# Construct Demo Input Deck

**Brian R. Hirshman, Geoffrey P. Morgan,**
**Jesse R. St. Charles, and Kathleen M. Carley**

June, 2010
CMU-ISR-10-118

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Center for the Computational Analysis of Social and Organizational Systems
CASOS technical report.

This work was supported in part by the IRS project in Computational Modeling, the Air Force Office of Sponsored Research (MURI FA9550-09-1-001 mathematical methods for assisting agent-based computation), and the NSF IGERT in CASOS (DGE 997276). In addition support for Construct was provided in part by Office of Naval Research (N00014-06-1-0104 and MURI N000140-81-1-186 a structural approach to the incorporation of cultural knowledge in adaptive adversary models), and the National Science Foundation (SES-0452487). Additional support was provided by the Air Force Office of Sponsored Research (MURI N00014-08-1-1186 cultural modeling of the adversary). Further support was provided by CASOS - the Center for Computational Analysis of Social and Organizational Systems at Carnegie Mellon University. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Internal Revenue Service, the National Science Foundation, the Office of Naval Research, the Air Force Office of Sponsored Research, or the U.S. Government.

**Keywords:** Construct, multi-agent simulation, dynamic network analysis, agent-based modeling, input

## Abstract

This technical report describes a demo input deck for Construct, the CASOS dynamic network simulation tool. It describes an example problem – understanding the difference in information diffusion patterns when bridging agents span two otherwise separate groups – and then how such a simulation would be simulated using Construct. In so doing, it describes the way in which data is supplied to Construct. Additionally, this report also describes how to run the simulation using the command line, and ways in which the simulation could be extended to analyze other problems.

Table of Contents

## List of Figures

## 1 Introduction

Construct, a simulation environment developed in the CASOS lab, is a complex, agent-based simulation model that is grounded in sociology and cognitive science. It seeks to model the processes and situations by which humans interact and share information. Construct is an embodiment of constructuralism (Carley 1986), a mega-theory which posits that human social structures and cognitive structures co-evolve so that human cognition reflects human social behavior, and that human social behavior simultaneously influences cognitive processes. Recent work with Construct has sought to improve the social and cognitive richness of the Construct model, and to use these improvements in order to investigate information diffusion (Carley et al 2009; Hirshman & St. Charles 2009), merger integration (Frantz & Carley 2007), and a variety of other topics. The goal of this technical report is to provide a gentle introduction and demonstration of the features of Construct, and present the reader with the tools that will allow him or her to investigate similar types of questions.

This technical report describes a demo input deck for Construct. It presents a motivating example problem, describes the key components of the input deck that can be used to approach this problem, and then explains how to run the input deck and the results that should be observed for the example problem. This document is designed to provide an introduction to the syntax of ConstructML, the input file format for Construct. After reading this technical report, the reader should be able to understand the motivating example, understand how Construct implements the motivating example, and be able to follow along with the demonstration provided.

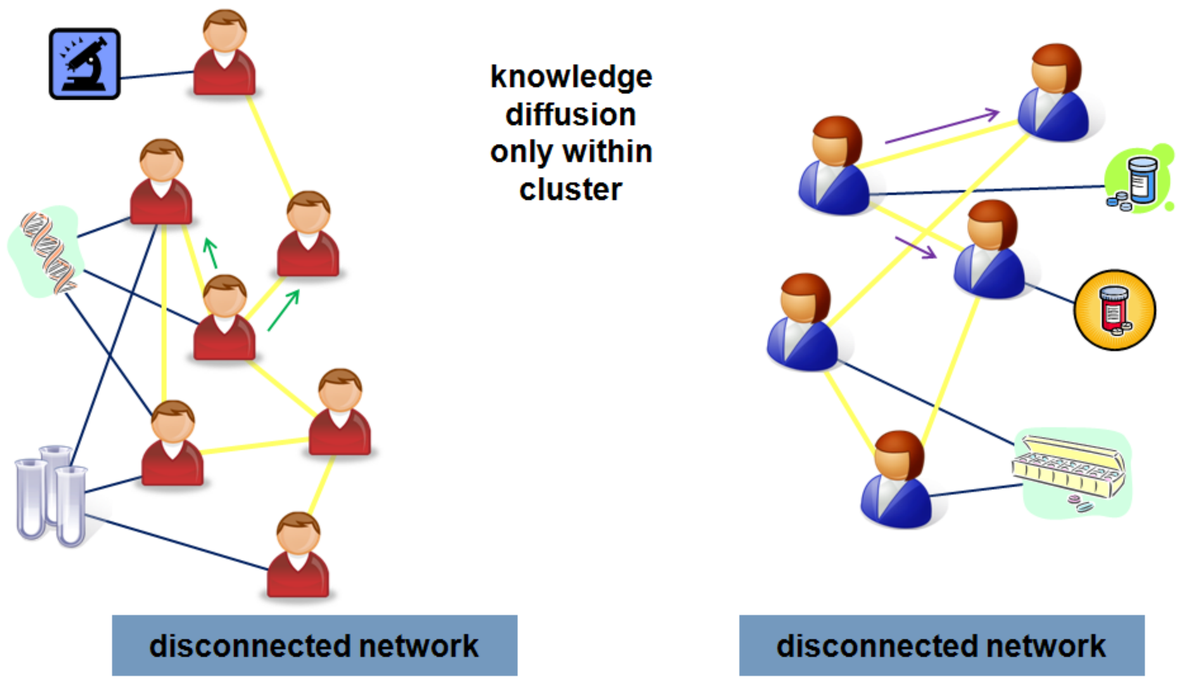
As this document will describe, Construct input decks are usually specified from the command line. As of this time, there is no GUI interface that allows access to all of Construct's features. While some of the simpler features can be accessed via the Near Term Analysis tool in *ORA (Moon & Carley 2007), the near term analysis does not allow the experimenter to manipulate all the input parameters available in Construct. To create inputs for more sophisticated experiments, it is necessary to manipulate the ConstructML directly and run the simulation via the command line. This technical report is designed to assist new users in understanding the ConstructML and how it can be used in order to build a successful simulation.

Construct is an evolving tool, and not all features described in this document will be state of the art in the coming years. The input deck described in this report is designed for use with Construct version 3.8.build016BRH, the public version available during the spring of 2010 and the 2010 CASOS Summer Institute. Future versions of Construct may modify components or parameters of this input deck. For best results with the included input deck, beginning users are encouraged to download the above version. All versions of Construct, including the version intended for use with this report, are available from the CASOS public web site, www.casos.cs.cmu.edu.

This technical report has sought to quote from an actual example of a Construct input deck (Appendix A) when possible. Such quotations will be specified in a `monospace font`. Code snippets will also be written in the `monospace font`, as these snippets are quotes from the demo input deck. Additionally, any Construct keywords, such as variable or network names, will use this `monospace font` to clearly differentiate an example from the surrounding text.

The remainder of this technical report is organized as follows. Section 2 describes the motivating example of information diffusion, which is helpful for understanding the Construct input as well as the assumptions made in the input deck. Section 3 describes the input deck in Construct, and specifically how the input deck is used in the motivating example. Section 4

**Figure 1: Two independent clusters of agents, each with their own sets of knowledge**



describes how to run Construct from the command line, and walks the reader through a small change that can be made to the input deck in order to see how information diffusion patterns can change. Section 5 concludes. An appendix presents the full demo input deck in a manner that can be lifted directly from this document.

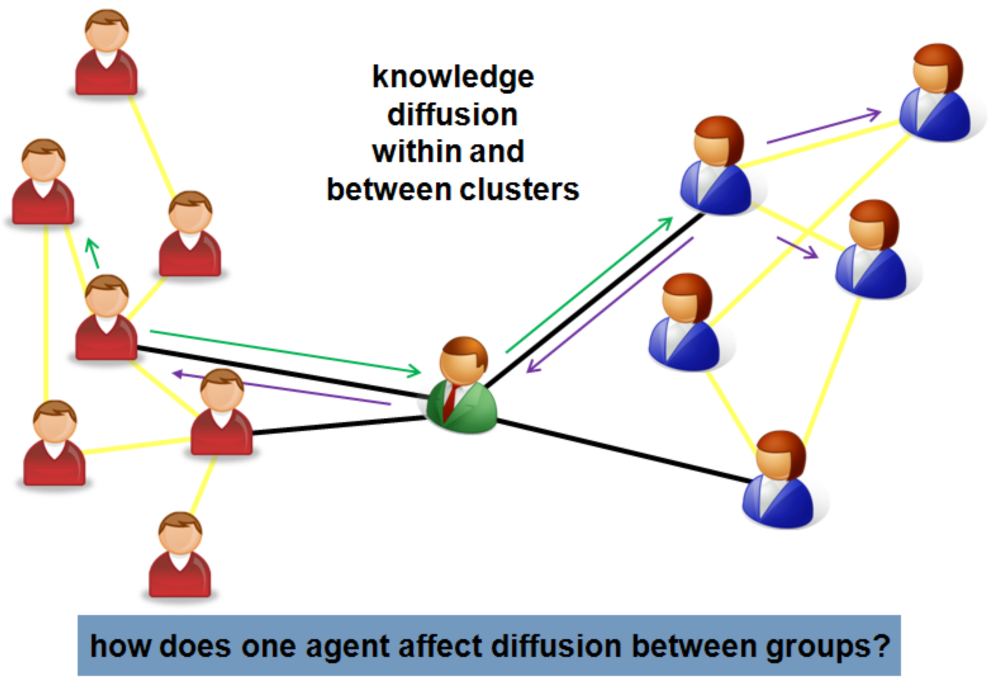
This document is meant to appeal to several types of readers. Readers who are new to Construct but want to get their hands dirty are encouraged to read Section 2 followed by Section 4 in order to understand the experiment and run it as quickly as possible. More advanced readers may wish to skip directly to Section 3 to understand the key features of the Construct input deck. Other users who are already building their own input decks may wish to use this document as a reference guide for understanding the various levers that this version of Construct provides. All users may find it more useful to use the table of contents in order to read sections of interest and reference chain to other parts of the document as necessary. This technical report has been written to accommodate all three reading styles as easily as possible.

## 2 Motivating Example

In order to demonstrate the features of the Construct simulation, an input deck describing a simple information diffusion scenario has been created.

Consider the following problem: suppose that there are two disconnected social networks each with their own domain-specific knowledge. For instance, consider the social networks pictured in Figure 1. The left-hand side network has seven red agents. This could be a network of researchers, sharing information specific to their domain. On the right hand side of the figure is a separate network of five blue agents. These agents could be considered industry analysts, each with knowledge specific to their domain. In both networks, the relationship between agents and knowledge are represented with blue lines. Note that the knowledge in each network is different, and there is no knowledge shared by the networks.

**Figure 2: Two clusters of agents linked by a bridging agent**


knowledge diffusion within and between clusters
how does one agent affect diffusion between groups?

In this simulated environment, the two networks are completely separate. Agents from one network cannot interact with those from the other. Thus, if the networks are evolved over time, information is expected to diffuse only within the networks. The red agents will continue sharing information relevant to their work, while the blue agents will share knowledge relevant to theirs. Over time, agents in each network will gain a richer picture of the knowledge in their own network, and thus will be connected to more knowledge items in their network. However, they will not have links to (read: they will not learn) any knowledge in the other network.

Now consider a slightly modified scenario. The two networks remain as before, but one or more agents are introduced which bridge them. Such a “bridge” agent or agents would have similar structural roles as the green agent in Figure 2. They would have ties to some of the red agents (researchers), and thus could learn knowledge relevant to that community, and also have ties to some of the blue agents (industry) to learn their knowledge. Thus, these agents could interact with agents in both networks and link the two communities together as a larger network.

More importantly, bridge agents would allow knowledge to diffuse from each community into the other community. For instance, the green arrow represents knowledge initially known by a red agent. This knowledge can still diffuse to other red agents as in Figure 1, but can now also diffuse through the green agent to the blue agents on the right-hand side. Similarly, the purple arrow represents knowledge initially known by the blue agents. This knowledge can still diffuse in the right-hand network, but can also diffuse through the bridge agent to the red agents.

Obviously, this is a very simplistic example. Very few human networks are so cleanly separated that information can only diffuse through exactly one bridging agents. However, such an example makes for an excellent simulation exercise. For instance, the input deck for the disconnected networks case (Figure 1) is relatively straightforward, and is the input deck already prepared in the Appendix. Running this input deck, as is discussed in Section 3.6.2, will allow the reader to observe how different information diffuses within the two separate networks.

However, it is also possible to make a relatively simple change to the demo input deck in order to convert it to the Figure 2 case. The user can then compare and contrast the results of these two simulations in order to understand how the small change made a difference.

Such an experiment would be relatively straightforward to implement in Construct, and could serve as a demonstration exercise. The remainder of this document will describe one way – the easiest way – that this experiment can be set up. While the networks in this example used will be slightly larger than those depicted in Figure 1 and Figure 2, the idea remains the same. A relatively small number of agents will be in two separate networks, and bridging agents can be added in order to understand how knowledge diffuses between them.

While this example is simplistic, it is not unworthy of simulation. For instance, consider the question of how rapidly will specific pieces of information diffuse in one network when the bridging agent is absent. This is a question that might be approached analytically, with survey data, or with lab experiments. However, consider what happens when the bridging agent becomes active. Does this increase the rate of information diffusion – i.e., do more agents end up learning more facts? Does the information learned coming from the other network supplement, or displace, information that would have diffused within an agent's own network? Do all pieces of information diffuse at the same rate, or is there substantial variation? Such questions are much more difficult to address with alternative techniques, and are ideal questions to investigate with a simulation approach. The demo input deck described in Section 3 and run in Section 4 will be used to address these questions.

While this technical report describes an input deck which can be used to investigate such questions, it is not meant to provide a definitive answer to them. Instead, the goal of this input deck is to allow the user to understand how these questions could be investigated using Construct and the input file provided in the Appendix. Some example analysis is done on the result of one experiment, but such results should be viewed with caution since these findings are designed as demonstrations. Users wishing to further explore this issue are invited to use the input deck as a jumping-off point in order to approach this question in a more principled manner than presented in this document.

## 3 Demo Input Deck Overview

The input deck described this technical report is an input deck designed to investigate the problem outlined in Section 2. This section outlines the key features of that input deck.

The input deck simulates the interaction between two groups of agents (called agent group A and agent group B), each of which have a separate set of knowledge facts (knowledge group 1 and knowledge group 2, respectively). The groups of agents are separate and cannot interact with agents in the other group. However, there is also a small group of bridging agents (agent group C); these agents span both groups and can allow information to diffuse between them when they are active. If the agents in agent group C are not active, however, then information will be confined to each group, and a clear boundary between both groups should be observed. The full input deck is available as an Appendix. In the default state of this input deck, the bridging agents are initially inactive in order to parallel Figure 1. Thus, the two networks will be separate, and no information will pass between them. Subsequent changes, like those made in Section 3.6.2, will activate these bridging agents to investigate how the simulation changes.

This demonstration input deck is designed to work with Construct version 3.8.build016 BRH (available April 2010). Future versions of Construct may not support the functionality or behavior described in this section, though effort will be made to keep Construct backwards-compatible with the features outlined in this section. However, it is possible that future versions of Construct may deprecate some of the features described in this input deck, so users are strongly advised to read the most recent literature when examining a new release of Construct.

As this section is quite extensive, it is useful to describe its high-level outline with an introduction. Section 3.1 describes the high-level organization of the input deck. Sections 3.2 to 3.7 describe the six crucial elements of the input deck: the construct variables which facilitate input deck creation, the construct parameters which control key simulation behavior, the nodes that comprise the core entities in Construct, the networks which relate the nodes, the transactive memory networks which serve as agent perceptions, and operations which provide output. Section 3.8 recapitulates many of the themes introduced throughout the remainder of this section, and ties the input deck design back to the motivating question described in Section 2.

## 3.1 Input deck organization

Construct input decks are written in a specialized form of XML called ConstructML. Because ConstructML is a variant of XML, inputs to the simulation are represented in XML tags. Most tags appear in pairs. The names of the tags are important, and serve to tell the simulation what kind of data is contained in the tag. Tags are organized in a specific structure, and their organization within the input deck is crucial for understanding what the tags mean. Users not familiar with XML syntax are strongly encouraged to examine an XML guide prior (or concurrent with) attempting to understand Construct input.

When parsed by the Construct executable, input decks are read line by line from top to bottom. Construct must be able to understand each line, or an error will be thrown and reported to the user. Some dependencies are implicit and will cause errors if violated. As such, it is strongly recommended that users following the structures of this input deck closely, and modify material in place.

At a very high level, the Construct input file has the following structure.

```
<construct>
  <construct_vars>
  <construct_parameters>
  <nodes>
  <networks>
  <transactivememory>
  <operations>
</construct>
```

The outermost tag in the input file is a tag named `<construct>`, which marks this document a Construct input file. The child tags define important properties of the construct input deck, including the entities to be simulated and the relationships between them. Thus, the `construct` tag must contain the following child tags, including:

- construct_vars. The `construct_vars` tag specifies a list of variables that are used in the rest of the file. Important variables can be declared here if they are to be referenced in the rest of the file. For instance, the number of agents can be declared as a variable in order to facilitate its use in other parts of the file. The `construct_vars` tag, and key variables in the demo input deck, is discussed in Section 3.2.

- construct_parameters. The `construct_parameters` tag specifies parameters critical for the operation of the simulation. Parameters can govern what models (key parts of the simulation logic) are in use, what additional features are enabled, and how Construct will deal with certain types of input. The `construct_parameters` tag, and key parameters in the demo input deck, is discussed in Section 3.3.
- nodes. The `nodes` tag specifies the entities that will be simulated. Nodes include agents in the simulation, knowledge that will be represented, beliefs, and a number of other entities. The `nodes` tag, and key nodeclasses in the demo input deck, is discussed in Section 3.4.
- networks. The `networks` tag specifies the relationships that occur between the nodes. For instance, the who-knows-what relationship between agents and knowledge is represented as a network. The `networks` tag, and key networks in the demo input deck, is discussed in Section 3.5.
- transactivememory. The `transactivememory` tag is a special tag which represents how agent nodes perceive the capabilities of other agents. For instance, there is a `knowledge transactive memory network` that describes how agents perceive the knowledge of others as well as a `beliefs transactive memory network` that describes how agents perceive the beliefs of others. The `transactivememory` tag, and its use in the demo input deck, is discussed in Section 3.6.
- operations. The `operations` tag specifies simulation outputs. Construct is capable of printing out a wide variety of simulation outputs, including networks that are manipulated by the simulation. The `operations` tag, and key operations in the demo input deck, is discussed in Section 3.7.

All of these tags are required, and Construct will error if these tags are not present. For simplicity, it is recommended that users seeking to create a new simulation start by modifying an existing input deck.

## 3.2 Variables

Construct variables are user specified constants that can be used throughout the program to improve readability. They are used to simplify the act of consistently changing experimental parameters of interest. For example, if a user is interested in running an experiment that varies the number of agents in the simulation, a construct variable can be initialized once and then referenced consistently whenever that value is necessary. The set of Construct variables is defined at the top of the input deck, and should be done inside the `<construct_vars>` tag, thus:

```
<construct_vars>
  <!-- Put your variables here -->
</construct_vars>
```

Between the `<construct_vars>` tags, each variable can be defined using the following syntax.

```
<var name="[name]" value="[value]"/>
```

The name of the variable must be unique sequence of alphanumeric characters (i.e., a variable with the same name cannot already have been defined), though the variable name must begin with a letter. Variable names are not case sensitive. Construct will convert all variable names, and most variable values, to lower case for internal use. The value of the parameter can be a string, a number, or an expression that can be evaluated as a script (described in Section 3.2.2). Variable values are not typed when they are defined. However, when the variable is used, a type is provided and a cast of the value will be made from one type to another.

While variables can only be declared within the `construct_vars` tag, they can be referenced just about anywhere in the file – including in the creation of other variables. To reference the variable and use its value, the below syntax should be used.

```
construct::[type]::[variable name]
```

To reference the variable, the bare word `construct` should be followed by two colons, followed by the type of the variable, followed by another two colons, followed by the variable name. The most common types are `boolvar`, `intvar`, `floatvar`, or `stringvar`, which can be used if the user desires the value treated as a bool, int, float, or string value respectively. The variable is only cast when it is used in the above manner. Additional information about this reference system can be described in the technical report CMU-ISR-09-126, which describes the scripting system in Construct (Hirshman et al 2009).

The input deck example includes several dozen parameters, many of which are referenced multiple times in the document. The most important variables are discussed in Section 3.2.1, while the remaining variable definitions are discussed in Section 3.2.2. The variables in Section 3.2.1 are introduced in the order in which they are encountered in the demo input deck.

Before continuing, it should be noted that the order in which the variables are declared is important. Variables are loaded into the simulation in the order in which they are read. As just described, variables can reference other variables. However, the referencing variable must be declared later in the file relative to the variable that it references. While many of the variables described in Section 3.2.1 are constants, it should be noted that the variables that are not constant will follow those variables that reference them.

### 3.2.1 Key constants

- short_experiment. This variable, and the associated `time_count` variable, change the number of simulated time periods in the simulation. By setting the short experiment variable to true, as it is by default, the experiment can be made to run more rapidly. Manipulating this variable is suggested as a follow-up exercise (Section 4.8.5). Note that the `time_count` variable is defined after the `short_experiment` variable, as the former is dependent on the value of the latter.

```
<var name="short_experiment" value="true"/>
```

```
<var name="time_count" value="
    if(construct::boolvar::short_experiment)
     {
       50
      }
     else
     {
       100
     }
"/>
```

- bridging_agents_active. This variable determines whether bridging agents are active or not, and is the primary variable of interest to the motivating example. It determines if the bridging agents will be active in the `agent active time period network` (Section 3.5.16, page 61) and thus able to interact with agents in the different agent groups. This is also the variable that will be manipulated by the user in the demo described in Section 3.6.2.

```
<var name="bridging_agents_active" value="false"/>
```

- agent groups. These variables specify the number of agent groups that will be simulated. This is provided for simulation bookkeeping purposes.

```
<var name="agentgroup_count" value="3"/>
```

- agent group sizes. These variables specify the size of each agent group. Note that there are three agent groups, as specified by the `agentgroup_count` variable. Also be aware that agent groups A and B are larger than group C, the group of bridging agents; this is intentional.

```
<var name="agentgroup_A_size" value="20"/>
<var name="agentgroup_B_size" value="20"/>
<var name="agentgroup_C_size" value="2"/>
```

- knowledge groups. These variables specify the number of groups that will be simulated. This is provided for simulation bookkeeping purposes.

```
<var name="knowledgegroup_count" value="2"/>
```

- knowledge group sizes. These variables specify the size of each knowledge group. There are two groups, as specified by the `knowledgegroup_count` variable. These knowledge groups are the same size for the purposes of this experiment.

```
<var name="knowledgegroup_K1_size" value="30"/>
<var name="knowledgegroup_K2_size" value="30"/>
```

- task sizes. These variables specify the number of tasks that agents are able to perform during the simulation. Binary tasks represent decision tasks that agents perform every turn based on their knowledge, while energy tasks represent tasks that agents conditionally perform based upon how much activity they have had during the simulation. Neither of these is used in the simulation, so both are set to zero.

```
<var name="binarytask_count" value="0"/>
<var name="energytask_count" value="0"/>
```

- belief sizes. This variable specifies the number of beliefs that will be used in the simulation. Beliefs represent agreement or disagreement with a principle, and can be manipulated by agents in a variety of ways. For simplicity, they are not used in this example and thus the number of beliefs is set to zero. Note that if the number of beliefs is set to a value greater than zero, it will be necessary to change the belief model (see Section 3.3.9).

```
<var name="belief_count" value="0"/>
```

### 3.2.2 Scripting

The variables not described in Section 3.2.1 are scripted variables. These variables make use of Construct's scripting system, which is described extensively in the CASOS technical report CMU-ISR-09-126 (Hirshman et al 2009). While that technical report goes into detail about the way in which scripting can be used to increase Construct's flexibility and power, for the purposes of this input deck it is sufficient to understand that scripting can be used in order to compute simple math operations on variables.

The included example computes other variables which are sums of these base values. For instance, consider the first variable that uses a mathematical operation, the definition of the variable `agentgroup_A_end`, which is defined at file line 85.

```
<var name="agentgroup_A_end" value="
     construct::intvar::agentgroup_A_start +
          construct::intvar::agentgroup_A_size - 1"/>
```

This variable is the mathematical result of adding or subtracting three values: adding the variable `agentgroup_A_start`, `agentgroup_A_size`, then subtracting the constant value `1`. The complicated syntax `construct::intvar::agentgroup_A_start` is a verbose method for telling Construct to treat the result of the construct variable `agentgroup_A_start` as an integer when performing the addition. The values of these variables have been specified earlier in the file: `agentgroup_A_start` is defined to be 0 on the immediately prior line, while agentgroup_A_size is defined to be 20 as seen in Section 3.2.1. It is not surprising, then, that this mathematical expression evaluates to 0+20-1 = 19, which would be the end-value expected for a zero-indexed group of twenty agents. Similar calculation can be used to show that `agentgroup_B_end` will evaluate to 39 and `agent_count` to 42.

While such syntax may seem cumbersome to a beginning user examining a Construct input deck for the first time, it is actually extremely helpful when one begins to try and manipulate it.

The majority of the networks in the simulation will refer to construct variables such as `agentgroup_A_start` or `agentgroup_B_end` in order to specify which agents should be assigned specific values. By using the scripting system to calculate these values, it makes it simple to change the number of agents in each group. If the user changes `agentgroup_A_size` (i.e., to a value like 30), Construct will automatically recompute the value of `agentgroup_A_end` (i.e., 29) when the simulation begins. Similarly, if the user changes the number of knowledge bits in knowledge group K1, all variables dependent on it will be recomputed. This greatly decreases the number of file locations that must be changed if the input deck is manipulated, and greatly decreases debugging.

## *3.3 Parameters*

Parameters are global values that help control how Construct operates as a simulation and which optional modules are enabled (by default most modules are disabled). All parameters should be set inside the `<construct_parameters>` tag.

```
<construct_parameters>
  <!-- Put all parameters here -->
</construct_parameters>
```

Within the construct parameters tag, each parameter is set using the following syntax.

```
<param name="[name]" value="[value]">
```

The name of the parameter must be a valid parameter name, such as the parameters specified below. Some advanced parameters have been omitted from this input deck and control other internal subsystems. The value of the parameter must be a valid value for the parameter. If the value is invalid, Construct will error and exit. Note that many but not all Construct parameters can be specified using Construct variables.

The parameters used in the demo input deck, as well as some alternate valid settings for these parameters, are described below. Parameters are presented in the order encountered in the demo input deck, which can be found in the appendix.

### 3.3.1 Seed

This parameter controls the random seed used to control the simulation. If a time-dependent seed is desired, set this value to `0`. If a constant seed is desired, set to a , otherwise set to any integer. The value used in this experiment is `1`, in order to ensure that the experiment results are constant if the simulation is performed multiple times. This has the added benefit of ensuring that user results will be identical to the figures shown in the walkthrough that is Section 3.6.2.

```
<param name="seed" value="1"/>
```

### 3.3.2 Verbose initialization

This parameter determines whether the value of variables as referenced print to screen as the simulation starts up, which can be very useful for debugging a simulation. It is strongly recommended that users enable verbose initialization when attempting to debug a simulation, as verbose initialization will help a user determine the value of every Construct variables (Section 3.2) and thus every value used when defining the nodes and networks. This value is binary, and the default is `false`.

Note that this parameter cannot be a Construct variable: i.e., it cannot be of the type `construct::boolvar::`. This parameter is read as a bare word before Construct attempts to read any variables, and thus it cannot be interpolated. Thus, this parameter's value must be `true` or `false`.

```
<param name="verbose_initialization" value="false"/>
```

### 3.3.3 Dynamic environment

This parameter determines whether the simulation includes an 'outside world' agent that possesses different knowledge from other agents and can change information every turn, allowing the introduction of 'new' information to the simulation. This value is binary. The default is `false`, and should not be modified except by advanced users.

```
<param name="dynamic_environment" value="false"/>
```

### 3.3.4 Default agent type

This parameter determines what kind of agent is used as the default agent type. The type must be defined as a special `agent_type`, as discussed in Section 3.4 (page 16). The default type is `human`, as long as a human agent type is defined.

```
<param name="default_agent_type" value="human"/>
```

### 3.3.5 Learning and forgetting

These variables determine how agents acquire or lose information during the simulation.

- Forgetting determines whether agents can lose (i.e. forget) knowledge facts they have learned. This is true or false. If set to true, agent knowledge can decay, and will decay at the rate specified in the `binary forgetting rate network` (Section 3.5.9). The default is `false`.
- Binary Forgetting is relevant if agents can lose knowledge (i.e., the `forgetting` variable is set to `true`). If so, this flag determines whether the decay of knowledge is all-or-nothing. If this is set to `true`, then the agent forgets only entire facts, not parts of facts; knowledge of that fact will decay to zero. If this is set to `false`, then the agent forgets parts of facts, so knowledge can be fractional. This flag has no effect if `forgetting` is set to `false`.
- Binary Learning determines how agents learn. This is binary. If set to `true`, agents learn the entire fact at once or not at all. If set to `false`, agents can learn part of a knowledge fact. If both Binary Learning and Binary Forgetting are set to the `true` (or if Forgetting is disabled), the agent knowledge matrix will only have 0 and 1 values. The default is `true`.

```
<param name="forgetting" value="false"/>
```

```
<param name="binary_forgetting" value="true"/>
<param name="binary_learning" value="true"/>
```

### 3.3.6 Use mail

This parameter determines if the mail communication subsystem is active. This is true or false. If this is set to `true`, then agents can use the mail communication mechanism, which allows agents to send a message in one period that agents will be able to read at a later period. For agents to use mail, they must use the communicationMechanism called `mail`, as described in Section 3.4.10, and must employ a variety of supplemental networks and parameters, as further described in Section 3.5.35. For more detail on the mail subsystem, see CMU-ISR-08-114, "Modeling Information Access in Construct" (Hirshman & Carley 2008) as well as the notes in this document which summarize changes made in versions up to Construct version 3.8.build016BRH. The default value for the `use_mail` parameter is "false", and should be set to false by default since the subsystem can be computationally expensive when agents are not actively making uses of mail communication.

```
<param name="use_mail" value="false"/>
```

### 3.3.7 Default communication weights

These parameters determine what kind of messages will be sent between agents when they communicate. Agents are capable of communicating complex messages with multiple components, including knowledge, belief, and transactive memory. These weights serve to determine the type of content of a message, though the specific content transmitted will depend upon the agent.

- Belief Weight. The probability that an agent chooses to include its belief on any belief in the message. Each belief is chosen with equal probability.
- Transactive Memory Belief Weight. The probability that an agent chooses to send its perception of any third party's belief in the message. Each agent, and each belief for that agent, is equally likely to be chosen as long as the sender has a perception of the relevant third party's belief.
- Fact Weight. The probability that an agent chooses to include a fact in the message. If the agent sends a fact, the agent also implicitly sends the message that the sender has transactive memory of that fact.
- Transactive Memory Fact Weights. The probability that an agent chooses to send its perception of any third party's knowledge in the message. Each agent, and each knowledge bit for that agent, is equally likely to be chosen as long as the sender has a perception of the relevant third party's belief.

These need to add up to 1, but will normalize if a mechanism is removed or a particular item cannot be transmitted. For instance, in the demo input deck in the appendix, no beliefs are modeled. This means that the weights will be renormalized to excluding both belief and beliefTM weights. Thus, the probability of choosing to include a knowledge bit is effectively 71.4%, while that of choosing to include any transactive memory bit is 29.6%.

```
<param name="communicationWeightForBelief" value="0.2"/>
```

```
<param name="communicationWeightForBeliefTM" value="0.1"/>
<param name="communicationWeightForFact" value="0.5"/>
<param name="communicationWeightForKnowledgeTM" value="0.2"/>
```

### 3.3.8 Thread count

This parameter determines the number of threads can Construct use in order to parallelize a construct process. This is an integer value, minimum `1`, which is the default value. This should only be set above 1 in Linux systems, as multi-thread builds for Windows are not supported. If running multiple Construct simulations at once, it is preferable to place N separate Construct runs in N different processes simultaneously than to run one simulation using N threads.

```
<param name="thread_count" value="1"/>
```

### 3.3.9 Active models

The active models parameter specifies the models that the simulation uses in order to govern interaction. The model system is designed to, in future releases of Construct, allow users to supplement the default Construct logic with logic that performs additional tasks. For instance, users will be able to write models for the way some kinds of agents interact with others, the way certain facts are treated, and other modifications to core simulation behavior. However, as this system is still being developed, the parameters supplied here should only changed by an expert.

There are three models that are used by the current simulation.

- standard interaction model. The standard interaction model uses homophily and expertise to guide interaction among the agents. All of the nodeclasses described in Section 3.4 and the majority of the networks described in Section 3.5 must be present in the input deck for the standard interaction model to function successfully.
- standard influence model. The standard influence model uses influence and influenceability (how susceptible is a particular agent to influence) to determine how an agent's belief is influenced by those around it. This influence model requires that the `agent belief network` (Section 3.5.12), `beInfluenced network` (Section 3.5.5) and `influenceability network` (Section 3.5.6) be defined in the input deck for it to run properly. Agents with high influenceability will be able to more strongly influence their peers; similarly, agents with low beInfluenced values will be more susceptible to such influence. Note that this influence calculation must be updated before it is used in a belief model.
- standard belief model. The standard belief model updates an agent's belief based on both the beliefs of others (the influence calculated by the influence model), the belief weights associated with facts that the agent knows, and the agent's knowledge in the previous time period. Thus, in this model, the agent is influenced both by what it knows, what it previously believed, and what others around it believe. The standard belief model requires the `agent belief network` (Section 3.5.12), the `belief knowledge weight network` (Section 3.5.13), and a hidden network set up by the influence model. It should be noted, however, that the parameter `disable` is passed as a keyword to this model. This specifies that the belief model will be disabled during the simulation. This is done because no beliefs are modeled (the

**Figure 3: Networks required for Key Construct Models**

| **Required Networks for Standard Interaction Model** | **Required Networks for Standard Influence Model** | **Required Networks for Standard Belief Model** |
|---|---|---|
| *for agent interaction (23)* | *for agent influence (3)* | *for agent belief (4)* |
| agent initiation count | beInfluenced | beInfluenced |
| agent reception count | influenceability | knowledge |
| agent message complexity | agent belief | agent belief |
| agent selective attention effect | | belief knowledge weight |
| agent learning rate | | |
| agent forgetting rate | | |
| agent learn by doing rate | | |
| knowledge | | |
| physical proximity | | |
| sociodemographic proximity | | |
| social proximity | | |
| physical proximity weight | | |
| sociodemographic proximity weight | | |
| social proximity weight | | |
| binarytask similarity weight | | |
| binarytask assignment | | |
| binarytask requirement | | |
| binarytask truth | | |
| knowledge similarity weight | | |
| knowledge expertise weight | | |
| interaction knowledge weight | | |
| transmission knowledge weight | | |
| knowledge priority | | |
| learnable knowledge | | |
| **The Five Core Networks (always required, regardless of model)** | | |
| interaction sphere | | |
| access | | |
| agent active timeperiod | | |
| agent group | | |
| knowledge group | | |
| *note: the word "network" has been omitted from the end of all network names* | | |

number of belief nodes is zero, as defined in Section 3.2.1 and used in Section 3.4.3). To perform follow-up simulation using beliefs, add one or more beliefs, specify the `belief knowledge weight network` appropriately, and change the keyword `disable` to `mask_mode`. Such changes are further discussed in Section 4.8.5.

The order in which the models appear in the `active_models` parameter are the order in which they are executed each time period. Thus, in the current simulation, agents will interact before influence is calculated, and only then will beliefs be updated.

The models in Construct require specific networks in order to operate, as listed in Figure 3. The standard interaction model requires the greatest number of networks, as it is the most complex. It requires networks that control which agents prefer to interact with which agents, how frequently agents can interact, what can be shared when agents communicate, and what kind of tasks can agents perform. The standard influence model specifies the networks that are used in calculating social influence among agents. It requires networks which specify how strongly agents influence each other and can, in turn, be influenced. The standard belief model, when operating in `mask_mode` as opposed to `disable`, specifies how strongly agents are influenced by other agents as well as their own knowledge. It requires both knowledge- and belief-related networks. Note that five networks, the `interaction sphere network`, `access network`, `agent active timeperiod network`, `agent group network`, and `knowledge group network`, are required by Construct regardless of which models are active. These networks must be present in any simulation involving Construct.

It is important to note that the `active_models` parameter contains a `with` variable, a meta-expression that determines how Construct reads the parameter. This with variable, `delay_interpolation`, specifies that Construct should not attempt to interpolate the `active_models` parameter when the parameter is first encountered by the system. The belief model has an expression in parentheses, which Construct's parser would like to interpret as a mathematical expression. As the larger expression is not a mathematical one, however, a fatal error would occur and Construct would exit. By specifying `delay_interpolation` as a `with` variable, Construct does not parse this variable right away and instead waits to parse it at a later time. This `with` expression should not be removed, even if the active models are modified. `With` variables are further described in CMU-ISR-09-126 (Hirshman et al 2009).

```
<param name="active_models" value="standard interaction
    model,standard influence model,standard belief
    model(type=>disable)" with="delay_interpolation"/>
```

### 3.3.10 Active mechanisms

This parameter specifies some network post-processes that are performed when setting up the simulation, no mechanisms are specified in this example. The literacy and information access mechanisms described in CMU-ISR-09-126 is one example of such a mechanism, though it is not described as such in that technical report (Hirshman & Carley 2008). The value `none` ensures that this parameter is disabled and no mechanisms are used. This parameter should not be modified except by advanced users.

```
<param name="active_mechanisms" value="none"/>
```

**Figure 4: Sample nodeclasses and important networks in the demo input deck**

| NODE CLASS | Agent | Belief | Knowledge | Binary Task | Timeperiod | Dummy_ Nodeclass |
|---|---|---|---|---|---|---|
| Agent | interaction sphere ntwk | belief network | knowledge network | task assign. network | agent active time ntwk | agent type network |
| Belief | | | belief weight ntwk | | | |
| Knowledge | | | | task truth ntwk | | |
| Binary Task | | | | | | |
| Timeperiod | | | | | | |
| Dummy | | | | | | |

## 3.4 Nodes

Nodes are the entities that are simulated in Construct. Nodes are grouped into classes of nodes, called nodeclasses, and are related to each other in terms of networks. This section describes some of the nodes and nodeclasses in Construct: specifically, the nodes and nodeclasses in the demo input deck.

The Construct simulation system uses the idea of “nodes” and “networks”, as opposed to the more common formulation of “agents” in the agent-based modeling community, because Construct grew out of the social and dynamic network analysis tradition (Carley 1991; Carley & Reminga 2004) and PCANS framework (Krackhardt & Carley 1998). Nodes are grouped together into nodeclasses, which can be seen on the top and left of Figure 4. All nodes of the same type are placed in one nodeclass; thus, all agent nodes are collectively placed in the agent nodeclass. Classes of nodes can be associated with other classes of nodes to create networks, examples of which can be seen in the remainder of Figure 4. Links in these networks are then manipulated when Construct is running. New links in the network can be added or modified: for instance, if the agent learns knowledge, a new link between the specific agent node and the relevant knowledge node can be created. Thus, as a Construct simulation runs, the relationship among different nodes will be modified.

Nodeclasses specify the node’s behavior in the simulation. For instance, agent nodes are the nodes that interact, learn, and hold beliefs. While all agent nodes are alike in the sense that they are in the same nodeclass, each agent node can be associated with (have links to) different knowledge or have different influentialness values. Agents in Construct are just one class of node. Another example nodeclass is the knowledge nodeclass. As with the agent nodeclass, different nodes in the knowledge nodeclass are alike in the sense that they represent knowledge from the simulation’s perspective, but are different in the way that they represent different knowledge bits. Other node classes include beliefs, timeperiods, groups, and other entities.

It is worth clarifying that the node classes described here are different than the agent classes described in CMU-ISR-07-107, which describes Construct agents and their features (Hirshman

**Figure 5: Required and optional nodeclasses in the demo input deck**

| **Required Nonempty Nodeclass** | **Required Nodeclass** | **Other Nodeclass** |
|---|---|---|
| *must have >= 1 nodes* | *may have 0 nodes* | *may be omitted* |
| agent<br>knowledge<br>agentgroup<br>knowledgegroup<br>timeperiod<br>agent_type<br>dummy_nodeclass | belief<br>beliefgroup<br>binarytask<br>energytask | (user-defined nodeclasses) |

& Carley 2007a). This terminology clash is unfortunate. Agent classes are groups of agents with similar properties, some of which are specified in the `agent type` nodeclass (Section 3.4.10). Other agent class properties are specified by using a variety of Construct networks (Section 3.5). Nodeclasses, on the other hand, are collections of nodes treated in a similar manner by the simulation. Agents are one example of a nodeclass, while knowledge is another. Multiple agent classes can be present in the agent nodeclass.

Nodeclasses in ConstructML are declared within a `<node>` tag in the input deck.

```
<node>
  <!-- Put all nodeclasses here -->
</node>
```

Within the `<node>` tag, nodeclasses can be declared in one of two ways. The syntax chosen depends on whether a large number of nodes or a small and fixed number of nodes are in the nodeclass. If there are a large number of nodes, the below syntax should be used.

```
<nodeclass type="[type]" id="[type]">
  <properties>
    <property name="generate_nodeclass" value="true"/>
    <property name="generator_type" value="count"/>
    <property name="generator_count" value="[number]"/>
  </properties>
</nodeclass>
```

In the above syntax, the keywords `[type]` and `[number]` should be replaced with the name of the nodeclass to generate and the number of nodes that fall in that nodeclass. The `generate_nodeclass` and `generator_type` properties of the nodeclass will ensure that the specified number of nodes is generated for the nodeclass. Note that the `generator_count` property can be and often is a Construct variable (Section 3.2), which allows the number of nodes in the nodeclass to be changed quickly. Nodeclasses generated in this way can have between 0 and 65535 nodes, the maximum number of nodes in a nodeclass.

**Figure 6: Key nodeclasses in the demo input deck**

| Nodeclass Name | Number of Nodes In Demo Input Deck | Function or Purpose in Demo Input Deck |
|---|---|---|
| **agent** | 42 (20+20+2) | actors in simulation that interact and learn |
| **knowledge** | 60 (30+30) | used to compute similarity and expertise |
| **belief** | 0 | describe agreement with a principle |
| **binarytask** | 0 | tasks based on knowledge |
| **energytask** | 0 | tasks based on amount of time spent |
| **timeperiod** | 50 | simulation time tick |
| **agent group** | 3 | collections of agents for statistical purposes |
| **knowledgegroup** | 2 | collections of knowledge |
| **dummy_nodeclass** | 1 | necessary for specifying attribute networks |
| **agent_type** | 1 | necessary for specifying key agent behaviors |

There is an alternate way to specify a small number of nodes, which is useful if the number of nodes that will be needed is fixed. For readability, it is suggested that this syntax be used only when there are fewer than ten or twenty nodes to specify.

```
<nodeclass type="[type]" id="[type]">
  <node id="[id]" title="[title]"/>
  <node id="[id]" title="[title]"/>
  ...
</nodeclass>
```

As in the other syntax, the nodeclass type must be specified in the `[type]` variable. However, in this syntax, the nodes in the nodeclass are specifically enumerated one line per node. These nodes must be given a unique ID as well as a user-specified title. Both types of nodeclass generation mechanisms will be demonstrated in the demo input deck.

Several different types of nodeclasses are described in this document. Some of these nodeclasses are optional, and can be added to or removed from the input deck without consequence. Others must be present, but can contain zero nodes in the nodeclass. Still others must be present and must contain at least one node in order for Construct to function. A breakdown of different types of nodeclasses is provided as Figure 5. Nodeclasses that are required must be present because they are used in defining networks that are referenced by Construct algorithms. While input decks can specify that some nodeclasses have zero nodes in them, meaning that they will not affect that simulation, they still cannot be omitted from the input file or else Construct will exist and error.

The remainder of this subsection introduces the key nodeclasses in the demo input deck, in the order that they are encountered in the Appendix. The key features of each nodeclass are described in Figure 6. While the nodeclasses can be initialized in any order, it is suggested that the order presented in the demo input deck be followed in subsequent Construct input decks.

### 3.4.1 Agent nodeclass

The agent nodeclass represents the actors in the simulation. Agents are central to Construct's function. Agents can interact with other agents, can hold beliefs, and perform tasks.

Many of the networks in the input deck are networks associated with agents: agent x knowledge, agent x belief, and agent x timeperiod networks are the most common types of networks in the simulation. This nodeclass must be present in every simulation and must contain at least one node.

Agent nodes have an associated agent type, a collection of properties that is specified in the `agent_type` nodeclass (Section 3.4.10). The agent type is associated with the agent via the `agent type name network`, which is described in Section 3.5.1. The agent type describes such properties as whether the agent can send or receive knowledge when communicating, whether an agent can send or receive beliefs when communicating, and several other properties that define important agent behavior.

It is important to note that agents in the simulation are often human, but not all agents in the simulation must be (Carley et al 2009; Hirshman & Carley 2008). Agents are any actors that should interact with the population, which can include interventions like newspapers as well as collections of agents such as organizations. Human agents can choose to interact with these non-human agents for a variety of reasons; the likelihood of interaction can be modified by adjusting weights on, for instance, knowledge shared between the human and non-human agent. Whether the agent is designed to be human or non-human should be specified in the agent type nodeclass, described in Section 3.4.10. In the agent nodeclass declaration, these agents will be treated identically.

In the demo input deck, there are `agent_count` agents the nodeclass, all of which will be defined as human agents (in Section 3.4.10 and Section 3.5.1). The construct variable `agent_count` is a defined in Section 3.2. The value of the `agent_count` variable should be 42, as it is the sum of the twenty agents in agent group A, twenty in agent group B, and two in the bridging agent group (agent group C).

```
<nodeclass type="agent" id="agent">
  <properties>
    <property name="generate_nodeclass" value="true"/>
    <property name="generator_type" value="count"/>
    <property name="generator_count"
      value="construct::intvar::agent_count"/>
  </properties>
</nodeclass>
```

### 3.4.2 Knowledge nodeclass

The knowledge nodeclass represents the knowledge information that agents can exchange during the simulation, with one node per knowledge bit represented in the simulation. The “meaning” of knowledge in Construct is defined by the user; knowledge can be stylized, meaning it is simply an abstract representation of something that brings agents together, or it can have a specific meaning. In many simulations, both stylized and specific knowledge will be used; often, some stylized social knowledge will be used to drive interaction among agents while other specific knowledge bits may have a more specific meaning that is of particular interest for simulation analysis (Carley et al 2009; Hirshman & St. Charles 2009). This nodeclass must be present and must contain at least one node.

