher agent needs to be matched with a learner agent. The same process is performed if an agent decided to teach. It will either be matched with the first available learner agent from the learner queue or be placed in the teacher queue.

4.2.3.2.2. taskAttemptStep method

This method calls the doTask method in order to perform the agent task attempt. If the agent fails to solve the task, then it randomly chooses a task from the list of unsolved tasks. Choosing a task is done by calling the method chooseTask. Of note, since this choice is random, the chosen task can be the current task that remained unsolved. If the agent solves the task, then the solved task is added to the list of solved tasks and removed from the list of unsolved tasks. Finally, the agent chooses another task to solve by calling the chooseTask method.

4.2.3.2.3. learn and teach methods

The learn method performs the agent learning algorithm by calling the learnAllocate and learnProcess methods found in the WorkingMemoryImpl class. The teach method performs the agent teaching algorithm by calling the teachAllocate and teachProcess methods found in the WorkingMemoryImpl class. learnAllocate and learnProcess methods implement the learning algorithms described in section 3.3.1. Similarly,

teachAllocate and teachProcess methods implement the teaching algorithms described in section 3.3.2.

4.2.3.2.4. decayKnowledge method

In order to perform the agent knowledge decay, this method calls the decayKnowledge method of the KnowledgeImpl class.

4.2.3.2.5. decideAction method

The decideAction method determines if the agent will teach or learn in the current time step and it consists of 3 steps. In the first step, a random number between 0 and 1 is generated. In the second step, this number is compared with the current probability to learn. If the number is less than the probability value, then the agent will learn during the current time step. Otherwise, it will teach. In the final step, the current probability to learn is decreased if the agent will learn during the current time step and is increased if it will teach. The rationale for the third step is to allow for a balance between the number of learners and the number of teachers across multiple time steps.

4.2.3.3. KnowledgeImpl class

The most important methods of this class are initTopology, knowledgeDecay and bfsDecayVisit. The method initTopology implements the algorithm initKnowledgeTopology described in section 3.6. Methods knowledgeDecay and bfsDecayVisit implement the algorithms decayKnowledgeGraph and bfsDecayVisit respectively (section 3.4). Other utility methods of this class include findConnectedConcepts, isIsolated and other setter and getter functions. The method findConnectedConcepts retrieves the list of all concepts connected to a given concept.

The method isIsolated checks whether a given concept is isolated or it is connected to other concepts within the agent knowledge.

4.2.3.4. MotivationImpl class

The most important method of this class is the updateMotivationScores. This method computes the motivation scores for all concepts in the agent knowledge. In order to compute the motivation score for a concept, it calls the utility method calculateMotivation which comprises of 3 steps.

In the first step, the knowledge related part of the motivation score (the self-efficacy component presented in Eq. 3.1) is computed by summing the inverse of confusion interval lengths for all agent knowledge connections. In the second step, the task related part of the motivation score (the expectancy of possible task rewards presented in Eq. 3.1) is computed by summing the rewards associated with each task that involves the given concept. The final motivation score is computed in the third step by multiplying the scores computed in the first two steps.

4.2.3.5. WorkingMemoryImpl class

WorkingMemoryImpl class implements the actual learning and teaching algorithms described in detail in sections 3.3.1 and 3.3.2.

4.2.3.6. ConceptImpl class

This class implements functionality required for a knowledge concept. It has attributes for describing and identifying a concept within a given agent knowledge. Since the C-ULM model allows for different abstraction layers, concepts can represent various objects according to a specific problem context. Thus, this class can be extended with

new classes that add specific domain functionality (such as neuron firing if a concept represents a neuron).

4.2.3.7. TaskImpl class

This class includes the methods that perform the actual task attempt described in algorithms taskAttempt_Basic, taskAttempt_Structure and task_Attempt_Structure_Weight (section 3.5.1). It also includes utility functions such as bfsVisit_Basic, bfsVisit_Structure and the checkDistances method (section 3.5.1). Furthermore, the task feedback algorithm presented in section 3.5.2 is also implemented in this class.

4.2.3.8. EdgeWeight class

This class includes the required attributes for each knowledge connection. Some of those attributes include the connection weight, the confusion interval length and a flag for marking the existence of a connection in an agent knowledge.

Chapter 5. RESULTS

5.1. Overview

The C-ULM simulation makes use of several simulation parameters that were discussed in Chapter 3. Below we present a table with a brief mention of their description location in Chapters 3 and 5 and their range of values as they are used in the simulation.

Table 5.1 Simulation parameters

Simulation parameters	Description	Range of values
Working memory capacity	Sections 3.2.3 & 5.3.1	3 – 9
Motivation factor (mf)	Section 3.2.3.2.2	Strictly

		positive
Spread normalization factor (D)	Sections 3.2.3.2.2 & 5.3.2	1, 2, 3, 4, 5
Learning coefficient for the confusion interval length (cil)	Section 3.2.3.2.2	0 – 0.01
Learning coefficient for the confusion interval center (cic)	Section 3.2.3.2.1	0.8 – 1.2
Awareness threshold (AT)	Section 3.2.3.1	0 – 1
Number of concepts	Section 5.3.3	10, 30, 50, 100
Number of agents	Section 5.3.4	10, 20
Number of tasks	Section 5.3.5	3, 10, 30, 50
Confusion interval update amount for failed task attempt feedback (FF)	Section 3.5.2	0.15 (constant)
Confusion interval for successful task attempt feedback (SF)	Section 3.5.2	0.005 (constant)

In order to investigate how the system behaves, we employ a set of performance metrics that describe the evolution of agent knowledge on one hand and the agent ability to solve tasks on the other hand.

The progress of agent knowledge over simulation duration is analyzed using the following 2 metrics (displayed in the Knowledge_evolution_data and Avg_evolution_data files mentioned in Section 4.1.2)

- **Number of agent connections**

  This metric shows how many connections an agent has at each simulation time step. It offers a step-by-step information on the agent knowledge connectivity. This metric is an average over all agents. This average is then averaged over 30 simulation runs each having a different random seed value.

- **Average confusion interval length**

  This metric shows the average length of the confusion interval for agent connections. It offers a step-by-step information on the agent certainty towards

existing knowledge. This metric is an average over all LTM agent connections. Similar to the number of agent connections metric, it is then averaged over 30 simulation runs.

Taken together, the above two metrics give a detailed image of both agent knowledge connectivity and certainty, key to evaluate the "quality" of an agent's knowledge.

The ability of agents to solve tasks is analyzed using the following 3 metrics (displayed in the Task_solving_data and Avg_evolution_data files mentioned in Section 4.1.2)

- **Average number of not acquired connections**

  This metric shows how many task connections are still missing from the agent knowledge. It offers information regarding how close is the agent knowledge structure to fully include the task structure. This metric is an average over all agents and 30 simulation runs. The average is computed in a similar manner to the number of agent connections metric.

- **Average weight differences**

  This metric shows the average weight difference between an agent knowledge weight and the corresponding task weight. It offers information regarding how close are the agent knowledge weights to the task required weights. This metric is an average over the weight differences between all LTM agent connections and their corresponding task connections. The average is computed in a similar manner to the average confusion interval length metric.

- **Total number of solved tasks**

This metric offers information on how many tasks have been solved by the entire multi-agent system at each simulation time step. It is averaged over 30 simulation runs so that it is not sensible to changes in the random seed.

Taken together, the 3 metrics offer a detailed image (knowledge structure and weights) of the agent progress towards solving tasks.

In order to analyze the system behavior using the aforementioned measures, we perform several experiments that can be grouped into two main categories:

- studying the impact of using the chunking mechanism on agent knowledge and task performance
- studying the impact of several key factors on the C-ULM with chunking system

5.2. Impact of chunking on agent knowledge and task performance

In this section we compare the results obtained in the C-ULM system without chunking and the one that uses the chunking mechanism.

5.2.1. Impact on agent knowledge

In this section we investigate the impact of using the chunking mechanism on the evolution of agent connections and confusion intervals. The significance of this investigation is to validate the use of chunking in relation to existing human studies. In order to analyze the impact of chunking on agent knowledge we present our findings through Figures 5.2.1.1 – 5.2.1.12.

Observation 1.

Chunking makes a positive impact over the non-chunking version in the average number of agent connections (Figures 5.2.1.1 and 5.2.1.2). In the chunking version (Figure 5.2.1.1) the agents reach a higher number of connections than in the non-

chunking version (Figure 5.2.1.2). Similar to what has been observed in human studies, chunking leads to a higher agent connectivity than the one obtained without using this mechanism.

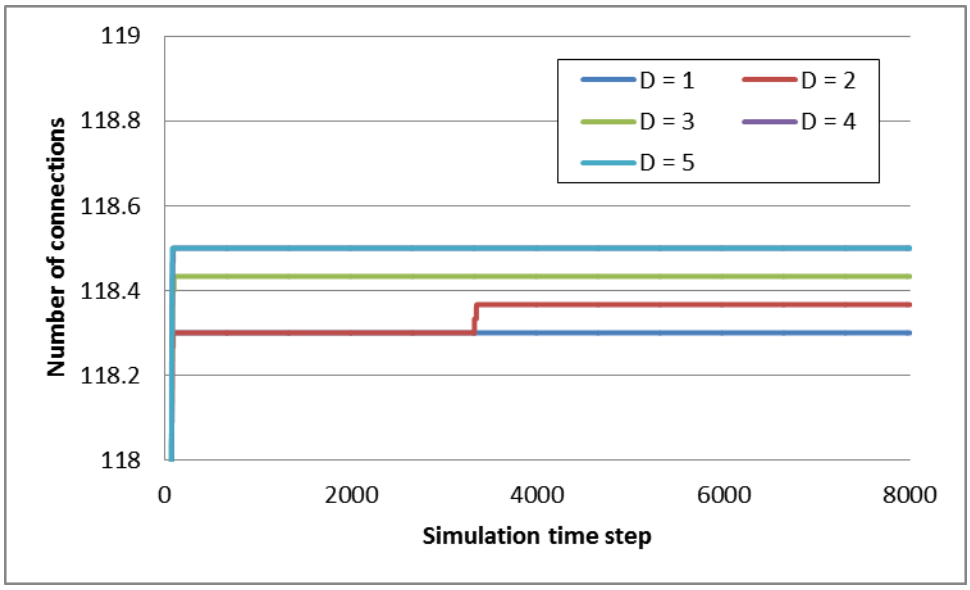

Fig. 5.2.1.1 Average number of agent connections: WM = 5; D = 1-5 (chunking); average task complexity: 8 (30 available concepts)

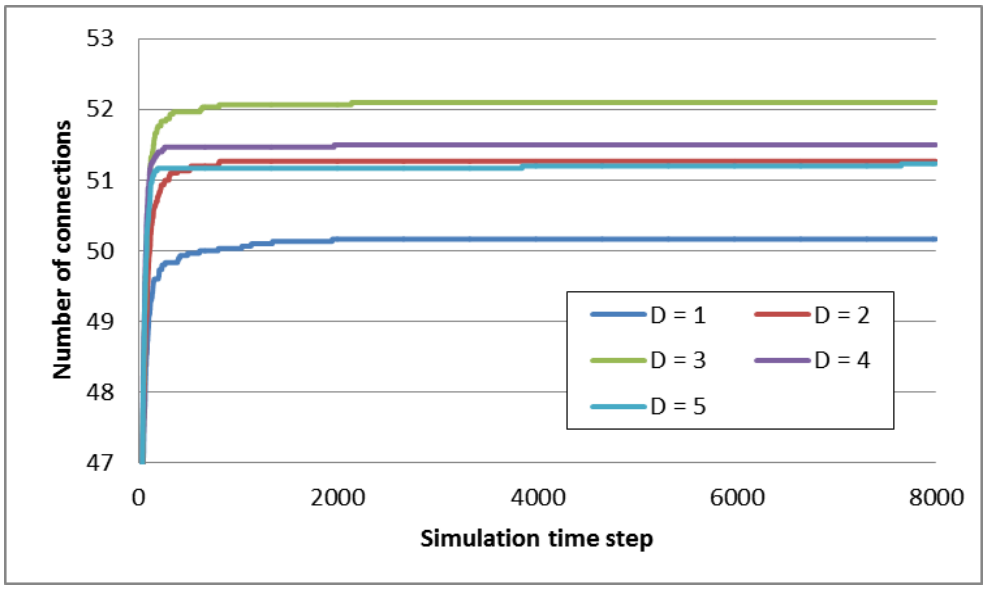

Fig. 5.2.1.2 Average number of agent connections: WM = 5; D = 1-5 (no chunking); average task complexity: 8 (30 available concepts)

Observation 2.

The evolution of confusion interval length is totally different in the chunking version as compared to the non-chunking version (Figures 5.2.1.3 and 5.2.1.4). In the chunking version, the confusion interval length drops extremely fast at values around 0.02 and then increases to values between 0.08 and 0.1 (Figure 5.2.1.3). In the non-chunking version, the interval length decreases very fast in the beginning and then continues to decrease asymptotically towards values between 0.01 and 0.02 (Figure 5.2.1.4).

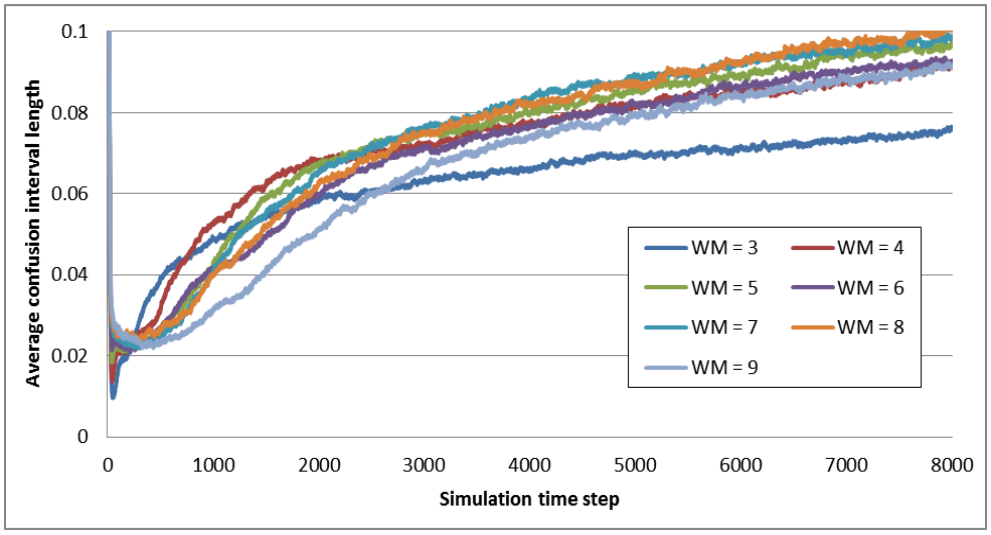

Fig. 5.2.1.3 Average confusion interval length: WM = 3-9; D = 5 (chunking); average task complexity: 8 (30 available concepts)

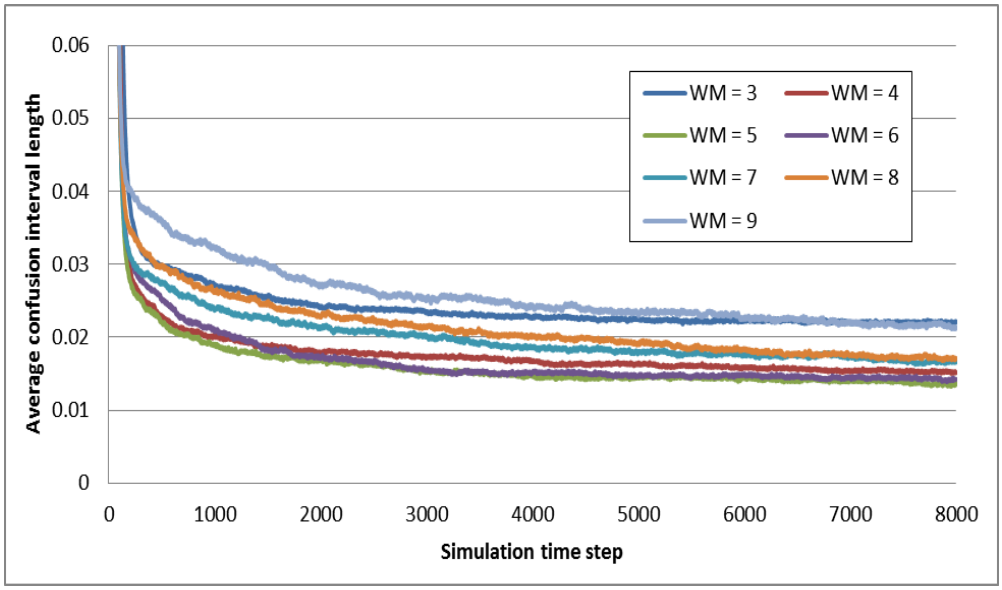

Fig. 5.2.1.4 Average confusion interval length: WM = 3-9; D = 5 (no chunking); average task complexity: 8 (30 available concepts)

The reason for the different behaviors observed in Figures 5.2.1.3 and 5.2.1.4 is given by the compounded effect of (1) having enough memory for learning and teaching and (2) the usage of the method enlargeConfusion whenever a task connection has not been matched by the corresponding agent weight. Specifically, only with enough memory capacity and the usage of enlargeConfusion calls is the system able to obtain the confusion interval behavior observed in Figure 5.2.1.3 as compared to Figure 5.2.1.4. As observed in Figures 5.2.1.4 and 5.2.1.7, omitting any of those two features leads to a confusion interval dynamic that doesn't present the curve shown in Figure 5.2.1.3.

