A Bayesian Network Approach to the Self-organization and Learning in Intelligent Agents

Ferat Sahin

Dissertation submitted to the Faculty of Virginia Polytechnic and State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

Hugh F. VanLandingham, Chair
John S. Bay
Pushkin Kachroo
A. Lynn Abbott
Charles J. Parry

August 25, 2000
Blacksburg, Virginia

Keywords: Bayesian networks, learning, intelligent agent, self-organization

A Bayesian Network Approach to the Self-organization and Learning in Intelligent Agents

Ferat Sahin

(ABSTRACT)

A Bayesian network approach to self-organization and learning is introduced for use with intelligent agents. Bayesian networks, with the help of influence diagrams, are employed to create a decision-theoretic intelligent agent. Influence diagrams combine both Bayesian networks and utility theory. In this research, an intelligent agent is modeled by its *belief*, *preference*, and *capabilities* attributes. Each agent is assumed to have its own belief about its environment. The belief aspect of the intelligent agent is accomplished by a Bayesian network. The goal of an intelligent agent is said to be the *preference* of the agent and is represented with a utility function in the decision theoretic intelligent agent. Capabilities are represented with a set of possible actions of the decision-theoretic intelligent agent. Influence diagrams have utility nodes and decision nodes to handle the preference and capabilities of the decision-theoretic intelligent agent, respectively.

Learning is accomplished by Bayesian networks in the decision-theoretic intelligent agent. Bayesian network learning methods are discussed intensively in this paper. Because intelligent agents will explore and learn the environment, the learning algorithm should be implemented online. None of the existent Bayesian network learning algorithms has online learning. Thus, an *online* Bayesian network learning method is proposed to allow the intelligent agent learn during its exploration.

Self-organization of the intelligent agents is accomplished because each agent models other agents by observing their behavior. Agents have belief, not only about environment, but also about other agents. Therefore, an agent takes its decisions according to the model of the environment and the model of the other agents. Even though each agent acts independently, they take the other agents behaviors into account to make a decision. This permits the agents to organize themselves for a common task.

To test the proposed intelligent agent's learning and self-organizing abilities, Windows application software is written to simulate multi-agent systems. The software, IntelliAgent, lets the user design decision-theoretic intelligent agents both manually and automatically. The software can also be used for knowledge discovery by employing Bayesian network learning a database.

Additionally, we have explored a well-known herding problem to obtain sound results for our intelligent agent design. In the problem, a dog tries to herd a sheep to a certain location, i.e. a pen. The sheep tries to avoid the dog by retreating from the dog. The herding problem is simulated using the IntelliAgent software. Simulations provided good results in terms of the dog's learning ability and its ability to organize its actions according to the sheep's (other agent) behavior.

In summary, a decision-theoretic approach is applied to the self-organization and learning problems in intelligent agents. Software was written to simulate the learning and self-organization abilities of the proposed agent design. A user manual for the software and the simulation results are presented.

This research is supported by the Office of Naval Research with the grant number N00014-98-1-0779. Their financial support is greatly appreciated.

Acknowledgment

First of all, I would like to take this opportunity to express my gratitude to my advisor Professor F. Hugh VanLandingham for his invaluable guidance and encouragement throughout this work. I am so grateful that he introduced me to the world of Artificial Intelligence. He has been very helpful and supportive both intellectually and personally. Without his help on typing my dissertation, this work would not be complete in time.

Second, I would like to express my deep appreciation to my co-advisor Dr. John Bay for his invaluable supervision and fortitude throughout the course of my M.Sc. and Ph.D. studies. He is my mentor and inspiration to be an academician. His supervision affected my research tremendously. Every time I visited him in his office, I was filled with hope and encouragement on my research. At the end of every meeting, he replaced my frustration with full of inspiration.

I am also thankful to my Ph.D. committee members Dr. Pushkin Kachroo, Dr. A. Lynn Abbott, and Dr. Charles Parry. Their expertise and assistance played an important role in the progress of my research.

Next, I would like to thank my parents, Aslan and Zehra, my brothers and sisters for their continuous love, understanding and encouragement during my study at Virginia Tech. I would also like to thank my friend Selhan and my brother Murat for their support and help throughout this work.

I would like to thank the Office of Naval Research. Their financial support is greatly appreciated. At last, but not least, to my machine intelligence laboratory (MIL) buddies and members of Multi-agent Bio-robotic Learning (MABL) group, thanks for the many memorable experiences.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

How can independent agents cooperate to solve a problem collectively in a real life environment? How do the agents explore the environment while organizing a common task? This research will attempt to answer these commonly asked questions from the machine learning literature. The heart of the problem is how the agents will learn the environment independently and then how they will cooperate to establish the common task. In the literature, these types of problems are referred to as self-organizing problems.

An agent is an entity that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [54]. The *Effector/Medium/Sensor* (*EMS*) paradigm explains this definition. Gerber stated that this paradigm provides an appropriate abstraction of a (human) agent acting and interacting with its environment and other (human) agents [54]. This idea comes from Malsch's work on *Generalized Media of Interaction* in sociology [55], where action and interaction are transmitted via appropriate media [51]. According to EMS, verbal communications are interpreted in the following manner: Each agent (human) has a speech *effector* (voice, speech apparatus) and an audio receptor (ear). The spoken language, i.e. the sound, is transmitted through the air [51].

In a multi-agent system, agents are independent in that they have independent access to the environment. Therefore, each agent should incorporate a learning algorithm to learn and/or explore the environment. Then, the agents should have some sort of communication between them to behave as a group. In other words, they must organize

themselves to act together. A sheepdog is a widely used example of self-organizing systems in the literature. Multiple dogs cooperate to put all the sheep into a pen together. Dogs have independent beliefs about the sheep and the environment, but they learn to cooperate with other dogs at the same time. Even though the dogs will have independent ideas about how to solve the problem, they also have to know how to solve the problem cooperatively. Since the multi-agent self-organization problem is a learning problem, the following paragraphs will explore how researchers in the artificial intelligence (AI) literature approach a learning system.

1.1 Learning systems in AI

There are two approaches to model a learning system in the AI literature. A learning system is modeled as either supervised or unsupervised. The first approach is called supervised learning in which the learning system has a world model. The learning system makes its decisions according to the world model. Some type of feedback from the environment is required to change the world model. This is also called a goal-driven learning system or a deliberative learning system. Figure 1.1 (a) illustrates a goal driven learning system.