Agents are associated with knowledge via the `knowledge network` (Section 3.5.11). Agents can have full knowledge or partial knowledge of each bit by setting the link weight in the knowledge network; a link weight of 1.0 represents full knowledge of the bit, while a link weight of 0.0 represents no knowledge. Agents cannot have negative knowledge.

Agents keep track of the knowledge of others via the `knowledge transactive memory network` (Section 3.6.1). These perceptions of others knowledge may be imperfect, meaning that agents may expect an alter to know a particular bit when it does not or may believe an alter is ignorant of a bit that the alter knows. Because some kinds of agents are boundedly rational, however, they will use these perceptions to drive their interactions.

In the demo input deck, there are `knowledge_count` knowledge bits the nodeclass, all of which are treated as stylized entities that can be easily exchanged among the actors in the simulation. The construct variable `knowledge_count` is a defined in Section 3.2. The value of the `knowledge_count` variable should be 60, as it is the sum of the thirty knowledge bits in knowledge group K1 and thirty knowledge bits in knowledge group K2.

```
<nodeclass type="knowledge" id="knowledge">
  <properties>
     <property name="generate_nodeclass" value="true"/>
     <property name="generator_type" value="count"/>
     <property name="generator_count" value=
                    "construct::intvar::knowledge_count"/>
  </properties>
</nodeclass>
```

### 3.4.3 Belief nodeclass

Beliefs in Construct represent agreement or disagreement with a principle. Nodes in the belief nodeclass represent these principles. Agents will then become associated with these beliefs, either positively (for agreement) or negatively (for disagreement) through an association in the `agent belief network` described in Section 3.5.12. The meaning or interpretation of each belief is left to the experimenter.

While the manipulation of beliefs is not included in this demo, it is important to mention several important characteristics about beliefs which may be useful for those seeking to model beliefs using Construct. First, the standard belief model (Section 3.3.9) defines complete agreement with a belief to be a value of +1.0 and complete disagreement to be -1.0. A value of 0.0 is treated both as neutral and serves as a default value. This implementation in Construct is a conscious design decision to represent agreement and disagreement as two extremes of a single belief, as opposed to representing two separate beliefs for “believes X” and “does not believe X”. Note, however, that it may be perfectly necessary to need multiple beliefs in order to examine a question of interest.

Additionally, continuing the theme of bounded rationality, agents in Construct do not have perfect perceptions of the beliefs that others hold. Agents can have the ability to store transactive memory about other agents’ beliefs in the `belief transactive memory network` (Section 3.6.2). Agents will refer to this transactive memory when calculating social influence, if using the standard influence model as described in Section 3.3.9. Thus, while the associations between agents and beliefs are important for decisions, the association between agents and perceived beliefs of others will be critical for social influence mechanisms.

In the demo input deck, there are `belief_count` beliefs in the simulation. In the construct variables section, however, this value has been set to zero. Thus, no beliefs are modeled in the demo input deck, and all networks involving beliefs will be ignored by the simulation. Note, however, that the belief nodeclass is a required nodeclass, and thus cannot be omitted from the input deck.

```
<nodeclass type="belief" id="belief">
  <properties>
     <property name="generate_nodeclass" value="true"/>
     <property name="generator_type" value="count"/>
     <property name="generator_count" value=
                    "construct::intvar::belief_count"/>
  </properties>
</nodeclass>
```

### 3.4.4 Binary task nodeclass

Binary tasks in Construct are actions that agents can perform during the simulation. Nodes in the belief nodeclass represent these tasks. These tasks require knowledge in order to perform and can potentially be attempted every simulation timeperiod. While binary tasks are not modeled in this demo, it is useful to describe the key features of the binary task in case users decide to use them in future investigations.

Agents who are eligible to perform a binary task (which is set in the `binary task assignment network`, Section 3.5.24) will then use the subset of their knowledge that is relevant for the task (set in the `binary task requirement network`, Section 3.5.25) and compare that subset to the true values (set in the `binary task truth network`, Section 3.5.26). This decision can then be inverted due to misrepresentation (set in the `agent misrepresentation probability network`, Section <TODO>). If binary tasks are present in the simulation, agents can be assigned to one or more binary tasks to perform. In each time period, agents will attempt to perform each task. Agents do so by selecting the subset of knowledge which the task requires.

Agents will use the following procedure when performing a binary task:

- The agent is initially assumed to be able to complete the task correctly.
- Loop over all required knowledge bits. For each bit:
  - If the agent's knowledge bit matches the value in the binary task truth network, the bit is ignored and the next bit should be examined. Such an event will occur, for instance if the truth network has a value of 1 and the agent's knowledge is non-zero, or if the truth network has a value of 0 and the agent's knowledge is also zero. In either case, the agent's accuracy is unchanged.
  - If the agent's knowledge bit does not match the value in the binary task truth network, it guesses with 50% probability. If it guesses correctly, the next bit is examined. If it guesses incorrectly, the agent is reported to complete the task inaccurately.
- Regardless of the whether or not the agent is accurate or inaccurate based it's knowledge and the truth, the agent has a certain chance to "misrepresent" its decision and report the wrong result. This effectively inverts the agent's decision that occurs based on knowledge and guesswork. If the agent would have completed the task accurately, the agent reports an inaccurate decision with a probability equal to that of

the misrepresentation rate. If the agent would have completed the task inaccurately, the agent reports an accurate decision with the same probability.

If there is no misrepresentation, this algorithm guarantees that agents will guess (with a fixed 50% probability) on all required that do not match the truth. Thus, there will be an increasingly unlikely probability of agents being accurate if they do not correspond with the truth values. As agents will guess independently on different time periods, it is possible for an agent to be accurate on a task during one time period but not accurate on a task during a future time period.

For most binary tasks, experimenters will probably want agents to know particular pieces of knowledge in order to complete the task. Thus, most of the required knowledge bits should have true values in the `binary task truth network`. However, the experimenter may wish to create a scenario where the knowledge of some facts actually impedes the performance of a particular task. For such tasks, the relevant value in the binary task truth network could be set to zero. Agents who learn that particular knowledge bit will then have to guess when performing the task, thus decreasing their accuracy. Such a setup can help with certain types of tasks in organizational settings. However, users can also prevent agents from learning particular knowledge facts by modifying the `agent learning rate network` (Section 3.5.8), should such decreases in accuracy be desired.

As of Construct version 3.8.build016BRH, the 50% probability of accurate guessing is an internal constant and cannot be modified by the user.

Note that if two agents are assigned to the same binary task, they will become more similar. First, agents will be more similar if they perform the same binary tasks, as Construct considers two agents to be more similar to each other if they are performing the same task. The importance of this similarity relative to that of other types of similarity can be changed by modifying the `binarytask similarity weight network` (Section 3.5.23). Note that in version 3.8.build016BRH of Construct, agents will always correctly perceive what tasks others are assigned to; there is no transactive memory for task assignment. Second, agents performing the same task will likely need similar knowledge, and thus will likely lead to greater knowledge similarity among these agents. As agents can gain knowledge from performing tasks (as set in the `learn by doing rate network`, Section 3.5.10), agents will further increase their similarity as both learn knowledge relevant to the task. However, when gaining knowledge via the learning by doing mechanism, agents will not gain transactive memory about others performing the same task.

During the simulation, an agent's success in performing a binary task can be displayed using the `ReadBinaryTaskAccuracy` operation, which will output an agent-by-binarytask network that records each agent's performance of a each task. If an agent cannot complete a task, because it is not assigned to it, the task accuracy will be zero (inaccurate). If agents can complete the task, the accuracy will be one (if accurate) and zero (if inaccurate). Note that it will be necessary in order to compare the binary task accuracy output to the assignment network in order to determine whether a zero output is due to an agent being inaccurate on a task or unable to complete it because it was not assigned to it.

In the demo input deck, `binarytask_count` binary tasks are modeled. In the construct variables section, however, this value has been set to zero. Thus, no binary tasks are modeled in the demo input deck, and all networks involving beliefs will be ignored by the simulation. Note, however, that the binary task nodeclass is a required nodeclass, and thus cannot be omitted from the input deck.

```
<nodeclass type="binarytask" id="binarytask">
  <properties>
    <property name="generate_nodeclass" value="true"/>
    <property name="generator_type" value="count"/>
    <property name="generator_count" value=
                        "construct::intvar::binarytask_count"/>
  </properties>
</nodeclass>
```

### 3.4.5 Energy task nodeclass

Energy tasks in Construct are actions that agents can perform that require a certain amount of effort, as opposed to knowledge. Each node in the energy task nodeclass represents a particular energy task type that agents may be able to perform. Depending on the amount of energy that an agent has, the task is attempted each time period, and possibly completed when enough energy has been expended on the task. Note that energy tasks are independent both of binary tasks and knowledge; it is run using a completely separate subsystem. While energy tasks are not modeled in this demo – and it is worth noting that energy tasks have not been used recently in Construct experiments – it is useful to describe the key features of the energy task in case users decide to use them in future explorations.

Agents who are eligible to perform a energy task (which is set in the `energy task assignment network`, Section 3.5.35) will then expend energy on the task until they meet the required amount of energy necessary to complete that task (set in the `energy task requirement network`, Section 3.5.35) as long an energy task instance has been created for that agent (set in the `energy task time network`, Section 3.5.35). Instances are agent-specific copies of the energy task which must be completed individually, and are used internally in the simulation in order to allow different agents to work on the same "type" of energy task. If energy tasks are present in the simulation, agents can be assigned to one or more energy tasks to perform, and instances of the task will spawn for each.

The total amount of energy each agent has to expend on energy tasks is fixed at one "unit". In each time period, agents will attempt to perform each task, which can consume some or all of the agent's energy each time period. Agents complete the task instance successfully once they have expended sufficient energy.

Agents will use the following procedure when performing an energy task:

- If an energy task instance is to be created during this time period, as set by the `energy task time network`, an energy task is created for this agent. Only this agent will be able to work on this instance of the energy task.
- The total number of active energy task instances is computed for this agent. New energy task instances are summed with old (previously uncompleted) energy task instances in order to determine the total number of instances to which the agent can devote energy.
- For each uncompleted energy task instance that the agent has available:
  - The amount of total energy devoted to this task is equal to the reciprocal of the total number of tasks. Thus, if an agent has two tasks remaining, the amount of energy added to each task is equal to ½.

    - If an agent has spent the requisite amount of energy on the task, as specified in the `energy task requirement network`, the task is marked as completed and the agent will not spend additional time working on the task.
- At the end of the time period, any energy not spent on energy tasks is wasted and cannot be "saved" for future time periods if all tasks have been completed.

Unlike binary tasks, there is no inherent measurement of "accuracy" with energy tasks. Instead, energy task accuracy is considered to be a measurement of the total number of energy tasks completed at the end of the simulation. As the number of completed tasks is marked by the simulation and the energy tasks started can be determined by the `energy task time network` for each agent, the overall accuracy on energy tasks is simply the this fraction.

As of Construct version 3.8.build016BRH, there are several unmodifiable features about the energy task implementation. First, energy tasks must all completed concurrently, with agents dividing their attention equally among multiple tasks. This may lead agents to be unable to complete energy tasks (and thus have a low energy task accuracy) because they cannot focus their attention to complete only one energy task at a time. Secondly, energy task instances are spawned for all agents at the time period specified in the `energy task time network`. There is no way to specify that some energy tasks should be spawned for some agents and not for others. Third, there is no way of specifying that some agents are able to expend more energy that others, a feature which might allow some agents to be more productive than others. Future modifications to the energy task algorithm might include such improvements, as well as tie the energy task system more closely with Construct's knowledge diffusion mechanisms.

In the demo input deck, `energytask_count` energy tasks are modeled. In the construct variables section, however, this value has been set to zero. Thus, no energy tasks are modeled in the demo input deck, and all networks involving beliefs will be ignored by the simulation. Note, however, that the energy task nodeclass is a required nodeclass, and thus cannot be omitted from the input deck.

```
<nodeclass type="energytask" id="energytask">
  <properties>
     <property name="generate_nodeclass" value="true"/>
     <property name="generator_type" value="count"/>
     <property name="generator_count" value=
                    "construct::intvar::energytask_count"/>
   </properties>
</nodeclass>
```

### 3.4.6 Time period nodeclass

Each node in the time period nodeclass represents one simulated time period in the simulation. Thus, the number of nodes in this nodeclass represents the length of the simulation, and individual nodes in this nodeclass represent each time period. Networks involving time periods, such as the `agent active time period network` (Section 3.5.16) determine properties that vary during the simulation. For instance, values in the `agent active time period network` can change by time period, allowing the experiment designer to specify when certain agents are to be active or inactive.

Users should be aware that some Construct models (Section 3.3.9) may treat time period zero – the first time period – as a special time period. Some models may run special setup routines in time period zero and may potentially employ slightly modified algorithms due to the lack of a previous time period from which to calculate a change. For such reasons, it is preferable to run simulations for a relatively large number of time periods (i.e. fifty or more) in order to minimize the effects of such edge conditions.

In the demo input deck, the number of time periods is specified by the variable `time_count`, which is set as a Construct variable. As described in Section 3.2, the value of the `time_count` variable, however, is determined by another variable, `short_experiment`. If the user is running a short experiment, fifty time periods will be simulated. If the user chooses to run a longer experiment, as is suggested as a follow-up experiment in Section 4.8.3, the experiment length will increase to one hundred time periods.

```
<nodeclass type="timeperiod" id="timeperiod">
   <properties>
      <property name="generate_nodeclass" value="true"/>
      <property name="generator_type" value="count"/>
      <property name="generator_count" value=
                     "construct::intvar::time_count"/>
   </properties>
</nodeclass>
```

### 3.4.7 Agent group nodeclass

The agent group nodeclass serves as a bookkeeping nodeclass for collections of similar agents. Construct is able to calculate a number of metrics over these node classes and display them as a simulation result. For instance, Construct is capable of determining how many agents have learned a particular fact in an agent group. This value can be printed out as a summary statistic. Such summaries may be much more useful than full matrices, especially with very large numbers of agents in the simulation.

The agent group nodeclass must be present in the input deck, and at least one agent group must be present. If all agents can be treated identically, use one agent group node in this nodeclass and assign all agents to the same agent group via the `agent group membership network` (Section 3.5.33).

In the demo input deck, there are three agent groups, A, B, and C as described in Section 2. Thus the `agentgroup_count` variable is set to 3.

```
<nodeclass type="agentgroup" id="agentgroup">
   <properties>
      <property name="generate_nodeclass" value="true"/>
      <property name="generator_type" value="count"/>
      <property name="generator_count" value=
                     "construct::intvar::agentgroup_count"/>
   </properties>
</nodeclass>
```

### 3.4.8 Knowledge group nodeclass

The knowledge group nodeclass serves as a bookkeeping nodeclass for collections of similar knowledge bits. Construct is able to calculate a number of metrics over these node classes and display them as a simulation result. For instance, Construct is capable of determining how many knowledge bits have been learned by the agents in an agent group. This value can be printed out as a summary statistic. Such summaries may be much more useful than full matrices, especially with very large numbers of knowledge bits in the simulation.

The knowledge group nodeclass must be present in the input deck, and at least one agent group must be present. If all knowledge bits can be treated identically, use one knowledge group node in this nodeclass and assign all knowledge bits to the same knowledge group via the `knowledge group membership network` (Section3.5.34).

In the demo input deck, there are two knowledge groups, K1 and K2 as described in Section 2. Thus the `knowledgegroup_count` variable is set to 2.

```
<nodeclass type="knowledgegroup" id="knowledgegroup">
  <properties>
     <property name="generate_nodeclass" value="true"/>
     <property name="generator_type" value="count"/>
     <property name="generator_count" value=
                    "construct::intvar::knowledgegroup_count"/>
  </properties>
</nodeclass>
```

### 3.4.9 Dummy nodeclass

The dummy nodeclass is a placeholder nodeclass that is designed to create column vectors for other nodeclasses. The dummy nodeclass is a special nodeclass, and should in no circumstance be changed when designing an input deck. Nevertheless, it is an essential nodeclass. Thus, it must be present in every input deck for Construct to function properly.

Construct works by manipulating two-dimensional input and internal networks, as described in Section 3.5. By using the `dummy_nodeclass`, it is possible to create an `agent` by `dummy_nodeclass` network that effectively acts as a one-dimensional network. One example of this occurs in the `beInfluenced network` that specifies a property that varies by agent (in this case, the agent's susceptibility to influence as described in Section 3.5.5) but is not designed to vary by belief or by time period. Two dimensional networks, and their one-dimensional variants made possible via the `dummy_nodeclass`, are designed to be easily visualized in network analysis tools such as *ORA, and indeed the syntax used to specify them is designed to be maximally compatible with *ORA's DynetML input format (Carley & Reminga 2004). Indeed, one of the goals of the Construct input deck format is to make Construct input easily to visualize and manipulate using *ORA. While this goal has not yet been realized, the `dummy_nodeclass` syntax represents one step along the way to making this possible.

The dummy nodeclass in the demo input deck is described below.

```
<nodeclass type="dummy_nodeclass" id="dummy_nodeclass">
  <node id="constant" title="constant"/>
</nodeclass>
```

### 3.4.10 Agent type

The node-types specify the various kinds of agent nodes in the simulation, at minimum the default actor type must be specified (defined previously in Section 3.3 as `default_agent_type`).

The actor-type has the following attributes:

- communicationMechanism. This attribute determines how the agent communicates. As of Construct version 3.8.build016BRH, there are two types of communication: `direct` and `mail`. Direct communication occurs when agents exchange messages during the current time period, and is the default style of communication. Mail communication occurs only when the mail subsystem is active (see Section 3.3.6), and will allow messages to be placed in a mailbox for delayed communication. In the demo input deck, this parameter value is set to `direct` as human agents are assumed to interact face-to-face and exchange messages immediately.
- canSendCommunication. This attribute determines whether nodes of this type are able to send a message to other nodes when communicating. Most agent types should be able to send communication, regardless of whether or not they are human or not. Agents who cannot send messages when communicating will be unable to influence the knowledge of their interaction partners. In the demo input deck, agents are able to send messages to each other when communicating, hence this parameter value is set to `true`.
- canReceiveCommunication. This attribute determines whether nodes of this type are able to receive messages sent by other nodes. Agents who are not able to receive communication are thus unable to learn any new knowledge or update beliefs and transactive memory. In the demo input deck, agents are able to receive communication from each other, hence this parameter is set to `true`.
- canSendKnowlege. This attribute determines whether nodes of this type are able to send knowledge when communicating. If agents are able to send communication, the content of the message sent can depend on whether agents of that type can send knowledge. Agents who cannot send knowledge will not examine their knowledge when choosing message content to communicate, and thus will not include knowledge in their messages. In the demo input deck, agents are able to send knowledge to each other, hence this parameter is set to `true`.
- canReceiveKnowledge. This attribute determines whether nodes of this type are able to receive knowledge from other agents. If agents are able to receive communication, the content of the message received can depend on the sender's knowledge. Agents who cannot receive knowledge will ignore all knowledge sent to them. In the demo input deck, agents are able to send knowledge to each other, hence this parameter is set to `true`.
- canSendBeliefs. This attribute determines whether nodes of this type are able to send beliefs when communicating. If agents are able to send communication, the content of the message sent can depend on whether agents of that type can send belief. Agents who cannot send beliefs will not examine their beliefs when choosing message content to communicate, and thus will not include beliefs in their messages. In the demo input deck, agents are able to send beliefs to each other, hence this

parameter is set to `true`; however, as no beliefs are modeled, this will have no effect on the demo simulation.

- canReceiveBeliefs. This attribute determines whether nodes of this type are able to receive belief from other agents. If agents are able to receive communication, the content of the message received can depend on the sender's belief. Agents who cannot receive belief will ignore all beliefs sent to them. In the demo input deck, agents are able to send beliefs to each other, hence this parameter is set to `true`; however, as no beliefs are modeled, this will have no effect on the demo simulation.
- canSendBeliefsTM. This attribute determines whether nodes of this type are able to send transactive memory about third party beliefs when communicating. If agents are able to send communication, the content of the message sent can depend on whether agents of that type can send belief TM. Agents who cannot send belief TM will not examine their perceptions of third party beliefs when communicating, and thus will not include belief TM in their messages. In the demo input deck, agents are able to send belief TM to each other, hence this parameter is set to `true`; however, as no beliefs are modeled, this will have no effect on the demo simulation.
- canReceiveBeliefsTM. This attribute determines whether nodes of this type are able to receive belief transactive memory from other agents. If agents are able to receive communication, the content of the message received can depend on the sender's transactive memory on third party beliefs. Agents who cannot receive belief TM will ignore all belief TM information sent to them. In the demo input deck, agents are able to send belief TM to each other, hence this parameter is set to `true`; however, as no beliefs are modeled, this will have no effect on the demo simulation.
- canSendKnowledgeTM. This attribute determines whether nodes of this type are able to send transactive memory about third party knowledge when communicating. If agents are able to send communication, the content of the message sent can depend on whether agents of that type can send knowledge TM. Agents who cannot send knowledge TM will not examine third party knowledge when communicating, and thus will not include knowledge TM in their messages. In the demo input deck, agents are able to send knowledge TM to each other, hence this parameter is set to `true`.
- canReceiveKnowledgeTM. This attribute determines whether nodes of this type are able to receive knowledge transactive memory from other agents. If agents are able to receive communication, the content of the message received can depend on the sender's transactive memory on third party knowledge. Agents who cannot receive knowledge TM will ignore all knowledge TM information sent to them. In the demo input deck, agents are able to send knowledge TM to each other, hence this parameter is set to `true`.
- canSendReferral. This attribute determines if agents can send referrals to other agents in addition to knowledge and belief. If the receiver is seeking a specific piece of information and the sender has transactive memory about a third party that has that information, the sender can include a referral in its message. Thus, agents that can send referrals can suggest that agents should interact with a specified third agent who has particular knowledge expertise. The default value for the demo input deck is set to `true`.

- canReceiveReferral. This attribute determines if agents can receive referrals from agents in addition to knowledge and belief. If the receiver is seeking a specific piece of information and the sender has transactive memory about a third party that has that information, the sender can include a referral in its message. Agents that can receive referrals will be able to take such information and thus be more likely to interact with the identified third party. The default value for the demo input deck is set to `true`.

Note that there are several additional properties that can be specified, though such properties should only be modified by an experienced user. Future versions of Construct may include new properties that may augment the list provided in this section.

The syntax for the `human` agent type node used in the demo input deck is the following.

```
<nodeclass>
 <node id="human" title="human">
   <properties>
      <property name="canSendCommunication" value="true"/>
      <property name="canReceiveCommunication" value="true"/>
      <property name="canSendKnowledge" value="true"/>
      <property name="canReceiveKnowledge" value="true"/>
      <property name="canSendBeliefs" value="true"/>
      <property name="canReceiveBeliefs" value="true"/>
      <property name="canSendBeliefsTM" value="true"/>
      <property name="canReceiveBeliefsTM" value="true"/>
      <property name="canSendKnowledgeTM" value="true"/>
      <property name="canReceiveKnowledgeTM" value="true"/>
      <property name="canSendReferral" value="true"/>
      <property name="canReceiveReferral" value="true"/>
   </properties>
 </node>
</nodeclass>
```

### 3.4.11 Other nodeclasses

The nodeclasses that can be used in the Construct simulation are not defined to the ten nodeclasses defined earlier in this section and included in the demo input deck. While these ten nodeclasses are the nodeclasses required for the active models currently in use (Section 3.3.9), it is important to note that the user is free to define other nodeclasses as well. In conjunction with the scripting system or new models, such nodeclasses can be used for a variety of purposes

User defined custom nodeclasses must follow the syntax described in Section 3.4 and contain names with only alphanumeric characters. Nodeclass names must be unique within the input deck; Construct's behavior is undefined if there are multiple definition of nodeclasses with the same name and will likely cause the simulation to crash.

Users may find custom nodeclasses can be appealing, as users may wish to supply additional network data to Construct. For example, one may wish to specify resources with which certain agents are associated. To supply such information to the simulation, one would need to add a resource nodeclass and then create a custom network which links the agent nodeclass to the resource nodeclass. However, while the nodeclass and accompanying networks can be initialized by Construct, the simulation's behavior will be unchanged. The presence of new nodeclasses and networks will be ignored by the simulation algorithms. While the Construct scripting system can be used in order to modify agent behavior in order to take into account a new nodeclass and new set of relationship, such scripting is only recommended for advanced users of Construct and is largely beyond the scope of this technical report (Hirshman et al 2009).

Nevertheless, models other than the standard interaction, influence, and belief models may require custom nodeclasses to be present in order to function successfully. For instance, the Near Term Analysis tool in *ORA will define an `isolation` nodeclass in order to remove key agents from the simulation at a specific time period. Eventually, future versions of Construct will allow users to write their own models for Construct; such a feature, however, will require the model writer to specify the functionality of the model as well as what kind of nodeclasses are required. Thus, while the nodeclasses described in this demo input deck are indicative of the types of nodeclasses required by Construct, future versions of the tool may allow the user to use new nodeclasses to further improve the quality of the simulation input.

## 3.5 Networks

Networks are the core data structures in Construct. As Construct is a network-based simulation, most of the inputs to Construct are provided in the form of networks. Networks relate the nodeclasses defined in Section 3.4, and can provide both inputs and scratchpads for the simulation to manipulate.

As introduced in Section 3.4, Construct uses the idea of "nodes" and "networks" in contrast to other agent-based simulation systems. Nodes are grouped together into nodeclasses, which can be seen on the top and left of Figure 4. Networks are relationships between different types of nodes. Links can serve multiple purposes, depending on the type of network. Some networks are static (unchanging) during the simulation, and thus serve as ways to provide input to the simulation. Examples of such networks include the `agent type network`, which assigns an agent type to an agent for the duration of the simulation. Other networks are more dynamic, and are manipulated when Construct is running. New links in the network can be added or modified: for instance, if the agent learns knowledge, a new link between the specific agent node and the relevant knowledge node can be created. Thus, as a Construct simulation runs, the relationship among different nodes will be modified. Examples of this type of network include the `knowledge network`, where new links are formed as agents learn knowledge and are removed if agents forget what they have learned.

Networks can relate most types of nodeclasses, and many pairs of nodeclasses can be related by networks. As can be seen in Figure 4 (page 16), there are networks that relate agents and agents, agents and knowledge, agents and belief, knowledge and belief, and a variety of other nodeclasses. The algorithms in Construct reference these networks in order to perform specific tasks. For instance, the agent by knowledge `knowledge network` performs the specific task of representing what knowledge is known by what agents. Note, however, the networks described in Figure 4 are not the only networks linking the nodeclasses; while the `knowledge network` is an agent by knowledge network, so is the `interaction knowledge weight network` and the `transmission knowledge weight network`. A list of all the networks included in the demo input deck, including the nodeclasses that are related by the networks, is provided in Figure 7.

**Figure 7: Key networks in the demo input deck**

| Network Name | Source & Target Nodeclasses | Function or Purpose in Demo Input Deck |
|---|---|---|
| **agent type name** | agent x dummy | specifies the agent type for each agent, thereby identifying key behavior |
| **agent initiation count** | agent x timeperiod | number of times agent can seek a partner, actively initiating communication |
| **agent reception count** | agent x timeperiod | number of times agent can be sought out, passively receiving communication |
| **agent message complexity** | agent x timeperiod | amount of info an agent can send when communicating |
| **beInfluenced** | agent x dummy | how resistant an agent is to changing its belief |
| **influentialness** | agent x dummy | how strongly an agent can influence the beliefs of others |
| **agent selective attention effect** | agent x dummy | percentage of agent knowledge that an agent will examine when communicating |
| **agent learning rate** | agent x knowledge | how quickly an agent will learn new knowledge when communicating |
| **agent forgetting rate** | agent x knowledge | how quickly an agent will forget old knowledge |
| **agent learn by doing rate** | agent x dummy | how quickly an agent will learn new knowledge when performing tasks |
| **knowledge** | agent x knowledge | the knowledge associated with an agent, i.e. what an agent currently knows |
| **agent belief** | agent x belief | the beliefs associated with an agent, i.e. what an agent currently believes |
| **belief knowledge weight** | knowledge x belief | the impact that each knowledge bit has on belief |
| **interaction sphere** | agent x agent | which agents are able to potentially interact with, and keep TM, of which |
| **access** | agent x agent | which agents have access to which (supplement to interaction sphere) |
| **agent active timeperiod** | agent x timeperiod | which agents are active during which timeperiods |
| **physical proximity** | agent x agent | how close each pair of agents are physically |
| **sociodemographic proximity** | agent x agent | how close each pair of agents are socio-demographically |
| **social proximity** | agent x agent | how close each pair of agents are socially |

**Figure 7: Key networks in the demo input deck, continued**

| Network Name | Source & Target Nodeclasses | Function or Purpose in Demo Input Deck |
|---|---|---|
| **physical proximity weight** | agent x timeperiod | weight placed on physical proximity when choosing interaction partner |
| **sociodemographic proximity weight** | agent x timeperiod | weight placed on s-d proximity when choosing interaction partner |
| **social proximity weight** | agent x timeperiod | weight placed on social proximity when choosing interaction partner |
| **binarytask similarity weight** | agent x timeperiod | weight placed on shared binary tasks when choosing interaction partner |
| **binarytask assignment** | agent x binarytask | which agents are assigned to which binary tasks |
| **binarytask requirement** | knowledge x binarytask | which knowledge bits are required to complete which binary tasks |
| **binarytask truth** | knowledge x binarytask | what values knowledge bits must have to complete which binary tasks |
| **knowledge similarity weight** | agent x timeperiod | weight placed on shared knowledge when choosing interaction partner |
| **knowledge expertise weight** | agent x timeperiod | weight placed on different knowledge when choosing interaction partner |
| **interaction knowledge weight** | agent x knowledge | weight placed on knowledge bits when choosing interaction partner |
| **transmission knowledge weight** | agent x knowledge | weight placed on knowledge bits when sending a message |
| **knowledge priority** | agent x knowledge | priorities placed on knowledge bits when sending a message |
| **learnable knowledge** | agent x knowledge | what knowledge bits can or cannot be ever be learned |
| **agent group membership** | agent x agentgroup | what agents are associated with which agent groups |
| **knowledge group membership** | knowledge x knowledgegroup | what knowledge bits are associated with which knowledge groups |

While it is possible for the user to create new networks that relate different types of nodeclasses – for instance, a network relating belief and the dummy_nodeclass – or to create new networks that are not listed in Figure 7, such networks are not currently referenced by the Construct executable. Thus, such networks would have no effect on the evolution of the simulation. Though new networks can potentially be used by new models or by the scripting system available in Construct, such features should only be employed by advanced users.

Networks in Construct are specified within a `<networks>` ConstructML tag in the input deck.

```
<networks>
```

```
  <!-- Put all networks here -->
</networks>
```

The `<networks>` tag contains the definition for a variety of networks that play critical roles in the simulation. Each networks tag must have five attributes: the network id (name), source nodeclass, target nodeclass, link type, and network type, as seen below.

```
<network src_nodeclass_type="[nodeclass]"
         target_nodeclass_type="[nodeclass]"
         id="[name]" link_type="[type]" network_type="dense">
  <!-- Set links and generators here -->
</network>
```

The network ID `[name]` is the name that refers to the network. Models in Construct (Section 3.3.9) each require certain kinds of networks in order to function. The networks that the model calls for have a specific role in the simulation. Thus, the network id defines what the `src_nodeclass_type`, `target_nodeclass_type`, `link_type`, and `network_type` should be in order for Construct to correctly understand the input provided. If the network ID is misspelled, Construct will not correctly interpret the input and will probably exit with an error (it will believe that the network is missing). Note that network IDs are case-insensitive – internally, all network names are converted to lower case – and can contain spaces.

The src_nodeclass_type and target_nodeclass_type `[nodeclass]` indicate the nodeclasses that are related by the network. Networks in Construct can be thought of as a set of weighted or unweighted relations between two nodeclasses. For instance, an agent by knowledge network such as the `knowledge network` (Section 3.5.11) is a directed network relating each node in the agent nodeclass to each node in the knowledge nodeclass. Relationships with weight zero are considered to be absent relationships, so a weight of zero on a link between an agent and a knowledge bit can be interpreted as an agent being ignorant of the particular knowledge bit. Similarly, the agent by timeperiod `agent active timeperiod network` (Section 3.5.16) relates agents to timeperiods, with a nonzero value indicating that an agent is active and a zero value indicating that the agent will not be active during the simulation. Note that in these examples, as in all networks, the source and target nodeclasses cannot be swapped; the links are directional from the source nodeclass to the target nodeclass. Examples of other types of networks, and their associated source (row) and target (column) nodeclasses can be found in Figure 4 on page 16.

The two other key network parameters are the `network_type` and `link_type`. The network_type specifies the storage mechanism used internally to represent this network. For most input decks, including the networks in this input deck, this should be `dense` storage. The link_type `type` defines the type of relation stored in this network. As of Construct version 3.8.build016BRH, there are four possible link types:

- bool. Bool links are the Boolean values `true` or false (or `1` or `0` when represented numerically). Either representation can be used when specifying links in the network.
- int. Integer links are whole numbers $-2^{31}$ to $2^{31}-1$. Integer values in networks can be positive, negative, or zero, though integers in certain types of networks (notably counts and priority values) should be nonnegative.

- float. Float links are IEEE 754 single-precision floating point values. Note that Construct only allows single-precision numbers (32 bit representations).
- string. String links can be any alphanumeric sequence of characters. Such links are usually found when relating one set of nodes to the dummy nodeclass, allowing the network to effectively act as a sequence of attributes for the nodeclass.

It is important to note that the link type must be the same for all links in the network. Thus, it is not possible to have floating point relationships in integer networks. However, the values taken by each link can vary.

There two ways to specify the value for a link in a network. The first way is to specify the value of a single link using the `<link>` tag. The second way is to specify a region of a network that should have links according to a pattern using the `<generator>` tag. Both of these will be discussed in turn.

The `<link>` method of link generation will generate a single link between a source node and a target node. The syntax for this generation mechanism is below.

```
<link src_node_name="[id]" target_node_name="[id]"
        value="[value]"/>
```

The link tag must have three attributes. The src_node_name `[id]` is the name of the source node in the source nodeclass, while the target_node_name `[id]` is the name of the target node. The name of the node depends on the way that it was generated in Section 3.4. If the node was generated via the generate_nodeclass syntax, which can generate multiple nodes, the name of the node will follow the pattern `[nodeclass name][index]`. Thus, the name of the third agent node will be `agent2` (since Construct nodes will be zero indexed) while that of the first time period node will be `timeperiod0`. If the node was generated using the node syntax, which generates one node at a time, the name of the node will be equal to the user-specified node id. If either the source or target node cannot be found in their respective nodeclasses, Construct will error and exit.

Note that the node value will be silently converted to an appropriate type for the network in use. Thus, if the network being generated is a string network, the link value will be interpreted as a string. If the network being generated is an int network and a float value is provided, the float will be truncated to an int as described in the Construct scripting system (Hirshman et al 2009). Users should verify that their link values are appropriate for the type of network that they are attempting to create.

The second way to add edges to networks is the `<generator>` mechanism. Adding links using generators has a number of advantages over the single-link `<link>` mechanism.

- more nuanced. A generator allows the experiment designer to create specific patterns within a network. While generators are most often used to set parts of networks to have constant values, generators are capable of much more. For instance, if a user wants to build a scale-free network, a specific generator is available to do so. Alternately, if the user wants to fill a network with random numbers from a normal or uniform distribution, generators can also provide such functionality. Lastly, generators can load in external csv files in order to read in more complicated patterns.

In contrast, links have static start and end nodes and will be unable to change values rapidly.

- flexible. Using a generator and appropriate construct variables, one does not have to use nodes with specific names. The `<link>` mechanism, on the other hand, adds a specific link from a named source to a named target. If either the source or target node is removed by shrinking the number of nodes in the source or target nodeclass, the edge will have to be manually deleted. Input deck designers will also have to remember to delete the edge, otherwise Construct will error when the source or target node is not found at simulation startup.
- able to introduce randomness. Many generators have a random element built into them. For instance, generators can set network values from uniform or normal distributions. If the random seed changes, the exact values generated will changed. This allows for a more complete exploration of the input space. The `<link>` mechanism, however, does not permit this random element.
- compact. Using a generator allows the experiment designer to slim down the size of the input deck. With large networks of moderate density, it often becomes necessary to use tens or hundreds of `<link>` tags to build up the network. This can greatly increase the input deck size. When running multiple replications of an input deck – which can require tens or hundreds of copies of the input file in separate directories – such costs may be non-trivial.
- faster. While network loading time is usually a minimal portion of overall simulation run-time, generating multiple links using a generator is significantly faster than generating individual links using a series of `<link>` tags.

The syntax for a prototypical generator is described below.

```
<generator type="[type]">
  <rows first="[firstrow]" last="[lastrow]"/>
  <cols first="[firstcol]" last="[lastcol]"/>
  <param name="[name1]" value="[value1]"/>
  <param name="[name2]" value="[value2]"/>
  ...
</generator>
```

Like the `<link>` tag, the `<generator>` tag must be placed inside the network tag. Unlike the link tag, the generator tag has only one attribute, the `type`. The type `[type]` must be a valid type of generator, examples of which can be found in the networks defined in the demo input deck as well as technical report CMU-ISR-07-116 (Hirshman & Carley 2007b). Child tags of the generator tag will indicate the first and last row (both inclusive) and first and last column (both inclusive) on which the generator should operate. Child tags will also indicate one or more parameters that are required for the generator. Note that many attributes in CMU-ISR-07-116 have been changed to child tags in the current version, meaning that most parameters to the generator are provided as child tags of `<generator>`.

Because there can be multiple types of links and generators inside a network, it is possible to have a particular link value between two nodes specified zero, one, or multiple times. For this reason, tags inside the network tag are read one after the other, starting at the top of the

`<network>` tag and continuing until the terminal `</network>` tag. The value of the node-node link will be the value specified by the last link or generator read. If the value is specified multiple times in other `<link>` or `<generator>` tags, the earlier link or generator values are overwritten and replaced with the later version. On the other hand, if the value is not specified in any `<link>` or `<generator>` tag, then the value will default to `false` in bool networks, `0` in int or float networks, and the empty string `“”` in string networks. This overwriting mechanism is further described in CMU-ISR-07-116 (Hirshman & Carley 2007b).

The remainder of this section introduces the key networks in the demo input deck, in the order presented in Figure 7. Additional detail on each type of network is provided, and the relevant values for the demo input deck are described. While the networks themselves can be initialized in any order, it is suggested that the order presented in the demo input deck be followed in subsequent Construct input decks.

### 3.5.1 Agent Type

The `agent type name network` indicates which agents are of which types. Agent nodes are specified in the agent node class, but do not have a specific type – i.e., their role in the simulation is not yet defined. Agents in Construct can be human or nonhuman, able to initiate communication or have to wait for agents to interact with them, able to learn new knowledge or static in nature, and have many other properties. However, in order to determine which properties an agent has, it must be associated with an agent type. This association is performed as a direct result of this network. Assigning an agent to an agent type will forbid or allow the agent type behaviors outlined in Section 3.4.10 including:

- <u>communication mode</u>: whether the agent uses direct or mail communication.
- <u>what can be communicated</u>: whether the agent can send or receive messages from other agents when interacting; if it can do so, the agent type also defines whether the agent can send or receive knowledge, beliefs, transactive memory, or referrals when interacting.

The `agent type name network` is an `agent` x `dummy_nodeclass` network of strings, where the string entry for each agent indicates the name of the associated agent type. The name of the associated agent type node must be specified as the link value in the network. If any link value is equal to the name of a node in the `agent_type` nodeclass (Section 3.4.10), Construct will error and exit. Values in this network should not be left blank, or else the agent will be assigned to the default agent type (Section 3.3.4).

The choice to use an `agent` x `dummy_nodeclass` network of strings (as opposed to, say, an `agent` x `agent_type` network of bool values) is a deliberate design decision. Since agents can only have one agent type, and the choice of agent types is mutually exclusive, only one entry in the Boolean network would have been set. For simplicity of use and clarity of assignment, a string network was chosen to represent the agent type. While this setup may clash slightly with the network-centric design of Construct, it is designed to be more intuitive for new users.

In the demo input deck, all agents are of the type `human`, the only agent type defined in Section 3.4.10. Humans use direct communication, and can send and receive all message content in their messages. This parameter is set serially, first for agent group A then agent group B then agent group C.  This setup allows an advanced user to quickly customize the agent type for each group.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="dummy_nodeclass"
        id="agent type name network" link_type="string"
        network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
             last="construct::intvar::agentgroup_A_end"/>
      <cols first="0"
            last="nodeclass::dummy_nodeclass::count-1"/>
      <param name="constant_value" value="human"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
             last="construct::intvar::agentgroup_B_end"/>
      <cols first="0"
            last="nodeclass::dummy_nodeclass::count-1"/>
      <param name="constant_value" value="human"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
             last="construct::intvar::agentgroup_C_end"/>
      <cols first="0"
            last="nodeclass::dummy_nodeclass::count-1"/>
      <param name="constant_value" value="human"/>
   </generator>
</network>
```

### 3.5.2 Agent Initiation Count

The `agent initiation count network` specifies the number of times each agent can actively select interaction partners.  Agent can interact with others in one of two ways.  First, agents can actively initiate communication with others by calculating a probability of interaction with all available partners and choosing a partner according this probability.  Alternately, agents can wait passively until another agent chooses to initiate interaction with them.  The initiation count specifies the number of times the former process can occur; the latter process is specified in the `agent reception count network` (Section 3.5.3).

The `agent initiation count network` is used by the standard interaction model (Section 3.3.9) and specifies the number of interactions that each agent can initiate every timeperiod.  The network is an agent x timeperiod network, meaning that the number of interactions an agent can initiate can vary by timeperiod.  The pseudocode for agent initiation is equivalent to an optimized version of the following.

- At the beginning of each timeperiod, pre-compute the absolute (non-normalized) probability of interaction between all pairs of agents as long as the latter is in the former's interaction `sphere` (Section 3.5.14).
- Examine the relevant column – i.e. the current timeperiod – of the `agent initiation count network` is examined. Each agent that can initiate interaction is added to a vector. If the agent can initiate interaction multiple times, it is added multiple times.
- While there are agent remaining in the vector:
    - Choose an agent at random. Call this the ego agent.
    - Examine all possible communication partners using the ego agent's `interaction sphere`. If the possible partner can still receive communication, keep this agent as a potential interaction partner for the ego.
    - Include the ego agent as a possible interaction partner in order to allow the agent to consider self-interaction. This concept of 'self-interaction' will be discussed at the end of the algorithm.
    - For each of the possible interaction partners of the ego, normalize the pre-computed absolute probabilities of interaction by the total absolute probability. Call this new set of probabilities the relative probability of interaction.
    - Select an interaction partner from the set of potential interaction partners with a probability equal to that of it is relative probability of interaction. If this interaction partner is not the ego agent, decrease the reception count of the interaction partner since the ego and partner have been paired off.
- When there are no more initiating agents, partner selection is considered complete for that time period. Interaction partners will then begin to exchange messages.

If an agent cannot find a suitable interaction partner, it will choose to 'interact with itself' or 'initiate a self-interaction'. This process mimics the notion of withdrawal from the larger interacting community, and allows an agent to decide that it does not want to interact with anyone. This process has two important consequences. First, agents will always consider themselves to be an interaction partner. This means that, when agents are computing their probabilities of interactions with alters, will have to rate their propensity to interact with any other agent against the propensity to initiate a self-interaction. Equally important, since agents will always consider self-interaction, an agent will always have at least one potential interaction partner, even if it is themselves. Second, self-interactions do not count against the agent's reception count. This also means that self-interactions can occur even after they have received the maximum number of communications from other agents. This also means that the reception count for agents does not have to be adjusted to compensate for potential self-interactions.

Note that the initiation count is the maximum number of times an agent can possibly interact during the timeperiod, and does not necessarily mean that an agent will interact with that number of unique alters. Agents who, by luck, can initiate multiple interactions at the beginning of the timeperiod may choose to interact multiple times with the same interaction partner (if that partner can receive communication multiple times and has not been paired off with other imitators). Agents who act at the end of the timeperiod will have fewer interaction partners to choose from since the number of agents who have not yet chosen a partner will be smaller. This may lead them to choose to interact with themselves. Thus, even though an agent may have a large initiation count, it may actually end up interacting with relatively few unique alters.

Increasing an agent’s interaction count will likely increase the number of partners that interacts with, but will not necessarily guarantee it.

It should also be observed that the order in which agents initiate communication each time period is random. A different random order will be chosen during each time period, meaning that agents who choose early during one timeperiod will likely not necessarily choose early in the next.

Values in the `agent initiation count network` must be integers, and should be values of size zero or greater. Agents that cannot initiate interaction in a time period should have a reception count of zero, while agents who should interact should have values one or greater. For practical purposes, the maximum reception count is effectively unlimited; however, users should be aware that the computation necessary per time period is proportional to the number of interactions that occur in that time period.

In the demo input deck described in this technical report, all agents can initiate interaction once per timeperiod. This means that each agent selects only one partner with which to exchange a message. In the input file, this parameter is set serially, first for agent group A then agent group B then agent group C. Such a setup allows an advanced user to quickly customize the initiation count for each group, should such a change be desired.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="timeperiod"
         id="agent initiation count network" link_type="int"
         network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
            last="construct::intvar::agentgroup_A_end"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
            last="construct::intvar::agentgroup_B_end"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
            last="construct::intvar::agentgroup_C_end"/>
       <cols first="0"
             last="nodeclass::timeperiod::count-1"/>
       <param name="constant_value" value="1"/>
   </generator>
</network>
```

### 3.5.3 Agent Reception Count

The `agent reception count network` specifies the number of times each agent can be chosen as an interaction partner by others. Agent can interact with others in one of two ways. First, agents can actively initiate communication with others by calculating a probability of interaction with all available partners and choosing a partner according this probability. Alternately, agents can wait passively until another agent chooses to initiate interaction with them. The reception count specifies the number of times the latter process can occur; the former process is specified in the `agent initiation count network` (Section 3.5.2).

The `agent reception count network` is used by the standard interaction model (Section 3.3.9) and specifies the number of interactions that other agents can initiate with each agent every timeperiod. The network is an agent x timeperiod network, meaning that the number of interactions an agent can receive can vary by timeperiod. The pseudocode for the interaction initiation and reception process was described in Section 3.5.2.