Incidentally, what we observed in Figure 5.2.1.3 is similar to what we also observed in another scenario where the WM is set to 30, and where no chunking is used for learning and teaching, as shown in Figure 5.2.1.5. This further confirms that memory capacity is indeed crucial in order to obtain a curve in the confusion interval dynamic as observed in Figure 5.2.1.3.

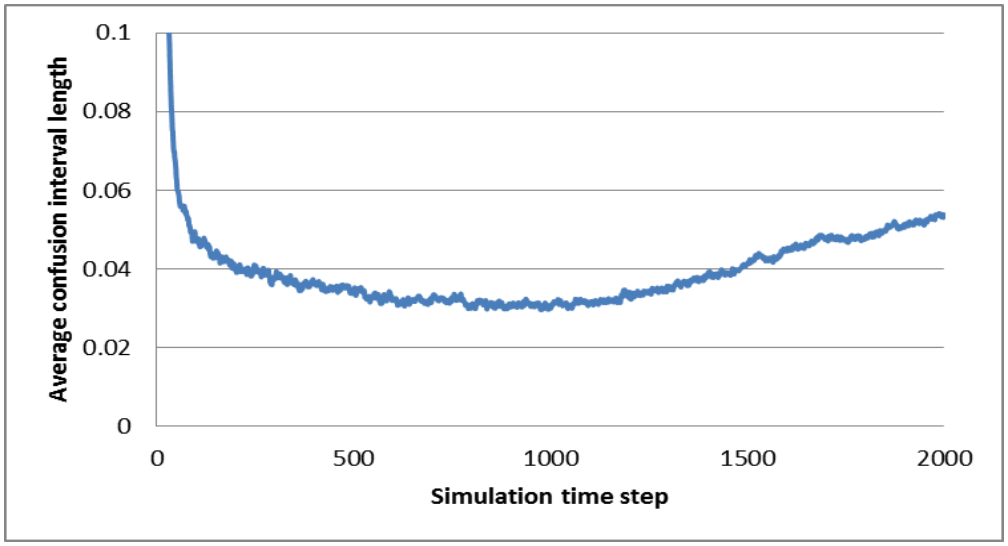

Fig. 5.2.1.5 Average confusion interval length: WM = 30; D = 5 (no chunking); average task complexity: 8 (30 available concepts)

Of note, chunking when attempting the task (retrieving the necessary information from agent knowledge in order to solve the task) is performed in both scenarios displayed by Figures 5.2.1.3 and 5.2.1.4. The difference is made by the fact that for Figure 5.2.1.3 chunking is also used when allocating WM for learning and teaching while for Figure 5.2.1.4 chunking is not used for those purposes and is only used when attempting to solve the tasks.

In order to further understand the behavior observed in Figures 5.2.1.3 and 5.2.1.4, we have drawn Figures 5.2.1.6 - 5.2.1.9. Those Figures help explain the impact of enlargeConfusion method calls on the confusion interval dynamics.

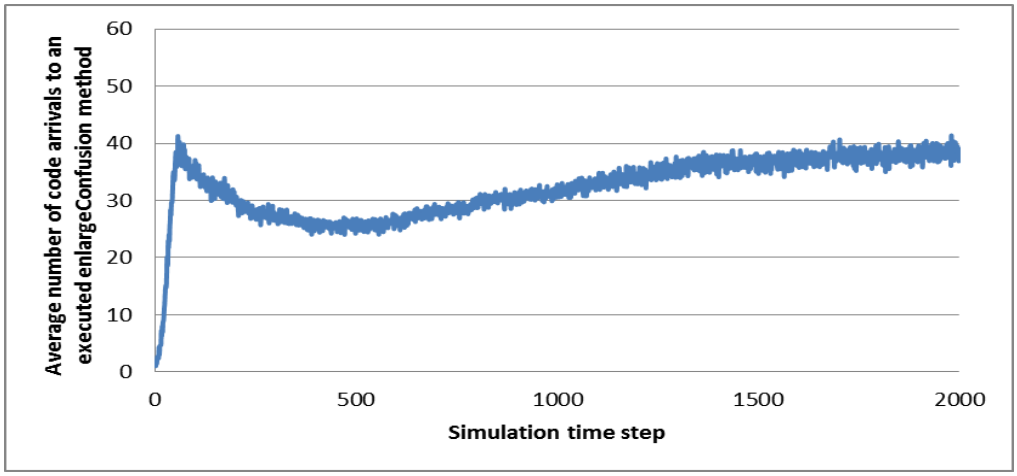

Fig. 5.2.1.6 Average number of code arrivals to an executed enlargeConfusion method: WM = 5; D = 5 (chunking);

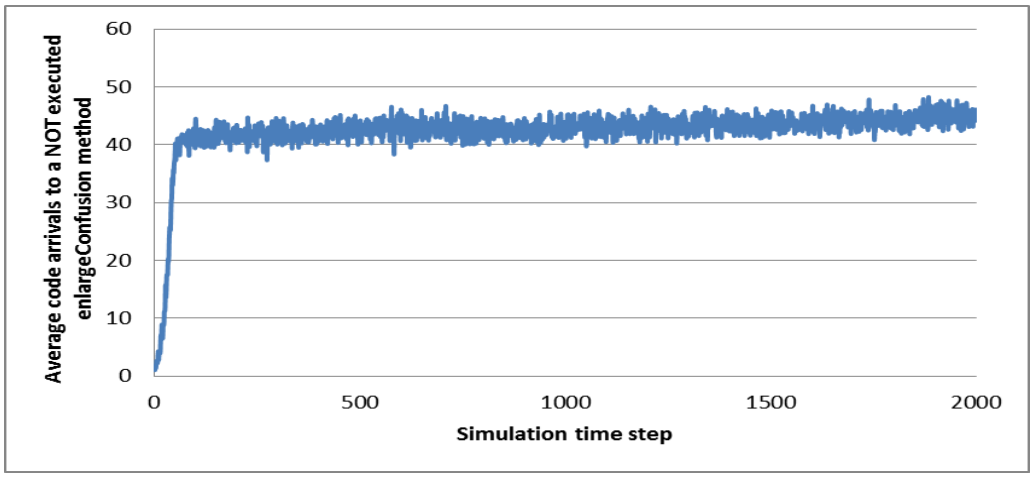

Fig. 5.2.1.7 Average number of code arrivals to a potential but NOT executed enlargeConfusion method: WM = 5; D = 5 (chunking);

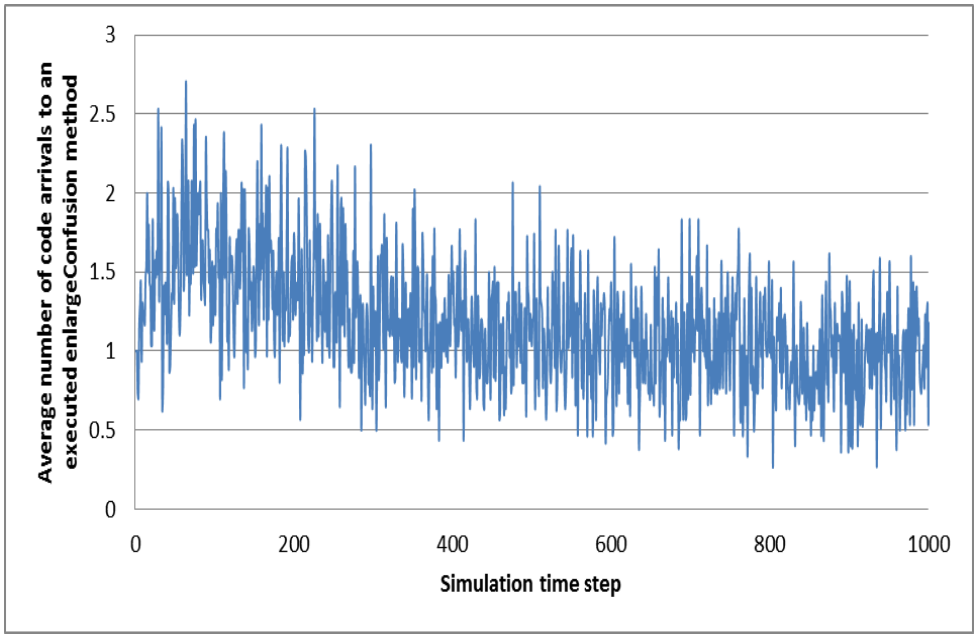

Fig. 5.2.1.8 Average number of code arrivals to an executed enlargeConfusion method: WM = 5; D = 5 (no chunking);

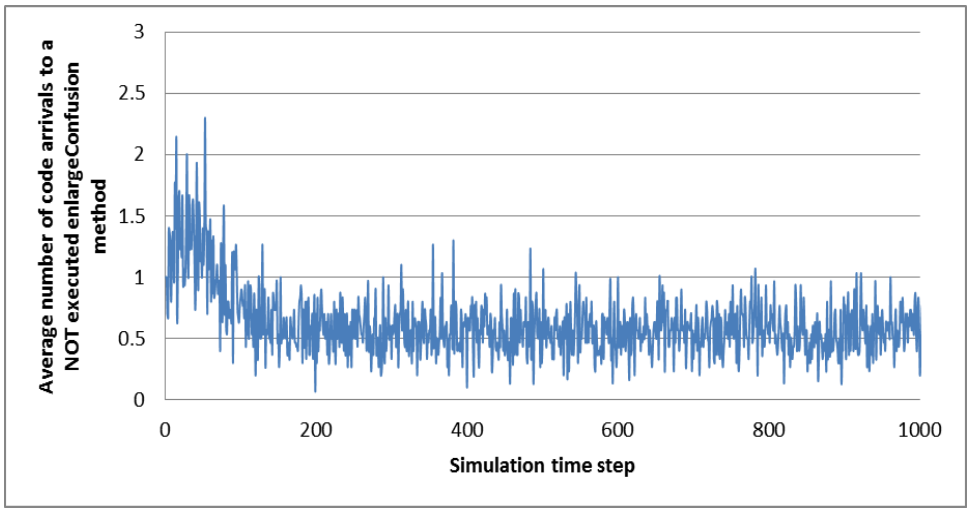

Fig. 5.2.1.9 Average number of code arrivals to a potential but NOT executed enlargeConfusion method: WM = 5; D = 5 (no chunking);

Figures 5.2.1.6 and 5.2.1.7 sustain the claim that the call to the enlargeConfusion method is absolutely necessary in order to obtain the behavior seen in Figure 5.2.1.3. In Figure 5.2.1.6, after an initial spike in the number of calls, we have a decrease and then an increase in the number of enlargeConfusion method calls. The obtained curve

resembles the one observed in Figure 5.2.1.3. In Figure 5.2.1.7 there is a similar initial spike but then the number of enlargeConfusion calls follows a slightly ascending trend.

The reason for the enlargeConfusion method making the observed impact is related to its purpose and to the manner this method is called. The purpose of the method is to increase the confusion interval length for an agent connection that didn't match in weight the corresponding task connection. In the subsequent steps, the agent weight is randomly picked from a larger interval thus increasing the chance that the new confusion interval includes the required task weight. Furthermore, this call to enlargeConfusion method is not done for agent connections where the weight did match. This discrimination – of calling this method only for connections where the weight did not match has an important role in the impact observed in Figures 5.2.1.3 and 5.2.1.6 as compared to Figures 5.2.1.4 and 5.2.1.7. This is clearly outlined by Figure 5.2.1.10 where the call to enlargeConfusion method is done for each agent weight in case of a failed task attempt (i.e. for both matched and unmatched connections). In this Figure, the confusion interval length is more similar to the one observed in Figure 5.2.1.4.

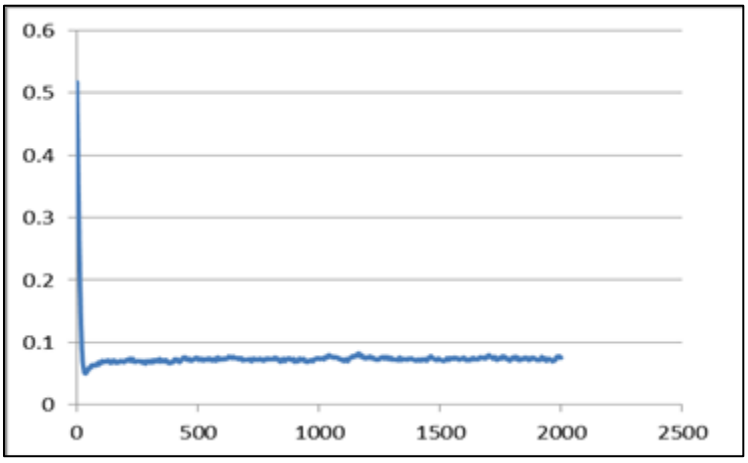

Fig. 5.2.1.10 Average confusion interval length: WM = 5; D = 5 (chunking);

Therefore, the agents receive, in the failed task feedback, a connection-by-connection information that leads to the observed confusion interval behavior and the associated gain in number of solved tasks. We mention a connection-by-connection information since the agents enlarge confusion interval only for unmatched connections. If the method is never called (Figures 5.2.1.7 and 5.2.1.9) or if it is called for each agent weight in case of a failed task attempt (Figure 5.2.1.10), there is nothing that discriminates between matched connections and unmatched connections in the case of a failed task attempt (in terms of feedback received by the agent). However, if the method is called only for unmatched connections during a failed task attempt, we obtain the behavior observed in Figures 5.2.1.3 and 5.2.1.6.

We have to point out though, that in the step following a failed task attempt, the same agent attempting the same task could come up with weights that do not match previously matched connections and vice-versa. Therefore, the enlargeConfusion method could be called in this step for a connection that hasn't 'experienced' this call in the previous step. Given this last mentioned fact, the added information given to agents on a connection-by-connection basis cannot fully account for the success obtained only in the scenarios with enough WM (either by chunking or high WM capacity) and the usage of enlargeConfusion method calls.

The comparison between Figures 5.2.1.6 and 5.2.1.8 shows that in the chunking version the number of enlargeConfusion method calls is much higher than in the non-chunking version. This is the main reason behind the confusion interval length increase observed in Figure 5.2.1.3 (chunking) as compared to Figure 5.2.1.4 (non-chunking).

The behavior observed in Figures 5.2.1.3 and 5.2.1.4 is also influenced by the number of knowledge decay calls made for each agent in each time step. In order to understand this impact we have drawn Figures 5.2.1.11 and 5.2.1.12.

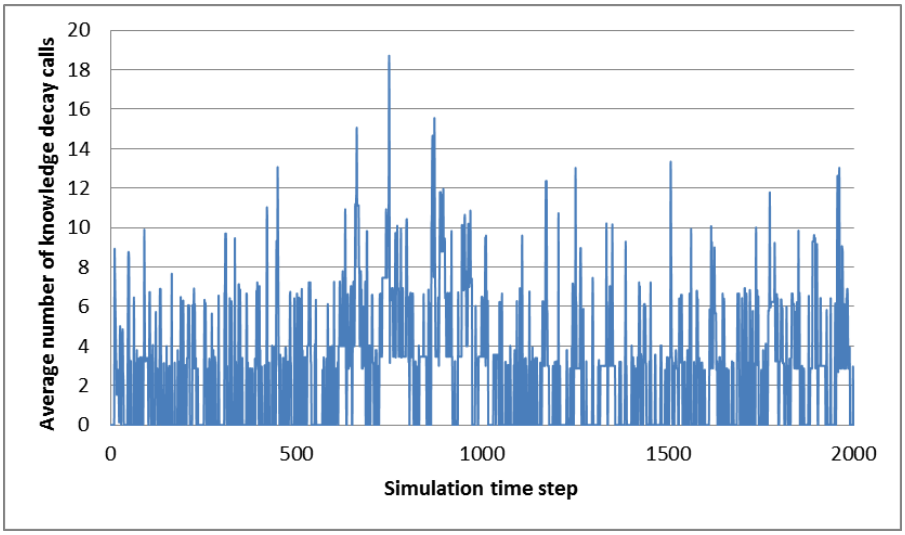

Fig. 5.2.1.11 Average number of knowledge decay calls: WM = 5; D = 5 (chunking) (with enlargeConfusion call);

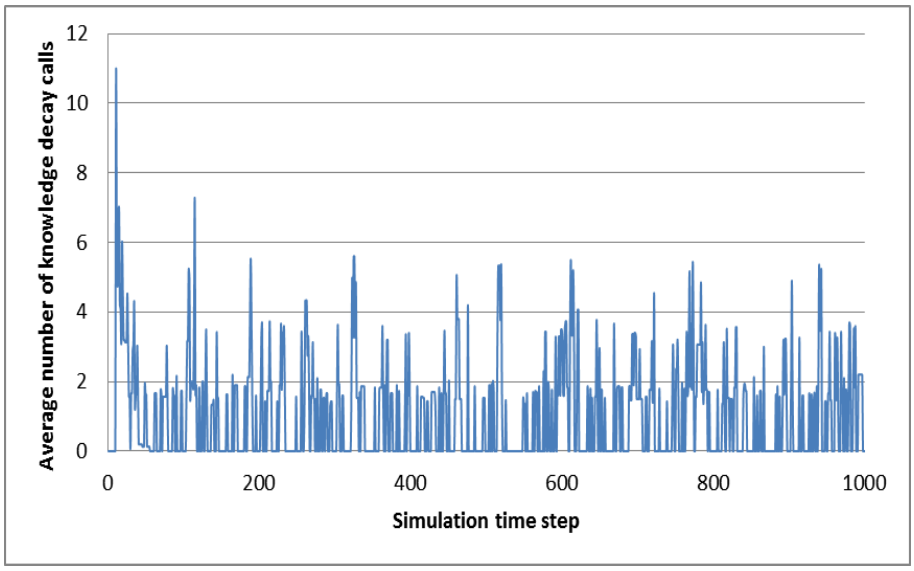

Fig. 5.2.1.12 Average number of knowledge decay calls: WM = 5; D = 5 (no chunking) (with enlargeConfusion call);

As we can see from Figures 5.2.1.11 and 5.2.1.12, chunking leads to more knowledge decay calls. This means that more connections experience knowledge decay. Therefore, more connections have their confusion interval enlarged in the chunking-version as compared to the non-chunking version due to knowledge decay. Just as the difference between Figures 5.2.1.6 and 5.2.1.8, the difference in knowledge decay calls can also help explain why the confusion interval length starts to increase in Figure 5.2.1.3 (chunking) as compared to Figure 5.2.1.4 (non-chunking).