The second approach is described as supervised learning in which the learning system explores the environment and takes actions to change it. This type of learning is also called a data-driven or a reactive learning system because the learning system depends on only data, and it does not have a model of the world. Figure 1.1 (b) illustrates an unsupervised/data-driven learning system model. There has been some research on a method that tries to combine the two learning models. The methods were combined often

in ad-hoc ways and usually with limited success. This work will propose an approach that combines both types of learning models. We will call the proposed learning model the *bi-directional learning model*. These two approaches are used consecutively in some learning systems, but they are not usually used simultaneously. Figure 1.2 illustrates the bi-directional learning model. After specifying what type of learning algorithm is needed for the self-organization problem, we need to explain the idea behind the self-organizing mechanism.

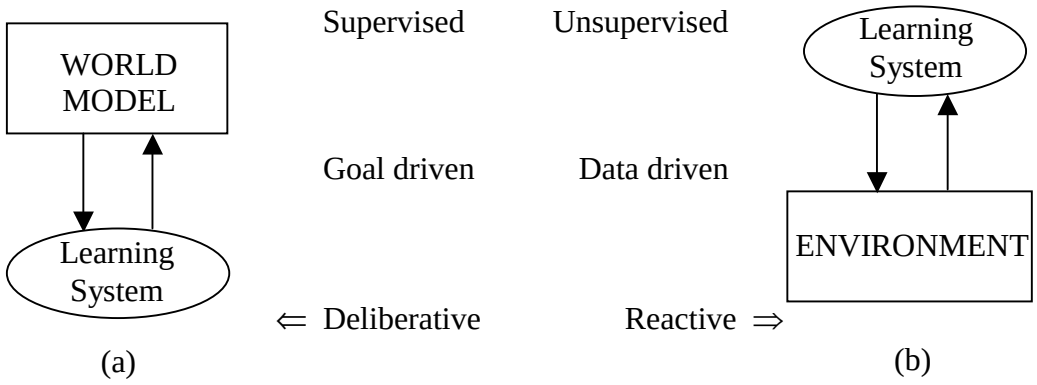


Figure 1.1. (a) Supervised learning model. (b) Unsupervised learning model.

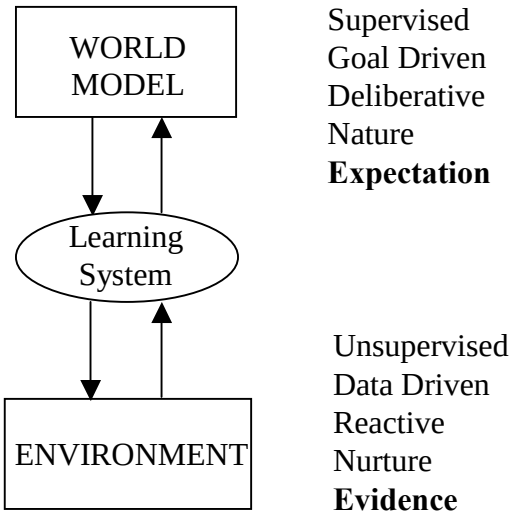


Figure 1.2. Bi-directional Learning System Model.

1.2 Self-organization systems

The main idea of a self-organizing mechanism is to control a society of autonomous agents through structurization and organization [51]. The task of adapting the structure of a group or a society of artificial agents to the environment is considered an optimization problem by characterizing a *search space* and an *objective function* to be optimized. The objective function denotes the current system's performance while a multi-dimensional search space describes the system's set of possible configurations [51]. The search space dimensions can be derived from principles of a multi-agent system application: *structural principles*, *communication principles*, and *agent architecture principles* [51].

Structural principles are, for example, the number of agents in the group, the number of specialists for a certain task, the organizational form of the group, migration (i.e. distribution of agents over the net), and so on. Communication principles can be expressed through the introduction of communication channels between subunits or even between agents belonging to a common subunit. Agent architecture principles are explicit resource distributions among the various agent modules [51]. A unified approach is provided by the paper [43].

Self-organization of multi-agent systems is commonly achieved by using some combination of *rule-based systems*, *Q-learning*, *Temporal Difference* $TD(\lambda)$, and *evolution-based algorithms*. Traditional Genetic algorithms (GAs) are well suited for off-line search, where search time is not important. Unfortunately, the domains where multi-agent systems are in use are generally highly dynamic since the environment may

change anytime. In addition, a traditional GA needs to process many individuals. This might require storing the configuration of tens of complete agent societies, which is intractable. That is why the evolution-based algorithms have to be modified greatly for on-line use. Thus, the performance of a GA is inefficient in multi-agent systems [51].

Temporal difference and Q-learning methods are also commonly employed to solve multi-agent learning and organization problems [50, 52]. Temporal difference methods require learning the value function for a fixed policy. Thus, they must be combined with other reinforcement learning methods that can use the value function to make policy improvements [53].

Temporal difference methods work in the following way. Let $V_{\Pi}(s)$ denote the current estimated value of state s under a fixed policy Π. When a sample $\langle s, a, t, r \rangle$ is received by performing action a in state s at time *t* with the reward *r*, the simplest TD-method (known as $TD(0)$) will update the estimated value to be

$$(1-\alpha)V_{\Pi}(s) + \alpha(r + \beta V_{\Pi}(t)) \tag{1.1}$$

Here α is the learning rate $(0 \le \alpha \le 1)$, governing to what extent the new sample replaces the current estimate. The symbol β is the discount factor. This is the basis of $TD(\lambda)$, where a parameter λ captures the degree to which past states are influenced by the current sample [40].

Q-learning is a straightforward and elegant method for combining value function learning (as in TD-methods) with policy learning. A Q-value, $Q(s, a)$, is assumed for each state-action pair $\langle s, a \rangle$. The Q-value provides an estimate of the value of

performing action a at state s. An agent updates its estimate $Q(s,a)$ based on sample $\langle s,a,t,r \rangle$ using the formula:

$$(1-\alpha)Q(s,a)+\alpha(r+\beta(\max_{a'}\{Q(t,a')\})) \quad (1.2)$$

Temporal difference and Q-learning methods are successful in multi-agent learning under the assumption of full observability. Full observability means that all states of the environment can be observed completely. If the environment is not fully observable or we have incomplete data, these methods easily fail to converge. Since an agent can adopt the best policy given its current knowledge, Q-learning is only guaranteed to converge to the optimal Q-function (and implicitly an optimal policy) if each state in the environment is sampled sufficiently [53].