Note that the reception count is the maximum number of times each agent can be selected as an interaction partner each timeperiod, and does not necessarily mean that an agent will interact with that number of unique alters. Some agents may be more sought after than others, perhaps due to the type of knowledge that others perceive that they have, and therefore will be selected as an interaction partner the maximum number of times. Other agents may be largely or completely ignored. The maximum number of times that an agent can receive communication has nothing to do with its popularity; agents who can only receive communication once might be popular (agents with whom others have a high probability of interaction), while agents who can receive communication many times may not be.

Also note that agents can choose to 'interact with themselves' during a timeperiod, and in so doing elect not to choose a different agent as an interaction partner. Such self interactions are not counted against the agent reception count.

Values in the `agent reception count network` must be integers, and should be values of size zero or greater. Agents that cannot receive interaction in a time period should have a reception count of zero, while agents who should interact should have values one or greater. For practical purposes, the maximum reception count is effectively unlimited.

In the demo input deck described in this technical report, all agents can receive interaction once per timeperiod. This means that each agent is selected only by one (non-self) partner with which to exchange a message. In the input file, this parameter is set serially, first for agent group A then agent group B then agent group C. Such a setup allows an advanced user to quickly customize the reception count for each group, should such a change be desired.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="timeperiod"
         id="agent reception count network" link_type="int"
         network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
            last="construct::intvar::agentgroup_A_end"/>
      <cols first="0"
           last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
             last="construct::intvar::agentgroup_B_end"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
             last="construct::intvar::agentgroup_C_end"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
</network>
```

### 3.5.4 Agent Message Complexity

The `agent message complexity network` specifies the number of items an agent can include in its message when communicating with others. Once an agent has selected an interaction partner, it will attempt to send a message to it. The message complexity specifies how many items that message will contain, with longer messages being more complex and shorter messages less so.

The `agent message complexity network` is used by the standard interaction model (Section 3.3.9). It is an agent x timeperiod network, meaning that the length of a message that an agent will communicate can vary by timeperiod. However, within a given timeperiod, all messages will be of the same length. While agents will send different messages to different interaction partners, the total number of items in each message will be the same even if the individual items differ from agent to agent.

The algorithm for message creation is an optimized version of the following.

- As long as the agent's message is less than the message complexity
  - Randomly choose which type of message item to include according to the communication weights specified in the Construct parameters (Section 3.3.7). If the agent cannot send that message item, or there are no items of that type to send, choose again.
  - Select an item of the appropriate type, according to the appropriate weight scheme. Verify that this item has not already been added to the message to send. If it has already been selected, choose again.
- At this point, the message should now have the desired number of items and can be sent to the interaction partner.

Values in the `agent message complexity network` must be integers, and should be values of size one or greater. Values larger than the total number of knowledge bits and beliefs are discouraged.

In the demo input deck described in this technical report, all agents can send a message of length one each time period. This means that each agent will send one knowledge bit or knowledge TM bit to its interaction partner, regardless of whether it is initiating or receiving communication. In the input file, this parameter is set serially, first for agent group A then agent group B then agent group C. Such a setup allows an advanced user to quickly customize the message complexity for each group, should such a change be desired.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="timeperiod”
         id="agent message complexity network" link_type="int"
         network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
            last="construct::intvar::agentgroup_A_end"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
            last="construct::intvar::agentgroup_B_end"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
            last="construct::intvar::agentgroup_C_end"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
   </generator>
</network>
```

### 3.5.5 Susceptibility

The `beInfluenced network` – a legacy name preserved for many iterations of Construct – specifies how susceptible an agent is to influence. As is well known, social influence can affect agent beliefs. The `beInfluenced network` affects how strongly other alter agents can affect an ego’s beliefs. Egos with a high susceptibility to influence will be more likely to change their beliefs to fit the prevailing belief in their interaction sphere, while those with a low susceptibility will be more independent.

The `beInfluenced network` is used by the standard influence model (Section 3.3.9). It is an agent x dummy_nodeclass network, meaning that the susceptibility is a static property of the agent and will not change by belief or over time. It should also be observed that susceptibility to influence is independent of agent knowledge. Agents can be susceptible to influence even if they have a large amount of knowledge, as susceptibility is set as an exogenous property of the agent.

In the standard influence model, an alter's influence on an ego is determined by multiplying the alter's influentialness (Section 3.5.6) by the alter's belief. When such values are averaged over all alters in ego's interaction sphere (Section 3.5.14), this process creates the net social influence on the ego. This net social influence is then multiplied by ego's susceptibility value in order to determine what the net effect of social influence will be on belief. This influence is then used in the belief model in order to determine how belief will change.

Values in the `beInfluenced network` must be floats, and should be values between 0.0 and 1.0. Agents who are impervious to the beliefs of others should have a value of 0.0, while agents who are maximally susceptible to the beliefs of others should have a value of 1.0. Values less than 0.0, or greater than 1.0, will lead to undefined behavior and should be avoided.

In the demo input deck described in this technical report, agents are given a random susceptibility value between 0.0 and 1.0. However, as no beliefs are modeled, this will have no net effect on the evolution of the simulation. Since no beliefs are modeled, this parameter is not set serially; instead, it is set for the entire network using a single generator.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="dummy_nodeclass"
         id="beInfluenced network" link_type="float"
         network_type="dense">
   <generator type="randomuniform">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="0"/>
     <param name="min" value="0.0"/>
     <param name="max" value="1.0"/>
   </generator>
</network>
```

### 3.5.6 Influentialness

The `influencialness network` specifies how strongly an agent can influence others. As is well known, social influence can affect agent beliefs. The `influencialness network` affects how strongly ego agents can affect the beliefs of alters. Egos with high influence values will have a greater effect on the beliefs of others in their interaction sphere, while those with a low susceptibility will have less of an effect.

The `influencialness network` is used by the standard influence model (Section 3.3.9). It is an agent x dummy_nodeclass network, meaning that the influentialness is a static property of the agent and will not change by belief or over time. It is not possible to set influencialness values on a per-agent basis; instead one influencialness value must be used for all agent-agent pairs. It should also be observed that influencialness is independent of agent knowledge. Agents can be influential even if they have a very little knowledge, as susceptibility is set as an exogenous property of the agent.

In the standard influence model, an ego's influence on an alter is determined by multiplying the ego's influentialness value by its belief. When averaged over all agents in alter's interaction sphere (Section 3.5.14), this process creates the net social influence on the alter. This net social influence is then multiplied by the alter's susceptibility to influence (Section 3.5.5) in order to determine what the net effect of social influence will be on belief. This influence is then used in the belief model in order to determine how belief will change.

Values in the `influencialness network` must be floats, and should be values between 0.0 and 1.0. Agents who have no influence on the beliefs of others should have a value of 0.0, while agents who are maximally influential on beliefs of others should have a value of 1.0. Values less than 0.0, or greater than 1.0, will lead to undefined behavior and should be avoided.

In the demo input deck described in this technical report, agents are given a random susceptibility value between 0.0 and 1.0. However, as no beliefs are modeled, this will have no net effect on the evolution of the simulation. Since no beliefs are modeled, this parameter is not set serially; instead, it is set for the entire network using a single generator.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="dummy_nodeclass"
         id="influencialness network" link_type="float"
         network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0" last="0"/>
     <param name="min" value="0.0"/>
     <param name="max" value="1.0"/>
   </generator>
</network>
```

### 3.5.7 Selective Attention Effect

The `agent selective attention effect network` specifies how much of an agent’s knowledge it will examine when deciding what knowledge bit to communicate in a message. Agents in Construct are boundedly rational, and as such may not be able to examine all knowledge bits that they know when deciding what to communicate. The size of the selective attention effect determines how much of an agent’s knowledge it will examine when choosing knowledge bits to communicate. While less focused then the transmission weight placed on knowledge (Section 3.5.30) or knowledge priority (Section 3.5.31), the selective attention rate allows the agent to examine a random subset of its facts and thus send very different types of information in different timeperiods.

The `agent selective attention effect network` is used by the standard interaction model (Section 3.3.9). It is an agent x dummy_nodeclass network, meaning that selective attention rate is a static property of the agent and will not change by belief or over time. While the total number of bits that pass through the selective attention effect filter will be constant, the exact knowledge bits examined may change from timeperiod to timeperiod as a random subset of the agent’s knowledge bits are used. It should also be observed that selective attention is independent of agent knowledge, and indeed of the values in the `transmission knowledge weight network` (Section 3.5.30) and `knowledge priority network` (Section 3.5.31).

Values in the `agent selective attention effect network` must be floats, and should be between 0.0 and 1.0. Values outside this range may lead to undefined behavior. A value of 1.0 indicates that the agent is able to examine all of its knowledge when sending a message, while a value of 0.0 indicates that an agent will ignore all facts. An intermediate value indicates that some knowledge bits will be ignored and others examined when the agent decides what knowledge bits to send; for example, a value of 0.7 indicates that each knowledge bit has a 70% of being considered (put another way, a 30% chance of being ignored). Note that just because a knowledge bit is considered does not mean it will be sent; instead, factors such as the transmission weight and knowledge priority will influence exactly what bits are selected for transmission.

In the demo input deck described in this technical report, agents do not suffer from selective attention and are able to consider all knowledge bits. Thus, the selective attention effect value is set to 1.0, allowing agents to examine all of their knowledge when choosing a fact to send. This network is set using a `randomuniform` generator, allowing a user to manipulate the minimum value in order to examine the effects of selective attention. Since selective attention is not considered an important part of this example, however, this parameter is not set serially; instead, it is set for the entire network using a single generator.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="dummy_nodeclass"
         id="agent selective attention effect network"
         link_type="float"
         network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0" last="0"/>
      <param name="min" value="1.0"/>
      <param name="max" value="1.0"/>
   </generator>
</network>
```

### 3.5.8 Learning Rate

The `agent learning rate network` specifies how quickly agents learn particular pieces of knowledge. Experiment designers may want some agents to learn faster than others, or to have agents learn incompletely when first exposed to certain knowledge bits. The learning rate network allows agents to partially or incompletely learn particular knowledge bits that are sent to them, depending on the learning method in use. While agents can be prevented from learning a particular knowledge bit by using the `learnable knowledge network` (Section 3.5.32), modifying that network will also prevent agents from learning knowledge while performing tasks (i.e. the `learn by doing network`, Section 3.5.10).

The `agent learning rate network` is used by the standard interaction model (Section 3.3.9). It is an agent x knowledge network, meaning that the rate at which agents can learn knowledge bits may vary by agent. However, in the absence of scripting, it is not possible modify this network over time or to change what knowledge bits are learnable once other knowledge bits are learned.

Note that the effect of the learning rate is governed by the `binary_learning` simulation parameter (Section 3.3.5). If binary learning is enabled, then knowledge learning will be binary. Agents will perfectly learn a particular knowledge bit sent to them with some probability, and will not learn anything otherwise. If binary learning is not active, then agents will learn part of what is sent to them. This difference can be important for the type of analysis performed. If binary learning is enabled, knowledge may diffuse more slowly, but agents will always have full knowledge of whatever is sent to them. If binary learning is not enabled, agents may have partial knowledge of a larger number of knowledge bits. Additionally, if binary learning is not enabled, agents will send parts of facts to other agents, which may lead them to learn only parts of parts of facts.

Values in the `agent learning rate network` must be floats, and should be between 0.0 and 1.0. Values outside this range may lead to undefined behavior. A value of 1.0 indicates that the agent is able to learn the knowledge bit perfectly every time it is sent to the agent, while a value of 0.0 indicates that an agent will never be able to learn the bit from another agent. An intermediate value indicates that the agent will either partially learn the knowledge bit or learn the bit by chance, depending on the learning method. If binary learning is enabled, an ego's learning rate of 0.7, for example, will lead the ego to have a 70% chance of learning the bit that an alter sends. This means that 70% of the time the ego agent will learn a value of 1.0 if the alter sends that value, while 30% of the time the ego will not learn at all. If binary learning is not enabled, a learning rate of 0.7 will mean that the ego agent will always learn a value of 0.7 if the alter sends a value of 1.0. Regardless of the learning rate or learning rate mechanism, it is not possible for an agent to increase its knowledge to more than 1.0 via communicating with another agent.

In the demo input deck described in this technical report, agents have a learning rate of 1.0. Thus, all agents learn perfectly during the simulation. This network is set using a `randomuniform` generator, allowing a user to manipulate the minimum value in order to examine what might occur if the learning rate were changed. Since the learning rate is not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per agent group / knowledge group combination. Users should also note that binary learning was enabled for this experiment (Section 3.3.5), so if the learning rate is changed agents will still learn whole knowledge bits, but will do so with a smaller probability.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="knowledge"
         id="agent learning rate network" link_type="float"
         network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0"
            last="nodeclass::knowledge::count-1"/>
       <param name="min" value="1.0"/>
       <param name="max" value="1.0"/>
   </generator>
</network>
```

### 3.5.9 Forgetting Rate

The `agent forgetting rate network` specifies how quickly agents forget knowledge that they have learned. Experiment designers may agents to forget knowledge that they have learned, and to have some agents (or some knowledge bits) forgotten at faster rates. The forgetting rate network allows agents to partially or incompletely learn particular knowledge bits that are sent to them, depending on the forgetting method in use.

The `agent forgetting rate network` is used by the standard interaction model (Section 3.3.9) as long as forgetting is enabled (Section 3.3.5. It is an agent x knowledge network, meaning that the rate at which agents can forget knowledge bits may vary by agent and by knowledge bit. However, in the absence of scripting, it is not possible modify this network over time or to change the rate at which knowledge is forgotten as an agent's knowledge changes.

Note that the effect of the forgetting rate is governed by the `forgetting` and `binary_forgetting` simulation parameter (Section 3.3.5). If forgetting is not enabled, then the values in this network are ignored. If forgetting is enabled and binary forgetting is also enabled, then forgetting will be binary. Agents will forget all knowledge of a particular bit in a particular time period, with a probability equal to that specified in the forgetting rate network. If binary forgetting is not active, then agents will forget a percentage of the knowledge bit every time period, where the percentage forgotten is equal to the value in the binary forgetting network. The differences between these forgetting mechanisms can be important for the type of analysis performed. If binary forgetting is enabled, knowledge may be forgotten more unevenly and intermittently. Additionally, if binary forgetting is not enabled, agents will send parts of knowledge bits to other agents if they partially forget what has been sent to them, a result which may lead to the transmission of partial knowledge bits in the simulation.

Note that if binary forgetting is not enabled, it may be difficult for an agent to completely forget a knowledge bit (have knowledge return to 0.0). Construct's knowledge network uses 32-bit floating point values, meaning that the smallest value that can be represented is on the order of $10^{-100}$. When non-binary forgetting is active, the forgetting rate is multiplied by the agent's knowledge as long as it is above this minimum value. At this time, there is no threshold in place for the non-binary forgetting system, so this method of forgetting is repeated until underflow occurs before a knowledge bit is completely forgotten. Some algorithms may treat agents with any amount of knowledge – even knowledge on the order of $10^{-100}$ – as sufficiently knowledgeable. Thus, for instance, agents with miniscule amounts of knowledge will still use this bit when calculating knowledge similarity (Section 3.5.27) or performing tasks (Section 3.5.26). Until the bit returns to the 0.0 value, the agent will still continue to perform this functions, although sometimes at a diminished level. Note that this problem does not apply to binary forgetting, as binary forgetting will set knowledge to a 0.0 value with the probability specified in the forgetting rate network.

Values in the `agent forgetting rate network` must be floats, and should be between 0.0 and 1.0. Values outside this range may lead to undefined behavior. A value of 1.0 indicates that the agent will lose all knowledge of the bit following every single time period, while a value of 0.0 indicates that an agent will never forget this knowledge bit. An intermediate value indicates that the agent will either partially forget the bit or forget the bit by chance, depending on the learning method. If binary forgetting is enabled, an agent's forgetting rate of 0.7, for example, will lead the agent to have a 70% chance of forgetting the knowledge after every single time period. This means that 70% of the time the ego agent will have its knowledge set to 0.0 for the knowledge bit in question, while 30% of the time the agent's knowledge will be unchanged. If binary forgetting is not enabled, a forgetting rate of 0.7 will mean that the agent will always forget 70% of a knowledge bit's value; if an agent initially started with a value of 1.0, the value would become 0.3 after the first period and 0.09 after the second. Regardless of the forgetting rate or forgetting mechanism used, it is not possible for an agent to decrease its knowledge to less than 0.0.

In the demo input deck described in this technical report, agents have a forgetting rate of 0.0. Thus, agents do not forget knowledge during the simulation, and will retain all knowledge once learned. This network is set using a `randomuniform` generator, allowing a user to manipulate the maximum value in order to examine what might occur if the forgetting rate were changed. Since the forgetting rate is not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per agent group / knowledge group combination. In doing so, users should enable the `forgetting` simulation parameter (Section 3.3.5) and ensure that `binary_forgetting` is used.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="knowledge"
         id="agent forgetting rate network" link_type="float"
         network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0"
            last="nodeclass::knowledge::count-1"/>
      <param name="min" value="0.0"/>
      <param name="max" value="0.0"/>
   </generator>
</network>
```

### 3.5.10 Learn by Doing Rate

The `agent learn by doing network` specifies how quickly agents learn particular pieces of knowledge when performing binary tasks. When agents perform binary tasks, they can increase their knowledge of bits that are relevant to that task. The learning rate network allows agents to learn more about knowledge bits that they partially know. While agents primarily learn information from other agents, as occurs through interaction and the values in the `agent learning rate network` (Section 3.5.8), the learn by doing network can allow agents to hone their knowledge in order to more accurately perform a task.

The `agent learning rate network` is used by the standard interaction model (Section 3.3.9). It is an agent x dummy_nodeclass network, meaning that the rate at which agents can learn knowledge bits may vary by agent but is constant across all knowledge bits and across all tasks. In the absence of scripting, it is not possible modify this network over time or to change the learning by doing rate once other knowledge bits are learned.

When agents learn by doing, they can only learn knowledge bits that are required to complete tasks to which they are assigned. The required knowledge for tasks is assigned using the `binary task requirement network` (Section 3.5.25). Additionally, agents must have at least partial knowledge of the bit before any learning by doing can occur; learning by doing cannot occur in a vacuum. This partial knowledge must come either from initialization or via communication with another agent.

Only certain knowledge bits are affected by the learn by doing mechanism. Knowledge bits that are not required for any task will not be affected by this learning mechanism. Additionally, knowledge bits required for tasks to which an agent is not assigned will not be affected by this mechanism. Knowledge bits that are required for tasks to which an agent is assigned, however, are affected by the learn by doing rate. If the knowledge bit is required to complete multiple tasks, agents will learn the relevant knowledge bits more quickly.

Note that while the learn by doing rate is not directly governed by the `binary_learning` simulation parameter (Section 3.3.5), it will have a greater effect when binary learning is disabled. Agents will only learn by doing if they do not have full knowledge of a bit – i.e. that their knowledge value is greater than 0.0 but less than 1.0. If binary learning is disabled, agents will be more likely to have partial knowledge and thus have the opportunity to learn by doing. If binary learning is active, agents can only learn by doing if the relevant agent is initialized with partial knowledge of a knowledge bit (or learns part of a knowledge bit via an agent who is initialized with partial knowledge).

Values in the `learn by doing network` must be floats, and should be between 0.0 and 1.0. Values outside this range may lead to undefined behavior. A value of 1.0 indicates that the agent is able to learn the remainder of the knowledge bit instantly when attempting to perform the task, while a value of 0.0 indicates that the agent will never increase its knowledge when performing a task. It is not possible for an agent to increase its knowledge to a value greater than 1.0 while learning from a task.

In the demo input deck described in this technical report, agents have a learn by doing rate of 0.0. Thus, agents do not learn knowledge by performing tasks. This network is set using a `randomuniform` generator, allowing a user to manipulate the maximum value in order to examine what might occur if the learning by doing rate were changed. Since the learning rate is not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per agent group / knowledge group combination.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="dummy_nodeclass"
         id="agent learn by doing rate network"
         link_type="float"
         network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0"
            last="0"/>
      <param name="min" value="0.0"/>
      <param name="max" value="0.0"/>
   </generator>
</network>
```

### 3.5.11 Knowledge

The `knowledge network` specifies which agents have what knowledge. Agents in Construct have knowledge, which they can use to select interaction partners, perform tasks, and form beliefs. This knowledge can change over the simulation as agents learn knowledge via interacting or performing tasks, or lose knowledge via forgetting.

The `knowledge network` used by the core Construct engine and must be present no matter what models are used. The network is an agent x knowledge network, meaning that each agent can have different knowledge. While Construct will update knowledge as the simulation is running according to the algorithms in the standard interaction model, the knowledge set in this network cannot be exogenously changed once the simulation starts in the absence of scripting. Note, however, that the standard interaction model can update agent knowledge as agents interact, learn, and forget.

The knowledge network is a float network, meaning that knowledge does not have to be binary. Agents can have full knowledge and no knowledge of a particular knowledge bit; however, they can also have partial knowledge. This partial knowledge can be used to drive interaction (Section 3.5.29), can be shared (Section 3.5.30), can be used to perform tasks (Section 3.5.26), and can be forgotten (Section 3.5.9). In most cases, partial knowledge is treated differently than full knowledge. For instance, agents with partial knowledge of a knowledge bit will weight similarity on that bit less than they would if they had full knowledge (Section 3.5.29). On the other hand, some algorithms do not treat partial knowledge different than full knowledge. For instance, the binary task algorithms do not distinguish between full and partial knowledge of a particular bit (Section 3.5.26). As the use of partial knowledge is algorithm specific – and indeed, is specific to the type of Construct model (Section 3.3.9) being run – users should examine documentation before assuming that float knowledge will be treated differently than binary knowledge.

Knowledge can be used to perform tasks as well as to inform beliefs. When agents use knowledge to perform a binary task, they examine a subset of their knowledge and compare it to the true value necessary to perform that task accurately. Each bit that differs from the truth value will result in an agent guess, which has a 50% chance of being incorrect. Task accuracy can improve or decline as agents learn knowledge. This process is further described in Section 3.4.4. Agents can also use their knowledge to inform beliefs when the standard belief model is in use. Agent knowledge may have a particular weight that can lead an agent to change its belief. Knowledge will be weighted by the values in the belief knowledge weight network (Section 3.5.13), which can contribute to a shift in agent belief as agents gain or lose knowledge. This process is further described in Section 3.5.13.

Values in the `knowledge network` must be floats, and should be values between 0.0 and 1.0. A value of 1.0 signifies that an agent has maximum knowledge of a particular concept while a value of 0.0 indicates that an agent has no knowledge of the concept. Knowledge values greater than 1.0 or less than 0.0 are undefined, and may cause Construct to become unstable and crash.

In the demo input deck described in this technical report, the knowledge network is organized to reflect the experiment design specified in Section 2. Specifically, four `randombinary` generators are used:

- <u>To give agents in group A some K1 knowledge</u>. The first generator gives each group A agent a 20% chance to know each knowledge bit in the K1 group. As the generator works on each bit independently, it will not guarantee that each group A agent has 20% of the knowledge bits in group K1, but on the whole group A agents should have close to 20% of the 30 K1 knowledge bits at simulation start. This occurs because the `randombinary` mean is 0.2, meaning that 20% of all link values in this region will be set. Note that the random binary sets values to be either 0.0 or 1.0, meaning that agents will either have full knowledge of the bit or no knowledge of the bit.

- To give agents in group B some K2 knowledge. The second generator gives each group B agent a 20% chance to know each knowledge bit in the K2 group. This generator is similar to the previous generator, and thus this `randombinary` generator will guarantee that each group B agent has about 20% of the 30 K2 knowledge bits.
- To give agents in group C a little K1 knowledge. The third generator gives each group C agent a 10% chance to know each knowledge bit in the K1 group. This `randombinary` generator will guarantee that each group C agent has about 10% of the 30 K2 knowledge bits. Note that this is about half of what was given to a group A agent.
- To give agents in group C a little K2 knowledge. The last generator gives each group C agent a 10% chance to know each knowledge bit in the K2 group. This `randombinary` generator will guarantee that each group C agent has about 10% of the 30 K2 knowledge bits. Note that this is about half of what was given to a group B agent.

It is also important to note which regions of the knowledge network do not have explicit generators: group A’s knowledge of the K2 bits and group B’s knowledge of the K1 bits. In Construct, network regions that are not generated are assumed to have `false` (if Boolean), `zero` (if numeric), or `“”` (if string) values. As the knowledge network is comprised of floats, values that are not generated have a value of 0.0. This means that group A agents do not have knowledge of the knowledge bits in group K2, and agents in group B do not have knowledge of the knowledge bits in group K1. This divides the knowledge groups among the group A and group B agents, as was desired in the motivating example (Section 2).

The net result of these generators is to guarantee that each agent, whether in group A, group B, or group C, has about six knowledge bits initially. Agents in group A have about 20% of the 30 K1 knowledge bits and 0% of the 30 K2 knowledge bits, for a mean of about six bits per agent. Agents in group B have 0% of the 30 K1 knowledge bits and about 20% of the 30 K2 knowledge bits, for a mean of also about six bits per agent. Agents in group C have about 10% of the 30 K1 knowledge bits and about 10% of the 30 K2 knowledge bits. Thus, they have about three knowledge bits per group, and a total of six bits overall. While the exact bits known by each agent is likely to be different, each agent will have about the same number of bits overall. Equally importantly, the distribution of bits is likely to be different, with agents in group A and B having non-overlapping knowledge.

As mentioned, all networks are generated using a `randombinary` generator, allowing a user to manipulate the mean values in order to examine what might occur if initial knowledge were modified. Each of these generators can be modified in order to understand what might occur if certain agents were not able to interact with each other (or if interaction partner possibilities were not mutual). Specifically, users may wish to examine what happens if the agents in group C have more knowledge, and how such a change affects simulation dynamics.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="knowledge"
         id="knowledge network" link_type="float"
         network_type="dense">
   <generator type="randombinary">
      <rows first="construct::intvar::agentgroup_A_start"
            last="construct::intvar::agentgroup_A_end"/>
      <cols first="construct::intvar::knowledgegroup_K1_start"
            last="construct::intvar::knowledgegroup_K1_end"/>
      <param name="mean" value="0.20"/>
   </generator>
   <generator type="randombinary">
      <rows first="construct::intvar::agentgroup_B_start"
            last="construct::intvar::agentgroup_B_end"/>
      <cols first="construct::intvar::knowledgegroup_K2_start"
            last="construct::intvar::knowledgegroup_K2_end"/>
      <param name="mean" value="0.20"/>
   </generator>
   <generator type="randombinary">
      <rows first="construct::intvar::agentgroup_C_start"
            last="construct::intvar::agentgroup_C_end"/>
      <cols first="construct::intvar::knowledgegroup_K1_start"
            last="construct::intvar::knowledgegroup_K1_end"/>
      <param name="mean" value="0.10"/>
   </generator>
   <generator type="randombinary">
      <rows first="construct::intvar::agentgroup_C_start"
            last="construct::intvar::agentgroup_C_end"/>
      <cols first="construct::intvar::knowledgegroup_K2_start"
            last="construct::intvar::knowledgegroup_K2_end"/>
      <param name="mean" value="0.10"/>
   </generator>
</network>
```

### 3.5.12 Agent Belief Network

The `agent belief network` specifies how strongly an agent holds a particular belief. Agents can hold beliefs, which represent agreements or disagreements with abstract principles. A positive link between an agent and belief node indicates that the agent holds a positive belief – i.e. supports that position – while a negative link indicates that an agent holds a negative belief – i.e. opposes that position. Like the `agent knowledge network` (Section 3.5.11) but unlike most other networks specified in the input file, this network is updated as the simulation is running in order to reflect the agent's current beliefs. During the simulation, agent beliefs can become more positive or more negative, depending on what they learn and what the agents around them believe.

The `agent belief network` is used by the standard influence model and the standard belief model (Section 3.3.9). It is an agent x belief network, meaning that different agents can have different strengths on different beliefs. While Construct will update beliefs as the simulation is running according to the algorithms in the standard belief model, the beliefs set in this network cannot be exogenously changed once the simulation starts in the absence of scripting. Note, however, that the standard belief model can update agent belief as a function of the agent's knowledge and the beliefs of those in the agent's interaction sphere if the model is activated.

To activate the standard belief model in order to update agent beliefs over time, it is necessary to both add belief nodes (Section 3.4.3) and change the model type from `disable` to `mask_mode`. When the belief model is disabled, it will not update an agent's belief, meaning that the agent belief will not change from one timeperiod to another. Belief can be enabled by changing the active model parameter (Section 3.3.9) to the following:

```
<param name="active_models" value="standard interaction
    model,standard influence model,standard belief
    model(type=>mask_mode;already_initialized=>true)"
      with="delay_interpolation"/>
```

Note that any initial beliefs set in the agent belief network will ignored by the standard belief model by default. Unless otherwise specified, the standard belief model will ignore any initial values set in the agent belief network and will instead seed the network with beliefs that reflect an agent's initial knowledge. Since agent knowledge influences agent belief via the `belief knowledge weight network` (Section 3.5.13), an agent's initial knowledge should be in accordance with an agent's belief. Thus, agents with a large number of knowledge bits that are associated with negative belief weights should, for consistency's sake, have an initial negative belief.

To force the standard belief model to use the initial values specified in the agent belief network, the supplemental subparameter `already_initialized` should be set to `true` when specifying the standard belief model. Thus, the active models simulation parameter should be initialized in the following manner:

```
<param name="active_models" value="standard interaction
    model,standard influence model,standard belief
    model(type=>mask_mode;already_initialized=>true)"
      with="delay_interpolation"/>
```

Regardless of which initialization scheme is used, however, as long as the standard belief model is in mask mode each agent's belief will evolve such that its belief level is much closer to what its true knowledge (and equally importantly, its social influence) reflects. Using different initial values for belief may still lead to different evolution patterns for belief diffusion. If all agents are initialized to have a positive belief, this may lead to a different overall level of belief than if agents are initialized to follow their initial knowledge. If `mask_mode` is active, an agent's belief is a function of its knowledge, its past belief, and the social influence of those around it. If the social influence is very strong, agents may continue to hold the socially acceptable belief as opposed to shifting to the belief indicated by their knowledge.

Values in the `agent belief weight network` must be floats, and should be values between -1.0 and 1.0. A value of -1.0 indicates that an agent has the strongest possible negative score for this belief, while a value of +1.0 indicates that it has the strongest possible positive score. A value of 0.0 can be thought of as a neutral belief (or no belief). Initial values outside the -1.0 to +1.0 range should be avoided.

In the demo input deck described in this technical report, all agent beliefs are initialized to 0.0. However, any initialized belief assignments set in this network will be ignored because the `already_initialized` subparameter is not set in the standard belief model (Section 3.3.9). Instead, beliefs will be initialized based on an agent's initial knowledge and the corresponding belief knowledge in the `belief knowledge weight network` (Section 3.5.13). Users wishing to set initial beliefs using this network need to ensure that the already initialized variable is set the `already_initialized` subparameter and may decide to replace the single `constant` generator with multiple generators, one per agent group / belief combination

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="belief"
         id="agent belief network" link_type="float"
         network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::belief::count-1"/>
      <param name="constant_value" value="0"/>
   </generator>
</network>
```

### 3.5.13 Belief Knowledge Weights

The `belief knowledge weight network` specifies how much impact a particular fact has on an agent's belief. It is well known that knowledge can impact the beliefs that one holds; the belief knowledge weight allows the input deck designer to associate particular knowledge bits with beliefs. If the standard influence model is in use, an agent's knowledge – as well as social influence – can affect belief. The knowledge weights in this network can be positive or negative which can contribute to the shift of belief in a positive or negative direction, respectively. Agents who have knowledge associated with positive facts will have their beliefs shifted in a positive direction (towards +1.0 belief) those who have knowledge associated with negative facts will have their belief shifted in a negative direction (towards -1.0 belief).

The `belief knowledge weight network` is used by the standard belief model (Section 3.3.9). It is a belief x knowledge network, meaning that the impact of knowledge on belief can vary by knowledge bit and by belief. In the absence of scripting, it is not possible modify this network over time or to change how much impact a particular piece of knowledge has on a particular agent's belief.

Note that a positive or negative belief weight is not the same thing as enhancing or detracting from an agent's belief. What is meant by a positive or negative weight is merely the direction that impact has on the agent's belief direction. Positive belief knowledge weights will move an agent's belief towards the positive (+1.0) while negative belief knowledge weights will move an agent's belief towards the negative (-1.0). Such weights can enhance or detract from an agent's belief as determined by social influence. If an agent knows knowledge bits with a positive belief weight, such knowledge will enhance a pre-existing positive belief and detract from a pre-existing negative belief. Similarly, if an agent knows knowledge bits with a negative belief weight, such knowledge will enhance a pre-existing negative belief and detract from a pre-existing positive belief.

Construct will expect that every row in the `belief knowledge weight network` will sum to zero, meaning that there is an equal number of positive and negative belief weights. There can be a different number of weights, and all weights can be of different magnitudes, but the overall contribution of knowing all knowledge bits must be no net effect on belief. This means, for instance, that a weight sequence of (-1, -1, +2) on three knowledge bits would be valid but (-1, 0, +2) would not. A row sum of zero is due to the internal structure of the algorithm, which will attempt to find a 'zero point' in order to determine how strongly each knowledge bit impacts belief. Knowledge bits with weight +2.0 will have twice as strong an impact as knowledge bits with weight +1.0; however, as belief itself is normalized in the range -1.0 to +1.0, such weights will be adjusted to more reasonable amounts when the simulation starts.

By default, the values in the belief knowledge weight network will be used to set agent beliefs. Agents start off with initial knowledge as specified in the `agent knowledge network` (Section 3.5.11). Some of this initial knowledge may be associated with particular beliefs. The belief values specified by this initial knowledge are then used to seed the belief network. Users can disable this feature of the standard belief model by specifying an `already_initialized` model subparameter, as discussed further in Section 3.5.12. In doing so, agents can set initial beliefs that are not necessarily in alignment with the beliefs specified by an agent's knowledge. Regardless of which initialization scheme is used, an agent's belief will evolve such that its belief level is much closer to what its true knowledge (and equally importantly, its social influence) reflects.

Values in the `belief knowledge network` must be floats, and should be between 0.0 and 1.0. Values outside this range may lead to undefined behavior. A value of 1.0 indicates that the agent is able to learn the knowledge bit perfectly every time it is sent to the agent, while a value of 0.0 indicates that an agent will never be able to learn the bit from another agent. An intermediate value indicates that the agent will either partially learn the knowledge bit or learn the bit by chance, depending on the learning method. If binary learning is enabled, an ego's learning rate of 0.7, for example, will lead the ego to have a 70% chance of learning the bit that an alter sends. This means that 70% of the time the ego agent will learn a value of 1.0 if the alter sends that value, while 30% of the time the ego will not learn at all. If binary learning is not enabled, a learning rate of 0.7 will mean that the ego agent will always learn a value of 0.7 if the alter sends a value of 1.0. Regardless of the learning rate or learning rate mechanism, it is not possible for an agent to increase its knowledge to more than 1.0 via communicating with another agent.

In the demo input deck described in this technical report, knowledge does not impart belief so the weight of each knowledge bit is 0.0. This network is set using a single `constant` generator. Since the belief knowledge weight is not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per belief / knowledge group combination. In doing so, however, users should ensure that belief nodes are included in the simulation (Section 3.4.3).

```
<network src_nodeclass_type="belief"
         target_nodeclass_type="knowledge"
         id="belief knowledge weight network" link_type="float"
         network_type="dense">
   <generator type="constant">
      <rows first="0" last="nodeclass::belief::count-1"/>
      <cols first="0" last="nodeclass::knowledge::count-1"/>
      <param name="constant_value" value="0"/>
   </generator>
</network>
```

### 3.5.14 Interaction Sphere

The `interaction sphere network` specifies which agents can potentially interact with whom when Construct is running. Agents in Construct must keep track of each potential interaction partner's transactive memory (Section 3.6) in order to compute probabilities of interaction with them. Note one agent's presence in another's interaction sphere does not guarantee interaction during the simulation; access between the two agents can be restricted via the `access network` (Section 3.5.15), agents may not have a high probability of interaction, or agents may always choose to interact with others as opposed to with each other. On the other hand, exclusion from another agent's interaction sphere guarantees that two agents will never be able to interact during the course of the simulation.

The `interaction sphere network` is used by the standard interaction model (Section 3.3.9) and specifies which agents keep transactive memory of and can interact with which. The network is an agent x agent network, meaning that each agent can have different agents in their interaction sphere. This network must remain constant over the course of the experiment, and cannot over time or as agents learn knowledge. Note that this network cannot be modified, even with scripting, as after the simulation begins it becomes highly intertwined with transactive memory storage. If an agent's set of potential interaction partners is to be modified dynamically, users should allow that agent to have all agents in their interaction sphere and restrict agent access via the `access network` (Section 3.5.15) instead.

As noted in the first paragraph, an alter's presence in an ego's interaction sphere does not guarantee that the ego agent and alter agent will interact. Instead, only the converse is true: if an alter agent is not in an ego agent's interaction sphere, the two cannot interact in this version of Construct. Since interaction is driven by transactive memory, and egos only have transactive memory about alters in their interaction sphere, an ego will not consider interacting with an alter about whom it has no transactive memory. If an alter is in an ego's interaction sphere, however, there are still reasons why the two may not interact. First, the ego may not have access to the alter because of the values specified in the access network (Section 3.5.15). Second, an ego may not have sufficient reason to want to interact with a particular alter, thus leading to a low probability of interaction. This may be because the ego lacks sufficient physical (Section 3.5.17), socio-demographic (Section 3.5.18), or social (Section 3.5.19) proximity to the alter; alternately, the ego and alter may not have much to drive interaction via shared knowledge (Section 3.5.27), shared binary tasks (Section 3.5.23), or expertise (Section 3.5.28). Third, regardless of how similar the ego and alter are, the two may be too busy interacting with others in order to interact; if the ego can only initiate a few interactions each timeperiod (Section 3.5.2) or the alter only receive a few interactions each timeperiod (Section 3.5.3), both agents may interact too frequently with other people in order to ever choose to interact with each other.  Fourth, the interference of other agents may prevent interaction; for instance, if a third party constantly seeks out the alter agent, it may prevent the ego from ever interacting with that alter.  Last, bad luck may prevent an ego and alter from interacting.  When selecting an alter to interact with, there is an element of luck as to which agent is chosen based on probabilities of interaction.  In a short simulation, two agents may meet all the previous criteria but fail to interact due the stochastic nature of the simulation.

Even if another agent is not in an ego's interaction sphere, it can still have profound effects on the ego agent.  For instance, the alter agent may be able to pass information through one or more intermediates to the ego.  Thus, information can be exchanged via a chain of agents, even if this chain starts outside an ego's interaction sphere.  Additionally, the alter may choose to interact with another agent who is in the interaction spheres of both the ego and alter, thus preventing the ego agent from interacting with it.  This may force the ego agent to choose a different interaction partner than the one it preferred to interact with; this may force the ego to learn new types of knowledge, and may even contribute to a very different evolution of the simulation.  Furthermore, the alter agent may be able to influence the beliefs of other agents in the ego's interaction sphere, which can then contribute to social influence on the ego.  In all three scenarios, while the ego and alter do not directly interact, they can have a profound effect on each other's change and development over the course of the simulation. Thus, while the interaction sphere specifies which agents can directly communicate with the ego, it does not prevent agents outside the interaction sphere from influencing the ego agent.

Note that the interaction sphere network does not have to be symmetric.  Thus, even though an ego has a potential alter in its interaction sphere, that alter does not have to include the ego in its interaction sphere.  In this case, the ego can initiate communication with the alter, but the alter will not be able to initiate communication with the ego.  Should the ego select the alter as its interaction partner, however, both the ego and alter would select knowledge, beliefs, and transactive memory to exchange.  While the ego will be able to update its transactive memory of the alter following this communication (if the ego learns a knowledge bit, it will also learn that the alter also knows that knowledge bit), the alter will not update its transactive memory since it has no transactive memory of the ego.

The size of each agent's interaction sphere can have a profound effect on the running time and computer memory required for the simulation.  For each agent in the interaction sphere, it is necessary to compute a probability of interaction with that agent, a calculation which requires examining all facts that the alter knows.  Calculating the probability of interaction is the most time-intensive part of the simulation, and as such increasing the interaction sphere size without reason will cause unnecessary slowdown, though removing access to that agent (Section 3.5.15) will eliminate most of this slowdown.  More importantly, however, an ego must keep transactive memory of each alter in its interaction sphere.  While ego agents typically do not think that an alter knows every single knowledge bit, egos must reserve sufficient memory to represent what they perceive to be an alter's knowledge.  Increasing the number of alters in the interaction sphere, then, will greatly increase the amount of memory necessary in order to run a Construct simulation.

Values in the `interaction sphere network` are bools, and can be set using the values 1/0 or true/false.  A value of 1 or true indicates that an ego (row) agent has transactive memory of the alter (column) agent, while a value of 0 or false indicates that it does not.

In the demo input deck described in this technical report, the interaction sphere is organized to reflect the experiment design specified in Section 2. Specifically, six `randombinary` generators are used:

- To allow interaction between agents within group A. The first generator allows all agents in group A to have all group A agents in their interaction sphere. It thus will create a situation in which all group A agents to can potentially interact with all group A agents. This occurs because the `randombinary` mean is 1.0, meaning that 100% of all link values in this region will be “true”.
- To allow interaction between agents within group B. The second generator allows all agents in group B to have all group B agents in their interaction sphere. It also has a `randombinary` mean of 1.0.
- To allow interaction between agents within group C. The third generator allows all agents in group C to have all group C agents in their interaction sphere. It also has a `randombinary` mean of 1.0.
- To prevent interaction between group A agents and group B agents. The fourth generator prevents agents in group A from having any agents in group B in their interaction sphere. This `randombinary` generator has a mean of 0.0, meaning that links with value “false” will be generated. This ensures that agents in group A cannot interact with – or store transactive memory about – agents in group B. The generator is symmetric, so it also prevents agents in group B from storing transactive memory of having any agents in group A in their interaction sphere. Note that this generator is technically not necessary, but included here for completeness sake. As the default network value is “false”, the values in this region do not change as a result of this generator.
- To allow interaction between group A agents and group C agents. The fifth generator allows all agents in group A to have all group C agents in their interaction sphere. It has a `randombinary` mean of 1.0, meaning that all links in this region are generated. As the generator is symmetric, it will also generate links between group C agents and group A agents.
- To allow interaction between group B agents and group C agents. The last generator allows all agents in group B to have all group C agents in their interaction sphere. It also has a `randombinary` mean of 1.0, meaning that all links in this region are generated. As the generator is symmetric, it will also generate links between group C agents and group B agents.

As mentioned, all networks are generated using a `randombinary` generator, allowing a user to manipulate the mean values in order to examine what might occur if access between subgroups of agents is modified. All generators are symmetric, meaning that if agent X can potentially interact with agent Y, then agent Y can potentially interact with agent X. Each of these generators can be modified in order to understand what might occur if certain agents were not able to interact with each other (or if interaction partner possibilities were not mutual).

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="agent"
```

```
      id="interaction sphere network" link_type="bool"
      network_type="dense">
<generator type="randombinary">
   <rows first="construct::intvar::agentgroup_A_start"
         last="construct::intvar::agentgroup_A_end"/>
   <cols first="construct::intvar::agentgroup_A_start"
         last="construct::intvar::agentgroup_A_end"/>
   <param name="mean" value="1.0"/>
   <param name="symmetric" value="true"/>
</generator>
<generator type="randombinary">
   <rows first="construct::intvar::agentgroup_B_start"
         last="construct::intvar::agentgroup_B_end"/>
   <cols first="construct::intvar::agentgroup_B_start"
         last="construct::intvar::agentgroup_B_end"/>
   <param name="mean" value="1.0"/>
   <param name="symmetric" value="true"/>
</generator>
<generator type="randombinary">
   <rows first="construct::intvar::agentgroup_C_start"
         last="construct::intvar::agentgroup_C_end"/>
   <cols first="construct::intvar::agentgroup_C_start"
         last="construct::intvar::agentgroup_C_end"/>
   <param name="mean" value="1.0"/>
   <param name="symmetric" value="true"/>
</generator>
<generator type="randombinary">
   <rows first="construct::intvar::agentgroup_A_start"
         last="construct::intvar::agentgroup_A_end"/>
   <cols first="construct::intvar::agentgroup_B_start"
         last="construct::intvar::agentgroup_B_end"/>
   <param name="mean" value="0.0"/>
   <param name="symmetric" value="true"/>
</generator>
<generator type="randombinary">
   <rows first="construct::intvar::agentgroup_A_start"
         last="construct::intvar::agentgroup_A_end"/>
   <cols first="construct::intvar::agentgroup_C_start"
         last="construct::intvar::agentgroup_C_end"/>
   <param name="mean"
          value="construct::boolvar::bridging_agents_active"/>
   <param name="symmetric" value="true"/>
</generator>
<generator type="randombinary">
   <rows first="construct::intvar::agentgroup_B_start"
         last="construct::intvar::agentgroup_B_end"/>
   <cols first="construct::intvar::agentgroup_C_start"
```

```
            last="construct::intvar::agentgroup_C_end"/>
      <param name="mean"
             value="construct::boolvar::bridging_agents_active"/>
      <param name="symmetric" value="true"/>
   </generator>
</network>
```

### 3.5.15 Access Network

The `access network` serves as a soft supplement to the interaction sphere, and can restrict the alters with which an agent can communicate. While the `interaction sphere network` (Section 3.5.14) is a hard restriction on which agents can keep transactive memory of which alters, the access network can prevent agents from potentially interacting with some of those alters. Agents need to both be in the interaction sphere as well as have access to each other in order to potentially communicate.

The `access network` is used by the core Construct engine and must be present no matter what models are used. It specifies which agents have access to which; alter agents who lack access to an ego agent are ignored when looping over the potential partners for an ego agent. The network is an agent x agent network, meaning that each agent can have access to different subsets of agents. In the absence of scripting, it is not possible modify this network over time or to change agent access constraints as agents learn knowledge.

As mentioned previously, alter agents must meet two criteria in order to be considered as an interaction partner. First, an alter agent must be in an ego's interaction sphere via the `interaction sphere network` (Section 3.5.14). Second, the ego must have access to the alter in the `access network`. If an alter is not in an agent's interaction sphere, the access network has no additional effect as the alter cannot be considered a potential interaction partner.

Note that some potential active mechanisms (Section 3.3.10), such as the `internet access mechanism`, will act by the access network. These modifications will be made after the network has been initialized. It is not necessary to modify the access network in any way in order to facilitate these mechanisms during network initialization. Thus, while agents may be initialized with the ability to potentially interact with (for example) web site agents, their lack of access will prevent them from ever doing so.

Values in the `access network` are bools, and can be set using the values 1/0 or true/false. A value of 1 or true indicates that an ego (row) agent has access to an alter (column) agent, while a value of 0 or false indicates that the ego does not.

In the demo input deck described in this technical report, the interaction sphere is not restricted, meaning that each value in the access network is set to 1.0. This network is set using a `randomuniform` generator, allowing a user to manipulate the minimum values in order to examine what might occur if access between random groups of agents was modified. Since modifying access is not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per agent group / agent group combination.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="agent"
         id="access network" link_type="float"
         network_type="dense">
   <generator type="randomuniform">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::agent::count-1"/>
      <param name="min" value="1.0"/>
      <param name="max" value="1.0"/>
   </generator>
</network>
```

### 3.5.16 Agent Active Time Period

The `agent active timeperiod network` specifies which agents are active during which timeperiods. Agents who are active are able to initiate and receive interaction, exchange messages, perform tasks, and update their beliefs. Agents who are not active are not able to initiate or receive interaction, exchange messages, perform tasks, or update their beliefs. In effect, inactive agents are treated as if they do not exist for the purposes of the simulation.