On the other hand, in the chunking version, more connections are learned and taught in each time step which leads to more method calls that shorten the confusion interval length.

The most plausible conclusion is that the interplays among (1) enough working memory capacity, (2) enlargeConfusion method calls that are done only for unmatched connections in each time step, (3) more connections experiencing knowledge decay and (4) more connections having their confusion intervals shortened due to chunking while learning and teaching led to the behavior observed in Figure 5.2.1.3 as compared to Figure 5.2.1.4. The mix of those factors led to a balance between exploration and exploitation that eventually led to a totally different behavior in Figure 5.2.1.3 as compared to Figure 5.2.1.4. Specifically, Figure 5.2.1.3 displays a V-shape behavior of confusion interval length that doesn't appear in Figure 5.2.1.4.

From a neural point of view, there are four neural learning mechanisms that are modeled within C-ULM. The first two mechanisms are based on a long-term repetition process and thus they implement the law of exercise. The first one is neuronal synapse strengthening and is modeled by shortening confusion intervals during repeated learning

events. The second one refers to pruning out unused neural synapses. Similar to synapse strengthening, this process is also a long-term one and is modeled within C-ULM by lengthening the confusion intervals during the knowledge decay process.

The third and the fourth mechanisms are short-term reinforcement based processes and thus they implement the law of effect. The first reinforcement-based process is active inhibition and it refers to short-term inhibition of specific synapses and also to slowing down neural firing rates between neurons. This process is modeled within C-ULM by the calls to the enlargeConfusion method for unmatched connections whenever an agent failed to solve a task. By calling the enlargeConfusion method for unmatched connections, the motivation to learn about those connections in the future decreases. Because of this fact, the motivation scores for those connections will less likely be above the awareness threshold and as a result the chance of unmatched connections to enter working memory in future time steps decreases. Thus, active inhibition is modeled by the enlargeConfusion method calls that lead to a lower likelihood of unmatched connections to enter working memory.

The second reinforcement-based process is active excitation and it refers to short-term excitation of specific synapses and to increased firing rates between neurons. C-ULM incorporates this mechanism by shortening confusion intervals for agent connections involved in a successful task attempt. This leads to increased motivation to perform learning and teaching related to those connections and thus an increased likelihood that they will enter agent working memory in subsequent time steps.

Within the context of those four neural mechanisms, we can observe that by using chunking, learning activity performed with all four mechanisms increases. Thus, long-

term repetition based excitation increases as observed in Figure 5.2.1.1 as compared to Figure 5.2.1.2. Long-term repetition based inhibition (knowledge decay) increases as observed in Figure 5.2.1.11 as compared to Figure 5.2.1.12. Finally, short-term reinforcement based processes are also higher in the chunking mechanism as seen in Figure 5.2.1.6 and 5.2.2.1 as compared to Figures 5.2.1.8 and 5.2.2.2 respectively. Of note, Figures 5.2.2.1 and 5.2.2.2 refer to the number of solved tasks presented in section 5.2.2 and are also a direct representation of the number of short-term active excitations performed due to solved tasks.

Therefore, we can state that chunking increases all excitatory and inhibitory processes and in doing so the learning behavior is dramatically changed (Figure 5.2.1.3 as compared to Figure 5.2.1.4).

5.2.2. Impact on agent effectiveness and efficiency

The aim of this section is to study how chunking affects the ability of agents to solve tasks (agent effectiveness) and how fast are tasks solved (agent efficiency). This is important since it shows whether chunking is useful for boosting agent effectiveness and efficiency. Those two agent characteristics are described by the three performance metrics mentioned in Section 1: average number of not acquired connections, average weight differences and total number of solved tasks.

Observation 1.

In the chunking version (Figure 5.2.2.1), the total number of solved tasks is much higher than in the non-chunking version (Figure 5.2.2.2). This can be seen by looking at the asymptotic value reached in the two Figures. This asymptotic value represents how many tasks are solved at the end of the simulation and it describes the agent

effectiveness. Therefore, agent effectiveness is greatly enhanced by the use of chunking. Due to the tremendous difference in agent effectiveness, agent efficiency doesn't play a role when comparing Figures 5.2.2.1 and 5.2.2.2.

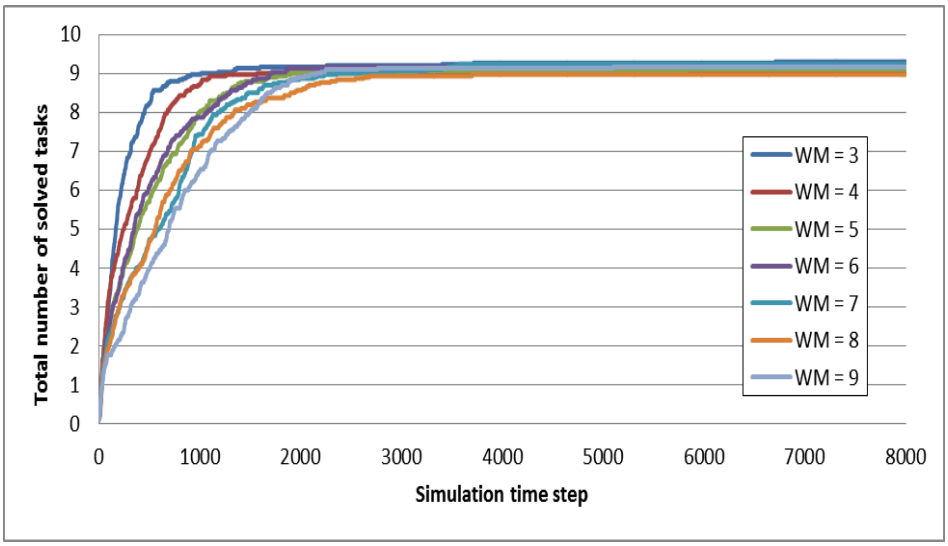

Fig. 5.2.2.1 Total number of solved tasks: WM = 3-9; D = 5 (chunking); average task complexity: 8 (30 available concepts)

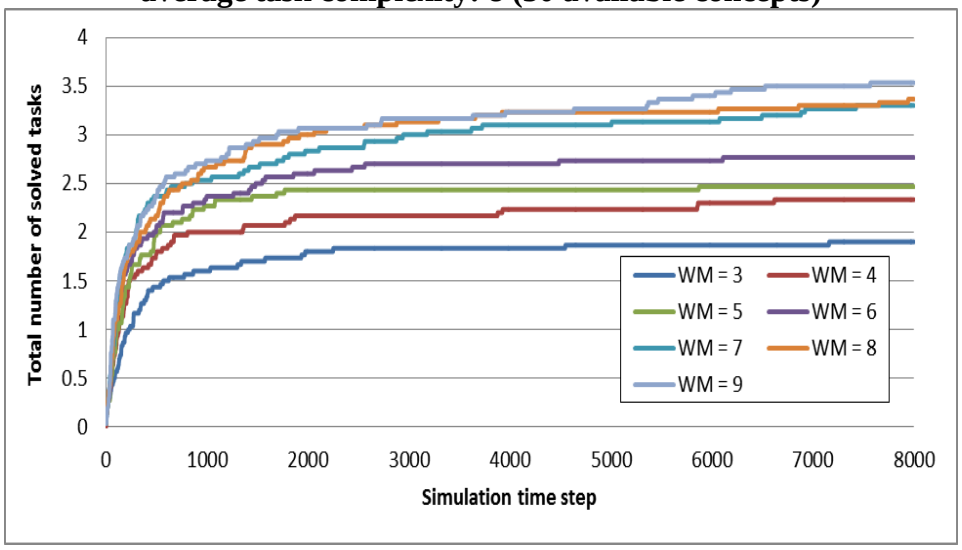

Fig. 5.2.2.2 Total number of solved tasks: WM = 3-9; D = 5 (no chunking); average task complexity: 8 (30 available concepts)

Observation 2.

In the chunking version (Figure 5.2.2.3), the average weight differences reach a lower asymptotic value than in the non-chunking version (Figure 5.2.2.4). A lower asymptotic value in the chunking version signifies that the agent knowledge weights get closer to the task required weights than in the non-chunking version. Because of this, the agents in the chunking version present increased chances of solving a task, therefore having higher effectiveness and efficiency than in the non-chunking version.

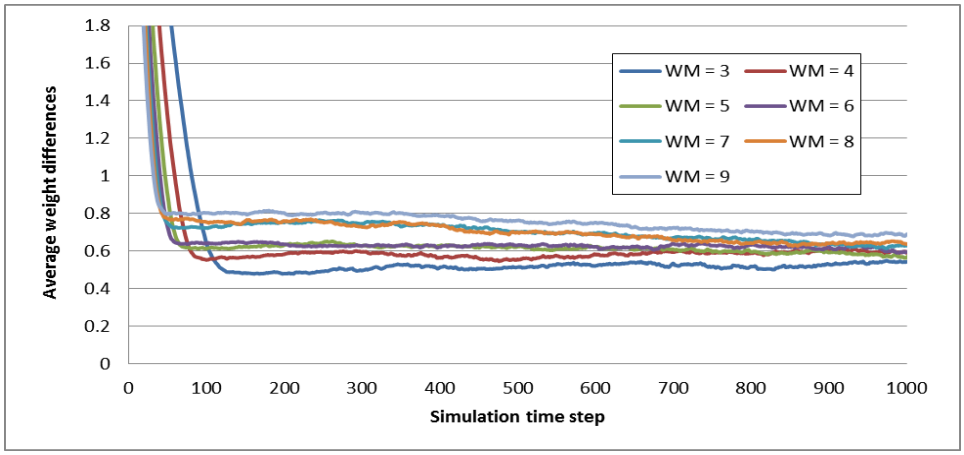

Fig. 5.2.2.3 Average weight differences: WM = 3-9; D = 5 (chunking); average task complexity: 8 (30 available concepts)

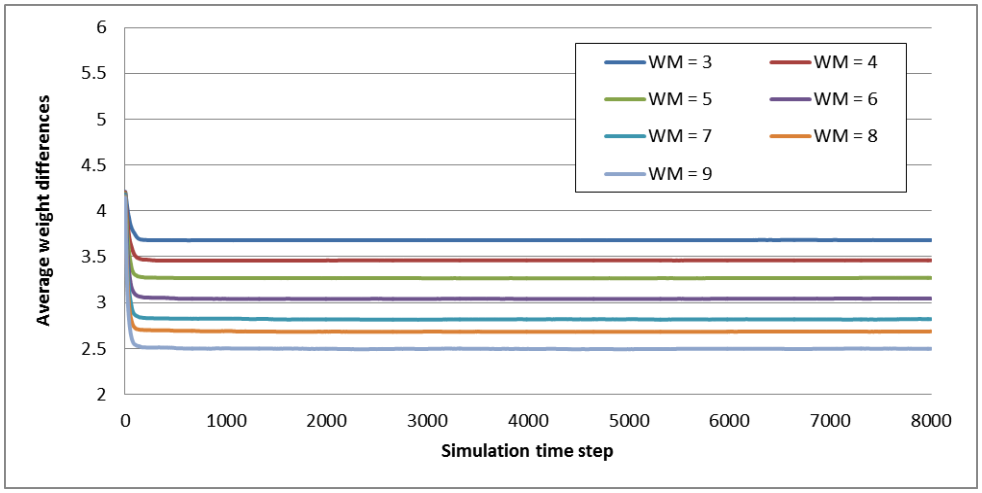

Fig. 5.2.2.4 Average weight differences: WM = 3-9; D = 5 (no chunking); average task complexity: 8 (30 available concepts)

Observation 3.

In the chunking version (Figure 5.2.2.5), the average number of not acquired connections reaches a much lower asymptotic value (0) than in the non-chunking version (Figure 5.2.2.6). The results described by the two Figures are in accordance with the previous two observations in the sense that the chunking version performs much better than the non-chunking one.

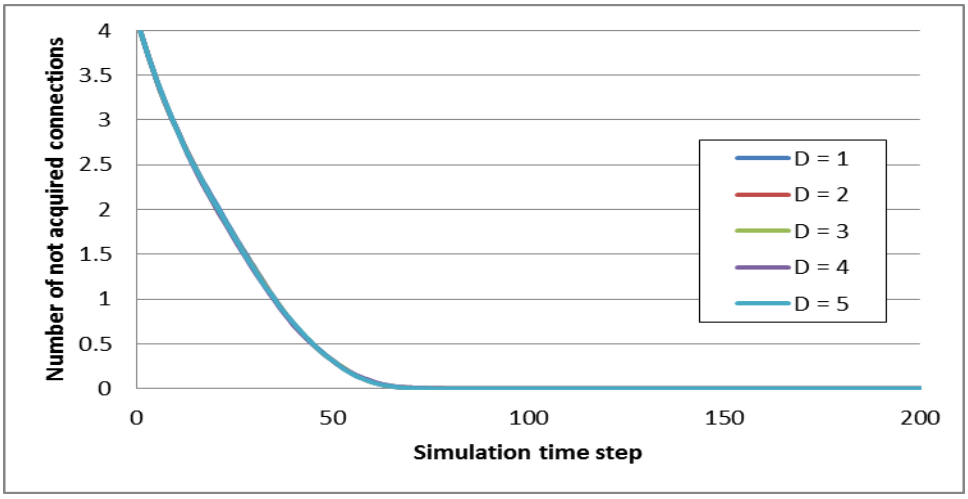

Fig. 5.2.2.5 Average number of not acquired connections:
WM = 5; D = 1-5 (chunking);
average task complexity: 8 (30 available concepts)

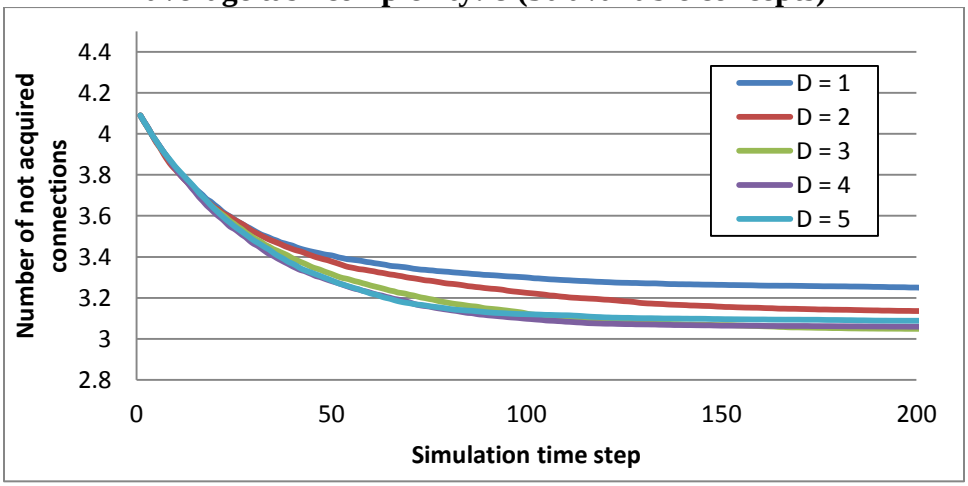

Fig. 5.2.2.6 Average number of not acquired connections:
WM = 5; D = 1-5 (no chunking);
average task complexity: 8 (30 available concepts)

Similarly to the reasoning on agent knowledge presented in section 5.2.1, agent effectiveness and efficiency also benefit from the chunking mechanism. This is because allocating one chunk on each working memory slot leads to more concepts being allocated in the agent WM during the learning process. This in turn leads to more connections having their weights updated and to more knowledge being transmitted by teachers and received by learners in each time step. As also mentioned in section 5.2.1, with the help of enlargeConfusion calls whenever a task connection hasn't been matched, agents manage to reach the agent effectiveness and efficiency presented by Figures 5.2.2.1, 5.2.2.3 and 5.2.2.5 in contrast to the non-chunking version presented in Figures 5.2.2.2, 5.2.2.4 and 5.2.2.6.