Learning classifier systems [Holland, 1986] also have been employed to solve multi-agent learning and self-organization problems. The *learning classifier system* (LCS) is a rule-based, message-passing, machine learning paradigm designed to process environmental stimuli, much like the input-to-output mapping provided by a neural network. The LCS provides learning through genetic and evolutionary adaptation to changing task environments. The operation of the LCS is centered around a list of rules or *classifiers*. These rules are essentially a set of "if-then" statements, where the "if" part of a rule is called *condition*, and the "then" part is called an *action.*

Learning classifier systems are genetic-algorithm-based machine learning mechanisms for developing action policies to optimize environmental feedback. Sen and Sekaran insist that learning classifier systems perform very competitively with the Q-learning algorithm, and are able to generate good solutions to both a resource sharing and a robot navigation problem [52]. They also claim that learning classifier systems can be

used effectively to achieve near-optimal solutions more quickly than the Q-learning algorithm does. Even though some [52] claim that learning classifier systems perform better than the Q-learning algorithm, these systems tend to have some deficiencies in decision-making because they are rule-based systems. Partial observability (incomplete data) is hard to handle for learning classifier systems too. Main problem with the LCS is the "bucket-brigade", which cannot converge.

Evolution-based algorithms are not efficient enough because they are not able to perform well on-line. Q-learning algorithms perform well online, but they are not able to handle the partial observability of the environment. Even though some claim that learning classifier systems perform better than Q-learning algorithms, they are not able to perform well with incomplete data. They also have some conceptual and computational difficulties to overcome.

Last, but not least, the methods described above are not completely bi-directional learning models although there is some bi-directionality in them. The importance of bi-directional learning comes from its potential to combine the supervised learning and unsupervised learning and facilitates them at the same time. The present research attempts to provide a new approach that overcomes the difficulties described above paragraphs. The new approach is based on *Bayesian networks*, directed acyclic graphs (DAG) that are constructed by a set of variables coupled with a set of directed edges between variables.

1.3 Why Bayesian Networks?

The main driving force to choose Bayesian networks is that Bayesian networks have a bi-directional message passing architecture. Learning from the evidence can be interpreted as unsupervised learning. Similarly, expectation of an action can be interpreted as supervised learning. Since Bayesian networks pass evidence (data) between nodes and use the expectations from the world model, they can be considered as bi-directional learning systems. In addition to bi-directional message passing, Bayesian networks have several important features such as allowing subjective a priori judgements, direct representation of causal dependence, nonmonotonic reasoning, distillation of sensory experience and the ability to imitate human thinking process.

A Bayesian network is a graphical model that finds probabilistic relationships among variables of the system. There are a number of models available for data analysis, including rule bases, decision trees and artificial neural networks. There are also several techniques for data analysis such as classification, density estimation, regression and clustering. One may wonder what Bayesian networks and Bayesian methods have to offer to solve such problems. The following paragraphs provide four answers to the question.

First, Bayesian networks handle incomplete data sets without difficulty because they discover dependencies among all variables. When one of the inputs is not observed, most models will end up with an inaccurate prediction. That is because they do not calculate the correlation between the input variables. Bayesian networks suggest a natural way to encode these dependencies.

Second, one can learn about causal relationships by using Bayesian networks. There are two important reasons to learn about causal relationships. The process is worthwhile when we would like to understand the problem domain, for instance, during exploratory data analysis or when an agent is exploring the environment. Additionally, in the presence of intervention, one can make predictions with the knowledge of causal relationships.

Third, considering the Bayesian statistical techniques, Bayesian networks facilitate the combination of domain knowledge and data. Prior or domain knowledge is crucially important if one performs a real-world analysis; in particular, when data is inadequate or expensive. The encoding of causal prior knowledge is straightforward because Bayesian networks have causal semantics. Additionally, Bayesian networks encode the strength of causal relationships with probabilities. Therefore, prior knowledge and data can be put together with well-studied techniques from Bayesian statistics.

Finally, in conjunction with Bayesian networks and other kinds of models, Bayesian methods give an efficient approach to avoid the over-fitting of data. Models can be "smoothed" in such a way that all available data can be used for training by using Bayesian approach [3].

Rule based systems are also commonly used for data analysis. After their first successes, it became clear that rule-based systems have their shortcomings. One of the major problems of rule-based systems is that they are not able to treat uncertainty coherently. The reason why rule based systems cannot capture reasoning under uncertainty is that dependence between events changes with knowledge of other events.

Another deficiency of the rule-based system is that the transition between the rules might result in incorrect decisions. For example, assume the system has the following rules:

"If the bottle is broken, then the grass is wet" and "if it rains, the grass is wet"

The rule-based system might make an incorrect conclusion considering these two rules. The system might decide that "if the bottle is broken, then it rained". This statement is not a logical statement, and it is not possible to make this kind of decision with the Bayesian networks. This is called a "dead end" in the machine learning literature [6].

Bayesian networks ease many of the theoretical and computational difficulties of rule-based systems by utilizing graphical structures for representing and managing probabilistic knowledge [1]. Their basic properties and abilities can be combined as described below.

Independencies can be dealt with explicitly. They can be articulated by an expert, encoded graphically, read off the network, and reasoned about, yet they forever remain robust to numerical impression.

Graphical representations uncover opportunities for efficient computation. Distributed updating is feasible in knowledge structures that are rich enough to exhibit intercausal interactions (e.g., “explaining away”). The explaining away property illustrates human-like behavior of the Bayesian Networks. No other expert systems or rule-based systems have this property. Additionally, when extended by clustering or conditioning, tree-propagation algorithms are capable of updating networks of arbitrary topology [1, 47].

The combination of predictive and abductive inference resolves many problems encountered by the expert systems and renders belief networks a viable model for

cognitive functions requiring both top-down and bottom-up inferences [6]. As stated above, Bayesian networks allow bi-directional learning and/or massage passing.

1.3.1 The relationship between Bayesian networks and neural networks

Even though Bayesian networks can model a broad spectrum of cognitive activity, their original strength is in causal reasoning, which performs reasoning about actions, explanations and preferences. Such abilities are not easily established in neural networks, whose strengths lie in quick adaptation of simple motor-visual functions [6]. Pearl states that neural networks cannot do reasoning between events [6]. A Bayesian network gives a model of the environment rather than, as in many other knowledge representation methods (e.g., rule-based systems and neural networks), a model of the reasoning process. In fact, it simulates the mechanisms that operate in the environment, and makes easier diverse models of reasoning, including prediction, abduction and control [6].