The `agent active timeperiod network` is used by the core Construct engine and must be present no matter what models are used. It is an agent x timeperiod network, meaning that activity is a property that varies with time. It should be noted that, in the absence of scripting, activity has to be set prior to the start of the simulation and cannot be changed while the simulation is running. Thus, it is not possible to dynamically activate or inactivate agents base on the knowledge or task performance of other agents.

If the agent is to be removed permanently from the simulation after a particular timeperiod, the agent can also be isolated from the simulation. This can be done by adding the `isolation model` as an active model (Section 3.3.9), then by adding an `isolation` nodeclass to the simulation. While this is an advanced technique and is used by *ORA's near term analysis, users are strongly encouraged to perform an isolation by setting an agent to inactive using the agent active timeperiod network. The majority of this model's behavior can be mimicked by marking an agent as inactive for all future periods.

Values in the `agent active timeperiod network` are bools, and can be set using the values 1/0 or true/false. A value of 1 or true indicates that an agent is active during that particular timeperiod, while a value of 0 or false indicates that it is not.

In the demo input deck described in this technical report, agents in agent group A and B are always active while agents in agent group C are active conditional on the value of the `bridging_agents_active` Construct variable (Section 3.2.1). In the input file, this parameter is set serially, first for agent group A then agent group B then agent group C. Note the use of a construct variable in the setting of the `constant_value` for agent group C. As the value of this variable is determined when Construct starts running, the agents in agent group C can be assigned to be always active (with value 1) or always inactive (with value 0) depending upon the user selection at the top of the input file. This allows for rapid modification of a parameter at one convenient location.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="timeperiod"
         id="agent active timeperiod network"
         link_type="bool" network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
            last="construct::intvar::agentgroup_A_end"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
       <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
            last="construct::intvar::agentgroup_B_end"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
       <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
            last="construct::intvar::agentgroup_C_end"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value"
             value="construct::boolvar::bridging_agents_active"/>
   </generator>
</network>
```

### 3.5.17 Physical Proximity

The `physical proximity network` specifies how close two agents are physically. While Construct does not expressly represent physical locations, it uses a physical distance metric in order to determine how likely two agents are to interact. By weighting physical proximity with other forms of proximity – sociodemographic proximity (Section 3.5.18) and social proximity (Section 3.5.19) – an agent's propensity to interact with certain partners can be modified. Agents that are seen as physically 'close' to an ego agent will have higher interaction probabilities, while those that are physically 'far' will have smaller. This, in turn, should lead an agent to favor physically close interaction partners over far ones.

The `physical proximity network` is used by the standard interaction model (Section 3.3.9). It is an agent x agent network, meaning that every pair of agents has a physical distance to each other. This distance does not have to be symmetric; the distance between the ith and jth agents does not have to be the same as between the jth and ith (which might be the case, for instance, if one agent had access to a car and the other did not). Note that the physical proximity value can be considered a perception of physical distance, as it drives agent-agent interaction; experimenters can choose whether or not to make this value an accurate representation of the real world or a possible misperception by one or both of the agents in the agent-agent pair. Physical distance is assumed to be constant throughout the simulation. In the absence of scripting, it is not possible modify the physical proximity network values as the simulation evolves or as agents learn knowledge.

While physical proximity must be constant, the weight that agents place on physical proximity may change over time. The `physical proximity weight network` (Section 3.5.20) specifies how much emphasis agents put on physical proximity relative to other factors such as socio-demographic proximity and social proximity. The physical proximity network will only be examined if the physical proximity weight is nonzero; the values in the physical proximity weight network, socio-demographic proximity weight network and social proximity weight network will be normalized by these weights.

Note that the physical proximity, socio-demographic proximity, and social proximity networks (and their respective weight networks) are identical internally. The values in the physical proximity network and, say, the socio-demographic proximity network can be swapped with no change to the simulation algorithm or results (as long as the weight networks are appropriately swapped). Three separate networks have been provided in order to allow the experimenter three conceptually different ways to control agent behavior. In some experiments, it may be useful to keep concepts such as physical distance constant while changing other factors like socio-demographic similarity. Alternately, if some physical proximity data is known but socio-demographic or social data is missing, it should be possible to seed the simulation with this data while allowing the other factors to be randomized.

Values in the `physical proximity network` must be floats, and should be values between 0.0 and 1.0. A value of 1.0 represents two agents that are maximally close to each other while a value of 0.0 represents two agents that are maximally separated.

In the demo input deck described in this technical report, agents do not use the physical proximity network when computing probabilities of interaction. Thus, the values in this network are all marked as 0.0. This network is set using a `randomuniform` generator, allowing a user to manipulate the maximum value in order to examine what might occur if some agents had different physical proximities. Since physical proximity is not considered an important part of this example, however, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of physical proximity on the demo network should also make sure that the physical proximity weight (Section 3.5.20) is set to a non-zero value or else changes to this network will not have an effect on the simulation.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="agent"
         id="physical proximity network"
         link_type="float" network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
           last="nodeclass::agent::count-1"/>
      <cols first="0"
           last="nodeclass::agent::count-1"/>
      <param name="min" value="0.0"/>
      <param name="max" value="0.0"/>
   </generator>
</network>
```

### 3.5.18 Socio-Demographic Proximity

The `sociodemographic proximity network` (one non-hyphenated word) specifies how close two agents are physically. While Construct does not expressly represent sociodemographic attributes, it uses a socio-demographic distance metric in order to determine how likely two agents are to interact. By weighting socio-demographic proximity with other forms of proximity – physical proximity (Section 3.5.17) and social proximity (Section 3.5.19) – an agent's propensity to interact with certain partners can be modified. Agents that are seen as socio-demographically 'close' to an ego agent will have higher interaction probabilities, while those that are socio-demographically 'far' will have smaller. This, in turn, should lead an agent to favor socio-demographically close interaction partners over far ones.

The `sociodemographic proximity network` is used by the standard interaction model (Section 3.3.9). It is an agent x agent network, meaning that every pair of agents has a socio-demographic distance to each other. This distance does not have to be symmetric; the distance between the ith and jth agents does not have to be the same as between the jth and ith (which might be the case, for instance, if one agent has less prejudice against another agent). Note that the sociodemographic proximity value can be considered a perception of socio-demographic distance, as it drives agent-agent interaction; experimenters can choose whether or not to make this value an accurate representation of the real world or a possible misperception by one or both of the agents in the agent-agent pair. Socio-demographic distance is assumed to be constant throughout the simulation. In the absence of scripting, it is not possible modify the socio-demographic proximity network values as the simulation evolves or as agents learn knowledge.

While socio-demographic proximity must be constant, the weight that agents place on socio-demographic proximity may change over time. The `sociodemographic proximity weight network` (Section 3.5.21) specifies how much emphasis agents put on socio-demographic proximity relative to other factors such as physical proximity and social proximity. The socio-demographic proximity network will only be examined if the socio-demographic proximity weight is nonzero; the values in the socio-demographic proximity weight network, physical proximity weight network and social proximity weight network will be normalized by these weights.

Note that the physical proximity, socio-demographic proximity, and social proximity networks (and their respective weight networks) are identical internally. The values in the socio-demographic proximity network and, say, the social proximity network can be swapped with no change to the simulation algorithm or results (as long as the weight networks are appropriately swapped). Three separate networks have been provided in order to allow the experimenter three conceptually different ways to control agent behavior. In some experiments, it may be useful to keep concepts such as socio-demographic distance constant while changing other factors like social similarity. Alternately, if some socio-demographic proximity data is known but physical or social data is missing, it should be possible to seed the simulation with this data while allowing the other factors to be randomized.

Values in the `sociodemographic proximity network` must be floats, and should be values between 0.0 and 1.0. A value of 1.0 represents two agents that are maximally close to each other while a value of 0.0 represents two agents that are maximally separated.

In the demo input deck described in this technical report, agents do not use the sociodemographic proximity network when computing probabilities of interaction. Thus, the values in this network are all marked as 0.0. This network is set using a `randomuniform` generator, allowing a user to manipulate the maximum value in order to examine what might occur if some agents had different sociodemographic proximities. Since sociodemographic proximity is not considered an important part of this example, however, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of sociodemographic proximity on the demo network should also make sure that the sociodemographic proximity weight (Section 3.5.21) is set to a non-zero value or else changes to this network will not have an effect on the simulation.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="agent"
         id="sociodemographic proximity network"
         link_type="float" network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0"
            last="nodeclass::agent::count-1"/>
      <param name="min" value="0.0"/>
      <param name="max" value="0.0"/>
   </generator>
</network>
```

### 3.5.19 Social Proximity

The `social proximity network` specifies how close two agents are socially. Social proximity serves as a simple way of making two agents more or less likely to interact in order to capture differences in factors such as seniority, career type, introversion, or other factors which might affect how frequently two agents would want to interact. By weighting social proximity with other forms of proximity – physical proximity (Section 3.5.17) and socio-demographic proximity (Section 3.5.18) – an agent's propensity to interact with certain partners can be modified. Agents that are seen as socially 'close' to an ego agent will have higher interaction probabilities, while those that are socially 'far' will have smaller. This, in turn, should lead an agent to favor socially close interaction partners over far ones.

The `social proximity network` is used by the standard interaction model (Section 3.3.9). It is an agent x agent network, meaning that every pair of agents has a social distance to each other. This distance does not have to be symmetric; the distance between the ith and jth agents does not have to be the same as between the jth and ith (which might be the case, for instance, if one agent is more senior than another and would be less likely to be approachable). Note that the social proximity value can be considered a perception of social distance, as it drives agent-agent interaction; experimenters can choose whether or not to make this value an accurate representation of the real world or a possible misperception by one or both of the agents in the agent-agent pair. Social distance is assumed to be constant throughout the simulation. In the absence of scripting, it is not possible modify the socio-demographic proximity network values as the simulation evolves or as agents learn knowledge.

While social proximity must be constant, the weight that agents place on social proximity may change over time. The `social proximity weight network` (Section 3.5.22) specifies how much emphasis agents put on socio-demographic proximity relative to other factors such as physical proximity and socio-demographic proximity. The social proximity network will only be examined if the social proximity weight is nonzero; the values in the social proximity weight network, physical proximity weight network and socio-demographic proximity weight network will be normalized by these weights.

Note that the physical proximity, socio-demographic proximity, and social proximity networks (and their respective weight networks) are identical internally. The values in the social proximity network and, say, the physical proximity network can be swapped with no change to the simulation algorithm or results (as long as the weight networks are appropriately swapped). Three separate networks have been provided in order to allow the experimenter three conceptually different ways to control agent behavior. In some experiments, it may be useful to keep concepts such as social distance constant while changing other factors like physical similarity. Alternately, if some social proximity data is known but physical or socio-demographic data is missing, it should be possible to seed the simulation with this data while allowing the other factors to be randomized.

Values in the `social proximity network` must be floats, and should be values between 0.0 and 1.0. A value of 1.0 represents two agents that are maximally close to each other while a value of 0.0 represents two agents that are maximally separated.

In the demo input deck described in this technical report, agents use only social proximity when computing probabilities of interaction. As knowledge is the primary factor driving interaction, agents are designed to have equal social distances from each other. Thus, the values in this network are all marked as 1.0. This network is set using a `randomuniform` generator, allowing a user to manipulate the minimum value in order to examine what might occur if some agents had different social proximities.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="agent"
         id="social proximity network"
         link_type="float" network_type="dense">
   <generator type="randomuniform">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::agent::count-1"/>
      <param name="min" value="1.0"/>
      <param name="max" value="1.0"/>
   </generator>
</network>
```

### 3.5.20 Physical Proximity Weight

The `physical proximity weight network` specifies how strongly agents will value physical proximity when choosing an interaction partner. While Construct does not expressly represent physical locations, it uses a physical distance metric (Section 3.5.17) in order to determine how likely two agents are to interact. By weighting physical proximity with other forms of proximity – sociodemographic proximity (Section 3.5.18) and social proximity (Section 3.5.19) – an agent's propensity to interact with certain partners can be modified. Agents that are seen as physically 'close' to an ego agent will have higher interaction probabilities, while those that are physically 'far' will have smaller. This, in turn, should lead an agent to favor physically close interaction partners over far ones.

The `physical proximity weight network` is used by the standard interaction model (Section 3.3.9). It is an agent x timeperiod network, meaning the weight placed on physical proximity can differ by timeperiod. Thus, agents can weight physical proximity more strongly in certain timeperiods while weighting it less strongly in others. However, the weights set in this network cannot be changed once the simulation starts. In the absence of scripting, it is not possible modify the physical proximity weight values as the simulation evolves or as agents learn knowledge.

Note that the physical proximity, socio-demographic proximity, and social proximity networks (and their respective weight networks) are identical internally. The values in the physical proximity network and, say, the socio-demographic proximity network can be swapped with no change to the simulation algorithm or results (as long as the weight networks are appropriately swapped). Three separate networks have been provided in order to allow the experimenter three conceptually different ways to control agent behavior. In some experiments, it may be useful to keep concepts such as physical distance constant while changing other factors like socio-demographic similarity. Alternately, if some physical proximity data is known but socio-demographic or social data is missing, it should be possible to seed the simulation with this data while allowing the other factors to be randomized.

Values in the `physical proximity weight network` must be floats, and should be values between 0.0 and 1.0. A value of 1.0 represents a maximal weight on physical proximity (to the exclusion of socio-demographic and social proximity) while a value of 0.0 means that physical proximity has no effect on the simulation. If the physical proximity weight for an agent is ever 0.0 in a timeperiod, all values in the physical proximity network will be ignored. Note that the sum of the values in the physical proximity weight network, sociodemographic proximity weight network, and social proximity weight network must be 1.0 for each agent and each timeperiod. If the values do not sum to one, the values are re-normalized such that their sum is one.

In the demo input deck described in this technical report, agents do not use the physical proximity network when computing probabilities of interaction. Thus, the values in this weight network are all marked as 0.0. This network is set using a `randomuniform` generator, allowing a user to manipulate the maximum value in order to examine what might occur if some agents had different physical proximities. Since physical proximity is not considered an important part of this example, however, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of physical proximity on the demo network should also make sure that the physical proximity relationships (Section 3.5.17) are set to non-zero values or else changes to this network will not have an effect on the simulation.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="timeperiod"
         id="physical proximity weight network"
         link_type="float" network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="min" value="0.0"/>
      <param name="max" value="0.0"/>
   </generator>
</network>
```

### 3.5.21 Socio-Demographic Proximity Weight

The `sociodemographic proximity weight network` (one non-hyphenated word) specifies how strongly agents will value sociodemographic proximity when choosing an interaction partner. While Construct does not expressly represent socio-demographic attributes, it uses a socio-demographic distance metric (Section 3.5.18) in order to determine how likely two agents are to interact. By weighting socio-demographic proximity with other forms of proximity – physical proximity (Section 3.5.17) and social proximity (Section 3.5.19) – an agent's propensity to interact with certain partners can be modified. Agents that are seen as socio-demographically 'close' to an ego agent will have higher interaction probabilities, while those that are socio-demographically 'far' will have smaller. This, in turn, should lead an agent to favor socio-demographically similar interaction partners over far ones.

The `sociodemographic proximity weight network` is used by the standard interaction model (Section 3.3.9). It is an agent x timeperiod network, meaning the weight placed on socio-demographic proximity can differ by timeperiod. Thus, agents can weight socio-demographic proximity more strongly in certain timeperiods while weighting it less strongly in others. However, the weights set in this network cannot be changed once the simulation starts. In the absence of scripting, it is not possible modify the physical socio-demographic weight values as the simulation evolves or as agents learn knowledge.

Note that the physical proximity, socio-demographic proximity, and social proximity networks (and their respective weight networks) are identical internally. The values in the socio-demographic proximity network and, say, the social proximity network can be swapped with no change to the simulation algorithm or results (as long as the weight networks are appropriately swapped). Three separate networks have been provided in order to allow the experimenter three conceptually different ways to control agent behavior. In some experiments, it may be useful to keep concepts such as socio-demographic distance constant while changing other factors like social similarity. Alternately, if some socio-demographic proximity data is known but physical or social data is missing, it should be possible to seed the simulation with this data while allowing the other factors to be randomized.

Values in the `sociodemographic proximity weight network` must be floats, and should be values between 0.0 and 1.0. A value of 1.0 represents a maximal weight on socio-demographic proximity (to the exclusion of physical and social proximity) while a value of 0.0 means that physical proximity has no effect on the simulation. If the socio-demographic proximity weight for an agent is ever 0.0 in a timeperiod, all values in the sociodemographic proximity network will be ignored. Note that the sum of the values in the physical proximity weight network, sociodemographic proximity weight network, and social proximity weight network must be 1.0 for each agent and each timeperiod. If the values do not sum to one, the values are re-normalized such that their sum is one.

In the demo input deck described in this technical report, agents do not use the sociodemographic proximity network when computing probabilities of interaction. Thus, the values in this weight network are all marked as 0.0. This network is set using a `randomuniform` generator, allowing a user to manipulate the maximum value in order to examine what might occur if some agents had different socio-demographic proximities. Since socio-demographic proximity is not considered an important part of this example, however, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of socio-demographic proximity on the demo network should also make sure that the socio-demographic proximity relationships (Section 3.5.18) are set to non-zero values or else changes to this network will not have an effect on the simulation.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="timeperiod"
         id="sociodemographic proximity weight network"
         link_type="float" network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="min" value="0.0"/>
      <param name="max" value="0.0"/>
   </generator>
</network>
```

### 3.5.22 Social Proximity Weight

The `social proximity weight network` specifies how strongly agents will value social proximity when choosing an interaction partner. While Construct does not expressly represent social roles or core psychological attributes, it uses a social distance metric (Section 3.5.19) in order to determine how likely two agents are to interact. By weighting social proximity with other forms of proximity – physical proximity (Section 3.5.17) and sociodemographic proximity (Section 3.5.18) – an agent's propensity to interact with certain partners can be modified. Agents that are seen as socially 'close' to an ego agent will have higher interaction probabilities, while those that are socially 'far' will have smaller. This, in turn, should lead an agent to favor socially close interaction partners over far ones.

The `social proximity weight network` is used by the standard interaction model (Section 3.3.9). It is an agent x timeperiod network, meaning the weight placed on social proximity can differ by timeperiod. Thus, agents can weight social proximity more strongly in certain timeperiods while weighting it less strongly in others. However, the weights set in this network cannot be changed once the simulation starts. In the absence of scripting, it is not possible modify the social proximity weight values as the simulation evolves or as agents learn knowledge.

Note that the physical proximity, socio-demographic proximity, and social proximity networks (and their respective weight networks) are identical internally. The values in the social proximity network and, say, the physical proximity network can be swapped with no change to the simulation algorithm or results (as long as the weight networks are appropriately swapped). Three separate networks have been provided in order to allow the experimenter three conceptually different ways to control agent behavior. In some experiments, it may be useful to keep concepts such as social distance constant while changing other factors like physical similarity. Alternately, if some social proximity data is known but physical or socio-demographic data is missing, it should be possible to seed the simulation with this data while allowing the other factors to be randomized.

Values in the `social proximity weight network` must be floats, and should be values between 0.0 and 1.0. A value of 1.0 represents a maximal weight on social proximity (to the exclusion of physical and socio-demographic proximity) while a value of 0.0 means that social proximity has no effect on the simulation. Note that the sum of the values in the physical proximity weight network, sociodemographic proximity weight network, and social proximity weight network must be 1.0 for each agent and each timeperiod. If the values do not sum to one, the values are re-normalized such that their sum is one.

In the demo input deck described in this technical report, agents use only social proximity when computing probabilities of interaction. As knowledge is the primary factor driving interaction, agents are designed to have equal social distances from each other. For this reason, social proximity is weighted fully to the exclusion of physical and socio-demographic proximity; the weight placed on all social proximity values is 1.0. This network is set using a `randomuniform` generator, allowing a user to manipulate the minimum value in order to examine what might occur if some agents had different social proximities.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="timeperiod"
         id="social proximity weight network"
         link_type="float" network_type="dense">
  <generator type="randomuniform">
     <rows first="0"
           last="nodeclass::agent::count-1"/>
     <cols first="0"
           last="nodeclass::timeperiod::count-1"/>
     <param name="min" value="1.0"/>
     <param name="max" value="1.0"/>
  </generator>
</network>
```

### 3.5.23 Binary Task Similarity Weight

The `binarytask similarity weight network` specifies how much weight agents place on shared tasks. It is well known that people who work on the same tasks are more likely to interact with each other, and also will have a large amount of task-related knowledge in common. The binary task similarity weight focuses on this former part and makes two agents more likely to interact the greater the number of tasks to which they are assigned. The weight on task similarity competes with the weight placed on knowledge similarity (Section 3.5.27) and knowledge expertise (Section 3.5.28) in order to comprise the variable portion of the probability of interaction.

The `binarytask similarity weight network` is used by the standard interaction model (Section 3.3.9). It is an agent x timeperiod network, meaning the weight placed on binary tasks can differ by timeperiod. Thus, agents can weight task similarity more strongly in certain timeperiods while weighting it less strongly in others. However, the weights set in this network cannot be changed once the simulation starts. In the absence of scripting, it is not possible modify the knowledge expertise weight values as the simulation evolves or as agents learn knowledge.

Unlike knowledge, which agents perceive – or potentially misperceive – using transactive memory, there is no transactive memory for task assignment. Agents are assumed to perfectly perceive which agents perform which tasks. Agents are able to use the true network of which agents are assigned to which tasks as opposed to a transactive memory value.

When computing binary task similarity, each task that agents have in common contributes equally to binary task similarity. While knowledge bits can be weighted such that they contribute unevenly – and even possibly negatively – via the `interaction knowledge weight network` (Section 3.5.29), such weighing is not possible for binary tasks. Construct views all binary tasks as equally important for interaction. Should some binary tasks be more important for driving interaction, the task node itself can be repeated multiple times in order to make the group of tasks appear more important. Alternately, weights associated with task knowledge can be increased in interaction knowledge weight network, a factor that should indirectly boost coworker interaction.

Net binary task similarity is computed by summing over all tasks shared by a particular agent pair. The net binary task similarity is then multiplied by the value of the binarytask similarity weight in order to determine the how binary task similarity contributes to the probability of interaction.

Values in the `binarytask similarity network` must be floats, and should be values between 0.0 and 1.0. A value of 1.0 means that agents will compute interaction probabilities using only shared tasks to the exclusion of knowledge similarity and expertise, while a value of 0.0 means that agents will ignore knowledge expertise in favor of other factors. Note that the sum of the values in the knowledge similarity weight network, knowledge expertise weight network, and binary task similarity weight network must be 1.0 for each agent and each timeperiod. If the values do not sum to one, the values are re-normalized such that their sum is one.

In the demo input deck described in this technical report, interaction is driven by knowledge similarity and, to a lesser extent, by knowledge expertise. Because no binary tasks are modeled, shared task similarity should have no effect. By construction, then, the values in the binary task similarly network are 0.0 for all agents over all time periods. This network is set using a `randomuniform` generator, allowing a user to manipulate the maximum values in order to examine what might occur if some agents had different weights on expertise. Note that this network is set using a single generator, so experimenters interested in examining changes to similarity weights for subsets of agents may consider using multiple generators and breaking this network up into subsections.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="timeperiod"
         id="binarytask similarity weight network"
         link_type="float" network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="min" value="0.0"/>
      <param name="max" value="0.0"/>
   </generator>
</network>
```

### 3.5.24 Binary Task Assignment

The `binarytask assignment network` specifies which agents are assigned to which binary tasks. Agents assigned to binary tasks will be able to perform them each time period. Agents can learn knowledge by performing the binary task (Section 3.5.10) and will also have increased similarity with others who are performing the same binary task (Section 3.5.23). The binary task completion algorithm is described in Section 3.4.4.

The `binarytask assignment network` is used by the standard interaction model (Section 3.3.9). It is an agent x binarytask network, meaning that task assignment may differ by agent. However, task assignment cannot be changed once the simulation starts and must remain constant for the entire simulation. In the absence of scripting, it is not possible modify task assignment as the simulation evolves or as agents learn knowledge.

Agents can be assigned to one or more binary tasks, which they perform in the manner discussed in Section 3.4.4. Assignment to more than one binary task will not impact task performance, as it may for energy task assignment (Section 3.4.5). Instead, assignment to multiple binary tasks may lead agents to interact more with other agents, as shared binary tasks (Section 3.5.23) increase similarity and cause agents to prefer each other as interaction partners.

Agents assigned to multiple binary tasks will have the opportunity to attempt each task during each time period. Agents will continue to attempt the task in each time period, even if they were able to successfully complete the task in a previous time period. While it is likely that agents who completed a task in a previous timeperiod will also complete the task successfully in a later timeperiod, note that agents can complete some tasks by guessing and therefore may not necessarily be accurate in future attempts.

Values in the `binarytask assignment network` are bools, and can be set using the values 1/0 or true/false. A value of 1 or true indicates that an agent is assigned to a binary task, while a value of 0 or false indicates that an agent is not.

In the demo input deck described in this technical report, agents are not assigned to binary task so all values in this network are set to 0. This network is set using a `randombinary` generator, allowing a user to manipulate the mean value in order to examine what might occur if more agents were assigned randomly to particular task. Since task assignment is not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per agent group / binary task combination. It would also be necessary to add one or more binary tasks and set appropriate values for the `binarytask requirement network` (Section 3.5.25) and `binarytask truth network` (Section 3.5.26).

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="binarytask"
         id="binarytask assignment network"
         link_type="bool" network_type="dense">
   <generator type="randombinary">
```

```
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0"
            last="nodeclass::binarytask::count-1"/>
      <param name="mean" value="0"/>
   </generator>
</network>
```

### 3.5.25 Binary Task Requirements

The `binarytask requirement network` specifies which knowledge bits are examined when an agent attempts to complete a binary task. Agents assigned to binary tasks will be able to perform them each time period, and in so doing will examine the subset of their knowledge as specified by the binary task requirement network. For each required knowledge bit, if the agent's knowledge value is not equal to the value specified in the `binarytask truth network` (Section 3.5.26), the agent will have to guess and have a chance of completing the task incorrectly. The binary task completion algorithm is described in Section 3.4.4.

The `binarytask requirement network` is used by the standard interaction model (Section 3.3.9). It is a knowledge x binarytask network, meaning that task requirements may differ by task. However, task requirements cannot be changed once the simulation starts and must remain constant for the entire simulation. In the absence of scripting, it is not possible modify task requirements as the simulation evolves or as agents learn knowledge.

The binary task requirement network specifies which knowledge bits in the bool-valued `binary task truth network` (Section 3.5.26) are relevant to the task. Since binary tasks may require that the agents know particular facts (have truth value 1) or not know specific facts (have truth value 0), it is not possible to represent the value "not applicable". Thus, the binary task requirement network is provided in order to filter out irrelevant knowledge bits from the truth network. Note that the same set of knowledge bits must be required for all agents who perform the task; while two or more agents may perform a particular task, all agents must have the same task completion requirements.

Values in the `binarytask requirement network` are bools, and can be set using the values 1/0 or true/false. A value of 1 or true indicates that a knowledge bit is required to complete a binary task, while a value of 0 or false indicates that a bit is not.

In the demo input deck described in this technical report, knowledge is not required for binary tasks task so all values in this network are set to 0. This network is set using a `randombinary` generator, allowing a user to manipulate the mean value in order to examine what might occur if a random subset of knowledge were required for binary task completion. Since binary tasks are not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per agent group / binary task combination. It would also be necessary to add one or more binary tasks and set appropriate values for the `binarytask assignment network` (Section 3.5.24) and `binarytask truth network` (Section 3.5.26).

```
<network src_nodeclass_type="knowledge"
         target_nodeclass_type="binarytask"
         id="binarytask requirement network"
         link_type="bool" network_type="dense">
   <generator type="randombinary">
      <rows first="0"
            last="nodeclass::knowledge::count-1"/>
      <cols first="0"
            last="nodeclass::binarytask::count-1"/>
      <param name="mean" value="0"/>
   </generator>
</network>
```

### 3.5.26 Binary Task Truth

The `binarytask truth network` specifies what the values of the required bits must be for an agent to complete a task without guessing. Agents assigned to binary tasks will be able to perform them each time period, and in so doing will examine the subset of their knowledge as specified by the binary task requirement network and compare their knowledge against what is required by the task. For each required knowledge bit set in the `binarytask requirement network` (Section 3.5.25), if the agent's knowledge is not equal to the value specified in the truth network then the agent will have to guess and have a chance of completing the task incorrectly. The binary task completion algorithm is described in Section 3.4.4.

The `binarytask truth network` is used by the standard interaction model (Section 3.3.9). It is a knowledge x binarytask network, meaning that the truth values may differ by task. However, task truth cannot be changed once the simulation starts and must remain constant for the entire simulation. In the absence of scripting, it is not possible modify task truth as the simulation evolves or as agents learn knowledge.

The binary task truth network specifies the knowledge values that agents must have in order to avoid guessing when performing the task. If a value is set to true in the binary task truth network, then the agent must have a nonzero amount of knowledge of that bit in order to avoid guessing. If the agent does not have any knowledge of the bit, the agent will have to guess and thus will fail to be accurate 50% of the time. On the other hand, if the truth value is set to false, the agent will be accurate only if the performing agent has no knowledge of the fact and will have to guess if it has any knowledge, however small.

While most binarytask users will create scenarios in which all required binarytask knowledge bits must be known, there may be certain times when users may want to add bits that impede task performance (and therefore have a truth value of 0). Such a scenario can occur if the task performer has too much information, for instance, an overqualified candidate performing a menial task. Agents who learn too much may then actually perform the task less successfully, and their accuracy will suffer. However, most users will probably want to have all required binary task bits be associated with positive truth values, as knowledge should increase performance in most tasks.

Note that if a knowledge bit is not required to complete the task – i.e. not set as a required knowledge bit in the requirement network – then the value set in the truth network is irrelevant and will be ignored by the simulation. By convention, it is recommended that experiment designers set these ignored truth values to false.

Values in the `binarytask truth network` are bools, and can be set using the values 1/0 or true/false. A value of 1 or true indicates that an agent must have a nonzero knowledge value in order to complete the task without guessing, while a value of 0 or false indicates that the agent must not have any knowledge of the bit to avoid guessing. Note that it is not possible to specify a minimum knowledge threshold that an agent must know; if the agent has any amount of knowledge, the agent will be considered fully knowledgeable and will still have to guess if the truth value is zero for that bit.

In the demo input deck described in this technical report, binary tasks are not modeled so all values in this network are set to 0. This network is set using a `randombinary` generator, allowing a user to manipulate the mean value in order to examine what might occur if a random subset of knowledge were considered true for the purposes of binary task completion. Since binary tasks are not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per agent group / binary task combination. It would also be necessary to add one or more binary tasks and set appropriate values for the `binarytask assignment network` (Section 3.5.24) and `binarytask requirement network` (Section 3.5.25).

```
<network src_nodeclass_type="knowledge"
         target_nodeclass_type="binarytask"
         id="binarytask truth network"
         link_type="bool" network_type="dense">
   <generator type="randombinary">
      <rows first="0" last="nodeclass::knowledge::count-1"/>
      <cols first="0" last="nodeclass::binarytask::count-1"/>
      <param name="mean" value="0"/>
   </generator>
</network>
```

### 3.5.27 Knowledge Similarity Weight

The `knowledge similarity weight network` specifies how much weight agents place on shared knowledge when calculating probabilities of interaction. Knowledge similarity is a measure of perceived similarity between two agents, and is calculated by comparing an agent’s knowledge against its perception of another agent’s knowledge using transactive memory. The weight on knowledge similarity competes with the weight placed on knowledge expertise (Section 3.5.28) and shared tasks (Section 3.5.23) in order to comprise the variable portion of the probability of interaction.

The `knowledge similarity weight network` is used by the standard interaction model (Section 3.3.9). It is an agent x timeperiod network, meaning the weight placed on knowledge similarity can differ by timeperiod. Thus, agents can weight knowledge similarity more strongly in certain timeperiods while weighting it less strongly in others. However, the weights set in this network cannot be changed once the simulation starts. In the absence of scripting, it is not possible modify the knowledge similarity weight values as the simulation evolves or as agents learn knowledge.

**Figure 8: Similarity and expertise in Construct**

| | **Ego Perceives that Alter Doesn't Know** | **Ego Perceives that Alter Knows** |
|---|---|---|
| **Ego Doesn't Know** | No Effect | Increases Expertise |
| **Ego Knows** | No Effect | Increases Similarity |

As can be seen in Figure 8, bitwise knowledge similarity is increased when the ego knows a knowledge bit and perceives that an alter also knows the same knowledge bit. The alter's perceived knowledge, not the alter's actual knowledge, is crucial in computing bitwise knowledge similarity; if an ego perceives that the alter knows a particular knowledge bit, bitwise similarity will be increased regardless of whether or not the alter actually knows it. The increase amount will be equal to the agent's knowledge of the bit. If the agent knows the knowledge bit fully (knowledge value 1.0), the increase to similarity between the agents will be 1.0; if the agent only partially knows the knowledge bit, the increase will be somewhere between 0.0 and 1.0 depending on the level of the agent's knowledge. This increase amount is then multiplied by the value in the `interaction knowledge weight network` (Section 3.5.29), which serves as a metric of salience, in order to determine the contribution of this bit to similarity calculation, in order to determine the similarity contribution of this bit.

Net knowledge similarity is computed by summing knowledge similarity values over all bits shared by a particular agent pair. The net knowledge similarity is then multiplied by the value of the knowledge similarity weight in order to determine the how knowledge similarity contributes to the probability of interaction.

Values in the `knowledge similarity weight network` must be floats, and should be values between 0.0 and 1.0. A value of 1.0 means that agents will compute interaction probabilities using only knowledge similarity to the exclusion of expertise and shared tasks, while a value of 0.0 means that agents will ignore knowledge similarity in favor of other factors. Note that the sum of the values in the knowledge similarity weight network, knowledge expertise weight network, and binary task similarity weight network must be 1.0 for each agent and each timeperiod. If the values do not sum to one, the values are re-normalized such that their sum is one.

In the demo input deck described in this technical report, interaction is driven primarily by shared knowledge. By construction, 80% of the probability of interaction is due to shared knowledge, meaning that each agent has a value of 0.8 for the knowledge similarity weight during each time period. This guarantees that agents will primarily choose their interaction partners based on those that they perceive to have similar knowledge, but favor agents with more expertise (Section 3.5.28) assuming both potential partners have the same amount of shared knowledge. This network is set using a `randomuniform` generator, allowing a user to manipulate the minimum and maximum values in order to examine what might occur if some agents had different weights on shared knowledge. Note that this network is set using a single generator, so experimenters interested in examining changes to similarity weights for subsets of agents may consider using multiple generators and breaking this network up into subsections.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="timeperiod"
         id="knowledge similarity weight network"
         link_type="float" network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
           last="nodeclass::agent::count-1"/>
      <cols first="0"
           last="nodeclass::timeperiod::count-1"/>
      <param name="min" value="0.8"/>
      <param name="max" value="0.8"/>
   </generator>
</network>
```

### 3.5.28 Knowledge Expertise Weight

The `knowledge expertise weight network` specifies how much weight ego agents place on knowledge known only by the alter when calculating probabilities of interaction. Knowledge expertise is a measure of perceived expertise between two agents, and is calculated by comparing an agent's knowledge against its perception of another agent's knowledge using transactive memory. The weight on knowledge expertise competes with the weight placed on knowledge similarity (Section 3.5.27) and shared tasks (Section 3.5.23) in order to comprise the variable portion of the probability of interaction.

The `knowledge expertise weight network` is used by the standard interaction model (Section 3.3.9). It is an agent x timeperiod network, meaning the weight placed on knowledge expertise can differ by timeperiod. Thus, agents can weight knowledge expertise more strongly in certain timeperiods while weighting it less strongly in others. However, the weights set in this network cannot be changed once the simulation starts. In the absence of scripting, it is not possible modify the knowledge expertise weight values as the simulation evolves or as agents learn knowledge.

As can be seen in Figure 8, bitwise knowledge expertise is increased when the ego does not know a knowledge bit but perceives that an alter knows it. The alter's perceived knowledge, not the alter's actual knowledge, is crucial in computing bitwise knowledge expertise; if an ego perceives that the alter knows a particular knowledge bit, bitwise expertise will be increased regardless of whether or not the alter actually knows it. The increase amount is equal to the value in the `interaction knowledge weight network` (Section 3.5.29), which serves as a metric of salience. While knowledge similarity values (Section 3.5.27) also included a multiplicative effect for how much of the knowledge bit was known by the ego agent, this factor is omitted from expertise calculations. Since the ego knows nothing, it is assumed that salience is the only driving force for this particular bit.

Net knowledge expertise is computed by summing knowledge expertise values over all bits known by the alter but not by the ego. The net knowledge expertise is then multiplied by the value of the knowledge expertise weight in order to determine the how knowledge expertise contributes to the probability of interaction.

Values in the `knowledge expertise weight network` must be floats, and should be values between 0.0 and 1.0. A value of 1.0 means that agents will compute interaction probabilities using only knowledge expertise to the exclusion of similarity and shared tasks, while a value of 0.0 means that agents will ignore knowledge expertise in favor of other factors. Note that the sum of the values in the knowledge similarity weight network, knowledge expertise weight network, and binary task similarity weight network must be 1.0 for each agent and each timeperiod. If the values do not sum to one, the values are re-normalized such that their sum is one.

In the demo input deck described in this technical report, interaction is driven secondarily by shared knowledge (in contrast to shared knowledge, which is the primary driving factor). By construction, 20% of the probability of interaction is due to shared knowledge, meaning that each agent has a value of 0.2 for the knowledge expertise weight during each time period. This guarantees that agents will primarily choose their interaction partners based on those that they perceive to have similar knowledge, but favor agents with more expertise assuming both potential partners have the same amount of shared knowledge. This network is set using a `randomuniform` generator, allowing a user to manipulate the minimum and maximum values in order to examine what might occur if some agents had different weights on expertise. Note that this network is set using a single generator, so experimenters interested in examining changes to expertise weights for subsets of agents may consider using multiple generators and breaking this network up into subsections.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="timeperiod"
         id="knowledge expertise weight network"
         link_type="float" network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0"
            last="nodeclass::timeperiod::count-1"/>
      <param name="min" value="0.2"/>
      <param name="max" value="0.2"/>
   </generator>
</network>
```

### 3.5.29 Interaction Weight

The `interaction knowledge weight network` specifies how much weight agents will put on particular knowledge bits when computing probabilities of interaction with other agents. Some knowledge bits may be more important than others in driving interaction – they may reflect certain characteristics that are crucial for behavior – and therefore should be emphasized when agents seek interaction partners. This weight may be different than what is actually shared when agents interact, as is set in the `transmission knowledge weight network` (Section 3.5.30) or `knowledge priority network` (Section 3.5.31).

The `interaction knowledge weight network` is used by the standard interaction model (Section 3.3.9). It is an agent x knowledge network, meaning the weight placed on knowledge can differ by knowledge bit. This allows some agents to value a particular knowledge bit highly when choosing an interaction partner while others minimize the importance of that bit. The weights set in this network cannot be changed once the simulation starts and must remain constant for the entire simulation. In the absence of scripting, it is not possible modify the interaction weight values as the simulation evolves or as agents learn knowledge.

The interaction knowledge weight is a weight per knowledge bit. As agents and learn more knowledge and transactive memory, more knowledge bits will affect the calculation of the probability of interaction. Thus, the net contribution of each knowledge bit will diminish. If forgetting is not active, agents will only learn knowledge during the experiment; thus, the net contribution of each knowledge bit will diminish. Experiment designers seeking to set interaction weights should do so in a way that takes into account the fact that agent knowledge is dynamic.

Note that the interaction knowledge weight differs from the transmission knowledge weight. The interaction knowledge weight specifies the weight that an agent places on a knowledge bit for purposes of choosing an interaction partner, while the transmission knowledge weight (Section 3.5.30) specifies the weight an agent places on a knowledge bit when choosing knowledge to send to an interaction partner. The two weights are active at very different points of the Construct cycle, and can be used in order to enhance input deck design. For instance, interaction knowledge weights can be set very high on 'social facts' than bring agents together, while transmission weights can be set very high on 'key' facts whose diffusion may be of interest. Thus, the differences between interaction and transmission knowledge can ensure that the factors that drive interaction are not what are shared when agents communicate.

While the concept of a `knowledge priority network` (Section 3.5.31) exists for message transmission, no such concept exists for calculating the probability of interaction. Knowledge priority has no effect on the calculation of probabilities of interaction. All knowledge bits, regardless of priority level, are considered when calculating the probability of interaction. Instead of knowledge priority, the interaction knowledge weights described here can be used to make some knowledge bits appear more important than others.

Values in the `interaction knowledge weight network` must be floats and can be any float value. A positively-weighted value will mean that this particular knowledge bit will increase the propensity for two agents to interact (whether through similarity or expertise) while a negatively-weighted knowledge bits will decrease it. Interaction knowledge weights of zero will indicate that the knowledge bit has no effect on similarity or expertise calculations. A knowledge bit with an interaction weight of 2.0 will contribute twice as much to similarity and expertise scores as a knowledge bit with an interaction weight 1.0, and four times as much as a knowledge bit with an interaction weight 0.5. While the interaction knowledge weight value is technically unbounded, users are strongly recommended to keep the largest and smallest values within two orders of magnitude. Additionally, while negative weights are possible, users should use such weights sparingly in order to ensure that the net probability of interaction between two agents is positive.

In the demo input deck described in this technical report, all knowledge bits are equally salient and therefore are given weight 1.0. Thus, no one knowledge bit or set of knowledge bits drive agents to interact with others. This network is set using a `randomuniform` generator, allowing a user to manipulate the minimum or maximum values in order to examine what might occur if the weight changed. Since the interaction knowledge rate is not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per agent group / knowledge group combination.

```
<network src_nodeclass_type="agent"
```

```
         target_nodeclass_type="knowledge"
         id="interaction knowledge weight network"
         link_type="float" network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
           last="nodeclass::agent::count-1"/>
      <cols first="0"
           last="nodeclass::knowledge::count-1"/>
      <param name="min" value="1.0"/>
      <param name="max" value="1.0"/>
   </generator>
</network>
```

## 3.5.30 Transmission Weight

The `transmission knowledge weight network` specifies how much weight agents will put on particular knowledge bits when sending a message to a chosen interaction partner. Some knowledge bits may be more important than others in message creation – they may be topics that agents frequently return to in conversation, or may be key factors whose diffusion is critical for the simulation – and therefore should be emphasized when agents communicate. This weight may be different than what agents consider important when seeking an interaction partner, as is specified in the `interaction knowledge weight network` (Section 3.5.29).

The `transmission knowledge weight network` is used by the standard interaction model (Section 3.3.9). It is an agent x knowledge network, meaning the weight placed on knowledge can differ by knowledge bit. This allows some agents to value a particular knowledge bit highly when sending a message while others minimize the importance of that bit. The weights set in this network cannot be changed once the simulation starts and must remain constant for the entire simulation. In the absence of scripting, it is not possible modify the transmission weight values as the simulation evolves or as agents learn knowledge.

The transmission knowledge weight is a weight per knowledge bit. As agents and learn more knowledge, more knowledge bits will be considered when the agent decides which knowledge bit to send. Thus, the net contribution of each knowledge bit will diminish. If forgetting is not active, agents will only learn knowledge during the experiment; thus, the net contribution of each knowledge bit will diminish. Experiment designers seeking to set transmission weights should do so in a way that takes into account the fact that agent knowledge is dynamic.

Note that the interaction knowledge weight differs from the transmission knowledge weight. The interaction knowledge weight specifies the weight that an agent places on a knowledge bit for purposes of choosing an interaction partner, while the transmission knowledge weight (Section 3.5.30) specifies the weight an agent places on a knowledge bit when choosing knowledge to send to an interaction partner. The two weights are active at very different points of the Construct cycle, and can be used in order to enhance input deck design. For instance, interaction knowledge weights can be set very high on ‘social facts’ than bring agents together, while transmission weights can be set very high on ‘key’ facts whose diffusion may be of interest. Thus, the differences between interaction and transmission knowledge can ensure that the factors that drive interaction are not what are shared when agents communicate.

The knowledge transmission weight specified in this network is independent of the priority levels specified in the `knowledge priority network` (Section 3.5.31). While knowledge bits with high transmission weights are more likely to be chosen than knowledge bits with low transmission weights, such a result is by no means guaranteed. In contrast, all knowledge bits with a higher priority level must be chosen before any knowledge bits in a lower priority level are selected for transmission. Within each priority level, transmission weights can be used to make certain knowledge bits more likely to be selected for transmission; however, even the highest-weighted knowledge bit in a lower priority level will not be transmitted until all knowledge bits at the higher level have been selected for inclusion in a message.

Values in the `transmission knowledge weight network` must be floats and can be any nonnegative float value. Positive transmission knowledge weights will make a knowledge bit increasingly likely be selected for transmission, while a transmission knowledge weight of zero will ensure that the agent will never select the bit. A knowledge bit with an transmission weight of 2.0 will be sent twice as frequently as a knowledge bit with an transmission weight 1.0, and four times as frequently as a knowledge bit with transmission weight 0.5. While the transmission knowledge weight value is technically unbounded, users are strongly recommended to keep the largest and smallest values within two orders of magnitude. In contrast to the interaction knowledge weight, negative transmission knowledge weights are not possible and should be avoided.

In the demo input deck described in this technical report, all knowledge bits are equally salient and therefore are given weight 1.0. Thus, no one knowledge bit or set of knowledge bits is more likely to be shared than another. This network is set using a `randomuniform` generator, allowing a user to manipulate the minimum or maximum values in order to examine what might occur if the weight changed. Since the transmission knowledge rate is not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per agent group / knowledge group combination.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="knowledge"
         id="transmission knowledge weight network"
         link_type="float" network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
           last="nodeclass::agent::count-1"/>
      <cols first="0"
           last="nodeclass::knowledge::count-1"/>
      <param name="min" value="1.0"/>
      <param name="max" value="1.0"/>
   </generator>
</network>
```

### 3.5.31 Knowledge Priority

The `knowledge priority network` specifies the priority level of a particular fact when building a message. Some knowledge bits may be extremely important in message creation – such bits should always be present in messages – and therefore should be emphasized when agents communicate. This priority level may be independent of what agents consider important when seeking an interaction partner, as specified in the `interaction knowledge weight network` (Section 3.5.29). It is related to the concept of the `transmission knowledge weight network` (Section 3.5.30), but has important differences.