Therefore, similarly to agent knowledge and learning behavior, agent performance on solving tasks is also significantly enhanced by the chunking design.

5.2.3. Summary

In this section we showed that a C-ULM simulation using the chunking mechanism together with a task feedback leads to a faster acquisition of knowledge (number of agent knowledge connections and confusion interval lengths) and also to a better performance on solving tasks (number of not acquired agent connections, average weight differences and number of solved tasks).

5.3. Impact of various factors on the C-ULM with chunking system

The aim of this section is to study the impact of critical system parameters on the chunking design. We mainly focus on separately analyzing the impact that working memory capacity (WM), the spread normalization factor (D), the number of concepts, the number of agents and the number of tasks have on agent knowledge and agent

effectiveness and efficiency. Furthermore, in section 5.3.6 we investigate whether there is a correlation between the impact of varying the number of tasks and the impact of varying the number of concepts. Finally, in section 5.3.7 we study the impact of varying the amount of task information that agents have at the start of the simulation.

Below we present a table that lists the system parameters studied in this section together with the goal of each impact study.

Table 5.2 System parameters and their impact study goal

System parameter	**Impact study goal**
Working memory capacity (WM)	Better understanding of how WM capacity influences human learning
Spread normalization factor (D)	Better understanding how the spread activation phenomenon affects human learning
Number of concepts	Better understanding how the task complexity affects agent performance on solving tasks
Number of agents	Better understanding how the size of a group of agents can impact the performance on solving tasks
Number of tasks	Better understanding how the complexity of the agent environment (represented by agent tasks) influences task solving performance
Ratio ‘number of	Investigating whether there is specific

tasks / number of concepts'	relationship between the task complexity (number of concepts) and environment complexity (number of tasks).
Initial task information	Better understanding C-ULM's robustness when dealing with a small amount of initial task information

There are two main goals for the impact study of the factors presented in Table 5.1 shown above. The first goal is to better understand the cognitive aspects revealed by those factors. This is especially the case with the working memory capacity and the spread activation factor. These two are important aspects of human learning and by varying them in the simulation we could get more cognitive insight into the human learning processes. Furthermore, we can obtain simulation-related insights that can help in the development of new computational approaches to cognitive research.

The second goal is better understand the C-ULM's potential as a simulation for agent research. By better understanding what can be achieved with the simulation, we can also offer suggestions regarding its potential future use in multi-agent research or domain-specific problems that require a multi-agent solution.

5.3.1. Working memory capacity

In this section we study the impact that changing WM capacity has on agent knowledge and task solving performance.

As can be observed in Figure 5.2.2.1, increasing working memory capacity leads to the same asymptotic value for the number of solved tasks – around 9 tasks solved. This

means that the agent effectiveness remains unchanged as the working memory increases in the 3-9 WM range. However, the initial growth rate of solved tasks is lower as the working memory increases. This signifies that agent efficiency decreases as the working memory increases in the 3-9 WM range.

The reason for a lower initial growth rate as the working memory capacity increases is obtained by analyzing together Figures 5.2.2.3 and 5.3.1.1.

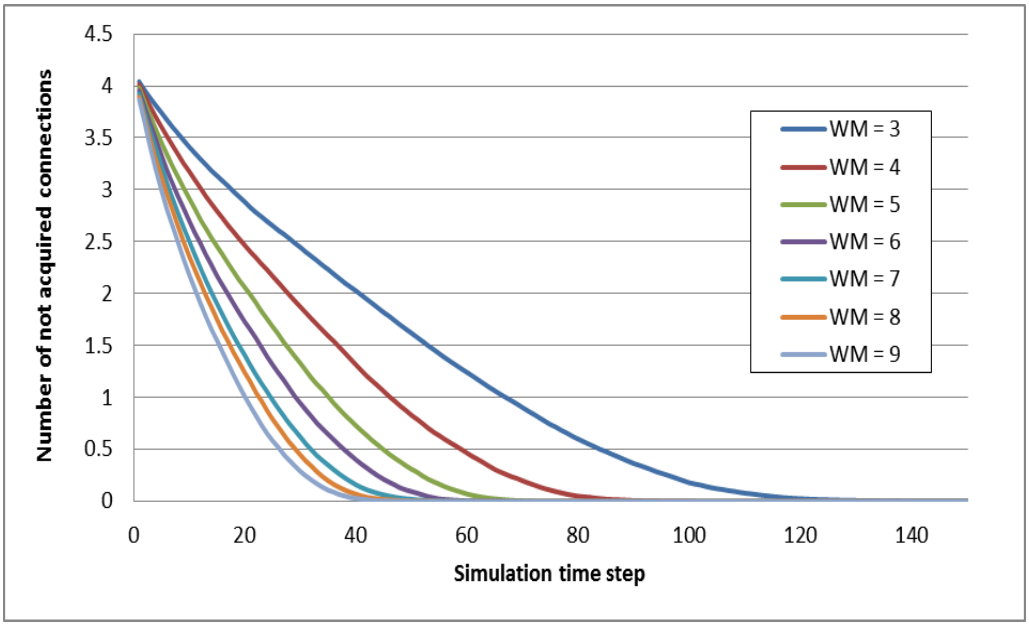

Fig. 5.3.1.1 Average number of not acquired connections:
WM = 3-9; D = 5 (chunking);
average task complexity: 8 (30 available concepts)

Figure 5.3.1.1 presents the average number of not acquired connections for the same system as the one presented in Figure 5.2.1.3. As can be seen in Figure 5.3.1.1, in the first 120 time steps all task connections are acquired by all agents (the average becomes 0 after at most 120 time steps). However, due to their limited capacities, the systems with WM =3 and WM = 4 have a slower decrease rate in the average number of not acquired connections (especially the system with WM = 3). Furthermore, as can be seen from Figure 5.3.1.1, the decrease rates are increasingly similar as the working memory

capacity increases. This indicates a level of redundancy as we increase the WM capacity in a system using chunking. However, due to their limited capacity, the systems with WM = 3 and WM = 4 are more selective systems with respect of what connections they receive in each time step. Thus, the WM = 3 and the WM = 4 systems focus on repetitive learning updates of fewer task connections in the first 150 – 200 time steps. This fact leads those systems to arrive to correct weights for the fewer connections they already received before learning of new connections. In contrast, the systems with high WM capacity (such as 8 or 9) have a fast decrease rate in the number of not acquired connections. This in turn leads those systems to perform repetitive learning updates on several connections but without arriving at approximately correct weights for some of them in the first place. This is why we observe more efficient behavior of the WM = 3 and WM = 4 systems in Figures 5.2.2.1 and 5.2.2.3.

The implication resulting from the initial system behavior (first 150 – 200 time steps) is that agents with a working memory capable of retaining only 3 or 4 chunks (of arbitrary size) are more focused on learning fewer task connection weights. This in turn leads to more efficient systems than systems with a higher WM capacity (Figure 5.2.2.1).

5.3.2. Spread factor D

In this section we study the impact that changing the spread factor D has on agent knowledge and task solving performance.

Figures 5.3.2.1-5.3.2.3 show that higher values for the D parameter (4 and 5) lead to a higher performance than values of 1-3. In Figure 5.3.2.1, the confusion interval length grows to a lower final value when D is 4 or 5. The same pattern can be observed for average weight differences (Figure 5.3.2.2) and also for the total number of solved tasks

(Figure 5.3.2.3). Those results are according to the expectation that a higher D value will lead to an increase in overall system performance.

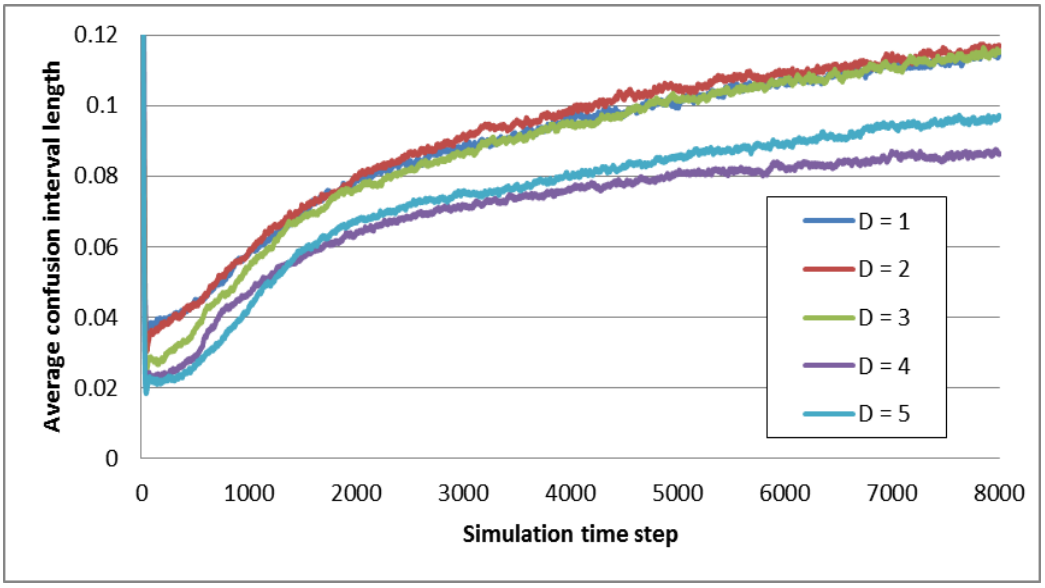

Fig. 5.3.2.1 Average confusion interval length: WM = 5; D = 1-5 (chunking); average task complexity: 8 (30 available concepts)

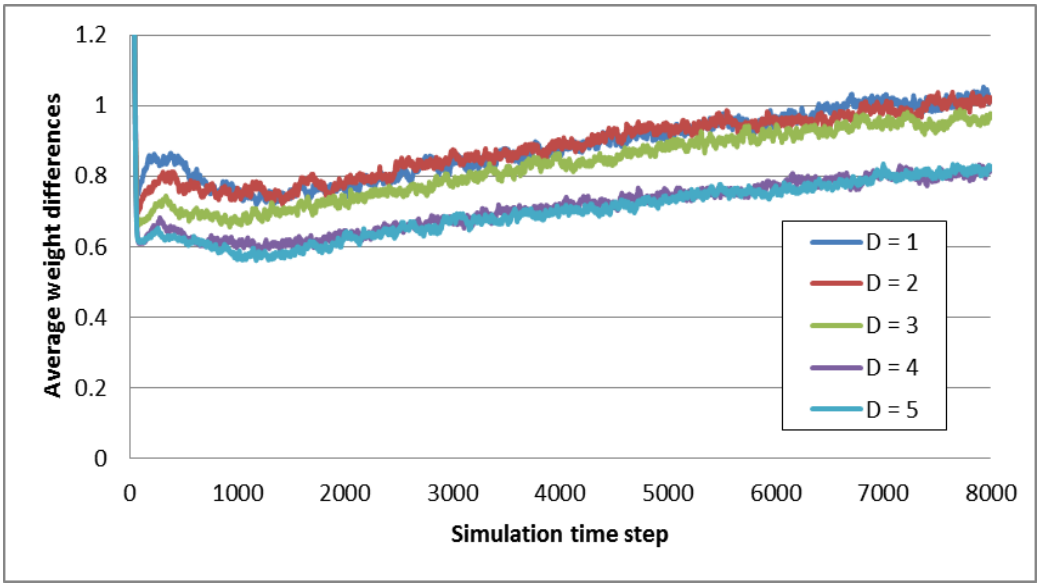

Fig. 5.3.2.2 Average weight differences: WM = 5; D = 1-5 (chunking); average task complexity: 8 (30 available concepts)

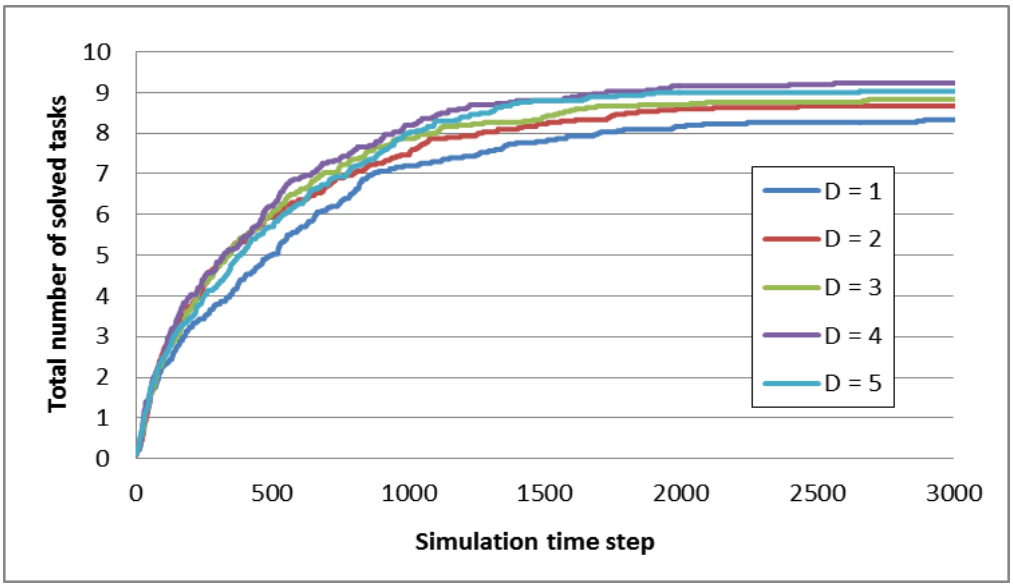

Fig. 5.3.2.3 Total number of solved tasks: WM = 3-9; D = 1-5 (chunking) average task complexity: 8 (30 available concepts)

The reason for having a D of 4 and 5 lead to a higher performance is given by the role of this parameter in updating the length of the confusion interval. Specifically, as the size of LTM chunks in the agent knowledge increases, the average distance from a given connection c to a connection c′ (connected by a path with connection c) also increases. According to Equation 3.8, this leads to larger chunks having more pairs of connections with a distance of 3 or 4 between. The confusion intervals of those connections are still updated in case D is high enough (such as 4 or 5). In contrast, if D is low, the distance between connections that are not close (distance at least 3) is capped to a lower D value which leads to a spread factor of 0 and no confusion update.

In other words, the connectivity of larger chunks is better exploited at higher D values and this leads to a faster learning process and a higher task performance.

5.3.3. Number of concepts

In this section we study the impact that changing the number of concepts has on agent knowledge and task solving performance. We present the results obtained with two systems, one with 30 and one with 100 available concepts. Of note, the average task complexity (and consequently the number of task connection weights that have to be matched) is 8 for the system with 30 concepts and is 26.1 for the system with 100 concepts.

Observation 1.

The two systems display the same pattern of growth in the number of agent connections (Figure 5.3.3.1). However, as expected, when the number of concepts is higher (100), both the initial and the final number of agent connections are higher (an increase from 150 to 450 connections for the system with 100 concepts as compared to an increase from 40 to 120 connections for the system with 30 concepts).

Observation 2.

The behavior of the average confusion interval length is rather similar in the two systems (Figure 5.3.3.2). In both systems, we have a relative brief period of little variability in the dynamic of the confusion interval length. After this period, this metric has an increasing trend in both systems. The difference between the two systems is given by the rate of increase in confusion interval length. Thus, in the 30-concept system, the confusion interval length increases faster than in the 100-concept system.