The relationship between Bayesian networks and neural networks is rather flimsy except for the usual ability to carry out distributed inferencing. For instance, there are a limited number of neural features in Bayesian networks: weights, sums and sigmoids play no momentous role; familiar linguistic notions are employed for all computational units; and placement of bi-directional messages in acyclic structures has no well-defined biological bias [5]. In these senses, Bayesian networks are not considered to be a kind of neural network in the machine learning literature.

1.4 Self-organizing system as a generalized graph of behaviors

This section explains how a self-organization problem can be considered as a generalized graph of behaviors and how they are related to cognitive learning and human thinking. Then, the reasons why Bayesian networks are employed to solve the self-organization problem of multi-agents will be provided.

A self-organizing system can be presented as a generalized graph of behaviors. Many viable cognitive learning models and brain function are spatially or temporarily localized, so that it is assumed that some ordered topology of behavior exists. A graph of behaviors actually does little to constrain the topology, but it offers a fixed model and analysis paradigm. The interconnections are somewhat better understood as a function of the behavior operations. The interconnections can be categorized in two ways: *quantitative* and *symbolic*. Symbolic messages might consist of command, queries or state information formatted in a textual form. Networks having these kinds of interconnections generally contain relatively high-level behaviors because they are assumed to be individually capable of generating, parsing, and interpreting the messages.

Quantitative interconnections may take various forms, such as in spreading activation networks [49], Bayesian networks [1, 2], neural modular networks (NMN), and mixtures of experts models (ME) [48]. Networks with quantitative interconnection exchange such information in a fixed format, which is not necessarily parsed at the receiving end. It is much more difficult to organize and adapt systems that require symbolic exchange of information than for a system that exchanges quantitative information because symbolic information has an essentially limitless dimension and the problem space for self-organization of such systems is extremely large. Thus, networks with quantitative

information exchange are more tractable, and analytical learning methods exist for many of them.

In recent years, Bayesian networks are commonly used networks with quantitative interconnections [1, 2]. Bayesian networks were developed in the 1970s to model distributed processing in reading comprehension, where both semantic expectations and perceptual evidence must be combined to form a cooperative interpretation. The coordination of bi-directional inferences is discovered in expert systems technology of the early 1980s. Lately, Bayesian networks have become known as a general representation scheme for uncertain knowledge [1, 2, 3]. The recent research is mainly focused on learning with Bayesian network [3, 15, 16, 17, 19, 24, 28, 29]. Learning with Bayesian networks will be discussed in Chapter 3.

Bayesian networks maintain prior and posterior probability estimates of optimal parameter sets describing a behavior [1]. Bayesian networks contain a number of nodes whose parameters specify a transformation on the incoming information assuming that a behavior is continuously parameterized. This is analogous to the view that a behavior is considered a mapping that depends on some numerical parameters.

In our research, Bayesian networks are employed to design independent agents because they support a human-like learning strategy. They have formal probabilistic semantics and yet can serve as a natural mirror of knowledge structures in the human mind [12]. Further information about the relationship between human reasoning and Bayesian networks will be discussed in the next section.

The last question that may arise is how Bayesian networks will be employed to solve our problem of the self-organization of independent agents. This brings up the following

related questions: What kind of methods will be employed in estimating parameters of the networks? How will the optimal structure of the network be estimated by using complete or incomplete data? How will the network adjust itself to environmental changes, etc.? These questions are answered in Chapter 3, broadly explaining the various methods discovered in the literature.

In our problem, each agent (the dog) will have its own Bayesian network whose nodes are obtained from the sensory data. Bayesian networks will be incorporated with *influence diagrams,* which allow agents to create actions according to the agent's objective and the state of the environment. Influence diagrams will be explained in Chapter 2. The detailed explanation of the structure of an agent will be provided in Chapter 5.

The agents have either no prior data, or limited data given by some sort of expert, when the agents start to explore the environment. Since data are not reliable at the beginning, the estimated Bayesian network is not going to model the real world properly. Therefore, the Bayesian network has to be updated while the agent explores the environment. In other words, the Bayesian networks should change its world modal by updating itself with the new data. An online Bayesian network learning is proposed to establish continuous learning in Bayesian networks. Chapter 4 explores the proposed online Bayesian network learning.

Online Bayesian network learning is one of the main contributions of this research. Online learning helps agents perform self-organizational behaviors. Since the agents learn during their exploration of the environment and observation of other agents, each agent takes its action according to the current state of the environment and its belief about

the other agents and environment. In Chapter 7, the simulations are performed without applying online Bayesian network learning in the agent design. Simulation results show that the agents cannot make cooperative actions since they do not learn/adapt their knowledge about the environment and the other agents by learning. Details of the simulations and the effectiveness of the online Bayesian network learning are explored in Chapter 7.

1.5 Outline of the dissertation

Chapter 2 will provide detailed descriptions of causal networks, Bayesain networks, and influence diagrams. Learning in Bayesian networks will be explored intensively in Chapter 3. Chapter 4 explores the proposed *online* Bayesian network learning. Chapter 5 will talk about how we will combine Bayesian networks and influence diagrams to create an intelligent agent model, namely the decision-theoretic intelligent agent. Then, the software, the IntelliAgent, is developed for creating and simulating intelligent agent design in Chapter 6. A herding problem is simulated by the IntelliAgent software. The problem definition and the simulation results are presented in Chapter 7. Chapter 8 concludes the research by presenting the main contributions of the research. Finally, Chapter 9 presents possible future work on the research.

CHAPTER 2

Causal Networks, Bayesian Networks and Influence Diagrams

This chapter provides a detailed explanation of causal networks and Bayesian networks along with the necessary probabilistic calculus. Subjects will be explained using an example: wet grass. First, causal networks will be explained along with basic principles of reasoning under uncertainty. Next, we will define the Bayesian networks. Finally, influence diagrams will be explored.

The causal information encoded in Bayesian networks facilitates the analysis of action sequences, their consequences, their interaction with observations, and their expected utilities, and hence the synthesis of plans and strategies under uncertainty [44, 46]. That is, Bayesian networks handle reasoning under uncertainty very well.

The isomorphism between the topology of Bayesian networks and the stable mechanisms that operate in the environment facilitates modular reconfiguration of the network in response to changing conditions, and permits deliberative reasoning about novel situations [6].

Since the reasoning under uncertainty is one of the advantages of causal and Bayesian networks it is necessary to provide some details on the principles of reasoning under uncertainty. The next section provides basic principles for reasoning under uncertainty.

2.1 Basic principles for reasoning under uncertainty

The basic problem when reasoning under uncertainty is whether information on some event influences our belief in other events. Rule-based systems cannot capture reasoning

under uncertainty because the dependence between events changes with the knowledge of other events. The problem will be explored with the following example.