The `knowledge priority network` is used by the standard interaction model (Section 3.3.9). It is an agent x knowledge network, meaning the knowledge priority can differ by agent. This allows some agents to prioritize a particular knowledge bit when sending a message while others do not. The priorities set in this network cannot be changed once the simulation starts and must remain constant for the entire simulation. In the absence of scripting, it is not possible modify the priority values as the simulation evolves or as agents learn knowledge.

The knowledge priority specified in this network is independent of the transmission weights specified in the `transmission knowledge weight network` (Section 3.5.30). While knowledge bits with high transmission weights are more likely to be chosen than knowledge bits with low transmission weights, such a result is by no means guaranteed. In contrast, all knowledge bits with a higher priority level must be chosen before any knowledge bits in a lower priority level are selected for transmission. Within each priority level, transmission weights can be used to make certain knowledge bits more likely to be selected for transmission; however, even the highest-weighted knowledge bit in a lower priority level will not be transmitted until all knowledge bits at the higher level have been selected for inclusion in a message.

Note that knowledge priority levels have a complex interplay with message length and the percentage of the message that will be knowledge bits. Message lengths are specified on a per-agent basis in the `agent message complexity network` (Section 3.5.4), while the number of knowledge bits is specified in the simulation parameter `communicationWeightForFact` (Section 3.3.7). A message can only include knowledge bits from a lower knowledge priority if the message length is long enough. If a message is too short or insufficient knowledge bits are included in the message, then messages will be created using only knowledge bits from the highest priority level. Unless this behavior is desired, users should verify that the message length is sufficiently long – or the number of knowledge bits at the highest priority level is set to a small enough level – in order to ensure that knowledge bits at the lower priority levels are included as well.

Values in the `knowledge priority network` must be positive ints. Knowledge bits with higher priorities will always be included in messages before messages with lower priorities; agents will only consider sending a message with priority level 1 after a message has been constructed containing all bits with priority level 2. Negative or zero priority levels should be avoided.

In the demo input deck described in this technical report, all knowledge bits have the same priority level, 1.0. Thus, no one knowledge bit or set of knowledge bits is more likely to be shared than another. This network is set using a `randomuniform` generator, allowing a user to manipulate the maximum values in order to examine what might occur if the weight changed. Since the knowledge priority is not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per agent group / knowledge group combination.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="knowledge"
         id="knowledge priority network"
         link_type="int" network_type="dense">
   <generator type="randomuniform">
      <rows first="0" last="nodeclass::agent::count-1"/>
      <cols first="0" last="nodeclass::knowledge::count-1"/>
      <param name="min" value="1.0"/>
      <param name="max" value="1.0"/>
   </generator>
</network>
```

### 3.5.32 Learnable Knowledge

The `learnable knowledge network` specifies which agents are able to learn what knowledge bits. Experiment designers may wish for some knowledge bits to be restricted to some subset of agents, and therefore would want to prevent other agents from learning such knowledge bits. The learnable knowledge network can prevent an agent from ever learning the knowledge bit. While learning from other agents can be mimicked by setting a rate of 0.0 in the `agent learning rate network` (Section 3.5.8), agents may still be able to learn knowledge by performing tasks via the `learn by doing network` (Section 3.5.10); however, marking a knowledge bit as unlearnable in the `learnable knowledge network` will guarantee that a knowledge bit can never be learned by an agent.

The `learnable knowledge network` is used by the standard interaction model (Section 3.3.9). It is an agent x knowledge network, meaning that which facts are learnable can vary by agent. However, in the absence of scripting, it is not possible modify this network over time or to change what knowledge bits are learnable once other knowledge bits are learned.

In most cases, if a knowledge bit is marked as unlearnable an agent will initially not have knowledge of it (as specified in the `knowledge network`, Section 3.5.11). In such cases, the agent will not be able to learn the knowledge bit. In some cases, however, a knowledge bit will not be considered learnable but the bit will be set in the `knowledge network`. Neither network will be changed, as Construct will assume that such behavior is intentionally desired by the simulation designer. This bit will be treated as all other bits for purposes of interaction partner selection and message creation. However, should the agent lose the bit via forgetting (Section 3.3.5), the agent will not be able to reacquire the bit through communication or performing a task.

Values in the `learnable knowledge network` are bools, and can be set using the values 1/0 or true/false. A value of 1 or true indicates that an agent can learn the knowledge bit during the simulation, while a value of 0 or false indicates that an agent cannot learn it.

In the demo input deck described in this technical report, agents are able to learn all knowledge bits. Thus, all knowledge is marked as learnable (with a value of 1.0) for all agents. This network is set using a `randomuniform` generator, allowing a user to manipulate the minimum value in order to examine what might occur if some agents were unable to learn certain types of knowledge. Since selective attention is not considered an important part of this example, this parameter is not set serially; instead, it is set for the entire network using a single generator. Users wishing to investigate the effect of this network on the demo input deck might decide that it would be appropriate to have multiple generators, one per agent group / knowledge group combination.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="knowledge"
         id="learnable knowledge network"
         link_type="bool" network_type="dense">
   <generator type="randomuniform">
      <rows first="0"
            last="nodeclass::agent::count-1"/>
      <cols first="0"
            last="nodeclass::knowledge::count-1"/>
      <param name="min" value="1.0"/>
      <param name="max" value="1.0"/>
   </generator>
</network>
```

### 3.5.33 Agent Group Membership

The `agent group membership network` is used to identify related sets of agents. This is intended to make statistical analysis and scripting easier, as some properties of agent groups can be easily computed within Construct. As these groups are intended for statistical tracking as well as logical reasoning, the presence or absence of certain types of groups will have no effect on how the simulation evolves. However, the proper use of such groups can greatly simplify data analysis.

The `agent group membership network` is used by the core Construct engine and must be present no matter what models are used. The network is an agent x agentgroup network containing Boolean values. Values are true if the agent is a member of the particular agent group and false otherwise. Agent group assignment must be fixed at the start of the simulation and cannot be changed dynamically at this time.

While nodes must be unique within their nodeclasses, nodes can be grouped in multiple ways. For instance, while agent nodes must be unique within the agent nodeclass, agents can be grouped in multiple ways. In the demo input deck, it would be possible to group the agents in agent group A into one agent group, but also group the agents in agent groups A and C jointly in another agent group. A name for such a group might be the “all agents who can interact with A-group agents” group; for clarity of presentation, however, this was not done.

Values in the `agent group membership network` are bools, and can be set using the values 1/0 or true/false. A value of 1 or true indicates that an agent is a member of the particular group, while a value of 0 or false indicates that an agent is not.

In the demo input deck described in this technical report, agents are assigned to unique agent groups. In the input file, the agents in agent group A are set as group 0, then B as group 1, then C as group 2. Note that the network is initialized to contain 0 values, so any values not explicitly set to 1 have value 0.

```
<network src_nodeclass_type="agent"
         target_nodeclass_type="agentgroup"
         id="agent group membership network"
         link_type="bool" network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
             last="construct::intvar::agentgroup_A_end"/>
      <cols first="0"
            last="0"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
            last="construct::intvar::agentgroup_B_end"/>
      <cols first="1"
            last="1"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
            last="construct::intvar::agentgroup_C_end"/>
      <cols first="2"
            last="2"/>
      <param name="constant_value" value="1"/>
   </generator>
</network>
```

### 3.5.34 Knowledge Group Membership

The `knowledge group membership network` is used to identify related sets of knowledge bits. This is intended to make statistical analysis and scripting easier, as some properties of knowledge groups can be easily computed within Construct. As these groups are intended for statistical tracking as well as logical reasoning, the presence or absence of certain types of groups will have no effect on how the simulation evolves. However, the proper use of such groups can greatly simplify data analysis.

The `knowledge group membership network` is used by the core Construct engine and must be present no matter what models are used. The network is a knowledge x knowledgegroup network containing Boolean values. Values are true if the knowledge bit is a member of the particular knowledge group and false otherwise. Knowledge group assignment must be fixed at the start of the simulation and cannot be changed dynamically at this time.

Values in the `knowledge group membership network` are bools, and can be set using the values 1/0 or true/false. A value of 1 or true indicates that a knowledge bit is a member of the particular group, while a value of 0 or false indicates that a knowledge bit is not.

In the demo input deck described in this technical report, knowledge bits are assigned to unique knowledge groups. In the input file, the knowledge bits in knowledge group K1 are set as group 0, then K2 as group 1. Note that the network is initialized to contain 0 values, so any values not explicitly set to 1 have value 0.

```
<network src_nodeclass_type="knowledge"
         target_nodeclass_type="knowledgegroup"
         id="knowledge group membership network"
         link_type="bool" network_type="dense">
   <generator type="constant">
      <rows first="construct::intvar::knowledgegroup_K1_start"
           last="construct::intvar::knowledgegroup_K1_end"/>
      <cols first="0" last="0"/>
      <param name="constant_value" value="1"/>
   </generator>
   <generator type="constant">
      <rows first="construct::intvar::knowledgegroup_K2_start"
            last="construct::intvar::knowledgegroup_K2_end"/>
      <cols first="1" last="1"/>
      <param name="constant_value" value="1"/>
   </generator>
</network>
```

### 3.5.35 Other Networks (not present in the file)

A number of networks are omitted from the demo input file. These networks are not used in the demo for one of several reasons. For instance, the networks related to energy tasks are not present in the input deck because no energy tasks are modeled (see Section 3.4.5), networks affecting the literacy and access mechanisms (see Section 3.3.10), and networks dealing with mail messages have been omitted since all agents use direct communication (see Section 3.3.6 and Section 3.4.10). While a brief description of the networks are described here, some of these networks are described elsewhere, such as CMU-ISR-07-107 or CMU-ISR-07-116 (Hirshman & Carley 2007a; b) and users are strongly encouraged to seek additional information on these networks and their use in various models from members of the CASOS center.

- Energy task assignment network. The `energy task assignment network` is an agent by energytask network which specifies which energy tasks to which an agent is assigned. This is a Boolean network, where a value of true means that a particular agent is assigned to an energy task. Agents can only attempt to complete energytasks that they are assigned to. Agents who are not assigned to energytasks cannot complete them.
- Energy task requirement network. The `energy task requirement network` is an agent by energytask network which specifies how much energy an agent must expend required in order to complete instances of this energy task. This is a float network, where the value in the network specifies how much energy an agent must expend in order to complete the task instance. Agents will continue to expend energy until the task is completed; agents will expend energy equally on all active task instances that they have. Note that the energy task mechanism is independent of knowledge, and that there is no “truth” mechanism as there is for binary tasks (see Section 3.5.26). The two subsystems are completely separate.
- Energy task time network. The `energy task time network` is an energytask by timeperiod network which specifies when instances of the energy task are spawned. During the specific time period, an energy task instance is spawned for all agents assigned to that task to complete. Thus, unlike binary tasks (which are completed

every period), energy tasks only occur at particular points in time. If an energy task is not completed during the first period, however, it persists; thus, it is possible that there can be a backlog of multiple instances of energy tasks available for an agent to complete.

- Literacy network. The `literacy network` is a boolean agent by dummy_nodeclass network which specifies whether or not each agent is literate, and becomes a required network if the literacy mechanism is active (see Section 3.3.10). If the value in this network is true for an agent, then the agent is literate and will not be affected when communicating with agents that require literacy. If the value is false, however, then the agent may receive truncated or incorrect information when communicating with an agent that requires literacy. Truncation occurs by shortening the message by removing each message element with the probability specified by the `literacy_coverage_mean` simulation parameter, a parameter between 0.0 and 1.0 which specifies how much of the original message never reaches the receiver. Misinformation occurs by randomly dropping each non-truncated fact in the message (and replacing it with a random fact that is of no relation to what the sender intended) with a probability specified by the `literacy_accuracy_mean` simulation parameter; this accuracy parameter is also a value between 0.0 and 1.0. Note that if the sender does not require literacy, as specified in the `literacy requirement network`, then the value in this `literacy network` will not affect communication.
- Literacy requirement network. The `literacy requirement network` is a boolean agent by dummy_nodeclass network which specifies whether an agent receiving communication from this agent must be literate in order to receive a full message. It becomes a required network if the literacy mechanism is active (see Section 3.3.10). If the value in this network is true for an agent, then the message that is communicated will be in written form – as would occur with a book or website – and illiterate agents will not receive the full message. If the value in this network is false, the literacy value is ignored as communication is assumed to be face to face.
- Mail check probability network. The `mail check probability network` is an agent by dummy_nodeclass network specifying the time periods at which agents will check their mail. If the mail subsystem is active (Section 3.3.6) and some agents have an agent type that allows them to send mail messages (Section 3.4.10), the mail check probability will allow receiving agents to receive these agents with some probability. At the end of every time period, after agents have initiated and received standard communication, agents will have the opportunity to check their mailbox, which they will do with the probability specified in this float network. If the agent checks its mail, it receives any messages in its mailbox; this means that an agent can potentially receive multiple mail messages at once. Learning from this mail message occurs as with standard face-to-face communication – for instance, agents can be forbidden from learning certain facts in the mail message (Section 3.5.32) – with the additional caveat that all message passed through mail will require agents to be literate in order to comprehend fully.
- Mail time to live network. The mail time to live network is as agent by dummy nodeclass network specifying how long a message will stay in this agent's mailbox before being deleted. If the mail subsystem is active (Section 3.3.6) and some agents

have an agent type that allows them to send mail messages (Section 3.4.10), the mail check probability will allow receiving agents to receive these agents with some probability. However, if agents do not check their mail within a certain period of time, it is assumed that the mail is cleared and this message is discarded. This is an integer network that specifies the number of periods in which a message will remain in the mailbox; a value of 4, for instance, will mean that a mail message sent to this agent will be available for four time periods before being deleted. Note that at the current time, the mail message's time to live is independent of the sender and the content of the message.

While there are several other different types of networks – for instance, networks dealing with the control of the dynamic environment (Section 3.3.3) – are also available, these networks are beyond the scope of this technical report.

It is also important to note that users can create custom networks. These networks can contain the default nodeclasses, as briefly introduced in Section 3.4.11, or can be new networks relating the base nodeclasses. For instance, a custom network relating the knowledge and timeperiod nodeclasses could be an "interaction relevance" network, a network that could make certain knowledge bits more important to all agents at a particular time period. In order to specify a new network such as this relevance network, the network name, source network nodeclass, target network nodeclass, and link type must all be specified by the designer. The network type, at least in Construct version 3.8.build016BRH, must remain `dense`. In this particular example, the name might be `interaction relevance network`, the source type `knowledge`, the target type `timeperiod`, and the link type `float` in order to represent different importance levels. The resulting ConstructML for the network will look similar to all of the networks described previously in this section, and any appropriate generator type can be used for creating the network. This example is demonstrated below.

```
<network src_nodeclass_type="knowledge"
         target_nodeclass_type="timeperiod"
         id="interaction relevance network"
         link_type="float" network_type="dense">
  <generator type="constant">
    <rows first="0" last="nodeclass::knowledge::count-1"/>
    <cols first="0" last="nodeclass::timeperiod::count-1"/>
    <param name="constant_value" value="1"/>
  </generator>
</network>
```

While additional networks like the hypothetical "interaction relevance" network can be created, they will not be used by Construct's core algorithms unless they have been previously specified in this technical report. Construct and its active models will expect certain types of networks, and will ignore networks that are not pre-specified in the executable. Nevertheless, user-generated networks can be important in one of two ways. First, it will be possible in future versions of Construct to write new models (Section 3.3.9) that Construct will use instead of the standard models. These models can make use of custom networks for special purposes, such as new types of similarity that affect which agents are likely to communicate with which. Second, it is possible to script behavior (using the Construct scripting system described in CMU-ISR-09-

126) in order to allow Construct to use these networks for input, output, and calculation purposes. Using the `ReadDecisionOutput` operation or the `lexer_based` network generator, it is possible to allow Construct to perform certain types of calculations using different types of networks. Both of these use cases are further described in CMU-ISR-09-126 (Hirshman et al 2009).

## *3.6 Transactive Memory*

Transactive memory networks are a second type of data storage mechanism in Construct. While networks store relations among nodeclasses, transactive memory networks store perceptions of networks. Networks, as discussed in Section 3.5, are assumed to be relationships between two nodeclasses. Transactive memory networks, on the other hand, are meant to store not just networks but each ego agent's perception of a network. Thus, while networks in Construct are two dimensional (or one dimensional if the dummy_nodeclass is in use), transactive memory data structures have three dimensions: the ego nodeclass and the two nodeclasses that are related in the perceived network.

As introduced in Section 3.4 and discussed further in Section 3.5, Construct uses the idea of "nodes" and "networks" in contrast to other agent-based simulation systems. Transactive memory networks, however, are specialized networks designed to hold an agent's perception of a network. The term "transactive memory" originated in group processes literature, and was first used extensively by Wegner (1986). As initially defined, the transactive memory system was the system in which groups encode, store, and retrieve knowledge. In this sense, it was the group's collective perception of the specialties and skills of individuals in the group. However, Construct uses the term slightly differently. In Construct, transactive memory serves as each agent's perception of the knowledge and skills of each other possible interaction partner. Perception is often imperfect and thus an agent's perception of a different agent often differs from that agent's actual skills. It is thus a type of meta-cognitive network (Metcalfe & Shimamura 1994).

It may be reasonable to ask: why are transactive memory networks necessary? In short, transactive memory is what sets Construct apart from other types of social simulation mechanisms. Agents in Construct are boundedly rational, meaning that they cannot perceive their surroundings accurately (Carley & Newell 1994; Simon 1957). While agents are socially bounded by their interaction spheres (Section 3.5.14) and access networks (Section 3.5.15), which restrict the types of agents with whom they can interact, agents also have imperfect perceptions of the knowledge of others. Thus, when agents compute their probabilities of interactions with others, they use their perceptions of the world – which may or may not match the underlying reality – in order to inform their decision. When agents compute the effect of social influence on their beliefs, they use their perceptions of others' beliefs – and which again may or may not match the underlying reality – in order to inform their decisions. Both of these perceptions are only possible due to the presence of transactive memory, which relates three nodeclasses: the egos (the ones who perceive), the source nodes (in both of the above cases, the alters that are perceived), and the targets (knowledge and belief, respectively). While more complex than the standard networks discussed in Section 3.5, transactive memory networks allow Construct agents to better match social theory and real-world behaviors.

**Figure 9: Key transactive memory networks in the demo input deck**

| Network Name | Source & Target Nodeclasses | Function or Purpose in Demo Input Deck |
|---|---|---|
| **knowledge transactive memory network** | agent x timeperiod | store agent perceptions of other agents' knowledge |
| **belief transactive memory network** | agent x timeperiod | store agent perceptions of other agents' beliefs |

A list of all the transactive memory networks included in the demo input deck, including the (non-ego) nodeclasses that are related by the networks, is provided in Figure 7. While it is possible for the user to create new transactive memory networks or to create new networks that are not listed in Figure 7, such networks are not currently referenced by the Construct executable. Thus, such networks would have no effect on the evolution of the simulation. Though new networks can potentially be used by new models or by the scripting system available in Construct, such features should only be employed by advanced users.

Transactive memory networks in Construct are specified within a `<transactivememory>` ConstructML tag in the input deck, as demonstrated below.

```
<transactivememory>
  <!-- Put all transactive memory networks here -->
</transactivememory>
```

The `<network>` tag contains the definition for the different types of transactive memory networks that can play critical roles in the simulation. Each transactive memory tag must have seven attributes: the network id (name), ego nodeclass, source nodeclass, target nodeclass, link type, network type, and associated network, as seen below.

```
<network id="[name]"
    ego_nodeclass_type="agent" src_nodeclass_type="[nodeclass]"
    target_nodeclass_type="[nodeclass]" link_type="[ltype]"
    network_type="[ntype]" associated_network="[network]">
  <!–Set TM generators here -->
</network>
```

The network ID `[name]` is the name that refers to the transactive memory network. Models in Construct (Section 3.3.9) each require certain kinds of networks in order to function. The networks that the model calls for have a specific role in the simulation. Thus, the network id defines what the `ego_nodeclass_type`, `src_nodeclass_type`, `target_nodeclass_type`, `link_type`, and `network_type`, and `associated_network` should be in order for Construct to correctly understand the input provided. If the network ID is misspelled, Construct will not correctly interpret the input and will probably exit with an error (it will believe that the network is missing). Note that network ids are case-insensitive – internally, all network names are converted to lower case – and can contain spaces.

The transactive memory network, as currently implemented, makes a number of assumptions about the ego_agent_nodeclass. Most importantly, the ego nodeclass is assumed to be the agent nodeclass in the current version of Construct. This is because only agents in Construct are designed to have perceptions of other types of nodeclasses. While future versions of Construct may relax this assumption, for now all ego_agent_nodeclass attributes should be of type agent. Additionally, while certain agents – i.e., nodes in the agent nodeclass – may not use these perceptions to drive their interactions, they still are given entries in the transactive memory network. While this may be wasteful if there are a large number of agents who do not employ transactive memory, it simplifies a number of algorithms and greatly increases the simulation run speed. Future versions of Construct may revisit these assumptions and allow for more flexible implementations of transactive memory.

The src_nodeclass_type and target_nodeclass_type `[nodeclass]` indicate the nodeclasses that are related by the network that each ego perceives. Transactive memory networks in Construct can be thought of as a stack of relations between the source and target nodeclasses, one network per ego. For instance, the agent by agent by knowledge network, `knowledge transactive memory network` (Section 3.6.1), can be considered a stack of network relating each node in the agent nodeclass to each node in the knowledge nodeclass, one network per ego. Thus, the first ego node has an agent-by-knowledge network of perceived knowledge, the second ego node has an agent-by-knowledge network of perceived knowledge, and so forth.

The associated_network `network` is the network that is perceived in this transactive memory network. Transactive memory networks represent perceptions of other networks in the simulation; the `knowledge transactive memory network` (Section 3.6.1), for instance, holds each agent's perception of the `agent knowledge network` (Section 3.5.11). Specifying the associated network can be crucial for helping to initialize the network, especially in conjunction with the `perception_based` generator. Additionally, future versions of Construct may use the associated network in order to help minimize the amount of space necessary to store the entire transactive memory network.

The network_type `ntype` specifies the storage mechanism for transactive memory network. Currently, there are two different storage mechanisms for the transactive memory network. The `TMBool` mechanism will store Boolean values. The `TMFloat` mechanism is a storage mechanism for specialized point values; specifically, it will store the values -1, 0, and +1. Both mechanisms are optimized in order to store very large amounts of data in as little space as possible. Even so, the sheer amount of space required to store transactive memory networks is often the primary reason why large amounts of RAM are necessary to run simulations with large numbers of agents.

The link_type parameter `ltype` defines the type of relation stored in this network. As of Construct version 3.8.build016BRH, there are two possible link types for transactive memory networks. Note that the link type of the transactive memory network need not be the same type as the associated_network; the link_type for the transactive memory network is often a design decision that is conscious of storage space as well as the necessary complexity of what must be stored.

- bool. Bool links are the Boolean values `true` or false (or `1` or `0` when represented numerically). Either representation can be used when specifying links in the network. If the link_type is Boolean, then the network type must be TMBool.

- float. Float links are stored as two-bit numbers representing the states -1, 0, and +1. If the link_type is float, then the network type must be TMFloat. Due to the large amount of space required to store a three-dimensional network of floating point values, the Construct design team made the decision to threshold float values to either -1, 0, or +1. This decreases the amount of memory necessary to store TMFloat networks by an order of magnitude. Any thresholding mechanism employed must be specified by the algorithm that uses this network.

Note that five of the above attributes – the id, source, target, link type, and network type – are attributes that are necessary for specifying a general network in the `<networks>` tag as discussed in Section 3.4. While the network types are different in transactive memory networks as compared to standard networks, the other parameters are very similar. This similarity is intentional, as transactive memory networks should be thought of as extensions to the standard networks that are used to initialize Construct. Each ego agent can be thought to have its own perception of key networks, and therefore will act according to their network-based perception of the world.

Transactive memory networks must be set using `<generator>` tags. While standard Construct networks can be set using `<link>` tags, which can be used to set a single link between two nodes, it is not possible to use the same syntax in order to relate the three nodes necessary to form a transactive memory link. This decision was made due to the cubic nature of the number of links that would have to be set using `<link>` tags: assuming that the number of agents is non-trivial, it would not be computationally- or space-efficient to use `<link>` tags to represent key links in the transactive memory network.

The syntax for a transactive memory generator is similar to that for a network, as described in Section 3.5. The syntax for a transactive memory generator is described below.

```
<generator type="[type]">
  <ego first="[efirst]" last="[elast]"/>
  <alter first="[afirst]" last="[alast]"/>
  <transactive first="[tfirst]" last="[tlast]"/>
  <param name="[name1]" value="[value1]"/>
  <param name="[name2]" value=="[value1]"/>
  ...
  <param name="verbose" value="[verbose]"/>
</generator>
```

The `[type]` attribute specifies the type of network that is to be generated. Note that many types of network generators that could be used to generate standard networks can be used to generate transactive memory networks. The `<ego>` child tag specifies the ego agent which stores the transactive memory, the `<alter>` tag describes the agent who is being perceived, and the `<transactive>` tag describes the entity that is being perceived. All three of these parameters are similar to the `<rows>` and `<cols>` tag of the standard network generators (Section 3.5); generation will be performed from the first value to the last value, inclusive. Various generator-specific `<param>`s must be supplied in order for the generator to function correctly. Lastly, a `<verbose>` parameter can be supplied in order to have Construct print out additional information about the generator as it is running. Specifically, the generator will print out a message every hundred agents it initialized, which can help users understand if Construct errors before or after a specific generator is executed.

The below example demonstrates how the constant value generator can be extended for use in generating transactive memory.

```
<generator type="constant">
  <ego first="[efirst]" last="[elast]"/>
  <alter first="[afirst]" last="[alast]"/>
  <transactive first="[tfirst]" last="[tlast]"/>
  <param name="constant_value" value="[value]"/>
</generator>
```

Note that this setup is very similar to other constant value generators, such as those used in generating the `agent belief network` (Section 3.5.12) in the demo input deck. The required parameter `constant_value` is the same in both networks, and will ensure that the region of the network is set to the same constant value. Note, however, that instead of `<rows>` and `<cols>` there are `<ego>`, `<alter>`, and `<transactive>` tags. This guarantees that a three-dimensional region of the network will be generated.

While generators for transactive memory networks have been extended into three dimensions, users may wish to hold some of these dimensions constant and only set values in particular subregions. For instance, users may wish to set to set all values for a particular ego agent; to do so, the ego's `first` and `last` attributes should be set to the index of the ego agent of interest while regions are specified for the `<alter>` and `<transactive>` tags. Similarly, if all agents' perceptions of one alter agent should be special, the alter can be held constant by setting the alter `first` and `last` values to the relevant index while varying indices specified in the `<ego>` and `<transactive>` tags. It is even possible to hold multiple tags constant when setting a value – for instance, to set an ego's perception of a particular alter – or to hold all values constant – to set an ego's perception of a particular alter's specific value.

Some generators have been specifically developed for transactive memory networks. One particular generator that is used in the demo input deck is the `perception_based` generator, below.

```
<generator type="perception_based">
  <ego first="[efirst]" last="[elast]"/>
  <alter first="[afirst]" last="[alast]"/>
  <transactive first="[tfirst]" last="[tlast]"/>
  <param name="false_negative_rate" value="[fnrate]"/>
  <param name="false_positive_rate" value="[fprate]"/>
  <param name="rounding_threshold" value="[threshold]"/>
  <param name="verbose" value="[verbose]"/>
</generator>
```

The perception_based generator will generate a network based on the `associated_network` of the transactive memory network. The `associated_network` attribute is an attribute of the original `<network>` tag which contains the generator as its child; for instance, the associated attribute of the `knowledge transactive memory network` (Section 3.6.1) is the `knowledge network` (Section 3.5.11). The `perception_based` generator will generate perception values based on the underlying reality specified in the appropriate associated network. This perception need not be completely accurate. The `false_negative_rate` and `false_positive_rate`, as well as the `rounding_threshold`, specify the kinds of perception errors that will be introduced by this generator; however, they have nuanced meanings depending on the type of network (TMBool versus TMFloat) and will be explained as they are used in the demo input deck.

Because there can be multiple types of generators inside a transactive memory network, it is possible to have a particular transactive memory value specified zero, one, or multiple times. For this reason, tags inside the network tag are read one after the other, starting at the top of the `<network>` tag and continuing until the terminal `</network>` tag. The value of the agent-node-node link will be the value specified by the last link or generator read. If the value is specified multiple times in other `<generator>` tags, the earlier generator values are overwritten and replaced with the later version. On the other hand, if the value is not specified in any `<generator>` tag, then the value will default to `false` in TMBool networks or `0` in TMFloat networks.

It is important to note that after all generators have finished running, Construct will run an internal generator which will guarantee that agents have perfect transactive memory of themselves. This means, for example, the last step in the initialization of the `knowledge transactive memory network` (Section 3.6.1) will guarantee that an agent will correctly perceive that it has knowledge of all the knowledge bits that it knows in the `knowledge network` (Section 3.5.11). A similar process will also occur in the initialization of belief transactive memory (Section 3.6.2) which holds transactive memory about the belief network (Section 3.5.12). Such a step is necessary in order to ensure the proper function of several of the Construct algorithms, which iterate over all transactive memory known by a particular agent. From the user's perspective, it is important to note that it will not be necessary to create special generators to set an agent's perception of itself.

### 3.6.1 Knowledge transactive memory

The `knowledge transactive memory network` specifies an ego agent's perceptions of other agent's knowledge. While an agent's actual knowledge is specified in the `agent knowledge network` (Section 3.5.11) other agents may misperceive this knowledge. Agents will use their perceptions of others' knowledge when calculating probabilities of interaction. For this reason, it is necessary for each agent to store knowledge transactive memory for each agent in its interaction sphere. In conjunction with the other networks required in the standard interaction model (Section 3.3.9), agents will use their perceptions of others in order to determine interaction partners as the simulation evolves.

The `knowledge transactive memory network` is used by the standard interaction models (Section 3.3.9). It is an agent x agent x knowledge network, meaning that perceptions can be different by ego agent, alter agent, and knowledge bit. Note that once initialized, it is not possible to modify knowledge transactive memory via scripting.

Knowledge transactive memory stores perceptions of knowledge. As such, it can be accurate or inaccurate: ego agents are able to correctly perceive an alter agent's knowledge or ignorance, to believe that an alter knows something when it does not, or to believe an alter does not know something when it does. There are several reasons why an alter may have an incorrect knowledge. First, an alter may have incorrect knowledge due to the way that knowledge was initialized. If the knowledge transactive memory is initialized with deliberate errors, then agents may not correctly perceive the knowledge of those in their interaction spheres. Second, alters may misperceive an alter’s knowledge through some failure when communicating. This may occur due to a miscommunication between agent and alter, or due to one or more of the Construct mechanisms that are active (Section 3.3.10). Third, an agent may receive incorrect information from a third party or chain of third parties. Thus, while the ego correctly receives the information from the third party, the third party’s information may be faulty. Note that the agent will never perceive this information to be faulty, as the agent does not know the underlying true values stored in the `knowledge network`.

When agents learn new information when interacting, they will update their transactive memory to reflect the new knowledge that they have learned. Agents who learn about the knowledge of third parties from another agent will always overwrite any information they previously knew with the information coming from the other agent. Agents are thus assumed to trust other agents completely, and to always assume that alter agents are more up-to-date than they are. The one exception to this is that ego agents will always perceive their own knowledge correctly. Such perceptions of an ego’s own belief are set when initializing transactive memory (Section 3.6), and will be constantly updated by the relevant algorithms that modify an ego agent’s knowledge.

Values in the `knowledge transactive memory network` are TMBools, and can be set using the two values 0 and 1. A value of 0 indicates that the ego thinks that the alter has no knowledge of a particular knowledge bit, while a value of 1 indicates that the ego thinks that the alter has knowledge of a particular knowledge bit. Note that these values may or may not correspond with what the alter actually knows

The `perception_based` generator for knowledge transactive operates on a TMBool network. Because the data being stored in the network is two-valued – only 0s and 1s are stored in this network – the parameters to the `perception_based` generator have relatively straightforward interpretations. The `rounding_threshold` parameter is used to round values coming from the associated network (in this case, the `knowledge network` of Section 3.5.11). Values that are below the rounding threshold are rounded down to 0, while values above the rounding threshold are rounded up to 1. At this point, misperception can occur. The `false_negative_rate` specifies how frequently ego agents will misperceive a 1 as a 0, erroneously failing to perceive that an alter has knowledge of a particular knowledge bit. The `false_positive_rate` specifies how frequently ego agents will misperceive a 0 as a 1, erroneously thinking that an agent has knowledge of a particular knowledge bit.

In the demo input deck described in this technical report, the `knowledge transactive memory network` is initialized using a perception-based generator. The settings in the input deck call for a rounding threshold at 0.0, meaning that only values of 0.0 will be perceived as no knowledge. The false negative rate is 50%, meaning that half of all knowledge held by alters will not be perceived correctly by the egos. The false positive rate is 0%, meaning that egos will correctly perceive that alters do not know knowledge bits that they lack. As each transactive memory entry is set independently for each agent and each alter, all false negative errors will be uncorrelated.

```
<transactivememory>
```

```
<network id="knowledge transactive memory network"
        ego_nodeclass_type="agent" src_nodeclass_type="agent"
        target_nodeclass_type="knowledge" link_type="bool"
        network_type="TMBool"
        associated_network="knowledge network">
  <generator type="perception_based">
    <ego first="0" last="nodeclass::agent::count-1"/>
    <alter first="0" last="nodeclass::agent::count-1"/>
    <transactive first="0"
                 last="nodeclass::knowledge::count-1"/>
    <param name="false_negative_rate" value="0.50"/>
    <param name="false_positive_rate" value="0.0"/>
    <param name="rounding_threshold" value="0.0"/>
    <param name="verbose" value="false"/>
  </generator>
</network>
```

### 3.6.2 Belief transactive memory

The `belief transactive memory network` specifies an ego agent's perceptions of other agent's beliefs. While an agent's actual beliefs are specified in the `agent belief network` (Section 3.5.12) other agents may misperceive these beliefs. Agents will use their perceptions of others' beliefs when computing their perceived social influence. For this reason, it is necessary for each agent to store belief transactive memory for each agent in its interaction sphere. In conjunction with the other networks required in the standard influence model (Section 3.3.9), agents will use their perceptions of others in order to update their beliefs as the simulation evolves.

The `belief transactive memory network` is used by the standard belief model and standard influence models (Section 3.3.9). It is an agent x agent x belief network, meaning that perceptions can be different by ego agent, alter agent, and belief. Note that once initialized, it is not possible to modify belief transactive memory via scripting.

Belief transactive memory stores perceptions of belief. As such, it can be accurate or inaccurate: ego agents are able to correctly perceive an alter agent's belief, to believe that an alter has a strong opinion when it really does not, to fail to believe an alter has a strong opinion when it really does, or to believe an alter holds the opposite opinion instead of what it really does. There are several reasons why an alter may have an incorrect opinion. First, an alter may have an incorrect opinion due to the way that belief was initialized. If the belief transactive memory is initialized with deliberate errors, then agents may not correctly perceive the beliefs of those in their interaction spheres. Second, an alter may learn an incorrect opinion by failing to correctly observe or learn an agent's belief. This may occur due to a miscommunication between agent and alter, or due to one or more of the Construct mechanisms that are active (Section 3.3.10). Third, an agent may receive incorrect information from a third party or chain of third parties. Thus, while the ego correctly receives the information from the third party, the third party's information may be faulty. Note that the agent will never perceive this information to be faulty, as the agent does not know the underlying true values stored in the `agent belief network`.

When the standard belief model is active and is acting in `mask_mode`, agents will update their transactive memory of other agents' belief. Agents who learn about the beliefs of third parties from another agent will always overwrite any information they previously knew with the information coming from the other agent. Agents are thus assumed to trust other agents completely, and to always assume that alter agents are more up-to-date than they are. The one exception to this is that ego agents will always perceive their own beliefs correctly. Such perceptions of an ego's own belief are set when initializing transactive memory (Section 3.6), and will be constantly updated by the relevant algorithms that modify an ego agent's belief.

Values in the `belief transactive memory network` are TMFloats, and can be set using the three values -1/0/+1. A value of -1 indicates that the ego believes that the alter opposes this belief, while a value of +1 indicates that the ego believes that the alter supports it. A value of 0 indicates that the ego believes that the alter does not have a strong opinion on this belief. Note that these values may or may not correspond with what the alter actually believes.

The `perception_based` generator for belief transactive memory works slightly differently than that for knowledge transactive memory. Because the belief transactive memory network is a TMFloat network with three values – -1, 0, +1 – several of the parameters to the generator must take on an expanded meaning. In the `perception_based` generator for TMFloat data, the `rounding_threshold` of the generator is used as an absolute value. Values whose absolute values are below this value are rounded to zero. This is because the agent belief network that is perceived may have data that is in the rage -1 to +1. Additionally, the false positive rate becomes the rate at which negative values are mistaken as positive ones, while the false negative rate becomes the rate at which positive values are mistaken for negative ones.

In the demo input deck described in this technical report, the `belief transactive memory network` is initialized using a perception-based generator similar to the one used to generate the knowledge transactive memory in Section 3.6.1. However, as no beliefs are modeled, this will have no net effect on the evolution of the simulation. Nevertheless, the network is set up in order to facilitate the addition of beliefs to the simulation. The settings in the input deck call for a rounding threshold of 0.0, meaning that only values of 0.0 will be perceived as no belief. The false negative and false positive rates are both 0.25, meaning that an average of 25% of all alters' beliefs will be misperceived by an ego agent. As each transactive memory entry is set independently for each agent and each alter, these errors will be uncorrelated.

```
<network id="belief transactive memory network"
         ego_nodeclass_type="agent" src_nodeclass_type="agent"
         target_nodeclass_type="belief" link_type="float"
         network_type="TMFloat"
         associated_network="belief network">
   <generator type="perception_based">
     <ego first="0" last="nodeclass::agent::count-1"/>
     <alter first="0" last="nodeclass::agent::count-1"/>
     <transactive first="0" last="nodeclass::belief::count-1"/>
     <param name="false_negative_rate" value="0.25"/>
     <param name="false_positive_rate" value="0.25"/>
     <param name="rounding_threshold" value="0.0"/>
     <param name="verbose" value="false"/>
   </generator>
 </network>
</transactivememory>
```

## 3.7 Operations

Operations are mechanisms to extract output from Construct. While the previous sections of this input deck have focused on how to provide input to the simulation, the operations section focuses on how to extract meaningful output as the simulation is running or after the simulation has completed. Such output can be used in order to debug the simulation, but more importantly it can be used to answer the research question that the experiment designer initially posited.

Construct provides three general classes of outputs, each with their own advantages and disadvantages.

- whole-network outputs. The most basic output in Construct is the whole-network output, which prints out a network at a particular period of time. For instance, users interested in understanding questions of information diffusion can print out the `knowledge network` (Section 3.5.11) at particular timeperiods in order to determine how the network has changed. The user has the option to print the entire network, or only specific rows and columns – i.e. specific nodes from the source nodeclass crossed by specific nodes from the column nodeclass – of that network. This network can then be read into post-processing scripts in order to answer specific research questions. The advantage of this approach is that the greatest amount of data is preserved: users who print out the entire knowledge network have access to the raw data used in Construct and not just summary statistics. This can help answer broad research questions (for instance, how did knowledge diffuse over time?), but can also allow for rapid drilldown into subquestions if desired (why did the diffusion of this knowledge bit differ from this one?). The disadvantage of this approach is that it often requires the development or use of tools to process large networks, which may be cumbersome to the experiment designer. Additionally, such networks can require large amounts of storage space, especially if many replications are performed. Last, experiment designers who use this method may suffer from information overload given the very large amount of data generated. These whole network outputs are described in Section 3.7.13.7.2.
- processed outputs. Construct is also able to provide a number of outputs that process network values. For instance, Construct can provide outputs that compute task accuracy, provide various metrics of information diffusion, and several other functions. The advantage of such outputs is that they can quickly summarize Construct data and do so with minimal overhead. Users wishing to compute these metrics can take advantage of built in summarizing routines in Construct in order to avoid printing out entire networks and writing post-processing scripts. However, these outputs are not always customizable: for instance, task accuracy will be computed on all tasks for all timeperiods, and information diffusion metrics will be calculated on for all knowledge bits as opposed to a relevant subsection. Users who wish to take advantage of such outputs should understand that certain assumptions are made by these types of outputs. Some of these processed outputs are described in Section 3.7.2.
- scripted outputs. The most complicated form of output, but potentially the most powerful output, is the scripted output. While the Construct scripting system can be used for specifying complex inputs, it can also be used to calculate complex functions

on Construct data while the simulation is running. The Construct scripting system will run a script once per agent in the simulation, allowing the experiment designer to calculate functions of agent knowledge, belief, and many other factors in order to come up with a desired output. While such functions can supplement the processed outputs described earlier, they can go further and actually modify the simulation data as Construct is running. The two main disadvantages of a scripting approach is the potential difficulty in understanding and debugging the scripting language, which may appear intimidating for beginning users. Additionally, it may be difficult to specify one's desired output in a way that is easy for the scripting language to run. While the scripting language is not described in this technical report, it is documented extensively in CMU-ISR-09-126, which describes the scripting system in Construct (Hirshman et al 2009).

Regardless of the type of operation used, operations in Construct are specified within a `<operations>` ConstructML tag in the input deck, as demonstrated below.

```
<operations>
  <!-- Put all operations here -->
</operations>
```

The `<operation>` tag contains the definition for the different types of operations that can provide output from the simulation. Operations have one attribute which specifies the type of operation in use, then a number of child tags which specify important properties of the operation. An example operation is shown below.

```
<operation name="[name]">
  <parameters>
    <param name="output_filename" value="[filename]"/>
    <param name="output_format" value="[csv or dynetml]"/>
    <param name="run" value="all"/>
    <param name="time" value="[first or last or timeperiod]"/>
    <param name="param1" value="[value1]"/>
    <param name="param2" value="[value2]"/>
    ...
  </parameters>
</operation>
```

The operation name `[name]` is required as it specifies the types of parameters that must be supplied to the operation. Most parameters to the operation are supplied as children to the child tag `<parameters>`, including the key parameters named `output_filename`, `output_format`, `run`, and `time`. These parameters are key for determining when the operation prints its output and the output that is printed.

The `output_filename` is the name of the file to create to store the output. The name of this output parameter must be specified with any applicable extensions, for instance "first.csv". By default, output will appear in the same directory where Construct is run; however, it is possible to prepend path information in order to store output in a different location. If this file already exists in the directory, the file will be overwritten without prompting. Thus, if Construct is run multiple times in the same directory, new data will overwrite the old. If the file cannot be created for some reason, Construct will exit and error (as discussed in Section 4.7).

The `output_format` must be either `csv` or `dynetml`. CSV is a comma-separated format that will separate values in the same row using commas. It is useful for importing Construct data into databases or for parsing with analysis scripts. DynetML is *ORA's data format that allows network data be quickly imported, visualized, and analyzed. Note that the csv data format is the only type that can be used to output data from multiple timeperiods.

The `run` param must be set to `all`. This parameter has been preserved for backwards compatibility with previous versions of Construct and should not be changed.

The time param specifies in which timeperiods output should be printed. Multiple values are valid for this parameter. The simplest possible value is the index of a single timeperiod, which is between 0 and the number of timeperiod nodes. Alternately, multiple timeperiods can be printed using a comma-separated list of timeperiods; to print the first three time periods, for instance, the value "`0,1,2`" can be used. If users wish to print out either the first or last timeperiod, the values `first` and `last` can be used as shorthands for the values `0` and `construct::intvar::time_count-1`, respectively, in order to ensure output on the first or last period. Lastly, the value `all` can be used to ensure that output is printed during multiple timeperiods.

If multiple timeperiods are to be printed for a CSV file, all values will be printed in the same file. During the first applicable period, a set of comma-separated values will be printed, followed by an empty line indicating the end of that time period. The values for the next applicable time period will be printed in comma-separated format following the blank line. If more than two time periods are printed, the file will alternate between comma-separated blocks of text and blank lines, with blank lines serving to separate different time periods. Users who wish to store output from different time periods in different files should use multiple generators (and print out only one time period per generator) in order to ensure that each file contains data unique to that timeperiod.

Note that multiple operations can be specified in the input deck. The operations will be performed in the order that they are specified in the input deck, which in many cases will not make a difference. Most operations will only run during the timeperiods in which they are specified, and thus will have minimal overhead on the simulation running time. The total number of operations is technically unlimited, but users may wish to avoid storing large amounts of data when performing multiple replications. Multiple large output files, combined with multiple runs, can quickly fill a hard disk with data.

The remainder of this section introduces the key networks in the demo input deck (Section 3.7.1) and other key outputs that experiment designers may wish to use (Section 3.7.2). Additional detail on each type of operation is provided, and the relevant values for the demo input deck are described.

### 3.7.1 Read networks

The `ReadGraphByName` is operation will print out a network (or a subset of a network) at particular points in time. The name of this operation may cause confusion as the name 'graph' is a legacy term for what has been referred to as a 'network' in this technical report. This operation is designed to read unprocessed data directly from Construct, and is best used in conjunction with external processing in order to best analyze simulation data. Users may choose to read as many networks as they wish – in separate operations, of course – and thus compute any function that they wish using the network values.

The `ReadGraphByName` operation, like any operation, can be used regardless of the Construct models that are in use. The dimensions of and values in the network outputted depend on the value of the Construct network. The ReadGraphByName operation can be used to print networks of bools (printed as 0/1), ints, floats, and strings.

The `ReadGraphByName` operation requires the `graph_name` param to be present, in addition to those listed in Section 3.7. This parameter specifies the name of the network that is to be read during the operation. The exact name of the network must be provided surrounded in single-quoted strings. Construct, by default, will remove spaces; however, it will not remove spaces in quoted strings. The network names specified in Section 3.5 all include spaces, so it is necessary to pass the network name to the operation while ensuring that the spaces are preserved. Thus, to read the `knowledge network`, it is necessary to pass the value `'knowledge network'` as the value of the `graph_name` parameter. Failing to include the quotes, or misspelling the network name, will cause Construct to error and exit as Construct will not be able to find the graph with that specific name.