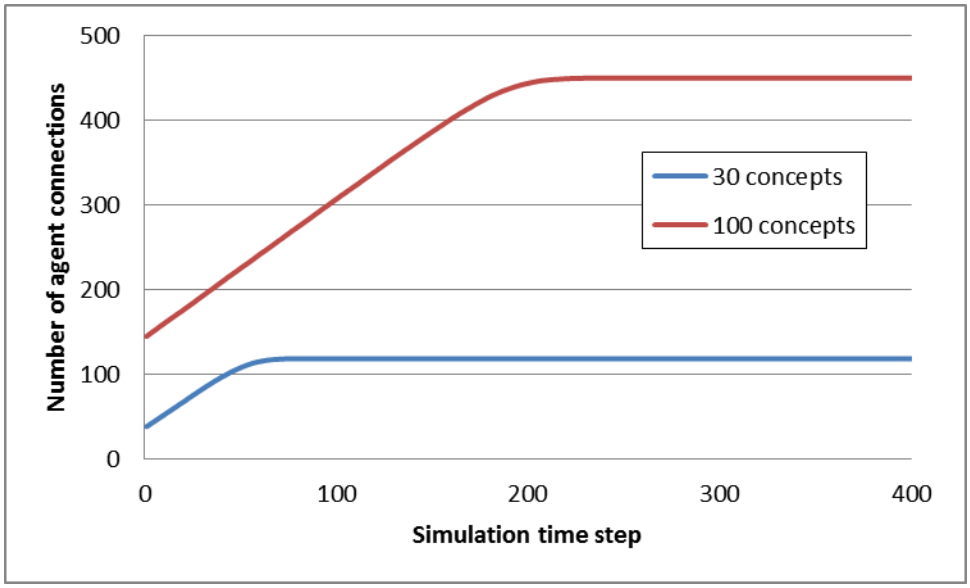

Fig. 5.3.3.1 Average number of agent connections: WM = 5; D = 5 (chunking)

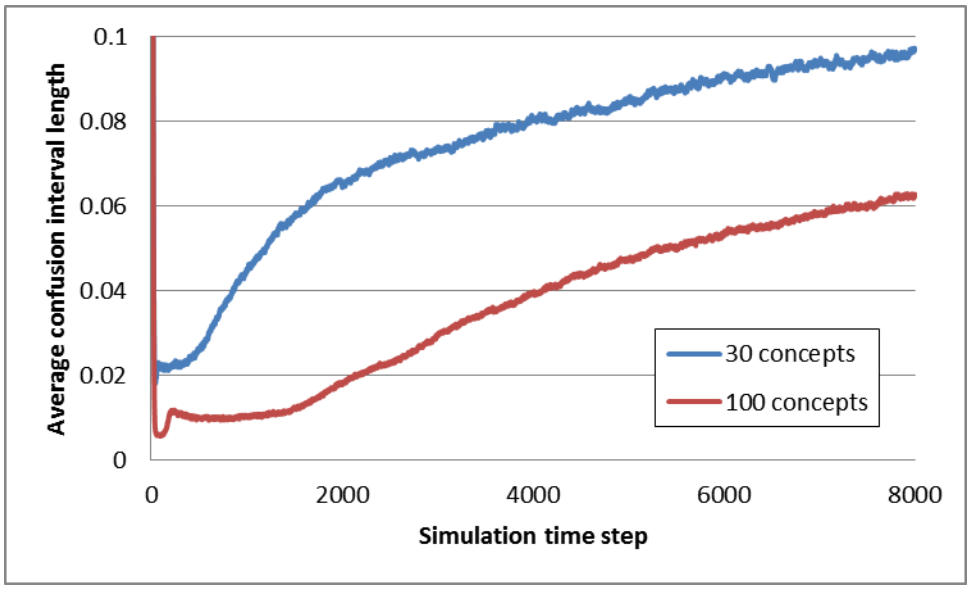

Fig. 5.3.3.2 Average confusion interval length: WM = 5; D = 5 (chunking)

Agents in a system with 100 concepts have more initial connections than the agents in a system with only 30 concepts. This is the reason why the average number of agent connections is much higher when the system has 100 concepts.

Observation 3.

The two systems display the same behavior of the average weight differences (Figure 5.3.3.3). As expected, when the number of concepts is higher (100) the average weight difference is higher than the one for a system with fewer concepts (30).

Observation 4.

As expected, the total number of solved tasks (Figure 5.3.3.4) is higher when the system has tasks with a lower complexity (the system with 30 concepts).

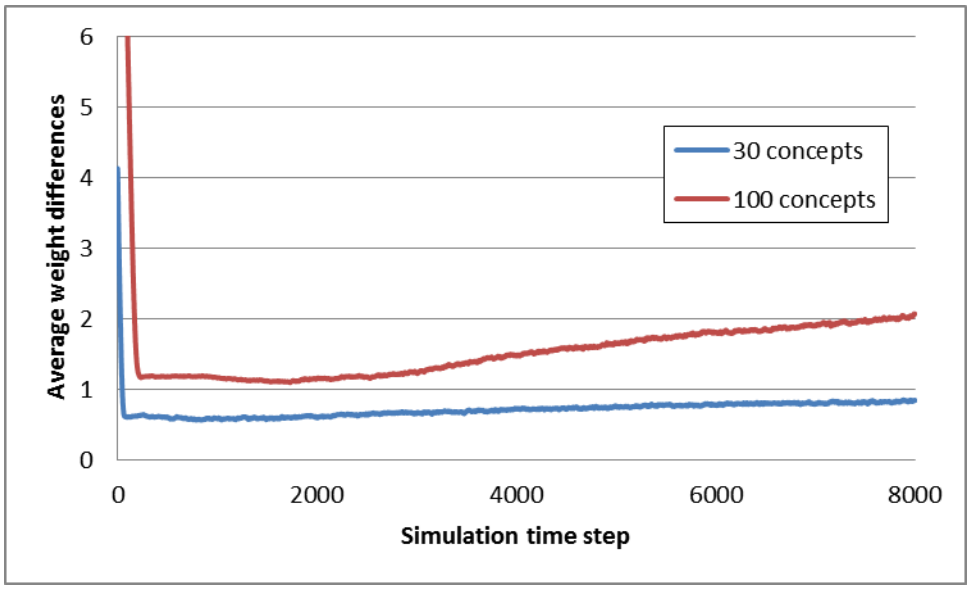

Fig. 5.3.3.3 Average weight differences: WM = 5; D = 5 (chunking)

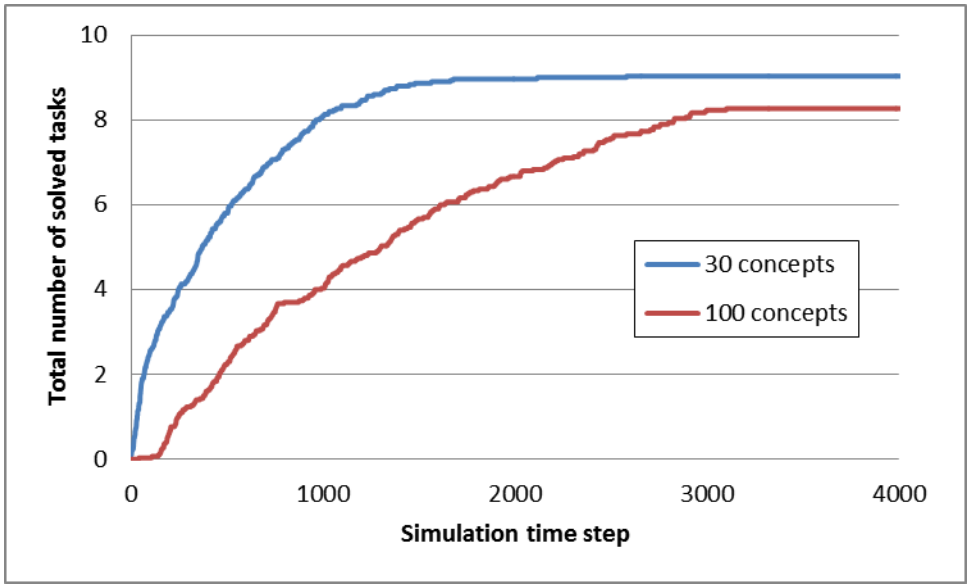

Fig. 5.3.3.4 Total number of solved tasks: WM = 5; D = 5 (chunking)

A system with 100 concepts presents more complex tasks—having more concepts and connections—than a system with 30 concepts. More task connections lead to more opportunities for agents to learn new connections. This in turn leads to more connections being acquired by the agents in the 100-concept system as compared to those in the 30-concept system. Furthermore, in the 100-concept system there are more connections that are being learned in the same time and thus more confusion intervals get shortened during the learning process. This leads to a slower increase in confusion interval length in the 100-concept system as compared to the 30-concept system (Figure 5.3.3.2). On the other hand, the number of task weights that have to be matched is higher for a system with 100 concepts as compared to one with only 30 concepts (Figure 5.3.3.3). This is also the reason for having a lower number of solved tasks in the system with 100 concepts (Figure 5.3.3.4). This is also in line with human observations since humans learn more when confronted with more complex tasks but the success rate of solving those tasks decreases.

The conclusion is that a system with more available concepts leads to more learning (confusion interval length is lower) and a higher knowledge acquisition in terms of task structure (agents acquiring more task connections) but a lower effectiveness at solving tasks.

5.3.4. Number of agents

In this section we study the impact that changing the number of agents has on agent knowledge and task solving performance. Below we present the results obtained with two systems, one with 10 and one with 20 agents. There are 100 available concepts in each system (the upper-bound task complexity is thus 100).

Of note, the average task complexity (and consequently the number of task connection weights that have to be matched) is 26.1. This is roughly 3.6 times higher than the average task complexity of 8 used in Sections 5.3.1, 5.3.2 and 5.3.3. An average task complexity of 8 is equivalent with a system having at most 30 concepts.

Observation 1.

The two systems display the same pattern of growth in the average number of agent connections (Figure 5.3.4.1). However, as expected, when the number of agents is higher (20), the final average number of agent connections is higher (around 440 for a system with 20 agents as compared to only 350 for a system with 10 agents).

Observation 2.

Figure 5.3.4.2 displays the same behavior for the confusion interval length when the number of agents varies. However, as expected, the growth of confusion interval length is lower when the number of agents is higher. Thus, the final confusion interval length

value for a system with 20 agents is around 0.05 as compared to 0.08 for a system with 10 agents.

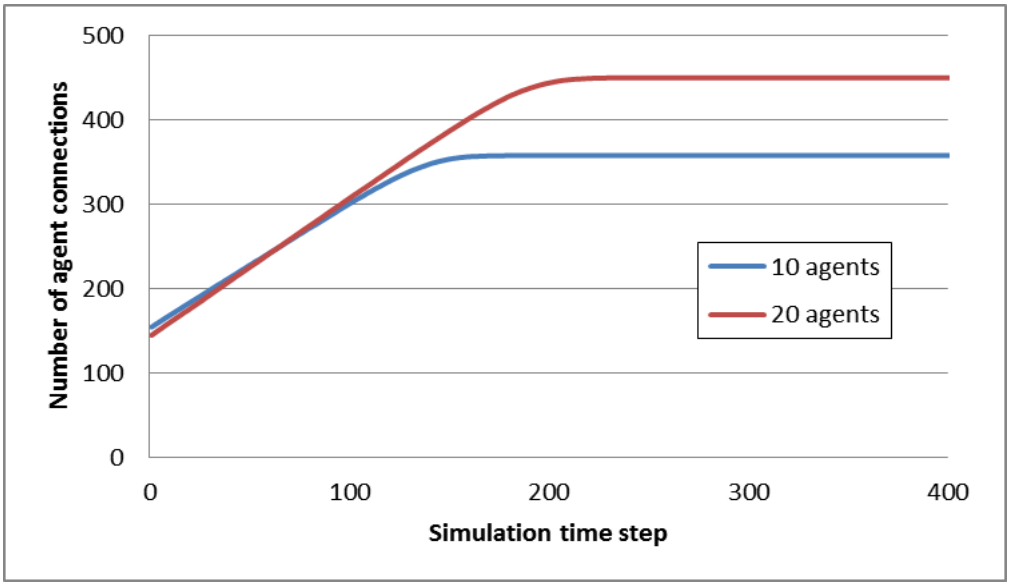

Fig. 5.3.4.1 Average number of agent connections: WM = 5; D = 5 (chunking); average task complexity: 26.1

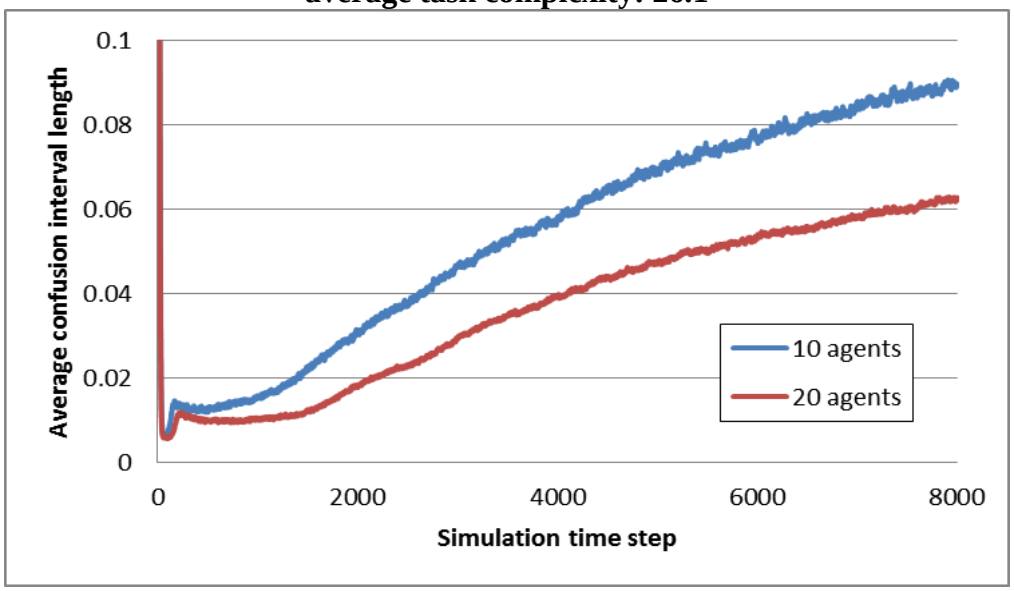

Fig. 5.3.4.2 Average confusion interval length: WM = 5; D = 5 (chunking); average task complexity: 26.1

The reason for acquiring more connections in the system with 20 agents as compared to the one with only 10 agents (Figure 5.3.4.1) is given by the fact that more agents have

connections in their initial agent knowledge. This leads to an increased probability that any connection will get shared among agents sometime during the simulation period. In turn, this increased probability of sharing any connection leads to a higher average of the number of agent connections in the 20-agents system as compared to the 10-agents system.

Observation 3.

Similar to the behavior observed for the confusion interval length, the average weight differences (Figure 5.3.4.3) end up lower for a system with 20 agents (2) as compared to one with just 10 agents (2.25).

Observation 4.

As expected, the total number of solved tasks (Figure 5.3.4.4) is higher when the number of agents is 20. Thus, the final number of solved tasks is 8.6 (out of 10 tasks) when the system has 20 agents and 8 when the system has only 10 agents.

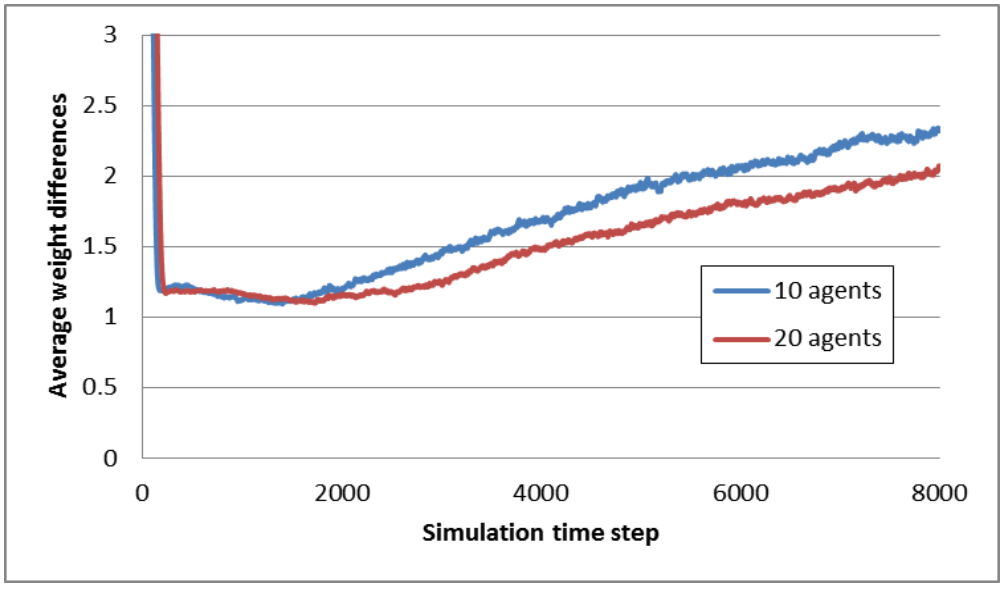

Fig. 5.3.4.3 Average weight differences: WM = 5; D = 5 (chunking); average task complexity: 26.1

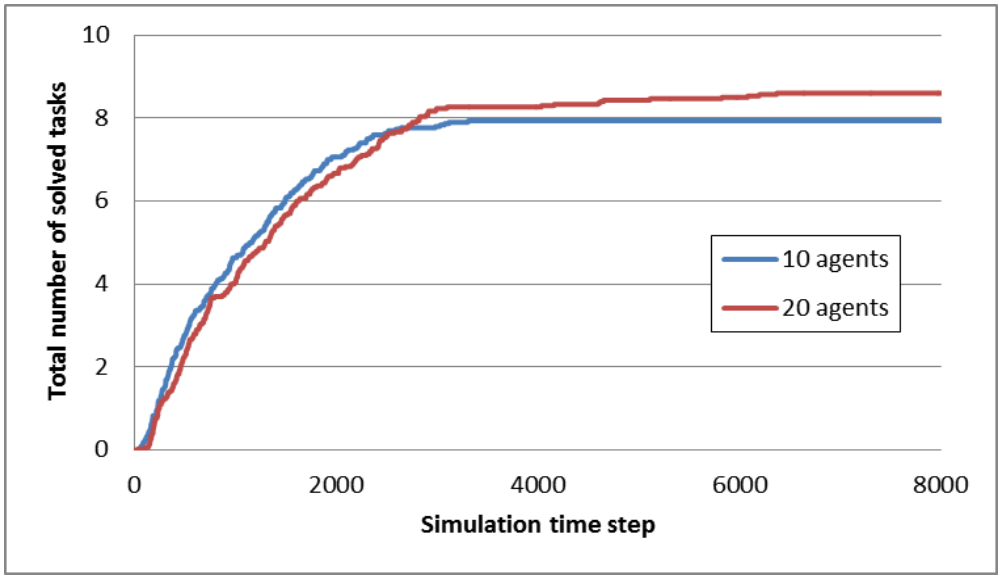

Fig. 5.3.4.4 Total number of solved tasks: WM = 5; D = 5 (chunking); average task complexity: 26.1

A system with more agents creates more opportunities for teaching and learning for each individual agent. As this communication protocol is enhanced by the number of options an agent has, confusion interval length and average weight differences increase slower toward the end of the simulation. This is also the reason why the number of solved tasks is higher when the number of agents is increased.