2.1.1 Wet Grass

The rest of the chapter will be explained with the wet grass example to show the reasoning process. Mr. Holmes leaves his house in the morning and notices that his grass is wet. He reasons it had been raining last night. Then he thinks that his neighbor, Mr. Watson's grass is most probably wet also. That is, the information that Mr. Holmes' grass is wet has an influence on his belief of the status of Mr. Watson's grass. Now, suppose that Mr. Holmes checks his rain meter, and it is dry. Then he will not reason as above, and information on Mr. Holmes' grass has no influence on his belief about Mr. Watson's grass.

Next, let us consider two possible causes for wet grass. Besides rain, Mr. Holmes may have forgotten to turn his sprinkler off. The next morning, suppose that Mr. Holmes again notices that his grass is wet. Mr. Holmes' belief of both rain and sprinkler increases. Then he observes that Mr. Watson's grass is wet, and he concludes that it had rained last night. The last step is virtually impossible through rules, but natural for human beings, called *explaining away*.

2.1.2 Explaining away

Explaining away is the process of decreasing one's belief in a causal event as a result in an increase in the belief of an alternative causal event. Let us explain this with our example. After seeing Mr. Watson's grass is wet in the next morning, Mr. Holmes concluded that it had rained. Consequently, Mr. Holmes' wet grass has been explained

by the rain, and thus there is no longer any reason to believe that the sprinkler has been on. Explaining away is another example of dependence changing with the information available [7]. The following section provides some details of dependence between the events.

2.1.3 Dependence of events

Dependence between two events is when the probability of an event depends on the knowledge of the other event. For example, when nothing is known in the initial state, the variables Rain and Sprinkler are *independent*. On the other hand, when the information on Mr. Holmes' grass is present, then Rain and Sprinklers become *dependent*. That is, change in the belief in whether it rained or not will change the belief in the sprinkler being on or off. If it rained, then the sprinklers were not on. Otherwise, the sprinklers were on. Of course, this is true only if there is no other variable that causes Mr. Holmes' grass being wet. On the other hand, if the information on Mr. Holmes' grass is not present, then we cannot relate the variables Rain and Sprinkler. Dependence between events will be clearer when we introduce the concept of causal networks.

The prior certainties are also an important concept in reasoning under uncertainty. The next paragraph will introduce the importance of the prior certainties for reasoning.

2.1.4 Prior Certainties

In the above example, it is obvious that if an event is known, the certainty on the other events must be changed. In a certainty calculus, if the actual certainty of a specific event has to be calculated, then the knowledge of certainties prior to any information is also

required. For instance, the certainty of Rain is still dependent on whether rain at night is rare (as in Los Angeles) or very common (as in London) given that Mr. Holmes' grass is wet [2].

Since basic principles of reasoning under certainty are provided above, now causal networks can be introduced. The following section introduces causal networks and provides related definitions such as connection types and *d-separation*.

2.2 Causal Networks

The reasoning above can be described by a graph. The events are nodes, and two nodes *A* and *B* are connected by a directed link from *A* to *B* if *A* has a causal impact on *B*. Figure 2.1 is graphical model for Mr. Holmes' small world of wet grass.

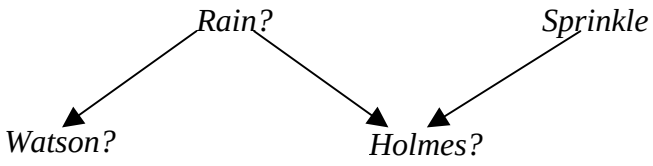


Figure 2.1. A graphical model for the wet grass example [7].

Figure 2.1 is an example of a causal network. A *causal network* is composed a set of *variables* and a set of *directed links* between variables. In mathematics literature, this composition is called a *directed graph*. In a directed graph, the terminology of family relations is adopted to explain the relations between the variables. If there exists a link from variable *A* to variable *B*, then *A* is called a *parent* of *B* and *B* is called a *child* of *A*. The variables symbolize events. Every variable in a causal network has two (yes and no) or more states (i.e. color of a car: blue, green, red, and black). In general, variables can

have continuous and discrete states. Reasoning about uncertainty also has a quantitative part such as the calculation and combination of *certainty numbers* [2]. The certainty numbers are the probabilities of the event (variables) given the data.

From the graph in Figure 2.1, one can read off the dependencies and independencies in the small world of wet grass. For example, one can see that if he knows that it has not rained tonight, then information on Mr. Watson's grass has no influence on Mr. Holmes' grass. The ways in which influence may run between variables in a causal network have been analyzed by Pearl [33] and Verna [20]. Two variables are said to be *separated* if new evidence on one of them has no impact on our belief of the other. If the state of a variable is known, then we say it is *instantiated*.

There are three types of connections in a causal network: *serial*, *diverging*, and *converging* connections. Figure 2.2 shows all type of connections in a causal network.

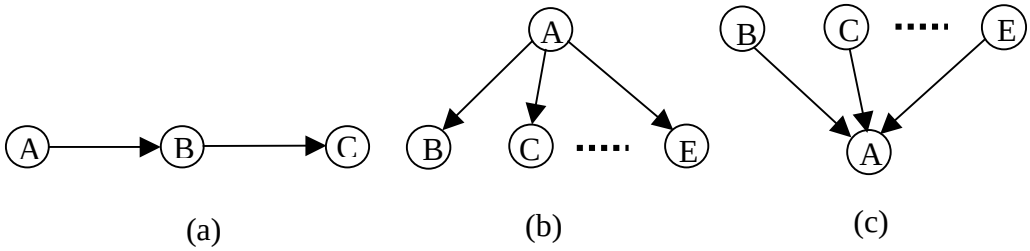


Figure 2.2. Serial, diverging, and converging connections respectively.

In Figure 2.2 (a), the variable *A* has a control on the variable *B* that then has control on the variable *C*. Apparently, the evidence on the variable *A* will affect the certainty of the variable *B* that in turn affects the certainty of the variable *C*. Analogously, the evidence on the variable *C* will affect the certainty of the variable *A* through the variable *B*. On the contrary, if the state of the variable *B* is given, then the link is blocked, and the

variable *A* and the variable *C* become independent. In other words, influence may run from *A* to *C* and vice versa unless *B* is instantiated.

As shown in Figure 2.2 (b), in a diverging connection the influence can pass between all the children of the variable *A* unless the state of the variable *A* is given. If the state of the variable *A* is known, then the variables *B, C,…, E* become independent from each other. Therefore, influence may run between *A*'s children unless *A* is instantiated.