Several optional parameters can be provided to the ReadGraphByName operation in order to improve the quality or readability of the output provided. The `print_row_names` and `print_col_names` parameters will cause Construct to print the row and column names that correspond to the row and column nodes, respectively. This can be useful for identifying which network entries correspond to which row and column entity. The params `first_row` and `last_row` can be used in order to specify that only network values from the first to last row (inclusive) be printed. Similarly, the params `first_col` and `last_col` allow a user to subset the column indices in a similar manner. These operations allow the user to only print a subset of the network, which can increase printing speed as well as decrease the amount of space necessary to store network output.

Note that it is not possible to read transactive memory networks using this operation.

Printing out large networks may be computationally costly, as Construct must write large amounts of data to the disk. While small networks can be printed quickly, large networks have src nodeclass x target nodeclass number of entries and may take a long time to output. Users are strongly advised to output the minimum amount of data necessary, as outputting unnecessary networks will slow down computation time.

In the demo input deck described in this technical report, three `ReadGraphByName` generators are provided.

- first.csv. The `first.csv` file stores the knowledge network after the first time period (timeperiod 0). As interaction does not occur during the first timeperiod, this network contains the initial values of the knowledge network. Each row represents one agent from agent 0 (a group A agent) to agent 41 (a group C agent), while each column represents one knowledge bit from knowledge bit 0 (a group K1 bit) to knowledge bit 59 (a group K2 bit). This network is further discussed in Section 4.3.
- last.csv. The `last.csv` file stores the knowledge network at the end of the simulation (timeperiod 49 in a short experiment, as discussed in Section 3.2.1). As it is a snapshot of agent knowledge at the end of the experiment, it is able to capture which agents know which knowledge bits. Again, each row represents one agent,

while each column represents one knowledge bit.  This network is further discussed in Section 4.3.

- prob.csv.  The `prob.csv` file stores the agent interaction probability network at the end of the simulation (timeperiod 49 in a short experiment, as discussed in Section 3.2.1).  The graph being read, the `agent interaction probability network`, is an internal network created by Construct to hold the probabilities of interaction as they are generated.  Each row represents one agent from agent 0 (a group A agent) to agent 41 (a group C agent), while each column also represents one agent from agent 0 (a group A agent) to agent 41 (a group C agent).  The probability of interaction specified is the probability of the row agent selecting the column agent as an interaction partner; the sum across all columns in the network should be 1.0 as it sums over all possible valid interaction partners.  Note that this is the instantaneous probability of interaction calculated during the last time period, not an average over all time periods in the experiment.  Thus, values in this network may be substantially different than interaction probabilities in the first time period.  This network is mentioned in Section 4.3, though not discussed extensively.

Users may customize any of these outputs, or add new ones to examine the values of particular networks of interest.

```
<operations>
 <operation name="ReadGraphByName">
   <parameters>
      <param name="graph_name" value="'knowledge network'"/>
      <param name="output_filename" value="first.csv"/>
      <param name="output_format" value="csv"/>
      <param name="run" value="all"/>
      <param name="time" value="first"/>
   </parameters>
 </operation>
 <operation name="ReadGraphByName">
   <parameters>
      <param name="graph_name" value="'knowledge network'"/>
      <param name="output_filename" value="last.csv"/>
      <param name="output_format" value="csv"/>
      <param name="run" value="all"/>
      <param name="time" value="last"/>
   </parameters>
 </operation>
 <operation name="ReadGraphByName">
   <parameters>
      <param name="graph_name"
             value="'agent interaction probability network'"/>
      <param name="output_filename" value="prob.csv"/>
      <param name="output_format" value="csv"/>
      <param name="run" value="all"/>
      <param name="time" value="last"/>
   </parameters>
 </operation>
</operations>
```

### 3.7.2 Other Operations (not present in the file)

While the `ReadGraphByName` is the only operation used in the demo input deck, it is worth mentioning several other important types of operations that users may find applicable to certain kinds of simulations. While code snippets of these operations are not provided, users should be able to build upon the examples provided previously in order to implement these operations in their input decks.

- ReadKnowledgeDiffusion. The `ReadKnowledgeDiffusion` operation prints out a one-dimensional vector, with as many entries in the network as there are agents in the simulation. Each entry in the vector specifies the ratio of the number of agents who have nonzero knowledge of that particular bit to the total number of agents. A value of 0 indicates that no agents know the bit, while a value of 1 indicates that all agents have full knowledge of the bit; an intermediate value suggests that some agents may have full knowledge while others either have incomplete knowledge or no knowledge. Note that this knowledge diffusion operation examines all agents and all knowledge bits; it is not possible to subset a particular group of agents or a particular group of knowledge bits for closer examination.
- ReadBinaryTaskAccuracy. `TheReadBinaryTaskAccuracy` operation prints out a one-dimensional vector, with as many entries in the network as there are agents in the simulation. Each entry in the vector specifies the agent's overall accuracy across all binary tasks to which it is assigned. For each task, an agent is given a 1 if accurate and a 0 if inaccurate according to the algorithm specified in Section 3.4.4. Note that this operation will not print out accuracy on any one individual binary task; instead, it reads out an average accuracy level per agent divided over all possible binary tasks in the simulation. If each agent is assigned to complete one task and one task only, a nonzero binary task accuracy can be thought of as an indication that the agent was able to complete the binary task successfully. If agents are assigned to complete multiple binary tasks, the ReadBinaryTaskAccuracy operation is not fine-grained enough to determine which binary task was completed.

Several other operations are available for advanced users.

It is also important to note that operations can be scripted using the `ReadDecisionOutput` operation. While not discussed in this technical report, scripting operations are discussed at length in CMU-ISR-09-126, which describes the scripting system in Construct (Hirshman et al 2009).

## *3.8 Key Features*

While the previous seven sections have attempted to introduce the key features of the Construct input deck, it may be useful to take a step back and examine the input deck from the perspective of the question asked in Section 2 of this technical report. As was discussed in that section, the input deck presented in Appendix A attempts to simulate the evolution of a society with three groups (group A, group B, and group C), the first two of which are separate and knowledge from separate knowledge groups (group K1 and group K2). The agents in the last group, agent group C, are only active conditionally; if they are active, they can interact with agents from both agent group A and agent group B.

Having introduced the input file in great detail for the majority of Section 3, is now possible to revisit the input deck at a high level. In so doing, it will be possible to where and how such features described in Section 2 are initialized. While the majority of Section 3 attempted to address the input file from the perspective of the Construct code, this section will revisit the file and attempt to describe the experiment thematically. In so doing, it will attempt to unify the various themes described throughout the rest of this document and to illustrate how various sections of the Construct input file work together.

- <u>There are three agent groups</u>. Three construct variables were used to specify the sizes of the agent groups (Section 3.2.1) and several helped variables were created to specify where each of these groups began and ended. Twenty agents were assigned to agent groups A and B, respectively, while two agents were assigned to group C. The number of agents in groups A, B, and C was summed to calculate the total number of agents nodes, `agent_count` (Section 3.2.1). The `agent_count` variable could then be used to initialize the agents nodeclass (Section 3.4.1). Agents in the agent nodeclass were then assigned to different agent groups for bookkeeping purposes (Section 3.5.33).
- <u>There are two knowledge groups</u>. Two construct variables were used to specify the sizes of the knowledge groups (Section 3.2.1) and several helped variables were created to specify where each of these groups began and ended. Thirty knowledge bits were assigned to knowledge groups K1 and K2, respectively. The number of knowledge bits in groups K1 and K2 was summed to calculate the total number of knowledge nodes, `knowledge_count` (Section 3.2.1). The `knowledge_count` variable could then be used to initialize the knowledge nodeclass (Section 3.4.2). Agents in the agent nodeclass were then assigned to different agent groups for bookkeeping purposes (Section 3.5.34).
- <u>Agents in groups A and B have unique knowledge</u>. Agents in group A were each assigned about 20% of the K1 knowledge, while agents in group B were each assigned about 20% of the K2 knowledge, in the `agent knowledge network` (Section 3.5.11). Agents in agent group C were assigned 10% of the knowledge in both K1 and K2. As there were 30 knowledge bits each in K1 and K2 (Section 3.2.1), this meant that all agents had about six knowledge bits initial.
- <u>Agents in groups A and B cannot interact directly</u>. Agents in agent group A were allowed to interact with all agents in their agent group, while agents in agent group B were allowed to interact with all the agents in their separate agent group, as specified in the `interaction sphere network` (Section 3.5.14).
- <u>Agents in group C bridge the two agent groups</u>. Agents in agent group C could interact with agents in both agent group A and agent group B, as also specified in the `interaction sphere network` (Section 3.5.14).
- <u>Agents in group C are conditionally active</u>. Agents in agent group C are conditionally active, as specified in the agent active timeperiod network (Section

3.5.16). If the `bridging_agents_active` parameter was true, agents in agent group C could interact, and therefore have the opportunity to interact with and learn information from agents in group A, then interact with and convey information to agents in group B (Section 3.5.14). If the `bridging_agents_active` parameter were false, as is the current setting in the appendix, this would not be allowed to happen and therefore information cannot be conveyed.

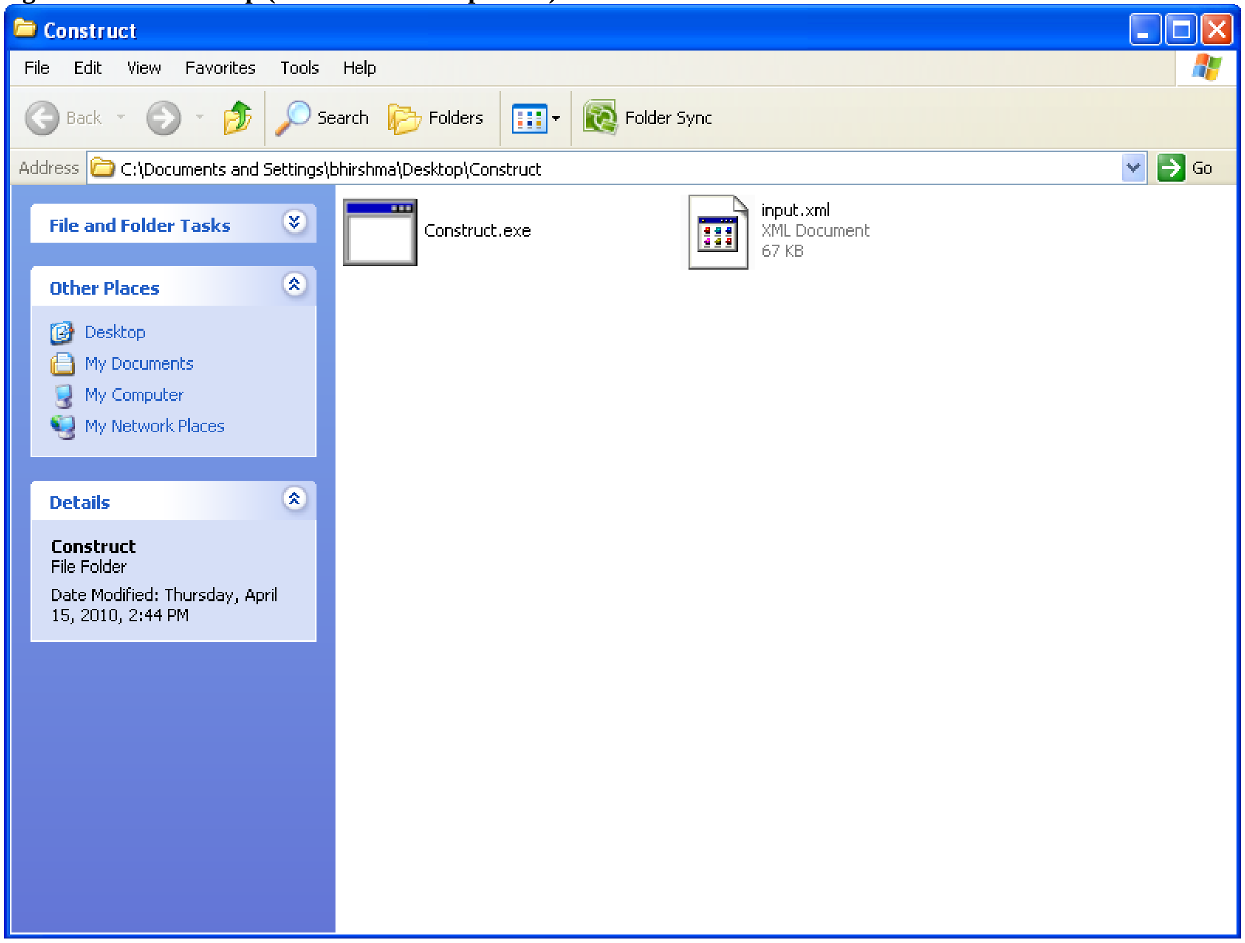
- Agents interact primarily by homophily. Agents in the simulation are primarily interacting due to homophily – roughly 80% of the probability of interaction calculation is due to perceived shared knowledge (Section 3.5.27). The remainder of the probability of interaction is due to perceived expertise (Section 3.5.28). Agents consider all knowledge bits equally important when choosing an interaction partner (Section 3.5.29). Note that perceptions matter, as agents use their transactive memory in order to decide which agents they would prefer to interact with; however, this information may not be accurate initially (Section 3.6.1).
- Agents interact twice per timeperiod and send short messages. Agents can initiate knowledge once per timeperiod (Section 3.5.2) and receive interaction once per timeperiod (Section 3.5.3). Agents who are able to obtain an interaction partner will send a single item to them (Section 3.5.4), which will likely be a knowledge bit (Section 3.3.7). If so, agents are equally likely to send any knowledge bit that they know (Section 3.5.30 and Section 3.5.31).
- The simulation lasts for fifty timeperiods. Because the simulation is a short simulation (Section 3.2.1), the `time_count` is set to 50 (Section 3.2.1), meaning that the timeperiod nodeclass has fifty nodes (Section 3.4.6). All agents are active for all of these time periods, except possibly the bridging agents (Section 3.5.14). Key weights, such as those placed on various proximities, do not change over the course of the simulation.
- Beliefs and tasks are not included. As specified in the construct variables section, there are no binary tasks, energy tasks, or beliefs (Section 3.2.1). Thus, these respective nodeclasses are empty, and simulation parameters and networks dealing with these concepts are either zeroed (i.e. Section 3.5.23), omitted (i.e. Section 3.5.35) or disabled (i.e. Section 3.3.9).

As can be seen, many of these themes require inputs from multiple parts of the input file in order to specify simulation behavior. Due to the internal organization of the input deck, it cannot be helped that many logical concepts are spread over multiple sections. Users attempting to understand the input deck should both keep in mind the high-level design and the low-level properties of the file. When designing or extending an input deck, beginning users may find it useful to diagram the desired behavior prior to coding the file in order to minimize this problem.

# 4 Running Construct

Construct can be run in several ways. Some of these ways have graphical user interfaces, such as the Near-Term Analysis tool in *ORA (Carley & Reminga 2004; Moon & Carley 2007). However, these tools, as yet, do not have support for many of the more complex features available in Construct. In order to access these features and run the experiment outlined in Section 2 and described in Section 3, however, it is necessary to run Construct in batch mode. This section describes how to run Construct in batch mode, and walks the user through the

**Figure 10: Initial setup (Construct and input file)**



C:\Documents and Settings\bhirshma\Desktop\Construct

Construct.exe

input.xml
XML Document
67 KB

Construct
File Folder
Date Modified: Thursday, April 15, 2010, 2:44 PM

process of making a small change to the input deck and comparing and contrasting the result of that change.

### *4.1 Gathering files for batch mode*

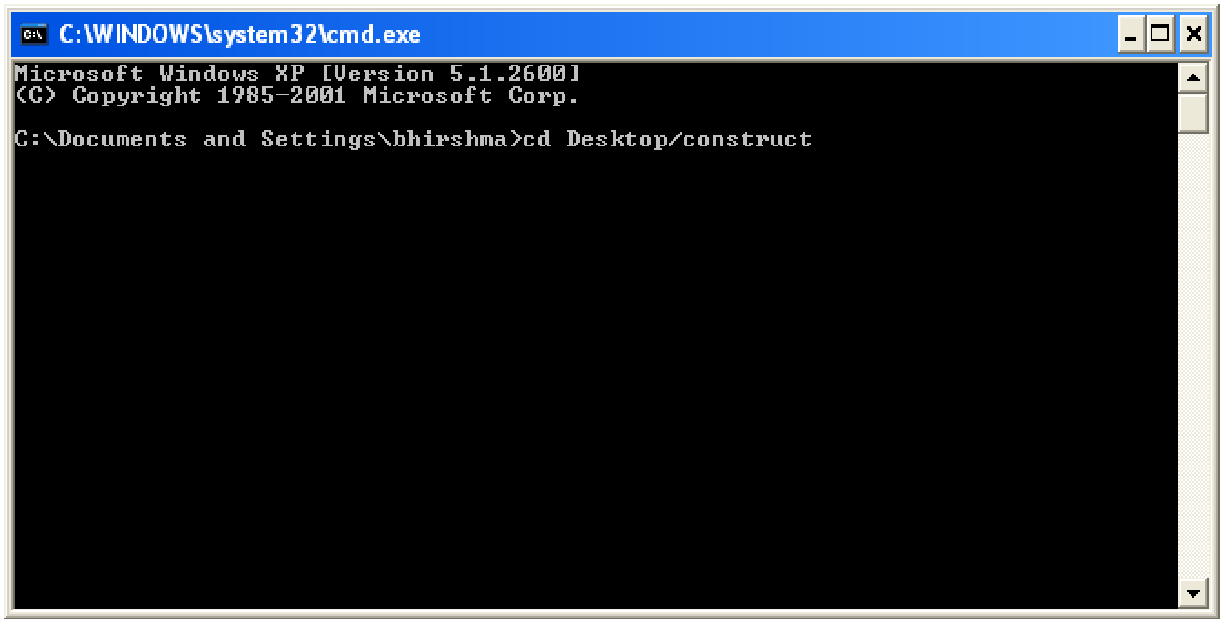
In order to run Construct in batch mode, it is necessary to obtain three items:

- the input deck
- the Construct executable
- any supplemental input files (such as CSV files referred to by the input deck)

These items should be placed on a in a folder on the user's hard drive. Each of these items will be described in turn.

The demo input file described in this document, available in Appendix A of this document, should be saved as the file `input.xml`. Create this input deck by copying the text following the initial `READ ME` comment on page 127 into a plain text editor such as Notepad. Copy the text from the initial `READ ME` until the terminal `</construct>` tag of the XML is reached. The final file should be 1216 lines long, ending with a line containing the terminal `</construct>` tag. While the file can have any tame, the name `input.xml` will be used throughout this document. Users are strongly recommended to use the name `input.xml` for the purposes of this demo.

**Figure 11: Navigate to the location of the Construct file**



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\bhirshma>cd Desktop/construct
```

The Construct executable version 3.8.016BRH should be downloaded from the CASOS web site. This version is the current version as of spring 2010. New users are strongly recommended to obtain this version of Construct (possibly by downloading the non-current version after summer 2010) in order to follow along with this demo. It is possible that new features in Construct may make slight changes to the random seed, which can lead to difficulty when comparing simulation output to the figures in this document. By using version 3.8.016BRH, any results obtained should be identical to those seen in this document.

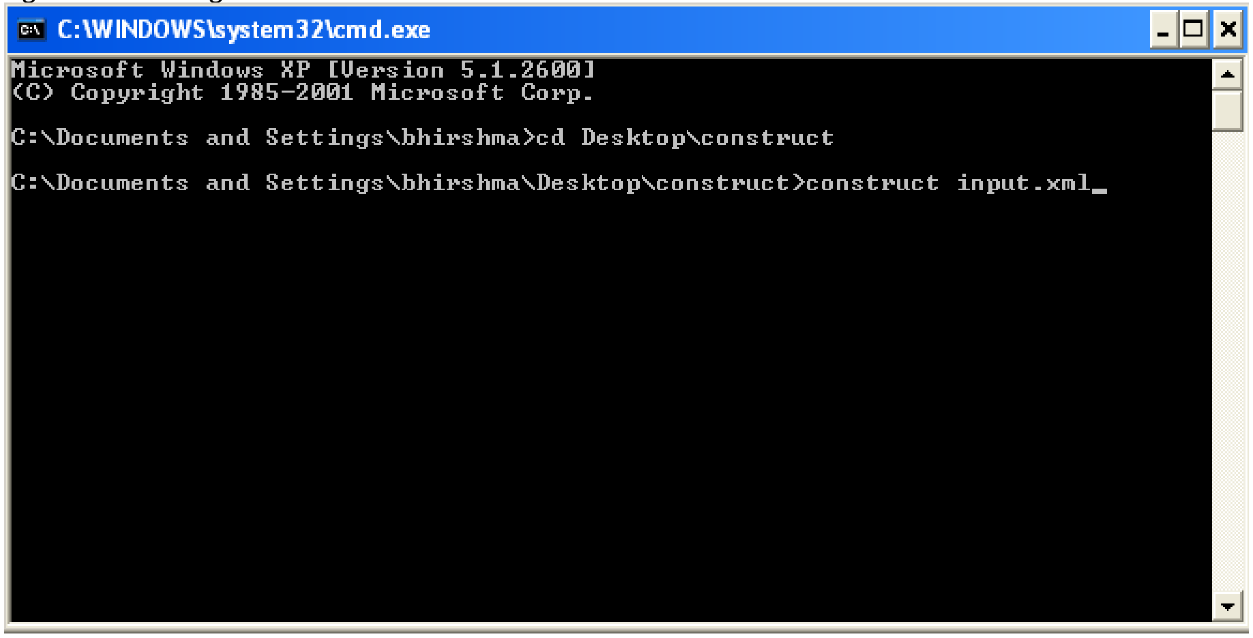
In this example, there are no supplemental files; thus, only the executable and the input file are needed.

The containing folder can be placed anywhere on the hard disk. It can even be placed on a remote server where the user has read, write, and execute access. For purposes of demonstration, it will be assumed that the user has placed the items in a folder named `construct` located on the desktop. Both the input file and the executable should be placed in this `construct` folder, as seen in Figure 10. While the demonstration input deck described in Appendix A creates very small files (~100KB total), Construct is capable of generating tens of megabytes per simulation; replications of the simulation, as well as additional experimental conditions, could require several hundred megabytes or even gigabytes. Thus, users should ensure that the folder is located on a hard disk of appropriate size.

As of Spring 2010, Construct is available for Windows as well as Unix. However, the input deck described in Appendix A is for Construct version 3.8.016BRH, which has only been built for Windows. Users wishing to follow along with the virtual experiment described in this document should verify that they are using this windows build in order to see identical results to those produced here.

## 4.2 Running Construct

**Figure 12: Running the executable**



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\bhirshma>cd Desktop\construct

C:\Documents and Settings\bhirshma\Desktop\construct>construct input.xml_
```

Construct is a batch mode program, and as such must be run from the command line. This can be done in one of several ways on a Windows machine. The simplest way, illustrated in Figure 11, is to use the Windows command line. The command line is available on all windows machines, and can be accessed from the Start menu (Start -> Run -> type `cmd` in the box that opens). For the remainder of this document, illustrations will be provided running the simulation via the Windows command line. It is important to note that Construct can also be run through a variety of other tools such as Cygwin. Advanced users who are creating scripts that call Construct multiple times (as described in Section 4.8.4) may wish to take advantage of these tools.

To run Construct, it is necessary to navigate to the folder created in Section 4.1. To do so, type the command `cd` and then the location of the folder. If the folder was called "construct" and placed on the desktop, as was recommended in Section 4.1, one can type the following command in the window to navigate to the folder:

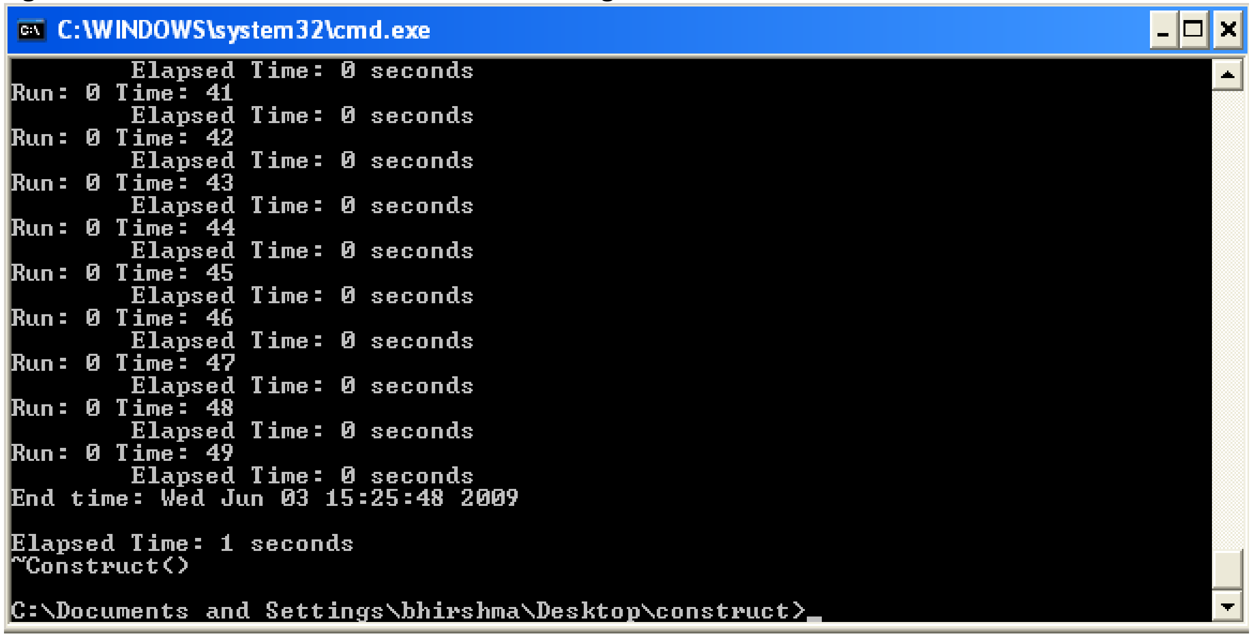
```
cd desktop/construct
```

This process is illustrated in Figure 11. Press enter for the command to be executed. Note that if the Construct folder is at another location, it may be necessary to use other shell commands to reach the folder's location. If the command is successful, the current directory should be changed to the location of construct executable and input deck. This can be verified by looking to the left of the curser, which should list the current location. If the location changes to include desktop and construct as the two rightmost entries, then the directory was successfully changed.

Once in the folder with the input deck and executable, the Construct simulation can be performed. To run the program on an input deck, type the command in the command line, as seen in Figure 12:

```
construct input.xml
```

**Figure 13: Results when Construct is finished running**



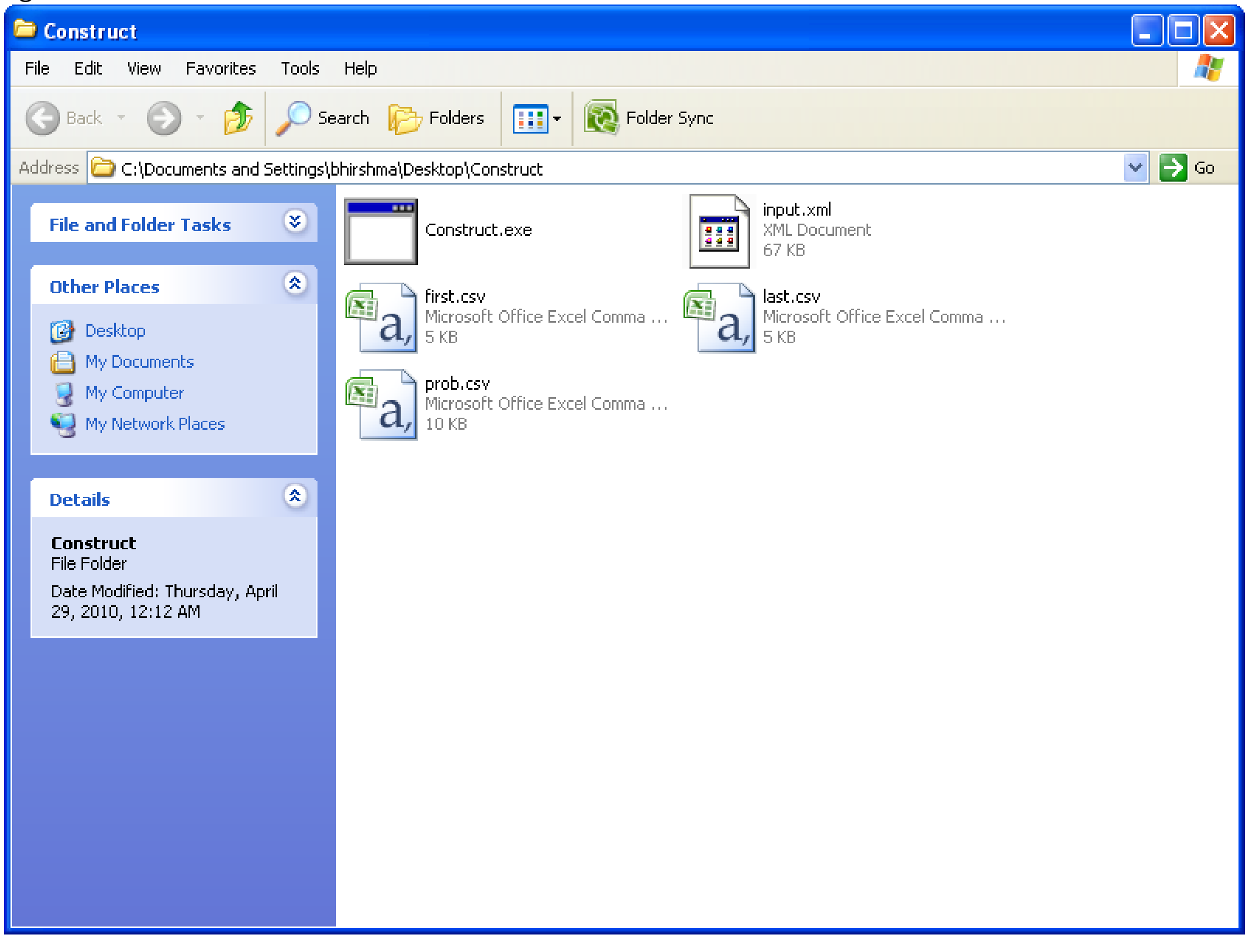
```
        Elapsed Time: 0 seconds
Run: 0 Time: 41
        Elapsed Time: 0 seconds
Run: 0 Time: 42
        Elapsed Time: 0 seconds
Run: 0 Time: 43
        Elapsed Time: 0 seconds
Run: 0 Time: 44
        Elapsed Time: 0 seconds
Run: 0 Time: 45
        Elapsed Time: 0 seconds
Run: 0 Time: 46
        Elapsed Time: 0 seconds
Run: 0 Time: 47
        Elapsed Time: 0 seconds
Run: 0 Time: 48
        Elapsed Time: 0 seconds
Run: 0 Time: 49
        Elapsed Time: 0 seconds
End time: Wed Jun 03 15:25:48 2009

Elapsed Time: 1 seconds
~Construct()

C:\Documents and Settings\bhirshma\Desktop\construct>_
```

Press enter to execute the command. This will call the Construct simulation on the input file named `input.xml`.

The simulation should begin, and output will be written to the terminal. If the simulation begins successfully, text should flash by rapidly in the output window. The end result of execution, which should occur just moments after pressing enter, should be identical to those of Figure 13. For the demo deck `input.xml`, fifty time periods will be simulated (as time periods are zero indexed, the final time period is time period 49), and the simulation running time should be only a second or two. If the result obtained is not equivalent to that seen in Figure 13, or any other error occurs, see Section 4.7 in order to debug the input deck.

Any time Construct runs, it will print out diagnostic messages regarding simulation setup and execution. First, Construct will print a number of debugging messages regarding the types of nodeclasses and networks loaded. Next, transactive memory will be initialized, outputs set up, and core simulation models initialized. Only after all this setup has occurred will the simulation run. As the simulation progresses, the program will print out information detailing how long each time period took to execute. When the simulation has completed successfully, it will display the end time of the simulation, then the elapsed time for the total simulation, and finally end with a line containing the characters `~construct()`. You can verify that all of this information was printed to the console by scrolling up using the scrollbar on the right-hand side of the window.

**Figure 14: After simulation**


Construct
File Edit View Favorites Tools Help
Back Search Folders Folder Sync
Address C:\Documents and Settings\bhirshma\Desktop\Construct Go
File and Folder Tasks
Other Places
Desktop
My Documents
My Computer
My Network Places
Details

Construct
File Folder
Date Modified: Thursday, April 29, 2010, 12:12 AM

Construct.exe

input.xml
XML Document
67 KB

first.csv
Microsoft Office Excel Comma ...
5 KB

last.csv
Microsoft Office Excel Comma ...
5 KB

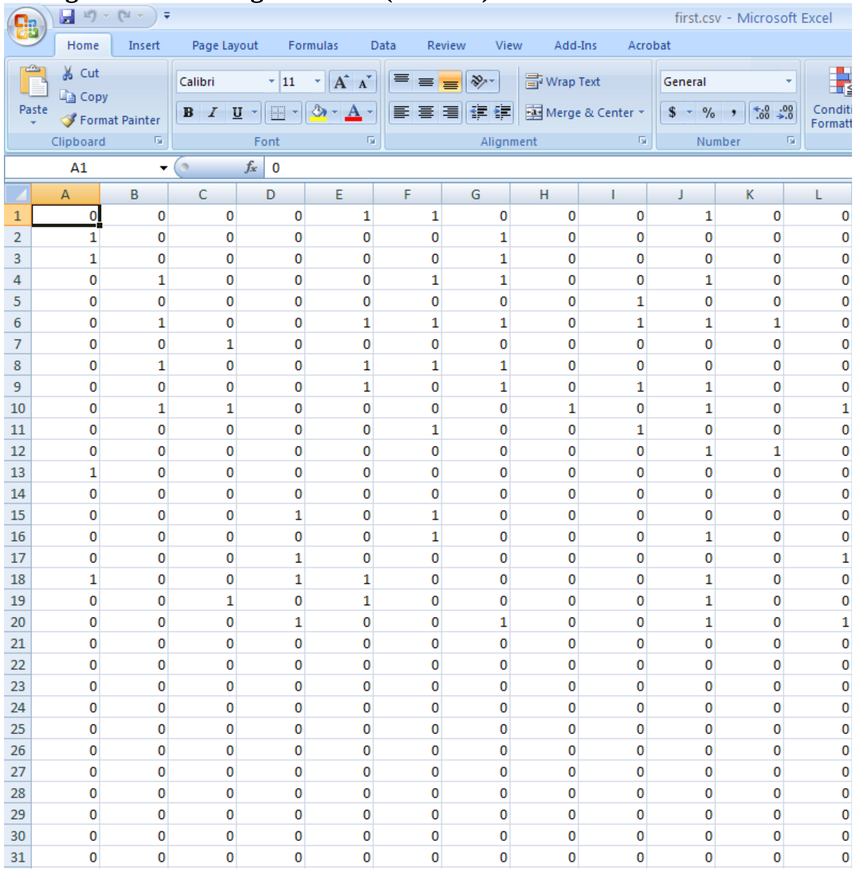
prob.csv
Microsoft Office Excel Comma ...
10 KB

### *4.3 Examining Construct output*

If the simulation is successful, the input deck `input.xml` is designed to create three output files: `first.csv`, `last.csv`, and `prob.csv`. These files will be generated in the same folder as the executable and input deck. When the simulation is complete, the folder should look like Figure 14, which shows the three new files created by the simulation. Compare this screenshot to that of the initial setup of Figure 10 (page 106) to see what output files should be when running the experiment.

As these output files are in CSV format, they can be opened in a spreadsheet or text editor program. While the examples in the remainder of this document use Microsoft Excel as the demonstration program, users are welcome to use any other program to open the files and examine the contents. The examples used in the remainder of this document will use Microsoft Excel's A1-style row-column format in order to reference particular entries in the file. While this format may differ between spreadsheet programs, the particular value referenced should be straightforward to compute.

**Figure 15: The initial agent-to-knowledge network (first.csv)**



|  | A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 7 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 10 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 11 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 13 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 17 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 18 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 19 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 20 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 24 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 27 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 28 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 29 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 30 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Each of these files contains important information about the experiment as it runs. In Section 4.4, they will serve as the basis for analyzing the experiment to understand what occurred during the simulation.

- The file `first.csv` saves the knowledge network at the beginning of the simulation, a portion of which can be seen in Figure 15. It is an agent-to-knowledge network, meaning that each row in the file represents an agent and each column in the file represents a particular knowledge bit. The user can verify that the file contains 42 rows for the 42 total agents and 60 columns for the 60 total knowledge bits. Both the agents and knowledge are printed out in order, meaning that the first row corresponds to agent index 0 and the first column corresponds to knowledge index zero (though programs like Microsoft Excel are one-indexed, meaning that the first agent is in row 1 and the first knowledge bit in column A). This means that the first twenty rows correspond to the first twenty agents initialized – specifically, the knowledge of the first twenty agents of agent group A – the next twenty agents correspond to the knowledge of the agents in agent group B, and the last two rows represent the knowledge of the agents in agent group C. Similarly, the first thirty columns correspond to the knowledge bits in knowledge group K1 and the second thirty columns correspond to that in group K2. The values in the file should be binary (0 or 1) since knowledge is treated as a binary known / not known value in this experiment. Note that this network is printed at the beginning of the simulation but before any interaction has occurred; this network represents the initial knowledge of the agents in the simulation.

- The file `last.csv` saves the knowledge network at the end of the experiment after it has run for the full fifty iterations. Again, there should be 42 rows and sixty columns in the file, and the values should still be binary. The values in this file should differ from those in `first.csv` for reasons that will be discussed in Section 4.4.
- The file `prob.csv` saves the agent interaction probability network at the end of the experiment. As this network is an agent-by-agent network, the file should have 42 rows and 42 columns corresponding to the 42 agents in the simulation. The meaning of each value is that the row agent has that probability of selecting the column agent as its interaction partner for the simulation round if all other interaction partners are available. Thus, the values in each cell are floating-point values, and each row should sum to unity. While the interpretation of this network will be discussed in Section 4.4, it is worth noting that this network is printed at the end of the simulation and thus represents the evolved probability of interaction among agents. The initial probability of interaction, which was not saved during the simulation run, had slightly different values since the probability of interaction evolved over the course of the simulation.

## 4.4 Understanding and interpreting results

In order to understand what occurred in the experiment, it is first necessary to understand the low-level details of each agent's initial knowledge. To begin, open `first.csv`, which represents the knowledge held by each agent at the beginning of the experiment. This file records which knowledge bits that were assigned to each of the agents at the beginning of the simulation. The first thirty columns and twelve rows of this file should be identical to Figure 15 if the user employed Construct version 3.8.016BRH.

Several important concepts are illustrated in the file.

- First, from a ten-thousand foot view of this file divides the knowledge network into five broad blocks. Note that these blocks correspond to the regions that were defined in the input file, and the corresponding generators that operated on the knowledge network.

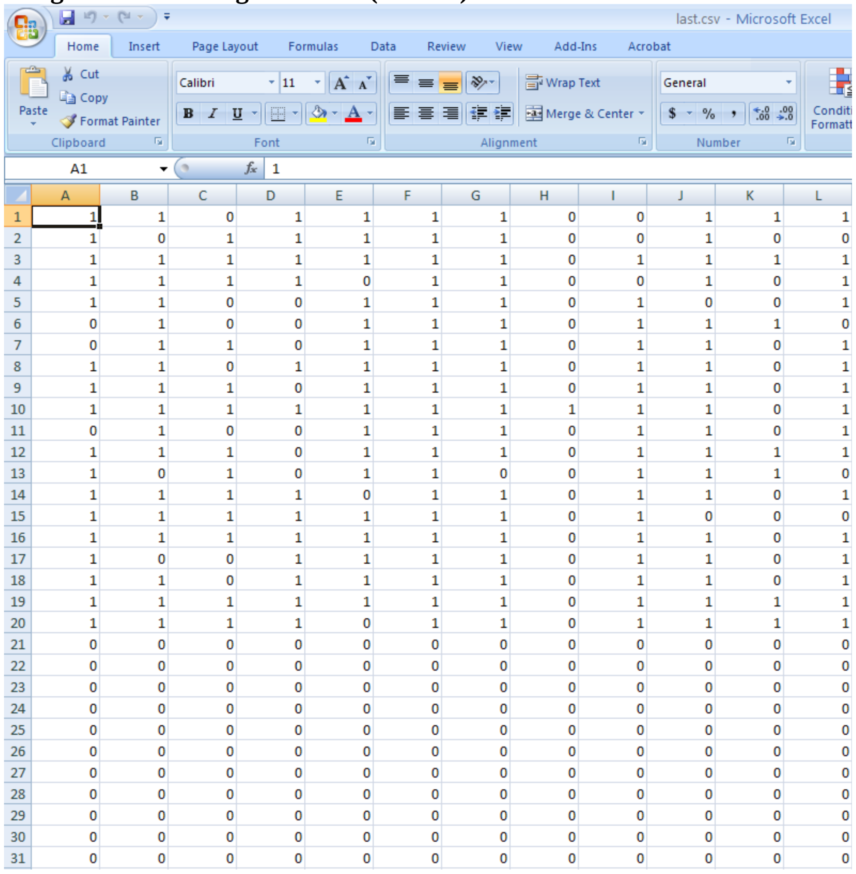
  Region I: The agents in group A (the first twenty agents, or rows in the file) have some of knowledge group K1 (the first thirty facts, or columns in the file). While no agent will have all of this knowledge, all group A will have at least some knowledge. The density of this area will be about 20%, meaning about one in five cells should have a value of 1. In Excel, this is the area between cells A1 and AD20. Part of this block can be seen in the top part of Figure 15.

  Region II: The agents of group A (the first twenty agents) have no knowledge group K2 (the last thirty facts). By design, no agent should have any of this knowledge, so all cells between AE1 and BH20 should have the value 0. Part of this area can be seen in the bottom part of Figure 15.

  Region III: The agents of group B (the next twenty agents) have no knowledge group K1 (the first thirty facts). By design, no agent should have any of this knowledge, so all cells between A21 and AD40 should have the value 0.

  Region IV: The agents in group B (the next twenty agents) have some of knowledge group K2 (the last thirty facts). While no agent will have all of this knowledge, all group A will have at least some knowledge. In Excel, this is the area between cells AE21 and BH40. The density of this area will also be about 20%.

**Figure 16: The final agent-to-knowledge network (last.csv)**



| | A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 4 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 6 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 7 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 8 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 9 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 10 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 11 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 12 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 13 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 14 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 15 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 16 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 17 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 18 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 19 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 20 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 24 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 27 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 28 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 29 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 30 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Region V: The knowledge that the agents of group C (the last two agents) have of some facts in group K1 and K2. In Excel, this is the region between cells A41 and BH42. While this block was technically defined by two generators in the input file,

- All agents should have about six knowledge bits. Each agent in agent group A or agent group B should know about six facts because that represents 20% of 30 facts in the relevant fact group (K1 or K2, respectively). Each agent in agent group C should have about 10% of the facts in both fact groups, meaning that they also have about six facts. This means that agents have the opportunity to learn a substantial number of facts when interacting with other agents. As will be seen shortly, such learning does occur.
- The facts that each agent knows will be different from the facts known by the neighboring agents, or indeed any other agent in the simulation. This is because the knowledge in the simulation was initialized randomly. This means that the interaction between any pair of agents means that both agents can learn something from the interaction. While agents may choose to share common knowledge when interacting, it is also possible for agents to learn new knowledge from each interaction partner.

The results outlined above represent the knowledge that was assigned to each agent at the beginning of the experiment. Once this data was stored, the agents began to interact with each other and to exchange information. Of paramount importance is the fact that the agent groups A and B were completely separate from each other. While agents in both groups were connected to group of bridging agents – agent group C – that could potentially convey information between both groups, in the simulation performed these bridging agents were inactive. Information should not be able to travel between the groups.

The file `last.csv` records the final knowledge of each agent, and the first thirty-row by twelve-column portion of the file can be seen in Figure 16. Keep the `first.csv` file open in one window, and open `last.csv` in another. By comparing the differences between `first.csv` and `last.csv`, it is possible to understand the nature of the changes that occurred during the experiment.

The first thing that should be apparent from the two files is that learning did occur. For instance, look at cell A1, which represents the first agent's knowledge of the first fact. In the initial time period (i.e., in `first.csv`), the agent did not have knowledge of that fact. However, during one of the fifty simulated time periods, it interacted with an agent that had that fact. Thus, in the last time period (i.e., in `last.csv`), the agent had knowledge of the particular fact. This process suggests that information diffusion is taking place within the simulated environment. Comparing the two files, it is easy to see that this is not an isolated example. Every (active) agent learns at least one new fact during the course of the simulation, and every fact is learned by one or more agents.

Learning and information diffusion occurs unevenly, and not all knowledge bits diffuse as extensively as others. Consider, for instance, column H (fact 7, or the eighth knowledge bit) in `last.csv`. Only five agents in agent group A ever learn of this knowledge bit during the simulation. While this might seem to be substantially less than the number of agents who know any other knowledge bit, this result is not necessarily surprising if one examines the `first.csv`. In that file, it can be seen that one agent knows the bit initially, meaning that information diffusion starts from only one point as opposed to many. However, explanation of variance due to changes in initial condition does not explain all differences. For instance, columns L and M (fact 11 and 12, respectively) are known by different numbers of agents at the end of the simulation, but both started with three agents knowing the bit. While a very thorough explanation of why these differences occur is beyond the scope of this technical report, it is sufficient to say that small changes in initial conditions – or even small changes in interactions that vary if and when the fact is shared – can have very important differences on final outcomes.

The main purpose of this simulation, however, is not necessarily to analyze the diffusion of individual bits. Instead, it is to analyze the patterns of information diffusion that occur. What kinds of patterns occur?

- Most obviously, it is clear that the knowledge density in Region I and Region IV increased. Agents in agent group A learn more facts in knowledge group K1, and agents in agent group B learn more facts in knowledge group K2, but do not learn any knowledge from group K1. Because agents cannot forget, all knowledge that the agents know in the initial stage of the simulation is known at the final stage of the simulation; agents learn new facts to supplement, not replace, their existing knowledge. Indeed, by the end of the simulation, most agents will learn more than eighty percent of all the knowledge bits in their associated knowledge group.
- Equally importantly, though, no information diffused between the two groups. Region II and Region III remain completely empty; agents in group A did not learn any knowledge from group K2 and agents in group B did not learn anything in group K1. Without the bridging agents active, the groups continue to be insular. Information is not entering each of the two systems. Instead, they act as if they are two completely independent diffusion simulations, each an independent microcosm.