The conclusion is that a system with more agents is more robust in terms of knowledge acquisition and task effectiveness.

5.3.5. Number of tasks

In this section we study the impact that changing the number of tasks has on agent knowledge and task solving performance. Below we present the results obtained with three systems, one with 3, one with 30 and one with 300 tasks. Of note, the average task complexity (and consequently the number of task connection weights that have to be matched) is 8 and there are 30 available concepts for each of the three systems.

Observation 1.

As expected, agents acquire more connections when the system is exposed to more tasks (Figure 5.3.5.1). Thus, the simulation ends up with an average of 50 connections for the 3-task system, 200 connections for the 30-task system and 440 connections for the 300-task system.

Observation 2.

In the 30 and 300-task systems, the average confusion interval length (Figure 5.3.5.2) and average weight differences (Figure 5.3.5.3) decrease sharply at the start of the simulation. In contrast, in the 3-task system, those metrics present a constant trend around the value of 0.4 and 2.4 respectively. Furthermore, Figure 5.3.5.4 shows that in the scenario with 30 tasks, the total number of solved tasks is very high, leading to a performance close to 100% (29.6 out of 30 tasks solved out of 30 available tasks). In contrast, the scenario with only 3 tasks displays only a 20% performance (0.6 tasks solved out of 3 available tasks). The scenario with 300 tasks is not conclusive since the system is still solving tasks after 8000 time steps.

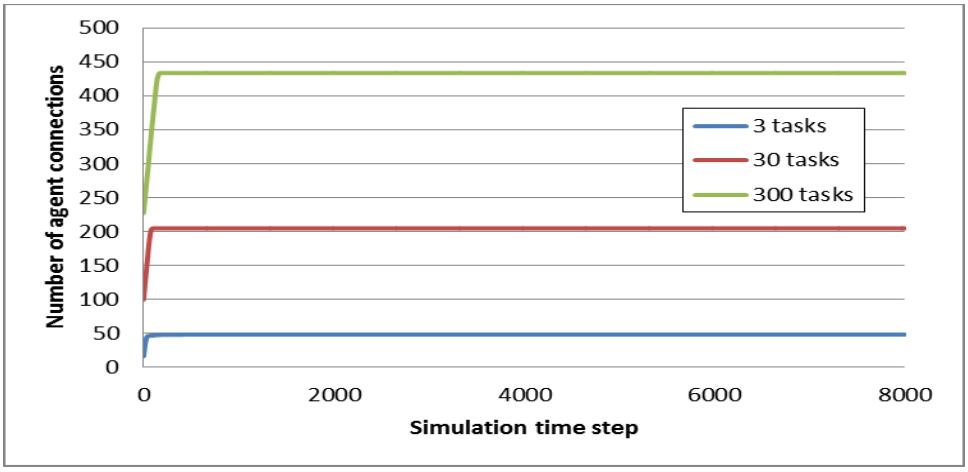

Fig. 5.3.5.1 Average number of agent connections: WM = 5; D = 5 (chunking); average task complexity: 8 (30 available concepts)

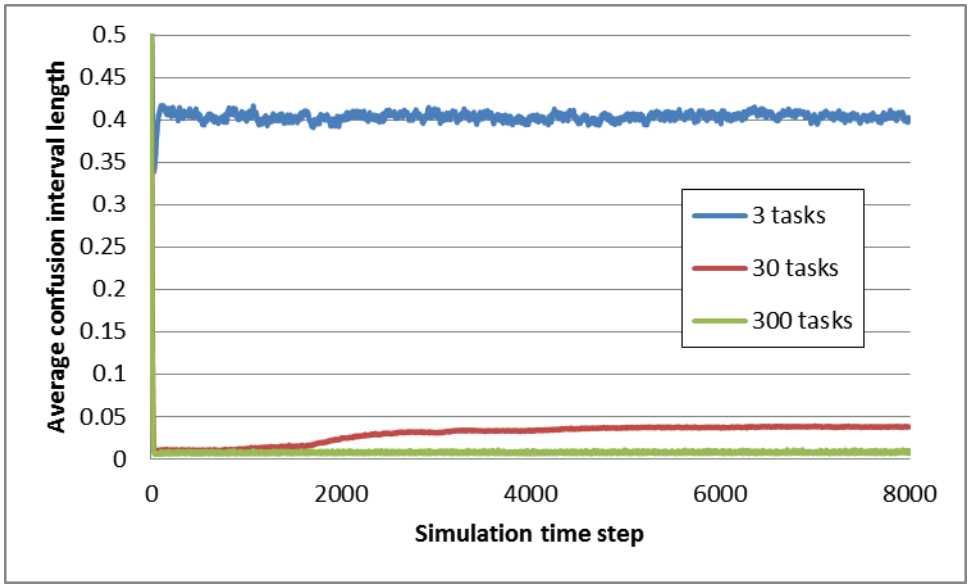

Fig. 5.3.5.2 Average confusion interval length: WM = 5; D = 5 (chunking); average task complexity: 8 (30 available concepts)

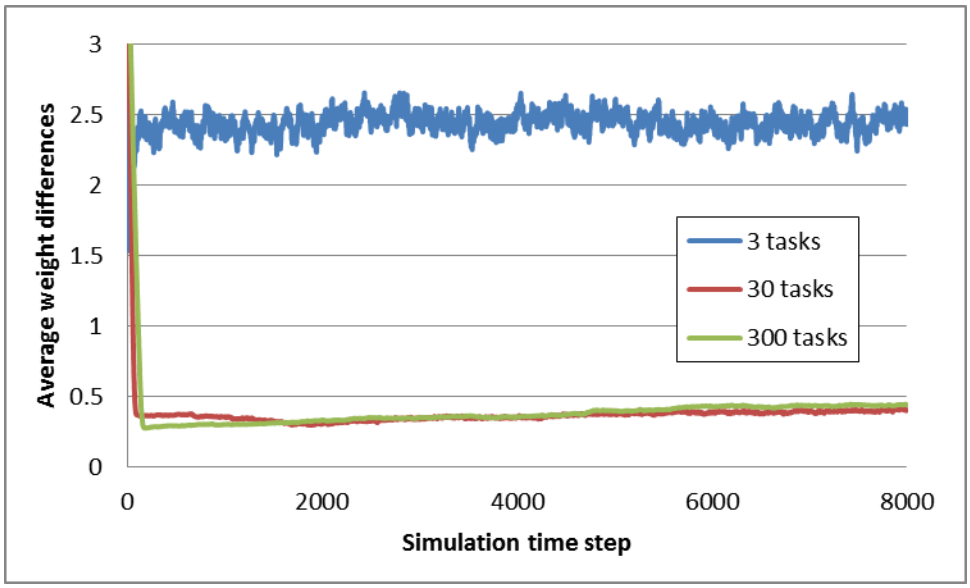

Fig. 5.3.5.3 Average weight differences: WM = 5; D = 5 (chunking) average task complexity: 8

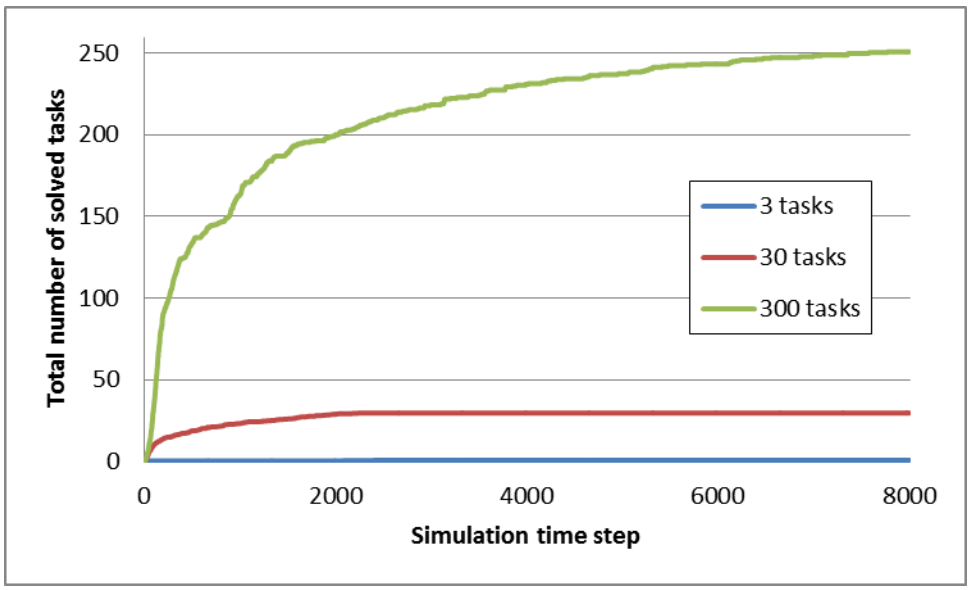

Fig. 5.3.5.4 Total number of solved tasks: WM = 5; D = 5 (chunking) average task complexity: 8

The most plausible explanation for the sharp performance contrast between the first scenario (3 tasks) and the other two scenarios (30 tasks and 300 tasks) is related to how much tasks overlap with each other. Task overlapping refers to two or more tasks that require some of the same connections and also the weights for those connections. Due to the initial task setup, if two or more tasks require a same connection, then they also require the same weight for that connection. In a system with 3 tasks, we have too few tasks in an environment with 30 concepts and the chance of those tasks overlapping is very small. In contrast, when we have 30 tasks in the same 30-concept environment, the chance for task overlapping is higher. When this happens, an agent that solved one task is in a good situation to solve in the near future a task that highly overlaps the solved task. Furthermore, by teaching the knowledge regarding the solved task to other agents, it helps other agents solve tasks that overlap with the solved task but were not yet solved by any agent.

This explanation might also be in line with the results obtained in section 5.3.3 when the environment contained 100 concepts. In that case, we had more tasks (10 as compared to 3) and their complexity was higher (26 as compared to 8). Due to this difference, we believe that the 100-concepts system presented in section 5.3.3 led to a much higher task overlap than the 3-task system presented in this section. Consequently, the 100-concept system presented in 5.3.3 achieved a much better performance than the 3-task system presented in this section.

The conclusion is that the degree of task overlapping plays a crucial role in the system performance. As the degree of task overlapping gets higher, the agents obtain more accurate task knowledge and they can reuse solved task information in order to solve new tasks that overlap the already solved ones. Thus, the agents are more knowledgeable about the tasks and consequently they are also more effective and efficient at solving them when the degree of task overlapping is high.

5.3.6. Ratio 'number of tasks / number of concepts'

In previous sections we investigated the impact of the number of concepts (in Section 5.3.3) and separately we investigated the impact of the number of tasks (in Section 5.3.5). The rationale for this experiment is investigating whether there is a specific correlation between those two impacts. Thus, we vary both the number of concepts and the number of tasks but we keep the same 'number of tasks / number of concepts' ratio.

For the purpose of this section, we note C = number of concepts and T = number of tasks. We present the results obtained when the ratio $\frac{T}{C}$ is equal to 1 for two different systems: one with 30 available concepts and 30 tasks and one with 50 available concepts and 50 tasks.

Observation 1.

As shown in the previous sections and reconfirmed in Figure 5.3.6.1, agents acquire more connections when the system is exposed to more concepts (Section 5.3.3) and to more tasks (Section 5.3.5).

Observation 2.

Figure 5.3.6.2 displays an S-shaped behavior for the confusion interval length in both systems. This behavior has been seen before in Figure 5.3.5.2 where we have the same 30/30 system (30 available concepts and 30 available tasks). However, the confusion interval length starts to increase at a later time (after 4000 time steps) in the system with the ratio 50/50 as compared to the one with a ratio of 30/30 (after 1000 – 1500 time steps).

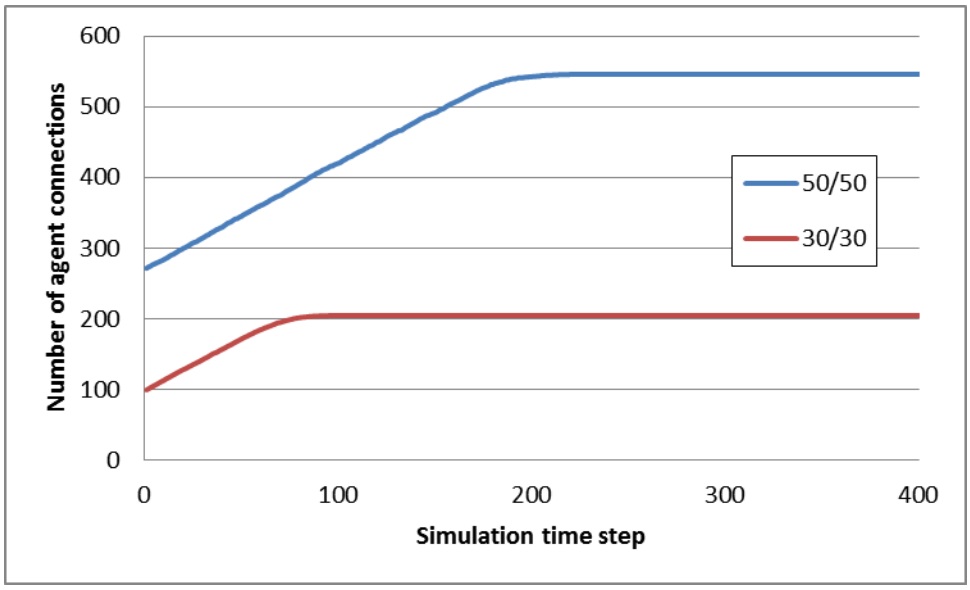

Fig. 5.3.6.1 Average number of agent connections: WM = 5; D = 5 (chunking)

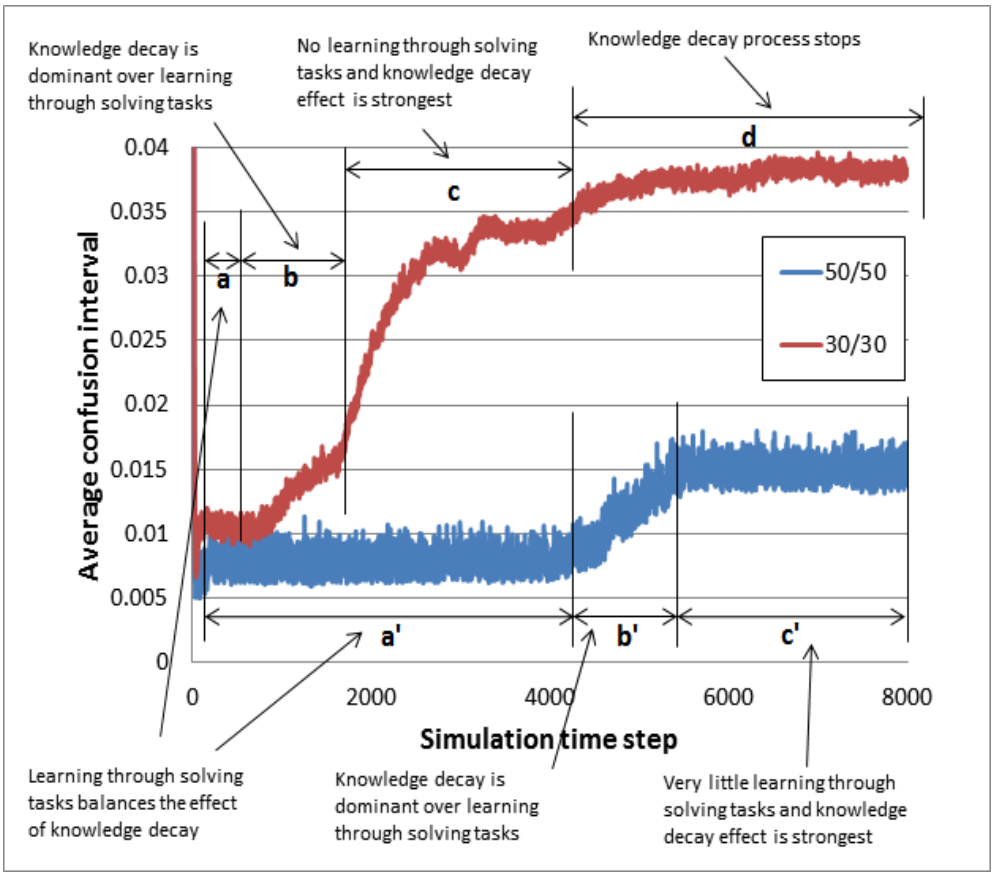

Fig. 5.3.6.2 Average confusion interval length: WM = 5; D = 5 (chunking)

The 30/30 system presents 4 stages in the behavior of confusion interval length. In the first stage (stage a) there is a sufficient number of tasks that are being solved so that the effect of learning from solved tasks balances the effect of knowledge decay. This balance leads to a constant confusion interval trend that is shown in stage a. In stage b, agents solve fewer tasks and the knowledge decay effect dominates the effect of learning through solving tasks. In stage c, there are no more tasks to be solved so that the effect of knowledge decay becomes even stronger leading to a sharper increase in confusion

interval. Finally, knowledge decay is duration-limited in the current C-ULM design and thus the knowledge decay effect eventually stops. This fact can be observed in the constant confusion interval trend of stage d.