In a converging connection shown in Figure 2.2 (c), if there is nothing known about the variable *A* other than what may be deduced from the knowledge of its parents *B, C,…, E,* then the parents are said to be independent. The independence means that evidence on one of the parents has no effect on the certainty of the others. If there is any other kind of evidence influencing the variable *A,* then the parents become dependent because of the principle of explaining away. Therefore, evidence may only be transmitted through a converging connection if either the variable in the connection or one of its descendants has received evidence. The evidence can be direct evidence on the variable *A,* or it can be evidence from one of its children. In causal networks, this fact is called *conditional dependence*.

Jensen stated that evidence on a variable is a statement of the probabilities of its states. If the statement supports the exact state of the variable it is called *hard* evidence. Otherwise, it is voiced *soft* evidence. For example, soft evidence can be evidence stating the probabilities of the states of a variable. Hard evidence is also referred as *instantiation*. In the case of serial and diverging connections, blocking a link requires hard evidence, while opening a link is possible for all types of evidence [2].

The three cases explained above wrap all the forms in which evidence may be transmitted through a variable. If the rules below are followed, it is conceivable to decide for any pair of variables in a causal network whether or not they are dependent knowing the evidence entered into the network. Two variables *A* and *B* are said to be *d-separated* if for all paths between variables *A* and *B* there is an intermediate variable *V* so that either

- the connection is serial or diverging and the state of *V* is known

or

- the connection is converging and neither *V* nor any of *V*'s descendants have received evidence [2].

If variables *A* and *B* are not d-separated they are said to be *d-connected*. For example, if the state of the variable *B* is given in Figure 2.2 (a), then the link is blocked, and the variable *A* and the variable *C* become independent. Therefore, it is said that the variable *A* and the variable *C* are d-separated given the variable *B*. Similarly, in Figure 2.1, *Sprinkler?* and *Watson?* are d-separated because the connecting trail is converging around the variable *Holmes?*

One should note that d-separation is a property of human reasoning [7], and therefore any calculus for uncertainty in causal structures must obey the principle that whenever *A* and *B* are d-separated then new information on one of them does not change the certainty of the other. To understand causal networks better, we need to establish the quantitative part of the certainty assessment. The next section will provide necessary probability calculus for certainty assessment.

2.3 Probability calculus

Even though various certainty calculi exist in the literature, this section provides the Bayesian calculus, which is *classical probability calculus*. The section starts with basic probability calculus. Then, the concept of subjective probability and conditional probability will be introduced.

2.3.1 Basic probability calculus

The basic concept in the Bayesian treatment of certainties in causal networks is *conditional probability*. When the probability of an event *A*, *P*(*A*), is known, then it is given conditioned by other known factors. A conditional probability statement has the following form:

Given the event B, the probability of the event A is x.

The mathematical representation of this statement is $P(A \mid B) = x$. This does not mean whenever *B* is true, then the probability for *A* is *x*. It means that if *B* is true, and everything else known is inapplicable to *A*, then $P(A \mid B) = x$.

The *fundamental rule* for probability calculus is given in the following way in [2];

$$P(A \mid B)P(B) = P(A, B) \tag{2.1}$$

where $P(A, B)$ is the probability of the joint event $A \wedge B$. Because probabilities ought to always be conditioned by a context *C*, the formula should be written as;

$$P(A \mid B, C)P(B \mid C) = P(A, B \mid C) \tag{2.2}$$

From (2.1), we can write that $P(A \mid B)P(B) = P(B \mid A)P(A)$ and this gives the famous *Bayes' rule*:

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}. \tag{2.3}$$

If we condition the Bayes' rule, we will get the following;

$$P(B \mid A, C) = \frac{P(A \mid B, C)P(B \mid C)}{P(A \mid C)}. \tag{2.4}$$

If A is a variable in a causal network with the set of states a_1, a_2,…, a_n, then the $P(A)$ is a probability distribution over this set of states:

$$P(A) = (x_1, x_2, \ldots, x_n) \qquad x_i \geq 0 \qquad \sum_{i=1}^{n} x_i = 1$$

where x_i is the probability of A being in the state a_i. One should note that the probability of A being in the state a_i is expressed as $P(A = a_i)$ and expressed as $P(a_i)$ if the variable is obvious from the context. Let B be another variable with the states b_1,b_2,…,b_m, then $P(A \mid B)$ is an n-by-m table consisting numbers $P(a_i \mid b_j)$. This table is called conditional probability table (CPT) for $P(A \mid B)$.

The joint probability for the variables A and B, $P(A, B)$, is also an n-by-m table containing the probabilities $P(a_i, b_j)$. The joint probabilities, $P(A, B)$, can be computed by utilizing the fundamental rule (2.1):

$$P(a_i, b_j) = P(a_i \mid b_j)P(b_j)$$

or equivalently,

$$P(A, B) = P(A \mid B)P(B) \tag{2.5}$$

The joint probability, $P(A, B)$, has $n \cdot m$ entries. The probability $P(A)$, can be computed from the table $P(A, B)$. Let a_i denote a state of the variable A. In the table $P(A, B)$, there are m different events for which the variable A is in state a_i, namely the mutually exclusive events (a_i, b_1), …, (a_i, b_m). Therefore, $P(a_i)$ can be calculated as;

$$P(a_i) = \sum_{j=1}^{m} P(a_i, b_j) \tag{2.6}$$

This operation is called *marginalization* and it is said that the variable *B* is marginalized out of $P(A, B)$ (producing *P*(*A*)). Thus, the notation can be written as follows:

$$P(A) = \sum_{B} P(A, B) \tag{2.7}$$

The definitions above work for only classical (objective) probabilities. Causal networks have another type of probability, called subjective probability. The subjective probability is one of the important features of causal networks because of their ability to explain one's belief on an event.

2.3.2 Subjective probabilities

Probability calculus does not require that the probabilities be based on theoretical results or frequencies of repeated experiments. Probabilities may also be completely subjective estimates of the certainty of an event. For example, a subjective probability may be my personal assessment of the chances of finishing my dissertation at the end of next Fall semester. Jensen provides a way of assessing this probability by comparing to gambling [2].

Subjective probability is also called as Bayesian probability or personal probability in the literature [3]. The Bayesian probability of an event *x* is a person's *degree of belief* in that event. A Bayesian probability is a property of the person who assigns the probability (e.g., your degree of belief that a coin will land heads), whereas a classical probability is a physical property of the world (e.g., the probability that a coin will land heads). In light of these statements, a degree of belief in an event is referred to as a Bayesian or personal

probability, and the classical probability is referred as the true or physical probability of that event [3].