- Lastly, the agents in group C – the last two agents in the input deck – were not active during the experiment. Thus, in comparing these rows in `first.csv` and `last.csv`, it is clear that these agents did not learn during the course of the experiment.

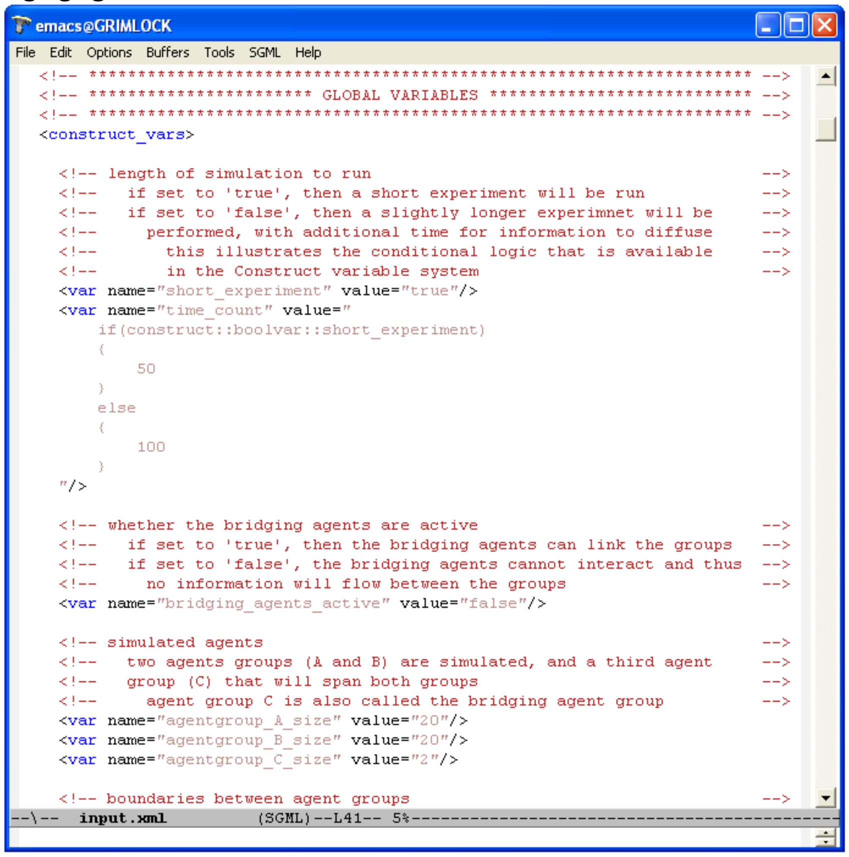
Before making a modification to the input deck to see how input changes can affect output, it is worthwhile to examine the file that recorded the final probability of interaction between agents, `prob.csv`. Open this file in a separate window. While there are multiple items of interest in this file, three of them are particularly illustrative of Construct features.

- Agents in agent group A and B have a non-zero probability of interaction with each of the agents in their agent group. This is because the input deck was designed to allow interaction between such pairs of agents. However, agents have zero probability of interacting with agents from the other group (B and A, respectively). Note that while agents technically had a probability of interaction with the agents in agent group C, such agents could not actually be chosen during the simulation and thus such values should be ignored for now.
- The non-zero numbers in each row vary from agent pair to agent pair. Agents in this simulation are interacting based largely due to homophily (similarity), but more importantly are interacting due to perceived homophily. While agents may appear similar in the `last.csv` file, the agents may not accurately perceive such similarity. Each agent's transactive memory – which was not recorded as a simulation output – governs that agent's behavior. If agents are not similar, or more commonly if agents do not perceive themselves to be similar, then agents will be less likely to interact. Thus, ranking agents in terms of similarity via the values in `last.csv` will not lead to the probabilities of interaction used in Construct.
- The probability of interaction is not symmetric, meaning that there are cases where one agent is more eager to interact with an alter than the alter is to interact with the ego. This lack of symmetry may occur for several reasons. For instance, agents value expertise as well as similarity; if one agent has knowledge that another agent lacks, the former will want to interact with the latter more than the latter will want to interact with the former. Alternately, one agent may have a more accurate perception of another agent's knowledge, and thus a higher probability of interaction since it perceives greater similarity. A third reason may be that one agent has less appealing third parties to choose from, making the alter agent appear more attractive by comparison. To understand which explanation is most accurate for each agent-agent pair is beyond the scope of this technical report. However, these reasons illustrate why this problem is ideal for simulation: such differences and non-linearities can be crucial for the evolution of a simulation at a micro level, and can have important impacts at the macro-level of analysis.

### *4.5 Making Changes to an existing input deck*

As can be seen in comparing Figure 15 and Figure 16, information diffused within group A and group B during the simulation. However, the two groups remained completely isolated from each other. Agents in agent group A, which had knowledge of facts in fact group F1, did not gain any knowledge of facts in fact group F2. Similarly, agents in agent group B did not gain any knowledge of facts in fact group F1. Because the bridging agents were inactive, no information was exchanged between the groups.

**Figure 17: The bridging agents active variable**



```
<!-- ******************************************************************** -->
<!-- *********************** GLOBAL VARIABLES *************************** -->
<!-- ******************************************************************** -->
<construct_vars>

  <!-- length of simulation to run                                        -->
  <!--   if set to 'true', then a short experiment will be run            -->
  <!--   if set to 'false', then a slightly longer experimnet will be     -->
  <!--     performed, with additional time for information to diffuse     -->
  <!--       this illustrates the conditional logic that is available     -->
  <!--       in the Construct variable system                             -->
  <var name="short_experiment" value="true"/>
  <var name="time_count" value="
      if(construct::boolvar::short_experiment)
      {
          50
      }
      else
      {
          100
      }
  "/>

  <!-- whether the bridging agents are active                             -->
  <!--   if set to 'true', then the bridging agents can link the groups   -->
  <!--   if set to 'false', the bridging agents cannot interact and thus  -->
  <!--     no information will flow between the groups                    -->
  <var name="bridging_agents_active" value="false"/>

  <!-- simulated agents                                                   -->
  <!--   two agents groups (A and B) are simulated, and a third agent     -->
  <!--   group (C) that will span both groups                             -->
  <!--     agent group C is also called the bridging agent group          -->
  <var name="agentgroup_A_size" value="20"/>
  <var name="agentgroup_B_size" value="20"/>
  <var name="agentgroup_C_size" value="2"/>

  <!-- boundaries between agent groups                                    -->
--\--  input.xml          (SGML)--L41-- 5%-----------------------------------------
```

To make changes to a Construct input deck, open the XML file `input.xml` using a text editor. As suggested in Section 4.1, programs such as Notepad are ideal for such editing. When beginning to edit the file, however, it is recommended that one launch the editor and then open the file (using the File -> Open menu option). This is because a typical Windows machine will open XML files using Windows Explorer when the XML file is double clicked, and changes cannot be made to a file using that program. If the file is accidentally opened in explorer, close the file and reopen `input.xml` in Notepad or any appropriate editor; no harm is done.

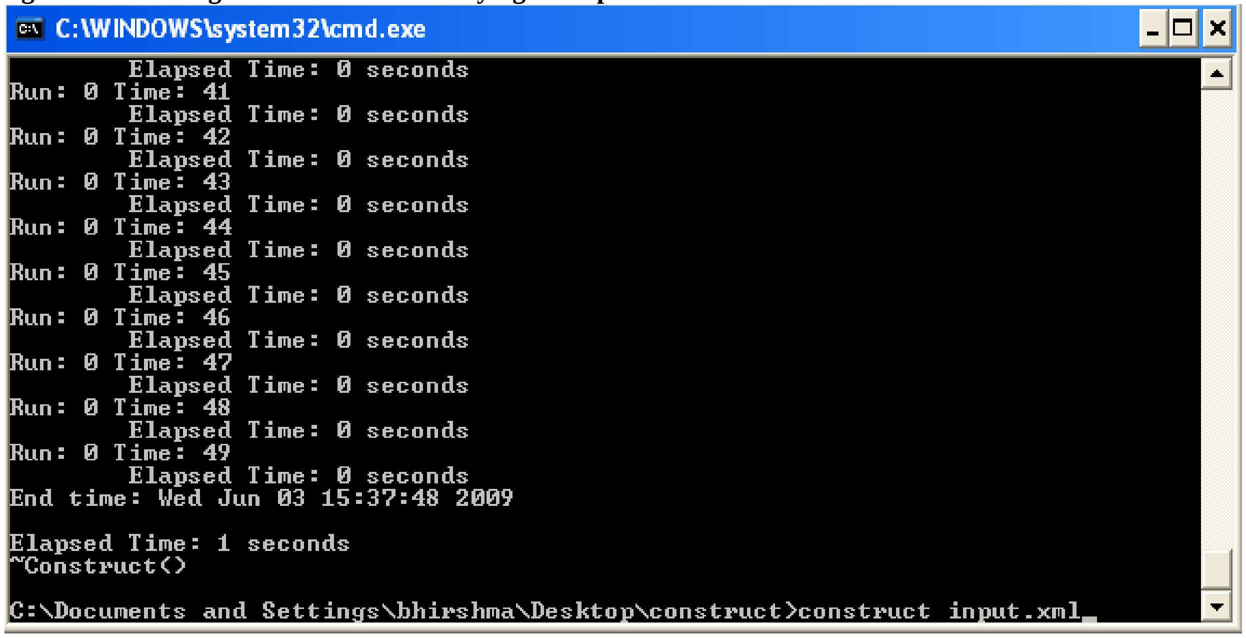
When `input.xml` is open in the text editor, find the line that defines the Construct variable `bridging_agents_active`. The line should read:

```
<var name=“bridging_agents_active” value=“true”/>
```

This line can be seen in the middle of Figure 17, which illustrates the first handful of variables defined in the input file. If one is using a text editor that can go to specific lines, this variable is found at line 68. One can also find the variable definition by searching for the comment text (i.e. “whether the bridging variables are active”) which is unique in the file; note that searching for the string “bridging_agents_active” may lead the user to both the variable definition at the top of the file as well as the locations where the variable is used.

The value of the `bridging_agents_active` variable is currently set to `false`. This value guaranteed that the bridging agents were not active during the simulation run in Sections 4.2 the subsequent analysis of Section 4.3. Setting this variable value to `true` will ensure that

**Figure 18: Running Construct after modifying the input deck**



```
C:\WINDOWS\system32\cmd.exe
        Elapsed Time: 0 seconds
Run: 0 Time: 41
        Elapsed Time: 0 seconds
Run: 0 Time: 42
        Elapsed Time: 0 seconds
Run: 0 Time: 43
        Elapsed Time: 0 seconds
Run: 0 Time: 44
        Elapsed Time: 0 seconds
Run: 0 Time: 45
        Elapsed Time: 0 seconds
Run: 0 Time: 46
        Elapsed Time: 0 seconds
Run: 0 Time: 47
        Elapsed Time: 0 seconds
Run: 0 Time: 48
        Elapsed Time: 0 seconds
Run: 0 Time: 49
        Elapsed Time: 0 seconds
End time: Wed Jun 03 15:37:48 2009

Elapsed Time: 1 seconds
~Construct()

C:\Documents and Settings\bhirshma\Desktop\construct>construct input.xml_
```

the bridging agents (agent group C) are active the next time the simulation is run. These agents will then be able to interact both with the agents of agent group A and agent group B and thus can serve as a conduit of information between the two groups.

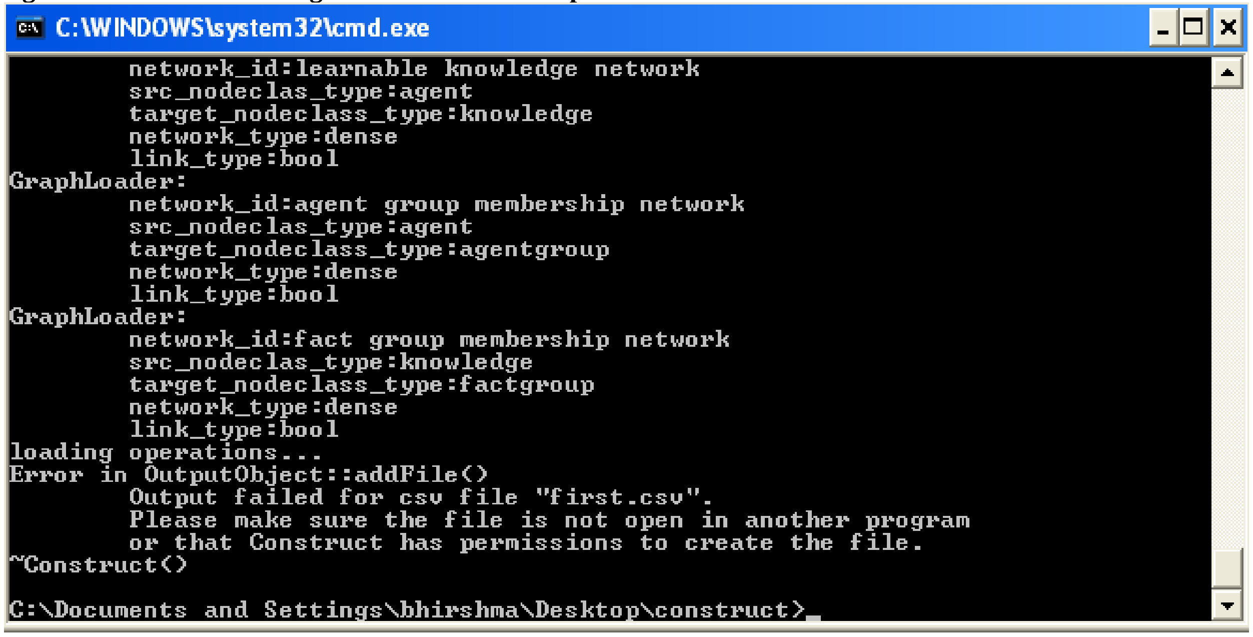
Thus, using the text editor, modify the value of the `bridging_agents_active` variable to be the value `true`. Following this change, the line should read:

```
<var name="bridging_agents_active" value="false"/>
```

This will allow the bridging agents to initiate and receive communication from agents in agent groups A and B. Remember to save the document prior to re-running the simulation. While it may not be necessary to close the text editor when running Construct, it is important to make sure that the changes are committed by saving the file; if the file is not saved, the changes to simulation setup will not be seen.

Before continuing with the remainder of this walkthrough, it is useful to rename the `first.csv` and `last.csv` files. In this section, the input deck was changed; in the section following, Construct will be re-run and these output files will be overwritten. As it may be useful to compare the results seen in Figure 16 to what occurs when the bridging agents are active, it would be useful to save both files in order to facilitate these comparisons. Any reasonable name for these files can be used, but names such as `briging-agents-inactive-first.csv` and `briging-agents-inactive-last.csv` are illustrative of the data contained in these files. Note that this step is completely optional, as the data currently in `first.csv` and `last.csv` can be recovered by undoing the changes made in this section and re-running the simulation.

**Figure 19: The error message that occurs if an output file is not closed**



```
        network_id:learnable knowledge network
        src_nodeclas_type:agent
        target_nodeclass_type:knowledge
        network_type:dense
        link_type:bool
GraphLoader:
        network_id:agent group membership network
        src_nodeclas_type:agent
        target_nodeclass_type:agentgroup
        network_type:dense
        link_type:bool
GraphLoader:
        network_id:fact group membership network
        src_nodeclas_type:knowledge
        target_nodeclass_type:factgroup
        network_type:dense
        link_type:bool
loading operations...
Error in OutputObject::addFile()
        Output failed for csv file "first.csv".
        Please make sure the file is not open in another program
        or that Construct has permissions to create the file.
~Construct()

C:\Documents and Settings\bhirshma\Desktop\construct>_
```

## *4.6 Comparing and contrasting output*

When the file has been saved, Construct should be re-run so that the simulation is performed on the updated input deck. In order to do this, repeat the command used in section 4.2 and seen in Figure 18:

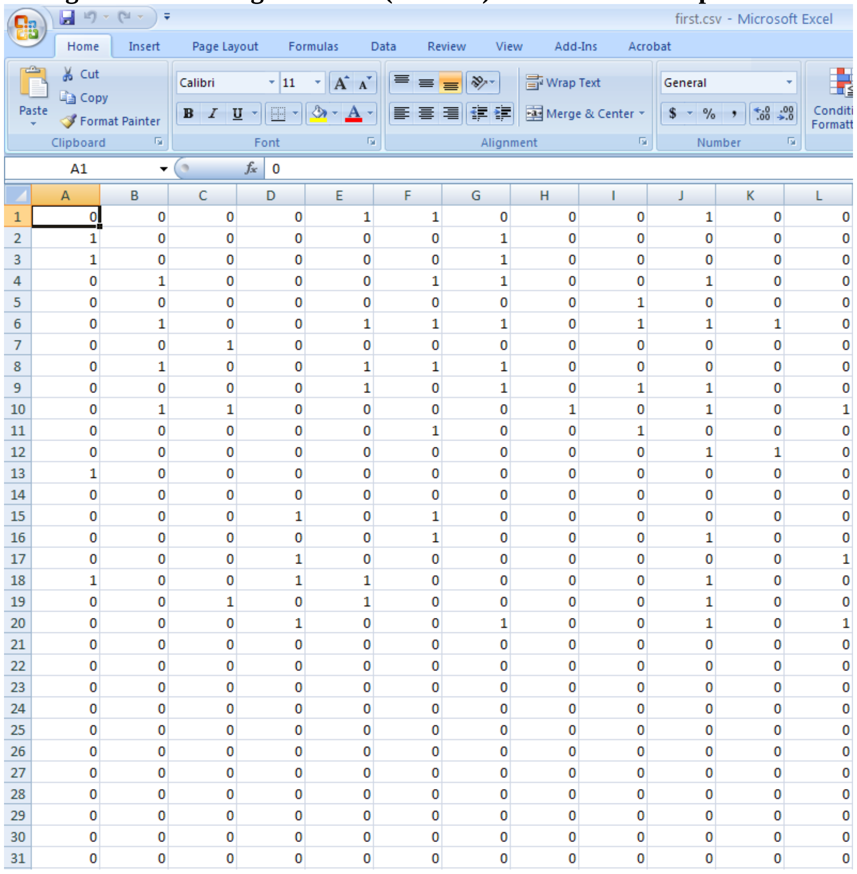
```
construct input.xml
```

This will call the construct executable on the (modified) input deck `input.xml`. Since the input deck was changed in the previous section, the simulation results will be different than those observed in the previous section, even though the execution command is the same.

If Construct runs successfully, it should first print out initialization information, then print out information regarding the time taken for each of the time periods. As before, simulation should take no more than a few moments to complete, and should end with a comment specifying the amount of time elapsed in performing the simulation. If the run is successful, the console should look like Figure 13 (page 109).

However, it is entirely possible that errors may be encountered when trying to run Construct. The most common problem which may occur when attempting to run simulation is that the program will fail to execute because a CSV file is locked by another program. Most commonly, the file `first.csv` or `last.csv` is open in a program such as Microsoft Excel. When Construct runs, it will overwrite any output files that currently exist; output files such as `first.csv` and `last.csv` will then contain data from the most recently completed simulation. If these files cannot be overwritten, Construct will exit and error with the error message seen in Figure 19. To correct this error, close the file in Excel (or rename it if one desires to keep it for comparison purposes) and re-run Construct using the directions outlined above.

**Figure 20: The initial agent-to-knowledge network (first.csv) of the modified input deck**



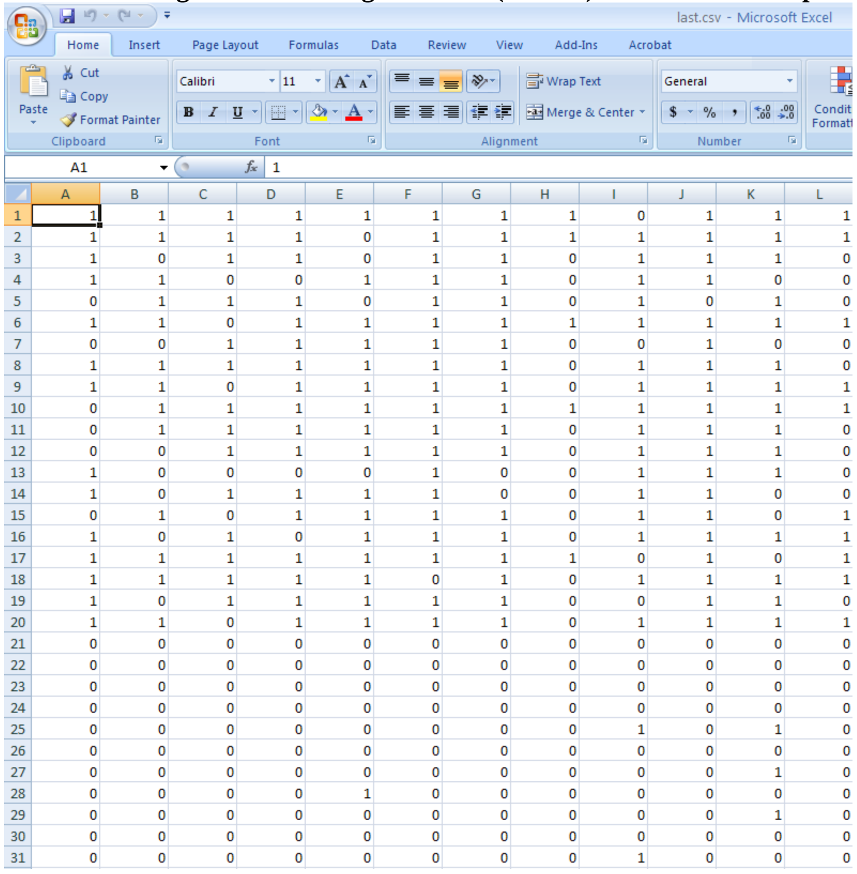
| | A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 7 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 10 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 11 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 13 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 17 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 18 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 19 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 20 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 24 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 27 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 28 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 29 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 30 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

When Construct execution completes successfully, three output files should again be generated: `first.csv`, `last.csv`, and `prob.csv`. Since these outputs were not modified in Section 4.4, they will contain the same type of information as before. Thus, these files will have important information about the status of the simulation: `first.csv` will again contain the initial knowledge state, `last.csv` will again contain the final knowledge state, and `prob.csv` will again contain the probability of interaction between agents in the final time period. Since the simulation setup is different, however, some of the data contained in these files may be quite different.

However, just because the simulation input deck is configured differently does not mean that all the output data will be different. For instance, open the file `first.csv`, which represents the knowledge known by the agents in the initial time period following simulation setup. The first thirty rows and twelve columns of this file should be identical to that of Figure 20. Compare this data with the experiment results when the bridging agents are inactive (`bridging-agents-inactive-first.csv` from Section 4.5, if the file was saved; Figure 15 if the file was not), and note that these values are exactly the same!

This is not an accident: these files are supposed to be identical. This result occurs because the modifications due to the `bridging_agents_active` parameter only occur when the simulation is running, not when the simulation is initialized. Even in the scenario where the bridging agents were inactive, the bridging agents were still initialized with knowledge, though later in the simulation they were not able to act on it. While the bridging agents are now active due to the modifications of Section 4.5, their activity does not affect the simulation initialization

**Figure 21: The final agent-to-knowledge network (last.csv) of the modified input deck**



| | A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 4 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 5 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 6 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 9 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 10 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 11 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 12 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 13 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 14 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 15 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 16 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 17 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 18 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 19 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 20 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 24 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 27 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 28 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 29 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 30 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

procedure. Since the random seed used in the simulation is the same whether the bridging agents are active or not, the "random" components of each of the simulation should be initialized identically. Thus, the simulations start from exactly the same point.

As can be seen in the `last.csv` file, however, the bridging agents can have an important impact on the evolution of the simulation. The `last.csv` file is the file saved at the end of the simulation, and contains information about how information diffused in the society after the agents have interacted. The initial part of this file can be seen in Figure 21.

What is most apparent in examining the `last.csv` file is that agents in agent group A learn some of the knowledge bits in knowledge group K2 (region II from Section 4.4) while agents in agent group B learn some of the bits in knowledge group K1 (region III). For instance, the cell E28 in Figure 21 has a value of 1, indicating that agent number 27 learned this knowledge bit during the course of the simulation. This agent could not have learned the knowledge bit from any other agent in agent group A, as none of the agents in group A initially had this K2 knowledge bit. The knowledge bit also could not have come from one of the bridging agents, as neither of the bridging agents started with this fact. Instead, the only way a group A agent could have learned of this fact would be for a group B agent to interact with a bridging agent, to teach the bridging agent this fact, and then for the bridging agent to pass it to a group A agent.

This result contrasts sharply with the simulation results when the bridging agents were not active, as depicted in Figure 16 (page 113). In that simulation, it was impossible for any information from agent group A to reach agent group B, and vice versa. Thus, the agents in agent group A exchanged only K1 knowledge (thus increasing the density of region I) while the agents in group B exchanged only knowledge in K2. By enabling the bridging agents, diffusion between the two groups was able to occur.

However, it is also possible to ask a number of follow-up questions regarding the diffusion of information. For instance, although agents in group A began to learn K2 knowledge when the bridging agents were active, might this new knowledge come at the expense of the old? To answer this question, it is possible to count the mean number of facts that each agent knew at the end of both simulations. This can be done by summing across each row, and finding the mean number of knowledge bits known by each of the twenty agents in each group. For instance, of the 2400 possible knowledge bits that can be known by agents in group A and group B combined, agents learn 927 (38%) of the knowledge when the bridging agents are not active. When the bridging agents are active, agents in group A and B learn a total of 980 bits (40% of the total), but only 99 of the bits were bits that started with the other group or the bridging agents. Thus, it appears that (with this particular random seed) agents in group A and B learn more knowledge overall when the bridging agents are present. The knowledge learned, however, is less concentrated: group A agents do not learn as much from the K1 group when it is possible to share and learn K2 information. While agents do learn more knowledge, some of what is learned comes at the expense of a more thorough learning of the old.

The purpose of this technical is to describe the Construct input deck, and not necessarily to analyze this particular example in detail. As the input deck was designed in a very abstract fashion for ease of demonstration, a more sophisticated analysis of the results should not be attempted without a discussion of the initial experiment assumptions. No conclusions should be considered reasonable unless they are replicated with different random seeds (see Section 4.8.1). Instead, the above analysis is meant to demonstrate a kind of analysis that can be done using Construct, and hopefully to provide enough information so that the user is able to follow along running the experiment and examining the results.

Such a disclaimer, however, should not prevent the user from understanding that what was presented was a thought-provoking use of the Construct simulation mechanism. Simulations like Construct can be used in order to investigate what-if questions of information diffusion, likelihood of interaction, and network evolution by designing an appropriate input deck. The one discussed in this document, and provided as an appendix, can be used as a template or stepping-stone to the design of a more complex simulation that investigates other phenomena of interest.

### *4.7 Encountering errors*

If the simulation can execute successfully, results similar to that of Figure 13 (page 109) or Figure 18 (page 117) will appear. These specific results should occur for the demo input deck, but analogous results should also appear for all Construct runs if the simulation completed successfully. If execution is unsuccessful, however, the output will be different, indicating that some form of error occurred. Several types of errors may have occurred when running the simulation, all of which would prevent Construct from running or execution from being successful. These include:

- The input file name is mistyped. If this occurs, Construct will error and exit noting that the input file specified is invalid. For instance, if a user types `construct imput.xml` instead of input.xml (replacing the n with an m), Construct will alert the user that the input file with that name cannot be found. The simulation will then exit. To correct this error, verify that the file name is spelled correctly.

- The input file name is completely missing. Should this occur, Construct will assume that the user is attempting to use the default file for the input file. If a user types `construct` at the command line without a following input file name, Construct will assume that the user is effectively using the default input deck `construct.xml`. Construct will look in the current directory for a file with this name, and the simulation will run using that file as the input deck if it is present. If the file cannot be found, Construct will error and exit. To fix this problem, include the name of the desired input file after the `construct` command.
- There is an error in the XML syntax of the input deck. If this occurs Construct will exit with an error indicating that the XML syntax cannot be parsed by Construct. For instance, if an XML tag is opened but not closed, Construct will be confused and unable to parse the XML file. An example of this error would be including the initial `<construct>` tag without also including the closing `</construct>`. Unfortunately, Construct is unable to provide a line number where the error occurred or provide the name of the tag that was opened but not closed. To correct this error when using the demo input deck copied from Appendix A, verify that the entire file has been copied and that there are no extraneous characters introduced when copying. To correct this input file for other input decks, use an XML debugger in order to determine the location of the error and correct the underlying problem.
- The simulation attempts to create an output file and the file cannot be created. If this occurs, the simulation will error and exit alerting the user to the fact that the file cannot be created. Most frequently, this is often due to the output file being open in another application, such as a CSV output file being open in Microsoft Excel. To fix this error, close the other application and allow Construct to overwrite the file. Note, however, that the current contents of the file will be destroyed when Construct runs so it will be necessary to rename the old file if one wishes to keep it. This error was discussed in Section 4.6, as it occurs frequently in trial simulations.
- A specific input is not present in the input deck. If this occurs, Construct will error and exit with an input-specific message. For instance, certain construct generators require certain kinds of inputs – constant value generators require a specific value to set, while generators which create random numbers variables will require a min and max to define a range from which to generate the value. If a user changes from one type of generator to another without updating all the tags, Construct will catch this error. In most cases, Construct will exit and error with a specific error message. Note, however, that Construct is usually unable to provide a specific line number where the error occurred. The user may have to comb the file manually in order to determine where the error occurred. Note that this type of error should not occur with the demo input deck copied from Appendix A, but may be common in other user-created input decks.
- One of the scripts in Construct is malformed. If this occurs, Construct will error and exit with a script-specific message. Construct uses a scripting language to specify more complicated inputs, as introduced in Section 3.2.2 and discussed more extensively in other technical reports. This language can use multiple different types of mathematical, logical, and programmatical operations, but requires a special syntax in order to be parsed correctly. If one of these scripts has an invalid syntax – for instance, a bracket is

misplaced – Construct will be unable to parse the script and will error with a specific message. To fix this problem, find the script that caused the error and determine why Construct was unable to parse it successfully. Note that this type of error should not occur with the demo input deck copied from Appendix A, but may be common in other user-created input decks.

Note that successful Construct execution does not necessarily mean that simulation was successful. Certain errors in logic may be syntactically valid but may lead to erroneous conclusions if the data is analyzed. Thus, the user should carefully examine both the input and the output of a simulation in order to understand what occurred.

## *4.8 Where to go from here*

The techniques outlined in this section merely demonstrated that a small change to the Construct input can lead to substantial changes in patterns of information diffusion. The experiment design was not rigorous, and the resulting change in diffusion pattern was far from proved. While it is beyond the scope of this technical report to tackle such issues, it is worth outlining a number of ways a user could go about further examining this data in his or her research.

### 4.8.1 Additional replications

One data point is simply one data point; it cannot prove a trend. In order to fully explore the phenomena outlined in Section 2, it is necessary to perform additional replications of the experiment and observe repeated patterns of information diffusion. The beauty of simulation is that it allows these replications to be performed quickly and easily.

Performing additional replications with Construct is straightforward. Every time that Construct is run, the simulation will evolve the user-specified initial conditions using a random seed. If the random seed is set to a new number, the results of each replication will be slightly different – agents may choose to interact with one partner over another, leading to highly non-linear changes to how quickly certain facts diffuse in the society. Thus, repeatedly running the simulation with `bridging_agents_active` equal to `true` for several iterations and then `false` for several others will allow for more principled analysis.

However, some care is needed in performing these replications. First, for demonstration purposes, the random seed parameter `seed` has been set to a constant value of 1. The user should change this value to a different number during each replication (or to a value of 0 to allow Construct to choose a random seed based on the time that the simulation was started). Otherwise, the exact same output will be returned each time Construct is run.

Second, Construct will overwrite any existing output files each time it is run. Thus, a user attempting to perform multiple replications using the command line will have to save or rename the output files every time a simulation is performed. This is a very human-intensive process, and while excellent for testing and debugging can be unwieldy for collection of large volumes of data. Alternative techniques such as Perl scripts that call Construct, or batch submission of Construct executables to a cluster, have been developed to facilitate this process.

### 4.8.2 Network analysis of output

Construct has the ability to export networks as output, as described earlier. The network output from Construct can be imported into network visualization tools such as *ORA in order to understand which agents are communicating with each other, which pieces of knowledge are being exchanged, and other network measures of interest. To facilitate this process, Construct can output data in DynetML (the *ORA standard data format) by changing the `output_format` line of the specified `ReadGraphByName` output tag. Alternately, *ORA can read the CSV output generated by Construct.

### 4.8.3 Changing the value of other variables

This input deck has two variables that can be easily manipulated: the `bridging_agents_active` variable, which was manipulated previously, and `short_experiment` at the beginning of the input deck. This variable is used to manipulate the `time_count` variable, which sets the length of the virtual experiment. In the experiments described earlier in this section, the `short_experiment` variable was set to `true` in line 52, as seen below:

```
<var name=“short_experiment” value=“true”/>
```

With `short_experiment` set to `true` the simulation will last for only fifty time periods. This is because the `if` statement of the `time_count` parameter will evaluate as true, meaning that the value of 50 will be used for the time count. It can be modified to take on the value `false`, as seen below.

```
<var name=“short_experiment” value=“false”/>
```

Modifying the `short_experiment` variable to `false` will lead to an experiment of one hundred time periods, since the else branch of the `time_count` parameter will be used.

This manipulation cause the experiment to run for a greater number of time periods, which in turn will mean that information will have more time to diffuse. While the experiment will take slightly longer to run, the increase in running time should be trivial.

More interesting, though, is the interaction between the longer experiment running time and the presence or absence of bridging agents. Because agents will have more time to interact and relay information that originally started with third parties, the bridging agents will have an even greater impact in the longer experiments. In the longer experiment, if the bridging agents are not active then information will diffuse more extensively within each cluster. If the bridging agents are active, though, information will diffuse extensively both within and between the clusters. The experiment results of Section 4.6 will be even more apparent. Construct users are encouraged to observe these results by manipulating the input deck in this fashion.

### 4.8.4 Printing out further information

In the current simulation, data is only gathered in the first and last time period. Thus, one can generalize about the extent of information diffusion but cannot necessarily analyze the rate of transmission in intermediate time periods. One can add additional `operations` to the simulation to print out the knowledge network at other key time periods.

The simplest way to add an additional operation is to copy and modify one of the existing operations. Copying the `ReadGraphByName` operation that prints out `first.csv`, then modifying `output_filename` and `time` attributes, will allow one to investigate how much information is known at an intermediate time period. Thus, if one wishes to examine how much information has diffused at time period ten, one could insert the following operation in the `operations` tag (i.e., after line 1172):

```
<operation name="ReadGraphByName">
  <parameters>
    <param name="graph_name" value="'knowledge network'"/>
    <param name="output_filename" value="intermediate.csv"/>
    <param name="output_format" value="csv"/>
    <param name="run" value="all"/>
    <param name="time" value="25"/>
  </parameters>
</operation>
```

Notice that this operation has the same structure as that which prints out first.csv, but has a different file name and print time. When one re-runs Construct and compares the result of `first.csv`, `intermediate.csv`, and `last.csv`, one can begin to more appropriately investigate how the information has diffused in the sample society.

### 4.8.5 Adding additional features

This input deck has two easy-to-manipulate variables, as have been previously presented. A number of other features of this deck could be changed in order to observe the changes. Such manipulations include:

- <u>Changing the number of possible links that exist in the underlying social network</u>. In the current experiment, agents in each group consider all other group members as potential interaction partners. The `interaction sphere` in Construct represents the agents with which another agent can interact with during an experiment. Thus, in the input deck, the density of the social network within each group is set to 1.0. Decreasing the density, or substituting a different type of network structure (such as a scale-free structure), can change the pattern of diffusion within each group.
- <u>Modifying proximities</u>. The current input deck does not prioritize socio-demographic, social, or physical proximities when agents interact. In fact, all agents are considered to have equal values, and equal weights, on these factors. These can be changed by modifying the values in the `socio demographic similarity network`, the `social similarity network`, the `physical similarity network`, or their corresponding `weight` networks. Modifying these proximities will lead agents to have static preferences for agents that are independent of any knowledge that they share. Modifying these proximities and weights may lead to very different patterns of information diffusion than occur in the stylized case presented.
- <u>Adding beliefs and examining belief diffusion</u>. In the current input deck, no beliefs are modeled. A belief can be added by increasing the `belief_count`, associating any number of facts with the belief by manipulating the `belief knowledge weight network`, and then changing the type of the `standard interaction model` to `mask_mode` as described in Section 3.3.9 and Section 3.5.12. With such changes, one might begin to examine what happens if the facts given to each group are on opposite sides of a particular belief. This could be performed, for instance, by setting all the K1

knowledge group bits to have a belief knowledge weight value of +1 for the belief and those of K2 a weight of -1 for the belief. Interaction within the group will lead to increased polarization, as agents will exchange information and provide each other with additional information that strengthens their initial belief. Interaction with the bridging agents, however, may lead to slightly different outcomes since the bridging agents can pass alternate information to each group.

- Others. The user is free to use this input deck as the basis of his or her own input deck fro examining a phenomena of choice. The three examples suggested here are jumping-off points for a further exploration of Construct and the types of studies that can be performed.

## 5 Conclusion and Closing Comments

This technical report has introduced the reader to a demonstration input deck for Construct version 3.8.build016BRH. Section 2 introduced a problem that motivated the input deck, while Section 3 described the input deck and Section 4 discussed how the deck could be run. While the demonstration problem was a toy problem, the input deck and use description are both indicative of ways to use the simulation.

The focus of this document was to present a demonstration input deck. While it also contains many features that may be found in a user's manual, it was not expressly designed as such. In the process of describing the input deck, though, many key features have been described at a high level. Users are strongly encouraged to examine other Construct literature for additional information regarding simulation behavior. For instance, additional information about agents in Construct can be found in CS-ISR-07-107 (Hirshman & Carley 2007a), more information about networks can be found in CS-ISR-07-116 (Hirshman & Carley 2007b), and more information regarding the scripting language in Construct can be found in CMU-ISR-09-126 (Hirshman et al 2009). Users interested in examining Construct use cases are encouraged to examine many of the papers cited in the bibliography.

As Construct is very much a research tool, its development cycle sometimes outpaces the speed at which new documentation can be generated. While this document has sought to describe the features of Construct version 3.8.build016BRH, this version is already being modified to include additional features. While many of these features supplement what already exists, other changes may require changes to the underlying structure of the parameters, nodeclasses, networks, or operations described in this document. Thus, the input deck and documentation provided here may become obsolete. Users are welcome to use the current document as a starting point for understanding Construct and the way the models and algorithms structured.

Readers of this technical report may wish to use this demo input deck as a stepping-stone to designing their own simulations using Construct. Such exploration is encouraged; there is no learning like first-hand learning! Nevertheless, readers are advised to modify the demonstration input deck described in this document and repurpose it to address their research question. Because the input deck already has many of the required parameters and networks in place, the file can be much more easily overhauled than it can be rewritten from scratch. Replacing the generators in specific networks, adding or modifying variables, and possibly fiddling with the simulation parameters would be an excellent way to begin developing a modified input deck. With frequent testing, users can identify what changes are causing the simulation to error or crash, and adjust appropriately.  Developing an input deck that addresses one's research questions will probably not occur instantaneously, but the end result will hopefully be rewarding.