On first glance, it seems that we don't have the a, b, c and d segments of confusion interval in the 50/50 system. However, we argue that the a' segment in the 50/50 system is indeed corresponding to the a segment in the 30/30 system. The 50/50 system has more tasks available to solve and it takes more time steps for the knowledge decay process to dominate over the learning through solving tasks process. In addition, b' corresponds to b but it is delayed with approximately 3000 time steps (starting at around 4000 time steps instead of 1000 time steps as it happens for b). Another important difference is that c' displays a much slower increase in confusion interval than stage c. This happens because in stage c' there are still a few very complex tasks to be solved (as compared to stage c). Once they are solved they lead to a very slow increase in confusion interval length. One consequence of this extremely slow increase is that in Figure 5.3.6.2 we can see only the beginning of stage c'. If we had allowed the experiment to continue, we would have observed that c' resembles c but at a much slower pace thus spreading on many more time steps than c. Since knowledge decay is duration-limited in both systems, stage d' corresponding to stage d naturally follows c' after many more steps than shown in Figure 5.3.6.2.

Observation 3.

The two systems display the same behavior for the average weight differences (Figure 5.3.6.3). As expected and also indicated in section 5.3.3, when the number of concepts is

higher (50), the average weight difference is higher than the one for a system with fewer concepts (30).

Observation 4.

The number of solved tasks is higher in the system with 50 tasks as compared to the system having only 30 tasks (Figure 5.3.6.4). This is expected and reconfirms section 5.3.5. However, the performance is lower in the 50/50 system as compared to the 30/30 system. This is expected and reconfirms what we observed in Figure 5.3.3.4 (section 5.3.3), i.e. a higher number of concepts leads to a lower task effectiveness.

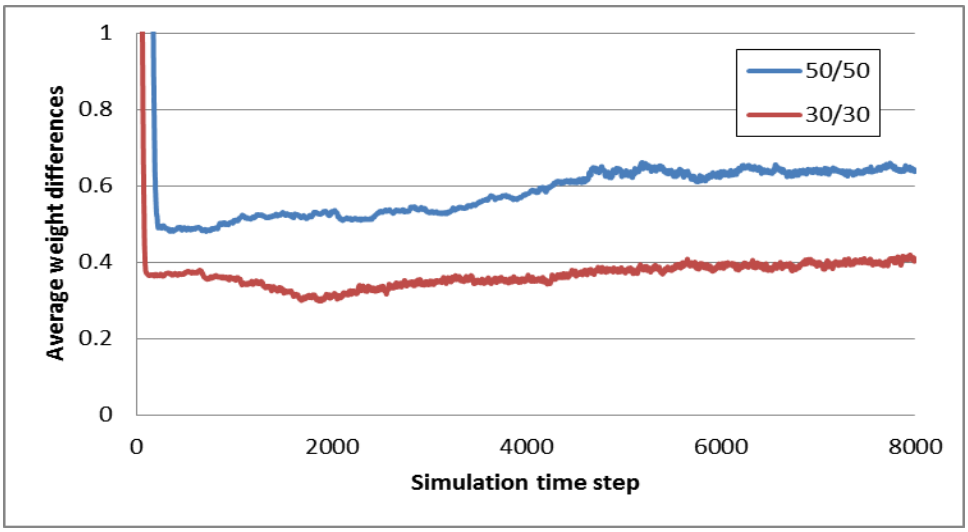

Fig. 5.3.6.3 Average weight differences: WM = 5; D = 5 (chunking)

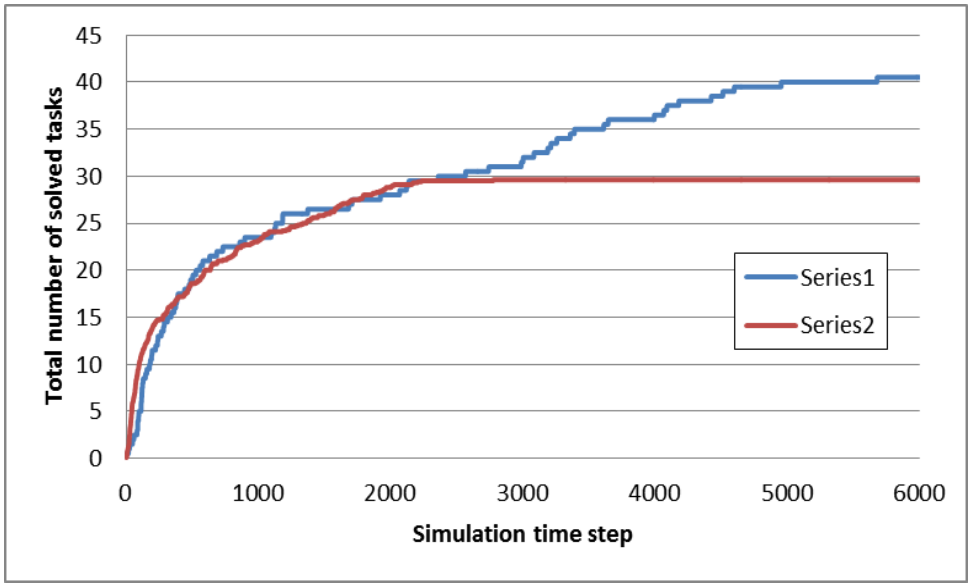

Fig. 5.3.6.4 Total number of solved tasks: WM = 5; D = 5 (chunking)

In conclusion, two systems with the same T/C ratio, one with 30/30 and one with 50/50 reconfirm the general patterns observed when varying only the number of concepts (section 5.3.3) and when varying only the number of tasks (section 5.3.5). That is, the ratio value does not directly play a role in the system performance. The performance hinges upon the number of tasks and the number of concepts and there is no clear correlation between the impacts of varying both parameters.

5.3.7. Initial task information

In all designs presented in previous sections, we ensure that at least one agent 'has some clue' about each task connection weight. This means that for each task connection weight, there is at least one agent that has an initial weight for the corresponding connection set to a random value picked from the interval $(w - 0.05, w + 0.05)$ where w is the actual task connection weight (Section 3.6). This set up can be compared to the idea of human collaboration for a complex task – at least one individual in a team has some information about a given task but they have to cooperate in order to solve it.

The impact of this initial task information is clearly critical for system performance. However, the purpose of this section is to see if, overall, a certain system would be robust enough to solve tasks mentioned in the previous section (tasks with an upper bound complexity of 30 concepts) but without given task weight information to the agents at the start of the simulation. In this design, we still make sure that all task connections still exist in at least one agent (task connections scattered among agents) but the initial agent weight for each of those connections is set to a random value in the interval $(0.5, 0.95)$. Therefore, in this design the agents have NO initial information about any task weights.

In order to achieve the described design we change line 3 in method insertTaskInformation (Section 3.6). Thus, instead of initializing the agent weight with a weight that is rather close to the task-required weight (within a margin of 0.05) we pick up a random weight value from the interval (0.5, 0.95). Thus, method insertTaskInformation becomes:

Algorithm insertTaskInformation

Input: AL – agent list

e – edge to be added to agent graph knowledge or only changed in terms of confusion interval and weight; it connects two task connected concepts

Returns void

1. **Loop** through all agents a_i in AL
2. $e.\mathrm{CI} = \mathrm{random}\ (0.1, 1)$
3. $\mathbf{e.weight = random}\ (\mathbf{0.5, 0.95})$ *// the only changed line for the purpose*
4. *// of this section*
5. **If** $(e \notin a_i.G_A)$ **then**

6. add e to $a_i.G_A$

7. **End If**

8. **End Loop**

End Algorithm

Below we present the results obtained with a system having 60 agents, WM = 5, D = 5, upper-bound task complexity of 30 concepts and a simulation duration of 16000 time steps.

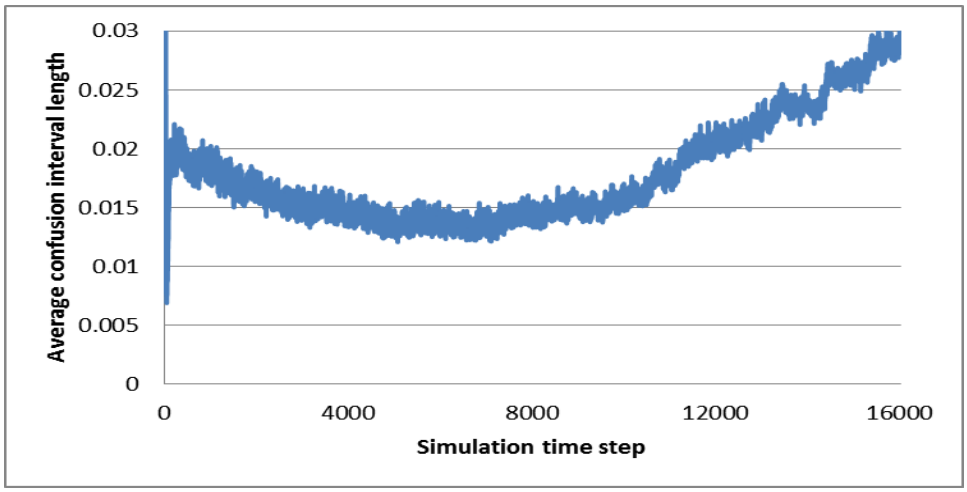

Fig. 5.3.7.1 Average confusion interval length: WM = 5; D = 5 (chunking)

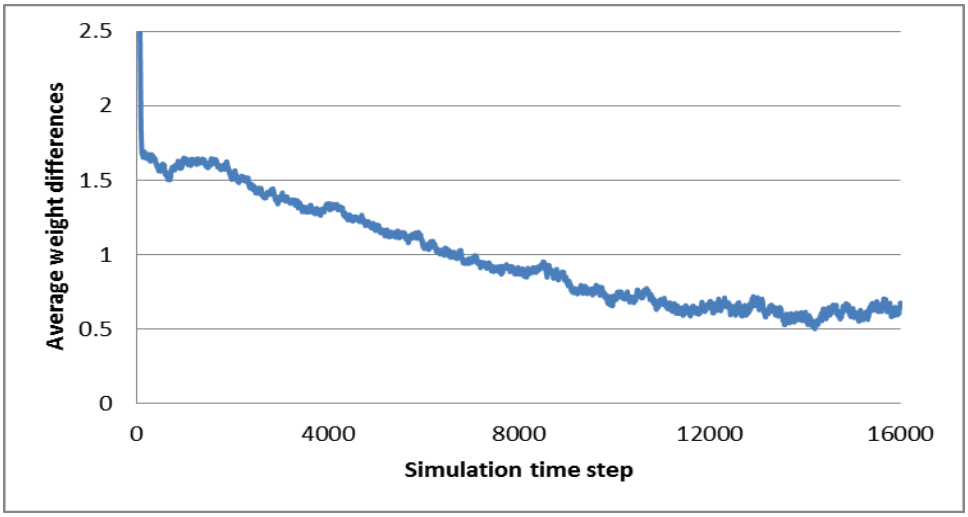

Fig. 5.3.7.2 Average weight differences: WM = 5; D = 5 (chunking)

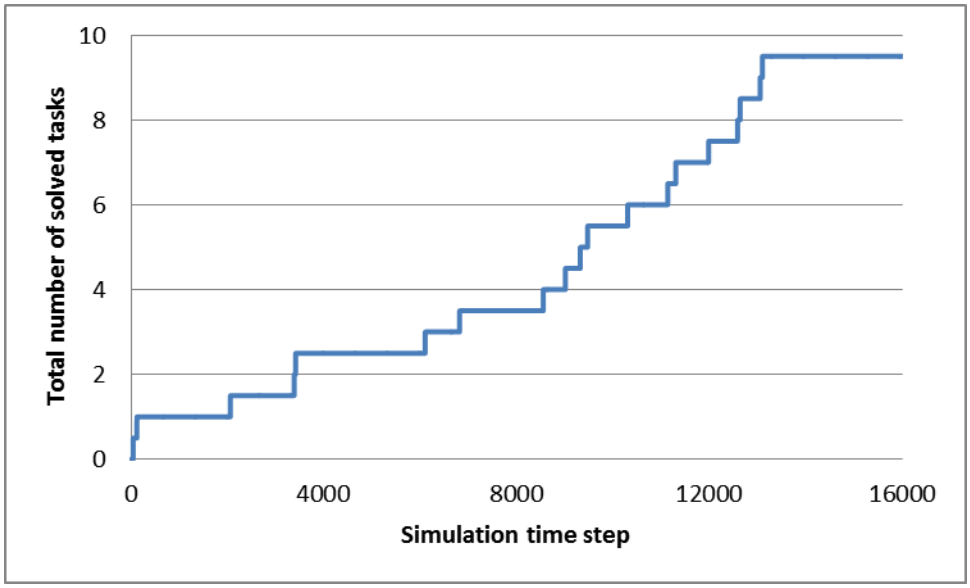

Fig. 5.3.7.3 Total number of solved tasks: WM = 5; D = 5 (chunking)

From Figures 5.3.7.1-5.3.7.3 we observe the following:

- The confusion interval behavior resembles the one observed in Figure 5.2.1.3 although the difference between the maximum final value and the minimum value observed at around 6000 time steps is much smaller than in Figure 5.2.1.3 (a difference of less than 0.02). Furthermore, the rate of decrease towards the minimum value and the subsequent rate of increase are much lower in Figure 5.3.7.1 as compared to Figure 5.2.1.3.
- The average weight difference starts to slowly but steadily decrease after the initial drop and does so for the entire simulation duration (Figure 5.3.7.2).
- After approximately 13000 time steps, the number of solved tasks reaches 9.5 out of 10 available tasks (Figure 5.3.7.3).

In this design, the agents have to explore various values for the task weights for a longer time since they start with no initial task weight information. The fact that the

system eventually reaches 95% performance on solving tasks indicates that the interplay between the usage of chunking in learning and teaching and the critical role of calling the enlargeConfusion method as a task feedback following unsuccessful task attempts is powerful enough to move agent weights towards the correct task weights. As mentioned in Section 5.2.1, part of this success is due to calling the enlargeConfusion method only for unmatched connections whenever a task attempt is unsuccessful. Since the confusion interval (Figure 5.3.7.1) and average weight differences (Figure 5.3.7.2) resemble those from Sections 5.2.1 (Figure 5.2.1.3) and 5.2.2 respectively (Figure 5.2.2.3), the main simulation difference required in order to solve tasks without any initial task weight information is the simulation duration (16000 time steps instead of just 8000 time steps in section 5.2.1). Thus, many more learning repetitions and weight center updates are necessary in this system in order for the agents to arrive to the correct task weights. Therefore, system efficiency is much lower as compared for example with section 5.2.1. Nevertheless, due to chunking and proper enlargeConfusion method calls, the system achieves a comparable effectiveness at solving tasks.

The results in this section seem to indicate that although initial task information plays a critical role in system performance, by increasing the simulation duration we can obtain a system robust enough to solve tasks without ANY initial weight information and at a comparable effectiveness with previous designs.

The implication here is significant: we have a system that is rather very usable. Without weight information, one would still be able to use the simulation to learn the weights and obtain a good performance on solving tasks. In order to confirm this, we understand that additional experiments with no initial weight information need to be

performed. This would include experiments where we vary the task complexity, the number of agents, the number of tasks and working memory capacity in order to better understand how those factors impact a system that does not have any initial weight information.

5.3.8. Summary

In this section we investigated the impact of several factors in the C-ULM simulation using the chunking mechanism. We have found out that a lower working memory can lead to a more efficient processing but the same overall effectiveness and that a higher spread factor leads to a faster learning and higher performance on solving tasks. We also concluded that a larger group of agents increases system effectiveness and that the system is robust enough to deal with a small amount of initial task information given to the agents. Furthermore, as shown in sections 5.3.3 and 5.3.5, an increased task complexity and a low degree of task overlapping lead to a slower learning rate and a lower agent performance on solving tasks.

5.4. Implications

An important implication resulting from Section 5.2 is the fact that a system that uses chunking is learning faster and it is also solving tasks more efficiently than a system without chunking. Furthermore, in Section 5.3, we found out that a higher spread factor also leads to faster learning and higher performance at solving tasks. According to the ULM model, those implications were expected and they show that the C-ULM model is accurate in the cognitive modeling of learning mechanisms such as chunking and spread activation factor.

The implication that the number of agents, task complexity, and the degree of task overlapping influence agent task solving performance was also expected. This shows that we can use the simulation to better understand how the multi-agent system behaves when parameter values are varied. Understanding how the system is changed by varying system parameters makes C-ULM an useful tool for performing "what-if" analysis.