An important difference between physical probability and personal probability is that there is no need for repeated trials to measure the personal probability. For example, consider the question: what is the probability that the Chicago Bulls will win the championship in 2001? The Bayesian method can assign a probability for this event. One common criticism of the Bayesian approach of probability is that probabilities seem arbitrary. This can be mainly observed as a probability assessment problem. Much research has been done to overcome this problem. A detailed construction of this criticism can be found in [3].

Another important concept in causal networks is the conditional independence between variables. The following subsection describes its importance in Bayesian calculus.

2.3.3 Conditional Independence

In the Bayesian calculus, the blocking of influence between variables is reflected in the concept of *conditional independence*. The variables A and C are independent given the variable B if

$$P(A \mid B) = P(A \mid B, C) \tag{2.8}$$

This expresses that if the state of the variable B is given then no information of the variable C will change the probability of the variable A. Conditional independence comes into view in the cases of serial and diverging connections. If (2.8) holds, then by the conditioned Bayes' rule (2.4) the following will be obtained

$$P(C \mid B, A) = \frac{P(A \mid C, B) \cdot P(C \mid B)}{P(A \mid B)} = \frac{P(A \mid B) \cdot P(C \mid B)}{P(A \mid B)} = P(C \mid B) \qquad (2.9)$$

So, Equations (2.8) and (2.9) hold simultaneously.

With this explanation of causal networks and Bayesian calculus, we can now explore Bayesian networks. The next section will describe the Bayesian network structure and provide its properties in detail.

2.4 Bayesian networks

As stated earlier, causal networks are introduced to define and understand Bayesian networks. The following paragraphs provide a detailed definition of Bayesian networks and related theorems. The chain rule theorem is introduced to do the necessary calculations in Bayesian networks.

Causal relations also have a quantitative side, namely their *strength*. This is expressed by attaching numbers to the links. Let the variable A be a parent of the variable B in a causal network. Using probability calculus, it will be normal to let the conditional probability, $P(B \mid A)$, be the strength of the link between these variables. On the other hand, if the variable C is also a parent of the variable B, then conditional probabilities $P(B \mid A)$ and $P(B \mid C)$ do not provide any information on how impacts from the variable A and the variable B interact. They may cooperate or counteract in various ways. Therefore, the specification of $P(B \mid A, C)$ is required.

It may happen that the domain to be modeled contains feedback cycles. Feedback cycles are difficult to model quantitatively. For causal networks no calculus coping with

feedback cycles has been developed. Therefore, it is necessary for the network not to contain cycles. Thus, A Bayesian network consists of the following elements:

- A set of *variables* and a set of *directed edges* between variables,
- Each set contains a finite set of mutually exclusive states,
- The variables coupled with the directed edges construct a *directed acyclic graph* (DAG),
- Each variable A with parents $B_1, B_2, \ldots, B_n$ has a conditional probability table $P(A \mid B_1, B_2, \ldots, B_n)$ associated with it [2].

If the variable A does not have any parent, then the table can be replaced by the unconditional probabilities $P(A)$. A graph is *acyclic* if there is no directed path $A_1 \to \cdots \to A_n$ such that $A_1 = A_n$. For the directed acyclic graph in Figure 2.3, the prior probabilities $P(A)$ and $P(B)$ have to be specified.

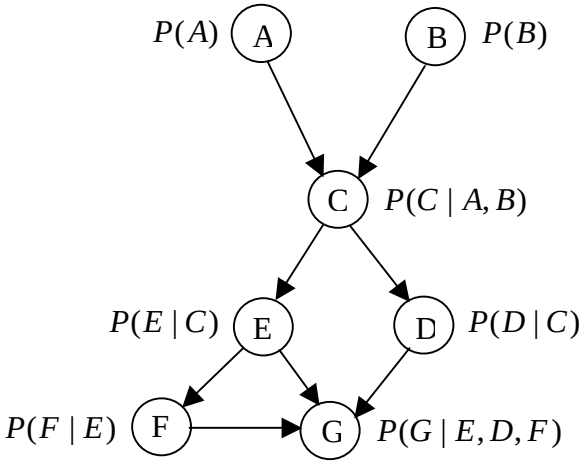


Figure 2.3. A directed acyclic graph. The probabilities to specify are shown.

It has been insisted that prior probabilities introduce an undesired bias to the model [1]. The necessary calculi have been developed in order to avoid this problem [1]. On the other hand, as explained before, prior probabilities are essential. They are important

not because of mathematical reasons but because prior certainty assessments are an integral part of human reasoning about certainty.

One of the benefits of Bayesian networks is that they admit d-separation. If the variables *A* and *B* are d-separated in a Bayesian network with evidence *e* inserted, then $P(A \mid B, e) = P(A \mid e)$. Therefore, d-separation can be used to *read-off* conditional independencies. Next, we will talk about one of the most crucial elements of Bayesian network calculations, namely *the chain rule*.

2.4.1 The chain rule

In a Bayesian Network, let $U = (A_1, A_2, \ldots, A_n)$ be a universe of variables. The chain rule provides a more compact representation of the joint probability $P(U) = P(A_1, A_2, \ldots, A_n)$ to make the probability calculations easier. If the joint probability table $P(U)$ is obtained, then the probabilities $P(A_i)$ can be calculated as well as the probabilities $P(A_i \mid e)$, where *e* is evidence. On the other hand, if the number of variables in the network increases, $P(U)$ expands exponentially. Therefore, a more compact representation of $P(U)$ is necessary: a manner of reserving information from which $P(U)$ can be computed if it is necessary [2].

Such a representation resides in a Bayesian network over *U*. $P(U)$ can be computed from the conditional probabilities defined in a Bayesian network if the conditional independencies hold for *U*. The following theorem explains this representation.

Theorem 2.1 (The Chain rule.)

Let BN be a Bayesian network over

$$U = (A_1, A_2, \ldots, A_n)$$

Then the joint probability distribution P(U) is the product of all conditional probabilities specified in BN:

$$P(U) = \prod_i P(A_i \mid pa(A_i)) \tag{2.10}$$

where pa(A_i) is the parent set of A_i.

Jensen proved this theorem by applying induction on the number of variables in the universe *U* [2]. The next section will provide theoretical and historical details on evidential reasoning using the chain rule.