## Appendix A: Demo Input Deck

The below input deck is a fully-functioning demo that will work with Construct version 3.8.build016BRH.  Compatibility with previous or future versions of Construct is not guaranteed.  In order to run this input deck, copy the below into a text editor and save the file as `input.xml` in order to follow the instructions outlined in Section 4 of this document or the `READ ME` section of the demo.

```
 <!-- *****************************************************************  -->
 <!-- ************************** READ ME ******************************  -->
 <!-- ****************************************************************** -->
 <!--                                                                    -->
 <!-- This is a demonstration input deck, and should work for Construct  -->
 <!-- version 3.8.build016 BRH (April 2010).  Future versions of Construct -->
 <!-- may not support the functionality or behavior described herein.    -->
 <!--                                                                    -->
 <!-- This input deck simulates the interaction between two groups of    -->
 <!-- agents (agent group A and agent group B), each of which have a     -->
 <!-- series of separate knowledge facts (knowledge group 1 and know-    -->
 <!-- ledge group 2, respectively).  These agents are separate and cannot  -->
 <!-- interact with each other.  However, there is also a group of       -->
 <!-- bridging agents (agent group C), agents which span both groups and   -->
 <!-- can allow information to diffuse between them.  If agents in group C -->
 <!-- are active, then information can diffuse from one group to another.  -->
 <!-- If they are not, though, then information will be confined to each   -->
 <!-- group, and a clear boundary between both groups should be observed.  -->
 <!--                                                                    -->
 <!-- In the default state of this input deck, the bridging agents are    -->
 <!-- initially disabled.  To enable them, change the Construct variable   -->
 <!-- "bridging_agents_active" to have the value "true".                 -->
 <!--                                                                    -->
 <!-- This version of Construct must be run from the command line.  To run -->
 <!-- it, open a command prompt, navigate to the location of both the     -->
 <!-- input deck and Construct executable, and execute the following:     -->
 <!--                                                                    -->
 <!-- "Construct.exe input.xml"                                          -->
 <!--                                                                    -->
 <!-- The output of this simulation will be a number of CSV files.  These  -->
 <!-- files should be examined in order to understand the effect of the    -->
 <!-- bridging agents and their effects on information diffusion.  See the -->
 <!-- associated slide presentations and technical reports for additional  -->
 <!-- information.                                                       -->
 <!--                                                                    -->
 <!-- This file is provided as-is.  Modifications to the input file will   -->
 <!-- change the behavior specified above.  For additional information     -->
 <!-- about Construct, see www.casos.cs.cmu.edu/projects/construct.       -->
<construct>

 <!-- ****************************************************************** -->
 <!-- ********************** GLOBAL VARIABLES ************************** -->
 <!-- ****************************************************************** -->
 <construct_vars>

   <!-- length of simulation to run                                      -->
   <!--   if set to 'true', then a short experiment will be run          -->
   <!--   if set to 'false', then a slightly longer experiment will be     -->
   <!--     performed, with additional time for information to diffuse     -->
```

```
<!--       this illustrates the conditional logic that is available     -->
<!--       in the Construct variable system                             -->
<var name="short_experiment" value="true"/>
<var name="time_count" value="
    if(construct::boolvar::short_experiment)
    {
        50
   }
   else
   {
       100
   }
"/>

<!-- whether the bridging agents are active                             -->
<!--   if set to 'true', then the bridging agents can link the groups   -->
<!--   if set to 'false', the bridging agents cannot interact and thus  -->
<!--     no information will flow between the groups                    -->
<var name="bridging_agents_active" value="true"/>

<!-- simulated agents                                                   -->
<!--   two agents groups (A and B) are simulated, and a third agent     -->
<!--   group (C) that will span both groups                             -->
<!--     agent group C is also called the bridging agent group          -->
<var name="agentgroup_A_size" value="20"/>
<var name="agentgroup_B_size" value="20"/>
<var name="agentgroup_C_size" value="2"/>

<!-- boundaries between agent groups                                    -->
<!--   these boundaries are defined so that each of the agent groups    -->
<!--   follow one another: A -> B -> C in the input file                -->
<!--     this illustrates a subset of the math ops available in the     -->
<!--     Construct variable system (which allows vars to be defined     -->
<!--     in terms of other variables) -->
<var name="agentgroup_A_start" value="0"/>
<var name="agentgroup_A_end" value="
   construct::intvar::agentgroup_A_start +
     construct::intvar::agentgroup_A_size - 1"/>
<var name="agentgroup_B_start" value="
   construct::intvar::agentgroup_A_end + 1"/>
<var name="agentgroup_B_end" value="
   construct::intvar::agentgroup_B_start +
     construct::intvar::agentgroup_B_size - 1"/>
<var name="agentgroup_C_start" value="
   construct::intvar::agentgroup_B_end+1"/>
<var name="agentgroup_C_end" value="
   construct::intvar::agentgroup_C_start +
     construct::intvar::agentgroup_C_size - 1"/>

<!-- the overall number of agents and agent group count                 -->
<!--   these are used to initialize the nodeclasses                     -->
<var name="agent_count" value="construct::intvar::agentgroup_C_end+1"/>
<var name="agentgroup_count" value="3"/>

<!-- simulated knowledge                                                -->
<!--   two knowledge groups, K1 given to agent group A and K2 to B      -->
<!--   note that agents in the bridging group will have some knowledge  -->
<!--     from K1 and some from K2; they do not have their own wholly    -->
<!--     unique set of knowledge                                        -->
<var name="knowledgegroup_K1_size" value="30"/>
<var name="knowledgegroup_K2_size" value="30"/>

<!-- boundaries between knowledge groups                                -->
```

```
  <!--   these boundaries are defined so that each of the knowledge       -->
  <!--   groups follow one another: K1 -> K2 in the input file            -->
  <!--   again, note that the bridging agents do not have their own       -->
  <!--     wholly unique set of knowledge, so a knowledge group start    -->
  <!--     and knowledge group end is not created for them              -->
  <var name="knowledgegroup_K1_start" value="0"/>
  <var name="knowledgegroup_K1_end" value="
    construct::intvar::knowledgegroup_K1_start +
      construct::intvar::knowledgegroup_K1_size - 1"/>
  <var name="knowledgegroup_K2_start" value="
    construct::intvar::knowledgegroup_K1_end + 1"/>
  <var name="knowledgegroup_K2_end" value="
    construct::intvar::knowledgegroup_K2_start +
      construct::intvar::knowledgegroup_K2_size - 1"/>

  <!-- the overall amount of knowledge and number of knowledge groups     -->
  <!--   these are used to initialize the nodeclasses                     -->
  <var name="knowledge_count" value="construct::intvar::knowledgegroup_K2_end+1"/>
  <var name="knowledgegroup_count" value="2"/>

  <!-- number of tasks                                                    -->
  <!--   binary, energy tasks are not used in this example                -->
  <!--     the binary and energy task nodeclasses must be present in the  -->
  <!--     input deck, but if they have zero nodes in the nodeclass then  -->
  <!--     they will not impact the simulation                            -->
  <var name="binarytask_count" value="0"/>
  <var name="energytask_count" value="0"/>

  <!-- number of beliefs                                                  -->
  <!--   no beliefs are modeled in this example                           -->
  <!--   (note: if zero beliefs are used and a warning message appears    -->
  <!--   make sure to verify that the value of the simulation parameter   -->
  <!--   "active_model" type is set to 'mask_mode' instead of 'disable')  -->
  <var name="belief_count" value="0"/>

</construct_vars>


<!-- ***************************************************************** -->
<!-- ***************** GLOBAL SIMULATION PARAMETERS ******************* -->
<!-- ***************************************************************** -->
<construct_parameters>

  <!-- seed: the seed for the simulation                                  -->
  <!--   set to zero for a time-dependent seed                            -->
  <!--   set to non-zero for a fixed seed                                 -->
  <param name="seed" value="1"/>

  <!-- verbose_initialization: whether info should be printed             -->
  <!--   the values of variables will be printed during startup           -->
  <param name="verbose_initialization" value="false"/>

  <!-- dynamic environment: whether an 'outside world' agent is modeled   -->
  <!--   the dynamic environment has knowledge which differs from other   -->
  <!--   agents and can change every time period (and thus can introduce  -->
  <!--   new knowledge to the simulation -->
  <param name="dynamic_environment" value="false"/>

  <!-- default type: the type of the agent                                -->
  <!--   the default agent type, if no other agent types are specified    -->
  <!--   this agent type must be defined in the agent_type nodeclass      -->
  <param name="default_agent_type" value="human"/>
```

```
    <!-- learning and forgetting: bool value that specifies how learning    -->
    <!--   and forgetting can occur; agents must always learn, but having   -->
    <!--   them forget is optional                                          -->
    <!--     binary valued learning and forgetting will mean that there is  -->
    <!--     a chance that knowledge will either be fully learned or fully  -->
    <!--     forgotten, and will not have float-valued intermediate states  -->
    <param name="forgetting" value="false"/>
    <param name="binary_forgetting" value="true"/>
    <param name="binary_learning" value="true"/>

    <!-- use_mail: whether mail system is active                            -->
    <!--   if the mail system is active, then agents can use the 'mail'     -->
    <!--   communication mechanism (in the agent_type nodeclass             -->
    <param name="use_mail" value="false"/>

    <!-- default communication weight parameters                            -->
    <!--   these are the weights for transmitting each of these factors     -->
    <!--   if a knowledge bit cannot be sent when communicating, the other  -->
    <!--   values are rescaled appropriately -->
    <param name="communicationWeightForBelief" value="0.2"/>
    <param name="communicationWeightForBeliefTM" value="0.1"/>
    <param name="communicationWeightForFact" value="0.5"/>
    <param name="communicationWeightForKnowledgeTM" value="0.2"/>

    <!-- thread count: for multithreading                                   -->
    <!--   note that multithreading only works on linux systems             -->
    <!--   if multiple runs are being performed, running many processes     -->
    <!--   using single threads is superior                                 -->
    <param name="thread_count" value="1"/>

    <!-- active models: for defining key behavior                           -->
    <!--   the standard interaction model specifies that agents will seek   -->
    <!--     to act via knowledge homophily and expertise, with slight      -->
    <!--     modifications due to static factors (socio-demographic, etc)   -->
    <!--   the standard influence model specifies that agent beliefs will   -->
    <!--     be subject to influence due to influence and influenceability  -->
    <!--   the standard belief model (disabled for this demo) specifies     -->
    <!--     what kind of belief propagation model will be simulated        -->
    <!--   note:                                                            -->
    <!--     1. the order of these models is important; do not change them  -->
    <!--        without good reason, as the simulation may crash            -->
    <!--     2. unlike other areas in the input deck, variable names cannot -->
    <!--        be expanded, and white space / punctuation is critical      -->
    <param name="active_models" value="standard interaction model,standard influence
model,standard belief model(type=>disable)" with="delay_interpolation"/>

    <!-- active mechanisms: for defining network post-processing after load -->
    <!--   literacy and information access parameters can affect the        -->
    <!--     'agent access network', specified in the networks section      -->
    <param name="active_mechanisms" value="none"/>

  </construct_parameters>

  <!-- ****************************************************************** -->
  <!--                  NODE SETS FOR SIMULATION ENTITIES                   -->
  <!--             these are the core entities being simulated              -->
  <!-- ****************************************************************** -->
  <nodes>

    <!-- I. agents: actors in the simulation                                -->
    <!--   this network is REQUIRED and CANNOT have zero nodes              -->
    <!--   all agents must be declared here, even if they are interventions -->
    <!--   agent properties are defined by agent_types and networks, below  -->
```

```
<!--     note that agents can be of multiple different types; the exact -->
<!--     type of agent is specified in the agent_types nodeclass        -->
<nodeclass type="agent" id="agent">
  <properties>
    <property name="generate_nodeclass" value="true"/>
    <property name="generator_type" value="count"/>
    <property name="generator_count" value="construct::intvar::agent_count"/>
  </properties>
</nodeclass>

<!-- II. knowledge: the knowledge represented in the simulation         -->
<!--   this network is REQUIRED and CANNOT have zero nodes              -->
<!--   all knowledge must be declared here                              -->
<!--   however, the relationships between agents and knowledge is       -->
<!--     specified a network                                            -->
<nodeclass type="knowledge" id="knowledge">
  <properties>
    <property name="generate_nodeclass" value="true"/>
    <property name="generator_type" value="count"/>
    <property name="generator_count" value="construct::intvar::knowledge_count"/>
  </properties>
</nodeclass>

<!-- III. beliefs: opinions on a subject                                -->
<!--   this network is REQUIRED and but can have zero nodes             -->
<!--   opinions can be affected by both knowledge and social influence  -->
<!--   as specified earlier, there are no beliefs in this simulation,   -->
<!--   but it is still necessary to initialize this nodeclass in order  -->
<!--   to initialize the corresponding nodeclass                        -->
<nodeclass type="belief" id="belief">
  <properties>
    <property name="generate_nodeclass" value="true"/>
    <property name="generator_type" value="count"/>
    <property name="generator_count" value="construct::intvar::belief_count"/>
  </properties>
</nodeclass>

<!-- IV. binary tasks: actions based on knowledge                       -->
<!--   this network is REQUIRED and but can have zero nodes             -->
<!--   all binary tasks, which are tasks which agents must have         -->
<!--   requisite knowledge for in order to successfully complete        -->
<!--   (or must guess correctly on unknown knowledge (unlikely)         -->
<nodeclass type="binarytask" id="binarytask">
  <properties>
    <property name="generate_nodeclass" value="true"/>
    <property name="generator_type" value="count"/>
    <property name="generator_count" value="construct::intvar::binarytask_count"/>
  </properties>
</nodeclass>

<!-- V. energy tasks: actions based on energy                           -->
<!--   this network is REQUIRED and but can have zero nodes             -->
<!--   tasks which require a constant amount of energy each time        -->
<!--   period in order to perform (energy allocated among tasks)        -->
<nodeclass type="energy task" id="energy task">
  <properties>
    <property name="generate_nodeclass" value="true"/>
    <property name="generator_type" value="count"/>
    <property name="generator_count" value="construct::intvar::energytask_count"/>
  </properties>
</nodeclass>

<!-- VI. agent groups: collections of agents                            -->
```

```
    <!--   this network is REQUIRED and but can have zero nodes             -->
    <!--   generally used for statistical purposes                          -->
    <nodeclass type="agentgroup" id="agentgroup">
      <properties>
        <property name="generate_nodeclass" value="true"/>
        <property name="generator_type" value="count"/>
        <property name="generator_count" value="construct::intvar::agentgroup_count"/>
      </properties>
    </nodeclass>

    <!-- VII. knowledge groups: collections of knowledge                    -->
    <!--   this network is REQUIRED and but can have zero nodes             -->
    <!--   generally used for statistical purposes                          -->
    <nodeclass type="knowledgegroup" id="knowledgegroup">
      <properties>
        <property name="generate_nodeclass" value="true"/>
        <property name="generator_type" value="count"/>
        <property name="generator_count"
value="construct::intvar::knowledgegroup_count"/>
      </properties>
    </nodeclass>

    <!-- VIII. time periods: simulation run periods                         -->
    <!--   this network is REQUIRED but CANNOT have zero nodes              -->
    <!--   note that time period zero is an initialization period where     -->
    <!--   agent knowledge is initialized prior to use                      -->
    <nodeclass type="timeperiod" id="timeperiod">
      <properties>
        <property name="generate_nodeclass" value="true"/>
        <property name="generator_type" value="count"/>
        <property name="generator_count" value="construct::intvar::time_count"/>
      </properties>
    </nodeclass>

    <!-- IX. dummy nodeclass: used for attributes                           -->
    <!--   this network is REQUIRED and SHOULD NOT BE CHANGED               -->
    <!--   nodeclass attributes are set up as <class> x dummy networks      -->
    <!--   note that this nodeclass contains a single node                  -->
    <nodeclass type="dummy_nodeclass" id="dummy_nodeclass">
      <node id="constant" title="constant"/>
    </nodeclass>


    <!-- X. agent types: major agent classes                                -->
    <!--   this network is REQUIRED but CANNOT have zero nodes              -->
    <!--   this contains key information about different agent types       -->
    <nodeclass type="agent_type" id="agent_type">

      <!-- human, a base entity with TM and beliefs                         -->
      <!--   TM (transactive memory) means that it acts on perceptions      -->
      <!--     of others, using perceived similarity to choose partners     -->
      <!--   beliefs means that it can hold, transfer, and send beliefs     -->
      <!--     to others when interacting                                   -->
      <node id="human" title="human">
        <properties>
          <property name="communicationMechanism" value="direct"/>
<property name="canSendCommunication" value="true"/>
          <property name="canReceiveCommunication" value="true"/>
          <property name="canSendKnowledge" value="true"/>
          <property name="canReceiveKnowledge" value="true"/>
          <property name="canSendBeliefs" value="true"/>
          <property name="canReceiveBeliefs" value="true"/>
          <property name="canSendBeliefsTM" value="true"/>
        <property name="canReceiveBeliefsTM" value="true"/>
        <property name="canSendKnowledgeTM" value="true"/>
        <property name="canReceiveKnowledgeTM" value="true"/>
        <property name="canSendReferral" value="true"/>
        <property name="canReceiveReferral" value="true"/>
      </properties>
    </node>
  </nodeclass>

</nodes>

<!-- **************************************************************** -->
<!--                 NETWORKS FOR SIMULATION ENTITIES                 -->
<!--           these are relations between the core entities          -->
<!-- **************************************************************** -->
<networks>

  <!-- 1. agent type name network: the agent_types of the agents          -->
  <!--   this network is a network of strings that correspond to nodes    -->
  <!--     in the agent_type nodeclass; agent_types define whether agents -->
  <!--     can initiate, receive, or send certain kinds of message in the -->
  <!--     simulation, and thus serve as an important shorthand for the   -->
  <!--     kinds of behavior that agents of that type can exhibit         -->
  <network src_nodeclass_type="agent" target_nodeclass_type="dummy_nodeclass"
         id="agent type name network" link_type="string"
        network_type="dense">
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
           last="construct::intvar::agentgroup_A_end"/>
      <cols first="0" last="nodeclass::dummy_nodeclass::count-1"/>
      <param name="constant_value" value="human"/>
    </generator>
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
           last="construct::intvar::agentgroup_B_end"/>
      <cols first="0" last="nodeclass::dummy_nodeclass::count-1"/>
      <param name="constant_value" value="human"/>
    </generator>
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
           last="construct::intvar::agentgroup_C_end"/>
      <cols first="0" last="nodeclass::dummy_nodeclass::count-1"/>
      <param name="constant_value" value="human"/>
    </generator>
  </network>

  <!-- 2. agent initiation count: number of interaction initiations       -->
  <!--   agents may initiate up to this number of interactions, but may   -->
  <!--   choose to "interact with themselves" if no suitable partner      -->
  <!--   is available at that time                                        -->
  <!--     note: interaction only, does not affect transmission           -->
  <network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
         id="agent initiation count network" link_type="int"
        network_type="dense">
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
           last="construct::intvar::agentgroup_A_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
    </generator>
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
           last="construct::intvar::agentgroup_B_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
    </generator>
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
           last="construct::intvar::agentgroup_C_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
    </generator>
  </network>

  <!-- 3. agent reception count: number of interaction receptions         -->
  <!--   agents are open to receiving this number of interactions each    -->
  <!--   period, but may not be contacted by other agents                 -->
  <!--     note: interaction only, does not affect transmission           -->
  <network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
         id="agent reception count network" link_type="int"
        network_type="dense">
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
           last="construct::intvar::agentgroup_A_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
    </generator>
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
           last="construct::intvar::agentgroup_B_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
    </generator>
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
           last="construct::intvar::agentgroup_C_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
    </generator>
  </network>

  <!-- 4. agent message complexity: number of items transmitted           -->
  <!--   agents send this number of knowledge bits in each interaction    -->
  <!--     note: transmission only, does not affect interaction           -->
  <network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
         id="agent message complexity network" link_type="int"
        network_type="dense">
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_A_start"
           last="construct::intvar::agentgroup_A_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
    </generator>
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
            last="construct::intvar::agentgroup_B_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
    </generator>
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
           last="construct::intvar::agentgroup_C_end"/>
      <cols first="0" last="nodeclass::timeperiod::count-1"/>
      <param name="constant_value" value="1"/>
    </generator>
  </network>
```

```
    <!-- 5. be influenced: measure of susceptibility to belief influence    -->
    <!--   note that the susceptibility does not change by source agent     -->
    <!--     note: transmission only, does not affect interaction           -->
    <network src_nodeclass_type="agent" target_nodeclass_type="dummy_nodeclass"
           id="beInfluenced network" link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0" last="nodeclass::agent::count-1"/>
        <cols first="0" last="0"/>
       <param name="min" value="0.0"/>
       <param name="max" value="1.0"/>
      </generator>
    </network>

    <!-- 6. influentialness: how strong this agent can influence others     -->
    <!--   note that the influentialness does not change by target agent    -->
    <!--     note: transmission only, does not affect interaction           -->
    <network src_nodeclass_type="agent" target_nodeclass_type="dummy_nodeclass"
      id="influencialness network" link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0" last="nodeclass::agent::count-1"/>
        <cols first="0" last="0"/>
       <param name="min" value="0.0"/>
       <param name="max" value="1.0"/>
      </generator>
```
**Error! Hyperlink reference not valid.**
```
    </network>

    <!-- 7. selective attention effect: percent of knowledge examined       -->
    <!--   due to bounded rationality, agents may not examine all their     -->
    <!--   knowledge when deciding what to send to an interaction partner   -->
    <!--     a value of 1.0 means that all knowledge is considered          -->
    <!--     a value less than 1.0 means that that each fact will be        -->
    <!--        examined with that probability and be considered for trans- -->
    <!--        mission (note that the facts that are actually sent are     -->
    <!--        governed by many factors, such as the message complexity)   -->
    <!--     note: transmission only, does not affect interaction           -->
    <network src_nodeclass_type="agent" target_nodeclass_type="dummy_nodeclass"
           id="agent selective attention effect network" link_type="float"
          network_type="dense">
      <generator type="randomuniform">
        <rows first="0" last="nodeclass::agent::count-1"/>
        <cols first="0" last="0"/>
       <param name="min" value="1.0"/>
       <param name="max" value="1.0"/>
      </generator>
    </network>

    <!-- 8. learning rate: percent of sent message learned by agent         -->
    <!--   learning rates are dependent on the type of learning enabled     -->
    <!--   the learning type is a simulation parameter:                     -->
    <!--     binary learning will mean that the agent will have an x%       -->
    <!--       chance of fully learning the knowledge, so an agent          -->
    <!--     non-binary learning will mean the agent's knowledge is         -->
    <!--       increased by x% of the sender's original message             -->
    <!--   for example, with a learning rate of 70% (0.70):                 -->
    <!--     if binary learning is enabled, the receiver will have a 7 / 10 -->
    <!--       chance of learning the sender's knowledge fully              -->
    <!--     if non-binary learning is enabled an agent's knowledge will    -->
    <!--       be increased by 0.70 if the sender sends a value of 1.00     -->
    <network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
           id="agent learning rate network" link_type="float" network_type="dense">
      <generator type="randomuniform">
```

```
        <rows first="0" last="nodeclass::agent::count-1"/>
        <cols first="0" last="nodeclass::knowledge::count-1"/>
       <param name="min" value="1.0"/>
       <param name="max" value="1.0"/>
      </generator>
    </network>

    <!-- 9. forgetting rate: boolean forgetting (if enabled)                 -->
    <!--   specifies the chance that a knowledge bit will be fully          -->
    <!--   forgotten during time period, meaning knowledge becomes 0.0      -->
    <!--     the 'forgetting' simulation parameter must be active for this  -->
    <!--     to have any effect                                             -->
    <!--     note: forgetting occurs after fact transmission                -->
    <network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
           id="agent forgetting rate network" link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0" last="nodeclass::agent::count-1"/>
        <cols first="0" last="nodeclass::knowledge::count-1"/>
       <param name="min" value="0.0"/>
       <param name="max" value="0.0"/>
      </generator>
    </network>

    <!-- 10. learn by doing rate: learning rate from tasks                  -->
    <!--   agents can learn new knowledge if the knowledge is associated    -->
    <!--   with a task they are performing                                  -->
    <network src_nodeclass_type="agent" target_nodeclass_type="dummy_nodeclass"
           id="agent learn by doing rate network" link_type="float"
network_type="dense">
      <generator type="randomuniform">
        <rows first="0" last="nodeclass::agent::count-1"/>
        <cols first="0" last="0"/>
       <param name="min" value="0.0"/>
       <param name="max" value="0.0"/>
      </generator>
    </network>

    <!-- 11. knowledge network: initial agent knowledge                     -->
    <!--   agent knowledge starts this way, but will increase via           -->
    <!--   interaction or decrease via forgetting                           -->
    <!--     note: in this example, the agents in group A start with        -->
    <!--     knowledge from group K1, while those in B learn from group K2; -->
    <!--     those in group C learn some from both groups, so that they     -->
    <!--     can bridge the divide between the two groups                   -->
    <network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
           id="knowledge network" link_type="float" network_type="dense">
      <generator type="randombinary">
        <rows first="construct::intvar::agentgroup_A_start"
             last="construct::intvar::agentgroup_A_end"/>
        <cols first="construct::intvar::knowledgegroup_K1_start"
             last="construct::intvar::knowledgegroup_K1_end"/>
       <param name="mean" value="0.20"/>
      </generator>
      <generator type="randombinary">
        <rows first="construct::intvar::agentgroup_B_start"
             last="construct::intvar::agentgroup_B_end"/>
        <cols first="construct::intvar::knowledgegroup_K2_start"
             last="construct::intvar::knowledgegroup_K2_end"/>
       <param name="mean" value="0.20"/>
      </generator>
      <generator type="randombinary">
        <rows first="construct::intvar::agentgroup_C_start"
             last="construct::intvar::agentgroup_C_end"/>
        <cols first="construct::intvar::knowledgegroup_K1_start"
             last="construct::intvar::knowledgegroup_K1_end"/>
       <param name="mean" value="0.10"/>
      </generator>
      <generator type="randombinary">
        <rows first="construct::intvar::agentgroup_C_start"
             last="construct::intvar::agentgroup_C_end"/>
        <cols first="construct::intvar::knowledgegroup_K2_start"
             last="construct::intvar::knowledgegroup_K2_end"/>
       <param name="mean" value="0.10"/>
      </generator>
    </network>

    <!-- 12. agent belief network: initial agent belief                     -->
    <!--   over time, this belief will change due to knowledge and          -->
    <!--   social influence factors                                         -->
    <network src_nodeclass_type="agent" target_nodeclass_type="belief"
           id="agent belief network" link_type="float" network_type="dense">
      <generator type="constant">
        <rows first="0" last="nodeclass::agent::count-1"/>
        <cols first="0" last="nodeclass::belief::count-1"/>
        <param name="constant_value" value="0"/>
      </generator>
    </network>

    <!-- 13. belief weight: how much each belief is affected by each knowledge   -->
    <!--   positive weights show agreement with a belief, while negative    -->
    <!--   weights show disagreement; typical values are in the -1 to 1     -->
    <!--   range with 0 being no impact on the belief; one knowledge can    -->
    <!--   potentially have impacts on many beliefs                         -->
    <!--     note: since there are no modeled beliefs, the belief weight    -->
    <!--       for all facts is set to zero                                 -->
    <network src_nodeclass_type="belief" target_nodeclass_type="knowledge"
           id="belief knowledge weight network" link_type="float"
           network_type="dense">
      <generator type="constant">
        <rows first="0" last="nodeclass::belief::count-1"/>
        <cols first="0" last="nodeclass::knowledge::count-1"/>
       <param name="constant_value" value="0"/>
      </generator>
    </network>

    <!-- 14. interaction sphere: possible agent-agent interactions          -->
    <!--   the interaction sphere is a hard cap on which agents keep track  -->
    <!--   of which others' transactive memory; agents not in the inter-    -->
    <!--   action sphere will not be contacted via standard homophily-      -->
    <!--   based interaction since the ego will not perceive the alter's    -->
    <!--   existence                                                        -->
    <!--     for a more malleable version of the interaction sphere, use    -->
    <!--     the agent access network, which can be used to prevent         -->
    <!--     agent-agent contact, or the access network, which can be       -->
    <!--     used to remove an agent at a particular time period            -->
    <!--        note: in the below example, every agent has every other     -->
    <!--        agent in its interaction sphere; the symmetric parameter    -->
    <!--        to the generators ensures that if the value at (row, col)   -->
    <!--        is set, then the value at (col, row) will be set as well    -->
    <network src_nodeclass_type="agent" target_nodeclass_type="agent"
           id="interaction sphere network" link_type="bool"
           network_type="dense">
      <generator type="randombinary">
        <rows first="construct::intvar::agentgroup_A_start"
             last="construct::intvar::agentgroup_A_end"/>
        <cols first="construct::intvar::agentgroup_A_start"
```

```
          last="construct::intvar::agentgroup_A_end"/>
   <param name="mean" value="1.0"/>
   <param name="symmetric" value="true"/>
  </generator>
  <generator type="randombinary">
    <rows first="construct::intvar::agentgroup_B_start"
          last="construct::intvar::agentgroup_B_end"/>
    <cols first="construct::intvar::agentgroup_B_start"
          last="construct::intvar::agentgroup_B_end"/>
   <param name="mean" value="1.0"/>
   <param name="symmetric" value="true"/>
  </generator>
  <generator type="randombinary">
    <rows first="construct::intvar::agentgroup_C_start"
          last="construct::intvar::agentgroup_C_end"/>
    <cols first="construct::intvar::agentgroup_C_start"
         last="construct::intvar::agentgroup_C_end"/>
   <param name="mean" value="1.0"/>
   <param name="symmetric" value="true"/>
  </generator>
  <generator type="randombinary">
    <rows first="construct::intvar::agentgroup_A_start"
          last="construct::intvar::agentgroup_A_end"/>
    <cols first="construct::intvar::agentgroup_B_start"
          last="construct::intvar::agentgroup_B_end"/>
   <param name="mean" value="0.0"/>
   <param name="symmetric" value="true"/>
  </generator>
  <generator type="randombinary">
    <rows first="construct::intvar::agentgroup_A_start"
          last="construct::intvar::agentgroup_A_end"/>
    <cols first="construct::intvar::agentgroup_C_start"
          last="construct::intvar::agentgroup_C_end"/>
   <param name="mean"
          value="construct::boolvar::bridging_agents_active"/>
   <param name="symmetric" value="true"/>
  </generator>
  <generator type="randombinary">
    <rows first="construct::intvar::agentgroup_B_start"
          last="construct::intvar::agentgroup_B_end"/>
    <cols first="construct::intvar::agentgroup_C_start"
          last="construct::intvar::agentgroup_C_end"/>
   <param name="mean" value="construct::boolvar::bridging_agents_active"/>
   <param name="symmetric" value="true"/>
  </generator>
</network>

<!-- 15. access network: agent is allowed access to another agent       -->
<!--   supplement to the interaction sphere since the int. sphere       -->
<!--   controls whether interaction can happen between any pair of      -->
<!--   agents (can be modified internally by Construct)                 -->
<!--     this network will be modified if any of the post-processing    -->
<!--     routines such as literacy are active                           -->
<!--       note: in this experiment, all agents can access all others   -->
<!--     note: interaction only (but necessarily means no transmission) -->
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
       id="access network" link_type="float" network_type="dense">
  <generator type="randomuniform">
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::agent::count-1"/>
   <param name="min" value="1.0"/>
   <param name="max" value="1.0"/>
  </generator>
```

```
    </network>

    <!-- 16. agent active time period: periods when agents are active       -->
    <!--   only active agents will interact, hold beliefs, etc              -->
    <!--   inactive agents will keep their knowledge, but will not update   -->
    <!--   until they become active again                                   -->
    <!--     note: agent groups A and B are always active, while agent      -->
    <!--     group C's activity depends upon the value of the variable      -->
    <!--     'bridging_agents_active'                                       -->
    <!--     note: interaction only (but necessarily means no transmission) -->
    <network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
           id="agent active timeperiod network" link_type="bool"
           network_type="dense">
      <generator type="constant">
        <rows first="construct::intvar::agentgroup_A_start"
             last="construct::intvar::agentgroup_A_end"/>
        <cols first="0" last="nodeclass::timeperiod::count-1"/>
       <param name="constant_value" value="1"/>
      </generator>
      <generator type="constant">
        <rows first="construct::intvar::agentgroup_B_start"
             last="construct::intvar::agentgroup_B_end"/>
        <cols first="0" last="nodeclass::timeperiod::count-1"/>
       <param name="constant_value" value="1"/>
      </generator>
      <generator type="constant">
        <rows first="construct::intvar::agentgroup_C_start"
             last="construct::intvar::agentgroup_C_end"/>
        <cols first="0" last="nodeclass::timeperiod::count-1"/>
       <param name="constant_value"
value="construct::boolvar::bridging_agents_active"/>
      </generator>
    </network>

    <!-- 17. physical proximity: how physically "close" two agents are      -->
    <!--   serves as a proxy for geographic distance;                       -->
    <!--   this is one of three factors which affects the static agent-     -->
    <!--   agent likelihood of interaction, and is affected by the weight   -->
    <!--   of the same name                                                 -->
    <!--     not used in this experiment, set to 0.0                        -->
    <!--     note: interaction only, does not affect transmission           -->
    <network src_nodeclass_type="agent" target_nodeclass_type="agent"
           id="physical proximity network" link_type="float" network_type="dense">
      <generator type="randomuniform">
        <rows first="0" last="nodeclass::agent::count-1"/>
        <cols first="0" last="nodeclass::agent::count-1"/>
       <param name="min" value="0.0"/>
       <param name="max" value="0.0"/>
      </generator>
    </network>

    <!-- 18. s-d proximity: how socio-demographically similar agents are    -->
    <!--   serves as a proxy for the "birds of a feather" principle;        -->
    <!--   this is one of three factors which affects the static agent-     -->
    <!--   agent likelihood of interaction, and is affected by the weight   -->
    <!--   of the same name                                                 -->
    <!--     not used in this experiment, set to 0.0                        -->
    <!--     note: interaction only, does not affect transmission           -->
    <network src_nodeclass_type="agent" target_nodeclass_type="agent"
           id="sociodemographic proximity network" link_type="float"
           network_type="dense">
      <generator type="randomuniform">
        <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::agent::count-1"/>
   <param name="min" value="0.0"/>
   <param name="max" value="0.0"/>
  </generator>
</network>

<!-- 19. social proximity: how similar two agents are socially          -->
<!--   serves as a proxy for other types of social distance;            -->
<!--   this is one of three factors which affects the static agent-     -->
<!--   agent likelihood of interaction, and is affected by the weight   -->
<!--   of the same name                                                 -->
<!--     used only as a minimal catchall here since physical and social -->
<!--     proximities are not being directly modeled                     -->
<!--     note: interaction only, does not affect transmission           -->
<network src_nodeclass_type="agent" target_nodeclass_type="agent"
       id="social proximity network" link_type="float" network_type="dense">
  <generator type="randomuniform">
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::agent::count-1"/>
   <param name="min" value="1.0"/>
   <param name="max" value="1.0"/>
  </generator>
</network>


<!-- 20. physical proximity weight: weight of physical proximity        -->
<!--   this weight, and the other proximity weights, must sum to 1.0    -->
<!--     not used for this experiment - set to 0                        -->
<!--     note: interaction only, does not affect transmission           -->
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
       id="physical proximity weight network" link_type="float"
       network_type="dense">
  <generator type="randomuniform">
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::timeperiod::count-1"/>
   <param name="min" value="0.0"/>
   <param name="max" value="0.0"/>
  </generator>
</network>

<!-- 21. s-d proximity weight: weight of similar s-d proximity          -->
<!--   this weight, and the other proximity weights, must sum to 1.0    -->
<!--     not used for this experiment - set to 0                        -->
<!--     note: interaction only, does not affect transmission           -->
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
       id="sociodemographic proximity weight network" link_type="float"
       network_type="dense">
  <generator type="randomuniform">
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::timeperiod::count-1"/>
   <param name="min" value="0.0"/>
   <param name="max" value="0.0"/>
  </generator>
</network>

<!-- 22. social proximity weight: weight of social proximity            -->
<!--   this weight, and the other proximity weights, must sum to 1.0    -->
<!--     used by default for this experiment - set to 1.0               -->
<!--     note: interaction only, does not affect transmission           -->
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
       id="social proximity weight network" link_type="float"
       network_type="dense">
  <generator type="randomuniform">
```

```
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::timeperiod::count-1"/>
   <param name="min" value="1.0"/>
   <param name="max" value="1.0"/>
  </generator>
</network>

<!-- 23. binary task similarity weight: similarity due to binary task   -->
<!--   this weight, and the other proximity weights, must sum to 1.0    -->
<!--     used by default for this experiment - set to 1.0               -->
<!--     note: interaction only, does not affect transmission           -->
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
       id="binarytask similarity weight network" link_type="float"
       network_type="dense">
  <generator type="randomuniform">
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::timeperiod::count-1"/>
   <param name="min" value="0.0"/>
   <param name="max" value="0.0"/>
  </generator>
</network>

<!-- 24. binary task assignment: which agents assigned to which tasks   -->
<!--   agents can only complete the tasks to which they are assigned,   -->
<!--   though agents can be assigned to one or more tasks               -->
<!--     agents assigned to similar tasks have greater similarity,      -->
<!--     a factor that is modified by the binary task similarity weight -->
<network src_nodeclass_type="agent" target_nodeclass_type="binarytask"
       id="binarytask assignment network" link_type="bool"
       network_type="dense">
  <generator type="randombinary">
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::binarytask::count-1"/>
   <param name="mean" value="0"/>
  </generator>
</network>

<!-- 25. binary task requirement: knowledge required to complete task   -->
<!--   specifies the knowledge required in order to perform the task    -->
<!--   if the agent does not have the value specified in the truth net- -->
<!--   work, it will have to guess with 50% accuracy (and thus will be  -->
<!--   highly unlikely to be able to complete the task that period      -->
<!--     not used in this experiment                                    -->
<network src_nodeclass_type="knowledge" target_nodeclass_type="binarytask"
       id="binarytask requirement network" link_type="bool"
       network_type="dense">
  <generator type="randombinary">
    <rows first="0" last="nodeclass::knowledge::count-1"/>
    <cols first="0" last="nodeclass::binarytask::count-1"/>
   <param name="mean" value="0"/>
  </generator>
</network>

<!-- 26. binary task truth: knowledge associated with task truth        -->
<!--   the value that the agent must have in order to successfully com- -->
<!--   plete the task; if the value in this matrix is a one, then the   -->
<!--   agent must know the knowledge, if it is a zero the agent must    -->
<!--   not have the knowledge                                           -->
<!--     not used in this experiment                                    -->
<network src_nodeclass_type="knowledge" target_nodeclass_type="binarytask"
       id="binarytask truth network" link_type="bool" network_type="dense">
  <generator type="randombinary">
    <rows first="0" last="nodeclass::knowledge::count-1"/>
    <cols first="0" last="nodeclass::binarytask::count-1"/>
   <param name="mean" value="0"/>
  </generator>
</network>

<!-- 27. knowledge similarity weight: per-agent weight on similarity    -->
<!--   an ego agent perceives that it has similarity to an alter when   -->
<!--   the ego knows a knowledge and knows that an alter knows it too   -->
<!--     similarity and expertise weights must sum to 1.0               -->
<!--     this simulation relies on 80% similarity, 20% expertise        -->
<!--     note: interaction only, does not affect transmission           -->
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
       id="knowledge similarity weight network" link_type="float"
       network_type="dense">
  <generator type="randomuniform">
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::timeperiod::count-1"/>
   <param name="min" value="0.8"/>
   <param name="max" value="0.8"/>
  </generator>
</network>

<!-- 28. knowledge expertise weight: per-agent weight on difference -->
<!--   an ego agent perceives that alter has expertise when the ego     -->
<!--   does not know a knowledge but knows that an alter knows it       -->
<!--     similarity and expertise weights must sum to 1.0               -->
<!--     this simulation relies on 80% similarity, 20% expertise        -->
<network src_nodeclass_type="agent" target_nodeclass_type="timeperiod"
       id="knowledge expertise weight network" link_type="float"
       network_type="dense">
  <generator type="randomuniform">
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::timeperiod::count-1"/>
   <param name="min" value="0.2"/>
   <param name="max" value="0.2"/>
  </generator>
</network>

<!-- 29. interaction weight: weight placed on knowledge for interaction -->
<!--   knowledge with high interaction weight is counted proportionally -->
<!--   higher for both similarity and expertise calculations (so a      -->
<!--   knowledge bit with weight 2.0 will have twice as much impact on  -->
<!--   similarity and expertise calculations as a bit with weight 1.0   -->
<!--     thus the former will contribute more to interaction partner    -->
<!--       selection by making the fact more salient                    -->
<!--     allows some knowledge bits to be perceived as more important   -->
<!--     for driving interaction, even if such knowledge is not         -->
<!--     necessarily often (or ever) communicated                       -->
<!--       compare with knowledge priority and transmission weight      -->
<!--     note: all knowledge is weighted equally in this experiment     -->
<network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
       id="interaction knowledge weight network" link_type="float"
       network_type="dense">
  <generator type="randomuniform">
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::knowledge::count-1"/>
   <param name="min" value="1.0"/>
   <param name="max" value="1.0"/>
  </generator>
</network>

<!-- 30. transmission weight: weight on knowledge for communication     -->
<!--   knowledge with a high transmission weight is counted propor-     -->
```

```
<!--   tionally higher when selecting a knowledge for transmission (so  -->
<!--   all things being equal, a knowledge bit with transmission weight -->
<!--     2.0 is twice as likely to be sent in a message than one with   -->
<!--     transmission weight 1.0                                        -->
<!--     allows certain knowledge bits to be perceived as more impor-   -->
<!--     tant for communication, even if such knowledge is not neces-   -->
<!--     sarily the driving force behind what drives agents to interact -->
<!--       compare with knowledge priority and interaction weight       -->
<!--     note: all knowledge is weighted equally in this experiment     -->
<network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
       id="transmission knowledge weight network" link_type="float"
       network_type="dense">
  <generator type="randomuniform">
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::knowledge::count-1"/>
   <param name="min" value="1.0"/>
   <param name="max" value="1.0"/>
  </generator>
</network>

<!-- 31. knowledge priority: priority on a knowledge bit                -->
<!--   while the transmission weight makes it more likely that a know-  -->
<!--   ledge bit will be selected when an agent is communicating, a     -->
<!--   high knowledge priority guarantees that a bit will be sent       -->
<!--     when selecting knowledge to transmit all knowledge from a      -->
<!--       higher priority level will be transferred before any know-   -->
<!--       ledge for a lower level are transferred, meaning that the    -->
<!--       transmission weight is striated by knowledge priority        -->
<!--     note: all knowledge has the same priority in this experiment   -->
<network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
  id="knowledge priority network" link_type="int" network_type="dense">
  <generator type="randomuniform">
    <rows first="0" last="nodeclass::agent::count-1"/>
    <cols first="0" last="nodeclass::knowledge::count-1"/>
   <param name="min" value="1.0"/>
   <param name="max" value="1.0"/>
  </generator>
</network>

<!-- 32. learnable knowledge: knowledge bits that agents can learn      -->
<!--   used to keep groups disjoint by ensuring that certain agents     -->
<!--   cannot learn certain knowledge bits                              -->
<!--     this network can, for instance, be used to guarantee that an   -->
<!--       agent will not learn a particular bit even if its inter-     -->
<!--       action partners attempt to send the bit in a message         -->
<!--     note: transmission only, does not affect interaction           -->
<network src_nodeclass_type="agent" target_nodeclass_type="knowledge"
       id="learnable knowledge network" link_type="bool"
       network_type="dense">
  <generator type="randomuniform">
   <rows first="0" last="nodeclass::agent::count-1"/>
   <cols first="0" last="nodeclass::knowledge::count-1"/>
   <param name="min" value="1.0"/>
   <param name="max" value="1.0"/>
  </generator>
</network>

<!-- 33. agent group membership: groupings for agent statistics         -->
<!--   assigns agents to agent groups for internal tracking             -->
<network src_nodeclass_type="agent" target_nodeclass_type="agentgroup"
         id="agent group membership network" link_type="bool"
         network_type="dense">
  <generator type="constant">
```

```
      <rows first="construct::intvar::agentgroup_A_start"
            last="construct::intvar::agentgroup_A_end"/>
      <cols first="0" last="0"/>
     <param name="constant_value" value="1"/>
    </generator>
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_B_start"
            last="construct::intvar::agentgroup_B_end"/>
      <cols first="1" last="1"/>
     <param name="constant_value" value="1"/>
    </generator>
    <generator type="constant">
      <rows first="construct::intvar::agentgroup_C_start"
           last="construct::intvar::agentgroup_C_end"/>
      <cols first="2" last="2"/>
     <param name="constant_value" value="1"/>
    </generator>
  </network>

  <!-- 34. knowledge group membership: groupings for knowledge statistics -->
  <!--   assigns knowledge to knowledge groups for internal tracking      -->
  <network src_nodeclass_type="knowledge" target_nodeclass_type="knowledgegroup"
         id="knowledge group membership network" link_type="bool"
         network_type="dense">
    <generator type="constant">
      <rows first="construct::intvar::knowledgegroup_K1_start"
           last="construct::intvar::knowledgegroup_K1_end"/>
      <cols first="0" last="0"/>
     <param name="constant_value" value="1"/>
    </generator>
    <generator type="constant">
      <rows first="construct::intvar::knowledgegroup_K2_start"
           last="construct::intvar::knowledgegroup_K2_end"/>
      <cols first="1" last="1"/>
     <param name="constant_value" value="1"/>
    </generator>
  </network>

</networks>

<!-- ****************************************************************** -->
<!--                 INITIAL TRANSACTIVE MEMORY VALUES                  -->
<!--               these are agent perceptions of others                -->
<!-- ****************************************************************** -->
<transactivememory>

  <!-- 1. knowledge transactive memory: perceptions of others' knowledge  -->
  <!--  the perception generator sets ego's perception of alter's know-   -->
  <!--  ledge for the bits in the specified range                         -->
  <!--    this generation sets up ego agents such that they               -->
  <!--    a. for every knowledge an alter knows, an ego agent has a 50%   -->
  <!--       probability of correctly perceiving the alter knows it       -->
  <!--    b. for every knowledge an alter does not know, an ego agent     -->
  <!--       has a 100% prob of correctly perceiving the alter doesn't    -->
  <!--       know it                                                      -->
  <!--    because transactive memory is represented as a boolean, it is   -->
  <!--    necessary to convert float-valued knowledge into booleans;      -->
  <!--    this is done by setting the binarization threshold              -->
  <!--      values greater than this value are "known" by the agent and   -->
  <!--      TM will be set according to the false negative rate           -->
  <!--      values less than this value are "not known" by the agent and  -->
  <!--      TM will be set according to the false positive rate           -->
  <network id="knowledge transactive memory network"
```

```
         ego_nodeclass_type="agent" src_nodeclass_type="agent"
        target_nodeclass_type="knowledge" link_type="bool"
        network_type="TMBool" associated_network="knowledge network">
    <generator type="perception_based">
      <ego first="0" last="nodeclass::agent::count-1"/>
      <alter first="0" last="nodeclass::agent::count-1"/>
      <transactive first="0" last="nodeclass::knowledge::count-1"/>
      <param name="false_negative_rate" value="0.50"/>
      <param name="false_positive_rate" value="0.0"/>
      <param name="rounding_threshold" value="0.0"/>
      <param name="verbose" value="false"/>
    </generator>
  </network>

  <!-- 2. belief transactive memory: perceptions of others' beliefs       -->
  <!--   this is not applicable to this experiment, as the belief model   -->
  <!--   is disabled and zero beliefs are modeled                         -->
  <!--     however, should the user wish to add beliefs to the model, the -->
  <!--     following generator will guarantee that agents have a false    -->
  <!--     positive rate of 25% (they will 25% of the time believe that   -->
  <!--     the alter holds the belief when it does not) and a false neg-  -->
  <!--     ative rate of 25% (they will 25% of the time believe that the  -->
  <!--     alter doesn't holds the belief when it actually does           -->
  <network id="belief transactive memory network"
         ego_nodeclass_type="agent" src_nodeclass_type="agent"
        target_nodeclass_type="belief" link_type="float"
        network_type="TMFloat" associated_network="agent belief network">
    <generator type="perception_based">
      <ego first="0" last="nodeclass::agent::count-1"/>
      <alter first="0" last="nodeclass::agent::count-1"/>
      <transactive first="0" last="nodeclass::belief::count-1"/>
      <param name="false_negative_rate" value="0.25"/>
      <param name="false_positive_rate" value="0.25"/>
      <param name="rounding_threshold" value="0.0"/>
      <param name="verbose" value="false"/>
    </generator>
  </network>
</transactivememory>

<!-- ****************************************************************** -->
<!--                 OPERATIONS FOR SIMULATION OUTPUT                   -->
<!--            these are the values that are printed and saved         -->
<!-- ****************************************************************** -->
<operations>

  <!-- 1. read graph: knowledge network                                   -->
  <!--   determine the agent's knowledge in the first time period of the  -->
  <!--   simulation (reads the knowledge network directly as imported)    -->
  <!--     the ReadGraphByName operation can read any of the networks     -->
  <!--     initialized above by replacing the graph_name parameter with   -->
  <!--     the (quoted) name of the network                               -->
  <operation name="ReadGraphByName">
    <parameters>
      <param name="graph_name" value="'knowledge network'"/>
      <param name="output_filename" value="first.csv"/>
      <param name="output_format" value="csv"/>
      <param name="run" value="all"/>
      <param name="time" value="first"/>
    </parameters>
  </operation>

  <!-- 2. read graph: knowledge network                                   -->
  <!--   determine the agent's knowledge in the last time period of the   -->
```

```
    <!--   simulation, for comparison with the initial knowledge network    -->
    <!--     this graph can be examined for evidence that any diffusion     -->
    <!--     occurs, as well as for seeing if diffusion of K1 knowledge     -->
    <!--     occurs in agent group B via the bridging agents                -->
    <operation name="ReadGraphByName">
      <parameters>
        <param name="graph_name" value="'knowledge network'"/>
        <param name="output_filename" value="last.csv"/>
        <param name="output_format" value="csv"/>
        <param name="run" value="all"/>
        <param name="time" value="last"/>
      </parameters>
    </operation>

    <!-- 3. read graph: interaction network                                 -->
    <!--   determine which agents interacted with whom over the course of   -->
    <!--   the simulation; the rows are the source agents and the columns   -->
    <!--   the receiver of the communication                                -->
    <!--     note that this operation reads the graph for the entire simu-  -->
    <!--     lation, meaning that multiple networks will be printed in the  -->
    <!--     same CSV file, with one empty line between each agent-by-agent -->
    <!--     block                                                          -->
    <operation name="ReadGraphByName">
      <parameters>
        <param name="graph_name" value="'agent interaction probability network'"/>
        <param name="output_filename" value="prob.csv"/>
        <param name="output_format" value="csv"/>
        <param name="run" value="all"/>
        <param name="time" value="last"/>
      </parameters>
    </operation>

  </operations>

</construct>
```

## References

Carley K. 1986. An Approach for Relating Social Structure to Cognitive Structure. *Journal of Mathematical Sociology* 12:137-89

Carley K. 1991. A Theory of Group Stability. *American Sociology Review* 56:331-54

Carley K, Martin M, Hirshman B. 2009. The Etiology of Social Change. *Topics in Cognitive Science* 1

Carley K, Newell A. 1994. The Nature of the Social Agent. *Journal of Mathematical Sociology* 4:221-62

Carley K, Reminga J. 2004. ORA: Organizational Risk Analyzer, Carnegie Mellon University School of Computer Science, Pittsburgh, PA

Frantz T, Carley K. 2007. The Impact of Knowledge Misrepresentation on Organization Performance Dynamics. In *Annual Conference of the North American Association for Computational Social and Organizational Sciences*. Atlanta, GA

Hirshman B, Carley K. 2007a. Specifying Agents in Construct, Carnegie Mellon University School of Computer Science, Pittsburgh, PA

Hirshman B, Carley K. 2007b. Specifying Networks in Construct, Carnegie Mellon University School of Computer Science, Pittsburgh, PA

Hirshman B, Carley K. 2008. Modeling Information Access in Construct, Carnegie Mellon University School of Computer Science, Pittsburgh, PA

Hirshman B, Carley K, Kowalchuck M, St. Charles J. 2009. Decisions, Variables, and Scripting in Construct. Carnegie Mellon University School of Computer Science: Carnegie Mellon University

Hirshman B, St. Charles J. 2009. Simulating Emergent Multi-Tiered Social Ties. In *Proceedings of the 2009 Human Behavior and Computational Intelligence Modeling Conference*. Oak Ridge National Laboratory, TN

Krackhardt D, Carley K. 1998. A PCANS Model of Structure in Organization. In *Proceedings of the 1998 International Symposium on Command and Control Research and Technology*. Monterey, CA

Metcalfe J, Shimamura A. 1994. *Metacognition: knowing about knowing*. Cambridge, MA: MIT Press

Moon I-C, Carley K. 2007. Estimating the Near-Term Changes of Organization with Simulation. *AAMAS*, pp. 127-35. Honolulu, HI

Simon H. 1957. A Behavioral Model of Rational Choice. In *Models of Man: Mathematical Essays on Rational Human Behavior in a Social Setting*, pp. 241-60 %@ 57-5933. London, England: John Wiley & Sons, ltd

Wegner D. 1986. Transactive memory: A Contemporary Analysis of the Group Mind. *Theories of group behavior*, pp. 185-208. New York: Springer-Verlag