Meanwhile, an interesting implication that was not expected is the fact that a lower working memory leads to a more efficient task solving process. This implication shows that the simulation can lead to a better understanding of the human learning mechanisms especially in the cases of long-term learning and problem solving where data from human subjects is generally not available.

Another unexpected yet significant implication is the fact that the system can learn task weights and solve tasks even if it does not have initial task weight information. This makes the C-ULM simulation usable to a large array of problems that can be represented as weighted graphs and where there is no weight information available to bootstrap the system to get it started.

5.5. Contributions to MAS research

Our contributions to the AAMAS community are at two levels. One level is the modeling of individual agent reasoning inspired by the functions and relationships between the three ULM components of knowledge, motivation and WM; and another level is the modeling of multi-agent interactions and knowledge transfer based on the principles of human teaching and learning processes.

At the agent reasoning level, most AAMAS efforts regarding modeling of human learning have been aimed to improve the performance of multiagent systems—i.e.,

whether agents utilizing a particular human-based learning model improves, for example, their utilities. The attractiveness of using a human-based learning model hinges upon the intuitive abstraction of human-to-human knowledge transfer behaviors in complex situations. Our multiagent simulation makes a step forward in the use of a human-based learning model by capturing the underlying learning mechanisms that tie knowledge, motivation, and WM together. This adaptation has several key benefits. First, because of ULM being a general cognitive model, incorporating the processes of knowledge, motivation, and WM into agent reasoning would allow AAMAS researchers to investigate multiagent systems that involve human learning, either with human agents interacting with each other or artificial agents working in tandem with their human counterparts in a hybrid environment.

Second, C-ULM can serve as a more general learning framework for agent reasoning, especially in situations where domain knowledge is not available or not sufficient to optimize or customize the learning processes. For example, in the RL approach, obtaining the utility of each state-action pair depends on the rewards and the gradients (both direction and amount of update in the utility), which requires domain knowledge to ensure learning convergence and efficiency. In a way, a system designer would have to estimate at the distance (or a function of the distance) between a state and the desired goal states, and guide the exploration and exploitation in the RL process to reach the goals. However, C-ULM also treats learning in a more intrinsic manner. An agent strives to get better by making its knowledge better, and the quality of its knowledge is the collective group of edge weights and the confusion intervals on the edge weights. This means, a MAS designer could get away without knowing much about the distance

between a state and the goal states, thereby, enabling one to bootstrap agent reasoning more efficiently and effectively.

Third, the C-ULM provides a practical, explicit framework for acquiring and retaining knowledge. Motivation guides how concepts or knowledge chunks are moved into the WM and it is integral to attention. Attention, important in acquiring and retaining knowledge, is a measure that could help MAS designers better model real-time agent focus and learning in dynamic environments. Further, grounding learning as a process to reduce confusion intervals of edge weights is akin to learning motivated by one's self-efficacy, i.e., one's confidence in one's knowledge and expertise. This allows agents to have intrinsic motivation for acquiring knowledge. Meanwhile, C-ULM agents are also extrinsically motivated by the rewards that can be obtained by solving tasks. Thus, our model integrates both intrinsic and extrinsic motivation, making it a more flexible solution framework for solving MAS-related problems.

At the multiagent learning and teaching level, C-ULM offers a solution on WM-level knowledge transfer between a teacher and a learner, allowing MAS designers to better design how agents decide on which knowledge to transfer, how to transfer, and the effectiveness of transfer. These decisions are neither arbitrary nor domain-driven, as indeed guided by the Unified Learning Model. Though our current work is not complete and do not yet comprehensively capture the ULM, we believe that it has the potential of offering at least an alternative to model and deliver knowledge transfer between agents.

Finally, the implications summarized in Section 5.4 can serve as a guideline on how to use the C-ULM simulation to address multi-agent research questions. Examples of such questions emerge directly from our experiments: What is the optimal number of agents

given a certain number of problems of a given complexity? What happens to the agent learning behavior if the value of the working memory capacity is increased from 3 to 7? How is the growth rate of long-term memory chunks affected by the value of the spread activation factor?

Chapter 6. CONCLUSIONS AND FUTURE WORK

6.1. Summary

In this work we have introduced C-ULM, a novel multiagent based cognitive architecture in order to address research problems in human cognition and in multi-agent systems. C-ULM follows the general learning principles outlined by the Unified Learning Model (ULM) and it integrates ULM's core components of long-term memory, motivation and working memory.

6.1.1. Related work summary

Within the field of cognitive architectures, the reference system is SOAR (Lehman et al. 2006) which is a production based system where knowledge is mainly described as sets of rules. By implementing C-ULM as a connectionist based model, we target the issue of limited expressivity found in rule based systems.

From the multiagent point of view, the reference model is the Belief-Desire-Intention architecture or BDI (Rao and Georgeff 1995) which uses beliefs as the agent knowledge representation, desires as the agent objectives and intentions as the agent possible actions. By the use of working memory and motivation, C-ULM embeds a cognitively-oriented filter that can be used to appropriately select a subset of beliefs, desires and actions of computationally reasonable size.

As compared to recent works in multiagent systems that incorporate cognitive components or principles (Sklar and Davies 2005; Spoelstra and Sklar 2008; Merrick 2011), C-ULM stands out by offering a more complex motivation model having both an intrinsic component given by the agent long-term knowledge and an extrinsic component given by the risks involved in attempting complex tasks.

From a cognitive informatics perspective, C-ULM implements in a practical model many of the abstraction layers outlined in a theoretic cognitive model called the Layered Reference Model of the Brain or LRMB (Wang & Chiew, 2010). In this model, the brain has seven abstraction layers of processes with primitive processes operating at the sub-conscious level and higher cognitive functions such as learning, problem solving and decision making operating at the conscious level and relying on the mechanisms of previous levels.

6.1.2. Framework summary

6.1.2.1. Single-agent model

C-ULM's single agent model incorporates the three main ULM components of long-term memory, motivation and working memory.

Long-term memory is represented as a weighted graph where each agent connection is represented by two elements: the connection weight and a confusion interval centered around the connection weight value.

Motivation is modeled by assigning motivation scores for each available concept in agent knowledge. Each score has two main parts. The first part is the intrinsic one and it influences the final motivation score proportionally to the strength of knowledge the agent has about the given concept. The second part is the extrinsic one and it influences

the final motivation score proportionally to the complexity of tasks that require the given concept.

Working memory is modeled as a subset of concepts found in agent knowledge. The size of this subset is given by the working memory capacity and the selection of knowledge concepts that get selected to enter working memory is given by comparing concept motivation scores to the awareness threshold (or AT).

The cognitive process of knowledge decay is also modeled within C-ULM. This process is modeled by lengthening the confusion intervals of the least used concepts in agent's knowledge.

6.1.2.2. Multi-agent framework

The multiagent communication and knowledge sharing protocol is performed by the actions of learning and teaching. Each of those agent actions is modeled in two major phases: the working memory allocation and the working memory processing (learnProcess and teachProcess algorithms). Each of the algorithms for allocation have two versions: one that allocates one concept in each working memory slot (learnAllocate_basic) and one that uses the chunking mechanism and fills each working memory slot with a group of interconnected concepts called a chunk (learnAllocate_chunking) instead of filling it with only one concept.

The communication flow starts with the working memory allocation algorithm for the teacher agent. This algorithm outputs the set of concepts that will be taught. This set of concepts is the input for the teacher agent's working memory processing algorithm. In turn, this algorithm produces the knowledge to be taught and also updates the confusion intervals for the teacher agent.

In the third stage, the WM allocation for the learner agent takes as input the learner agent's knowledge and the knowledge to be taught transmitted by the teacher. It then outputs a subset of this knowledge based on the learner's working memory. This knowledge subset is then processed by the WM processing module for a learner agent. This module further adjusts the weight centers and confusion intervals in the learner agent's knowledge.

6.1.2.3. Tasks

Similar to agent knowledge, tasks are also represented as weighted graphs but the task connections do not have an associated confusion interval. In order to solve a task, an agent has to have in its knowledge all the task connections and match all the task weights within a certain error margin. Task feedback is incorporated by shortening confusion intervals when an agent solves a task (positive feedback that leads to stronger knowledge) and by lengthening those intervals when an agent fails to solve a task (negative feedback that leads to weaker knowledge).

6.1.3. Implementation summary

The C-ULM simulation is written in the Java language and it uses Repast (North et al, 2006) as the framework for agent modeling. Similar to other Repast simulations, C-ULM divides each simulation run into time steps or "ticks". Each simulation run is defined by a set of parameters that are specified in a syntax specific parameter file. Those parameters include the number of agents, tasks and concepts, the agent WM capacity and the number of simulation time steps.

The simulation outputs are written in csv files that contain the values for specific C-ULM performance metrics such as the number of agent connections, the confusion interval length and the number of solved tasks.

The main simulation class is called ULMSimulationModel and it extends the Repast SimpleModel class. Agents, concepts, knowledge, motivation, working memory, concepts and tasks have associated abstract classes with the main features and method signatures and non-abstract classes extend the abstract ones for the actual implementation.

6.1.4. Results summary

The results obtained from various simulation runs indicate that a C-ULM system that uses the chunking mechanism leads to faster knowledge acquisition and also to a much improved agent effectiveness and efficiency on solving tasks.

Results further indicate that the agent knowledge acquisition rate and the multiagent system performance are significantly influenced by varying the values for various system parameters. Thus, we have found out that a lower working memory can lead to a more efficient processing but the same overall effectiveness in solving tasks. In addition, a higher level of spread activation also leads to a faster learning and higher performance on solving tasks. Furthermore, as shown in sections 5.3.3 and 5.3.5, an increased task complexity and a low degree of task overlapping lead to a slower learning rate and a lower agent performance on solving tasks. Finally, section 5.3.7 indicates that the system is quite usable and is robust enough to deal with problems where there is no weight information available to bootstrap the initial system configuration.

6.2. Future work

6.2.1. Cognitive research

As future work in the direction of cognitive and human learning research, we are interested in expanding and refining the C-ULM by experimenting with a larger parameter space. Of particular importance is investigating learning behavior and task solving performance when the task complexity is significantly increased (for example systems with 100, 200 or more available concepts).

Since the system is modeling a group of cognitive agents and not just one cognitive agent, by allowing for different working memory capacities for each agent, the system could potentially become more accurate in modeling learning and teaching between human agents.

Other cognitive research groups have shown that using a power law for the knowledge decay process leads to a more accurate decay modeling than the use of an exponential law (Kahana & Adler, 2002). Thus, changing the C-ULM decay exponential decay function to power function could provide future insight into knowledge decay and its impact on C-ULM learning and agent performance.

Another potentially fruitful line of research is to test the C-ULM generated data against human behavioral and neurological data. This could provide interesting insights on how to further improve the model accuracy.

Section 5.3 showcases that the C-ULM simulation can be used to better understand how the system changes when various system parameters are changed. This type of experiments could make C-ULM an useful tool for performing “what-if” analyses. One of the most interesting and unexpected results of this type is that a lower working

memory in a system using chunking leads to a more efficient task solving process than a system with a higher working memory capacity. Furthermore, such insights could lead to a better understanding of the human learning mechanisms especially in the cases of long-term learning and problem solving where data from human subjects is generally not available.

6.2.2. Multiagent system research

From the intelligent agent perspective, the C-ULM simulation could prove useful in the research of multi-agent systems that involve human learning. For example, such a system could be comprised of both artificial and human agents collaborating together in solving domain specific tasks that require learning. By endowing the artificial agents with C-ULM driven insights into the human learning behavior, their reactions to fellow human counterparts could improve and thus better serve the system as a whole.

In addition, motivated by our findings described in section 5.3.7, we believe that C-ULM could become useful in solving MAS problems where there is little amount of available information. For example, various transportation problems could amount to a weighted graph representation where connected locations are represented by two connected graph nodes. The weight connection can represent specific features such as the maximum allowed flow of transportation units between locations. It is rather probable that a complex problem of this type provides a general location map without including information about allowed transportation flows between locations. The robustness of C-ULM shown in section 5.3.7 could help to improve the prospects on such research problems.

Furthermore, C-ULM offers a general framework for knowledge transfer between agents. This is achieved by the design of the learning and teaching algorithms. Such knowledge transfer, driven by the working memory filtering and capacity constraint could be embedded in future multiagent systems. In such systems, large and distinct chunks of knowledge could be available to each agent but real world restrictions to agent-to-agent communication bandwidth and limited agent processing power would demand a performance-oriented selection of the most relevant pieces of knowledge to be transmitted among agents.

Lastly, another potentially useful research direction is experimenting with other types of agent interactions. For example, it could prove useful to explore one-to many teaching and learning interactions in order to improve the knowledge transfer protocol. In those interactions, a teacher agent could teach more learner agents and also a learner agent could learn from multiple teachers within the same time step. This could provide further insights into group learning and teaching and it would also more closely resemble a typical school-based educational process.

REFERENCES

1. J. F. Lehman, J. Laird, and P. Rosenbloom, "A gentle introduction to SOAR, an architecture for human cognition: 2006 Update", University of Michigan, 2006

2. A. S. Rao and M. P. Georgeff, "BDI Agents: From Theory to Practice", *Proc. of 1st International Conference on Multiagent Systems, AAAI, 1995*

3. J. Thangarajah, L. Padgham, and J. Harland, "Representation and Reasoning for Goals in BDI Agents", *ACSC Proc. of the 25th Australasian Conference on Computer Science, Vol. 4, pp. 259-265, 2002*

4. K. E. Merrick, "A Computational Model of Achievement Motivation for Artificial Agents (Extended Abstract)", *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems, AAMAS, 2011*

5. K. Shafi, K. E. Merrick, and E. Debie, "Evolution of Intrinsic Motives in Multi-agent Simulations", *Simulated Evolution and Learning (SEAL) 2012, Lecture Notes in Computer Science (LNCS) 7673, pp. 198–207, 2012*

6. M. K. D. Hardhienata, K. E. Merrick, and V. Ugrinovskii, "Task Allocation in Multi-Agent Systems using Models of Motivation and Leadership", *WCCI 2012 IEEE World Congress on Computational Intelligence, Brisbane, Australia, 2012*

7. E. Sklar and M. Davies, "Multiagent Simulation of Learning Environments", *Proc. of 4th Int. Conf. on Autonomous Agents and Multiagent Systems, AAMAS, 2005*

8. M. Spoelstra and E. Sklar, "Using Simulation to Model and Understand Group Learning", A*gent Based Systems for Human Learning, International Transactions on Systems Science and Applications,* 4(1), 2008

9. D. F. Shell, D. W. Brooks, G. Trainin, K. M. Wilson, D. F. Kauffman, and L. M. Herr, "The Unified Learning Model: How Motivational, Cognitive, and Neurobiological Sciences Inform Best Teaching Practices." Netherlands, Springer, 2010

10. G. Wilson-Doenges and R. A. R. Gurung. "Benchmarks for scholarly investigations of teaching and learning," *Australian Journal of Psychology*, vol. 65, pp. 63-70, 2013

11. S. J. Durning and A. R. Artino, "Situativity theory: A perspective on how participants and the environment can interact: AMEE Guide 52", Medical Teacher, vol. 33, pp. 188-199, 2011

12. A. L. Nebesniak, "Effective instruction: A mathematics coach's perspective," The Mathematics Teacher, vol. 106:5, pp. 354-358, December 2012/January 2013

13. T. Wasserman, "Attention, motivation, and reading coherence failure: A neuropsychological perspective," Applied Neuropsychology: Adult, vol. 19:1, pp. 42-52, 2012

14. N. Khandaker and L. K. Soh, "SimCoL: A simulation tool for computer-supported collaborative learning," *IEEE Transactions on Systems, Man, and Cybernetics-Part C: Applications and Reviews*, vol. 41:4, pp. 533-543, 2011

15. N. B. Turk-Browne, B. J. Scholl, M. M. Chun, and M. K. Johnson, "Neural evidence of statistical learning: Efficient detection of visual regularities without awareness." *J. Cognitive Neuroscience*, 21:1934-1945, 2008

16. M. J. North, N. T. Collier, and J. R. Vos, "Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit", *ACM Transactions on Modeling and Computer Simulation*, Vol. 16, Issue 1, pp. 1-25, 2006

17. J. R. Anderson, "A spreading activation theory of memory", *J. Verb. Learning & Verb. Behavior*, 22:261-295, 1983

18. D. J. Harris, "Pitch discrimination", *J. Acoustical Society of America*, 24:750-755, 1952

19. M. J. Kahana, and M. Adler, "Note on the power law of forgetting", Center for Complex Systems, and Program in Neuroscience and Department of Mathematics, Brandeis University, March 2002

20. D. F. Shell, C. C. Murphy, and R. H. Bruning (1989), "Self efficacy and outcome expectancy mechanisms in reading and writing achievement", *Journal of Educational Psychology*, 81, 91 100.

21. D. F. Shell, C. Colvin, and R. H. Bruning (1995), "Self-efficacy, attribution and outcome expectancy mechanisms in reading and writing achievement: Grade level and achievement level differences", *Journal of Educational Psychology*, 87, 386-398.

22. M. Bar (2000). Conscious and non-conscious processing of visual object identity. In Y. Rossetti & A. Revonsuo (eds.), "Beyond dissociations: Interaction between dissociable conscious and nonconscious processing". Amsterdam: John Benjamins (pp. 153-174).