2.4.2 Evidential Reasoning

As stated above, Bayesian networks accomplish such economy by pointing out, for each variable X_i, the conditional probabilities $P(X_i \mid pa_i)$ where pa_i are the set of parents (of X_i) which render X_i independent of all its other parents. After giving this specification, the joint probability distribution can be calculated by the product

$$P(x_1, \ldots, x_n) = \prod_i \mathrm{P}(x_i \mid pa_i)\,. \tag{2.11}$$

Using this product, all probabilistic queries can be found coherently using probability calculus. There are a number of algorithms for probabilistic calculations in Bayesian networks. Early algorithms employed message-passing architecture and they were limited to trees [18, 14]. In these algorithms, each variable was assigned a simple processor and allowed to pass messages asynchronously with its neighbors until equilibrium is accomplished. Some techniques have been developed to extend this tree propagation to general networks starting around the 1990s. Two of the most popular methods are Lauritzen and Spielgelhalter's method of join-tree propagation [22] and the method of loop-cut conditioning, which is explained in [1, 2]. Learning methods have

also been proposed for systematic updating of the conditional probabilities $P(X_i \mid pa_i)$, as well as the structure of the network in order to match empirical data [21]. The details of learning techniques are discussed in Chapter 3. We will explore some questions about the relationship between Bayesian Networks and the functionality of a human brain as our last topic in Bayesian networks.

2.4.3 Bayesian networks and the functionality of a human brain

Does an architecture like the Bayesian network exist anywhere in the human brain? If not, how does the human brain achieve those cognitive functions in which Bayesian networks excel? Pearl answers these questions in the following sentences: "Nothing resembling Bayesian networks actually resides permanently in the brain. Instead, fragmented structures of causal organizations are constantly being assembled on the fly, as needed, from a stock of functional building blocks" [6].

Every building block is concentrated on to accomplish a narrow context of experience and is presumably materialized in a structure of a neural network. For example, a network as in Figure 2.1 can be assembled from several neural networks each specializing in one variable. Such specialized networks will need to be stored in a permanent mental library, from which they are selected and assembled into a network structure. This is possible only when a specific problem displays itself, for instance, to resolve whether a working sprinkler could rationalize why Mr. Holmes' grass was wet in the middle of a dry season. Therefore, Bayesian networks are particularly beneficial in studying higher cognitive functions, where the organizing and supervising large assemblies of specialized neural networks is an important problem. As stated earlier,

Bayesian networks do human-like reasoning well not because the structure of the networks resembles the biological structure of a human brain but because the way Bayesian networks do reasoning resembles with the way humans do reasoning. The resemblance is more psychological than biological.

We have explained Bayesian networks and causal networks. Bayesian networks will be employed in our intelligent agent design because of their ability of reasoning the events and modeling the environment accurately. Modeling the environment is not enough for an intelligent agent to act rationally in the environment. The beliefs about the environment have to be converted into actions. The next section will introduce a method to convert beliefs of an agent into actions. In the literature, they are also called *influence diagrams* [2], or sometimes *decision networks* [54].

2.5 Influence Diagrams

A Bayesian network serves as a model for a part of the world, and the relations in the model reflect causal impacts between events. The reason for building these computer models is to use them when making decisions. That is, probabilities provided by the network are used to support some kind of decision-making [1]. In principle, there are two types of decisions, *test-decisions* and *action-decisions*. A test-decision is a decision to look for more evidence to be entered into the model. An action-decision is a decision to change the state of the world [1]. In this research, the action-decisions will be the focus.

Decision problems can be treated in the framework of *utility theory*. The utility of an action may depend on the state of some variables called *determining variables*. For example, the utility of a treatment with penicillin is dependent on the type of the infection and whether the patient is allergic to penicillin. The type of the infection and the patient's reaction to the penicillin are the determining variables of the utility of the treatment [2]. The utility theory and Bayesian network theory can be combined in a graphical representation, *influence diagrams*. An influence diagram (ID) is a compact representation emphasizing features of decision problems. The inference diagram formalism integrates the two components of knowledge, about beliefs and about actions.

Influence diagrams are directed acyclic graphs with tree types of nodes–*decision nodes, chance nodes,* and a *value node*. Decision nodes, shown as squares, represent choices available to the decision-maker. Chance nodes, shown as circles, represent random variables (or uncertain quantities) the same as for Bayesian networks. Finally, the value node, shown as a diamond, represents the objective (or utility) to be maximized.

The edges in an ID have different meanings, based on their destinations. An edge pointing to utility and chance nodes represent probabilistic or functional dependence, like the edges in Bayesian networks. They do not necessarily imply causality or time precedence although in practice they often do. Edges into decision nodes mean time precedence and are *informational*, i.e., they show which variables will be known to the decision-maker before the decision is made [2].

An influence diagram can be seen as a special type of Bayesian network, where the value of each decision variable is not determined probabilistically by its predecessors, but rather is imposed from the outside to meet some optimization objective. The domain of

each decision variable in an influence diagram varies according to previous decisions although the domains of the variables in a Bayesian network are fixed.

Figure 2.4 represents an influence diagram about weather and decision to carry an umbrella. *FORECAST* and *WEATHER* are chance nodes, just as in Bayesian networks.

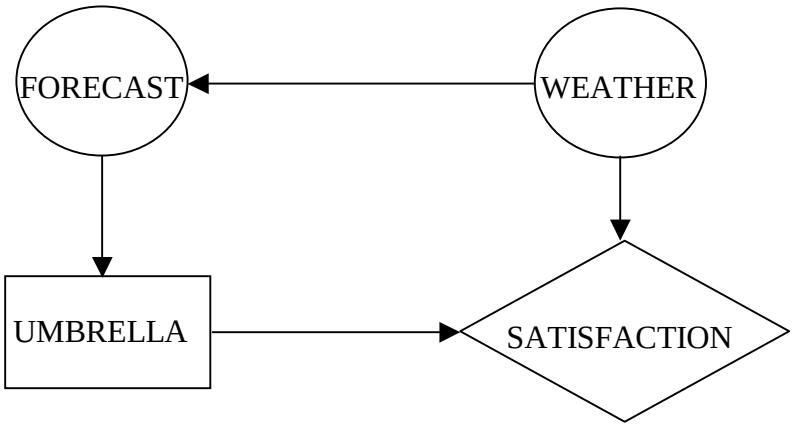


Figure 2.4. An influence diagram.

They have the probabilistic values about the weather and the forecast. *SATISFACTION* is a utility or value node, i.e. a node that measures our scoring of the system. *UMBRELLA* is a decision node, i.e. a node that we have to provide a value for. The objective is to maximize expected *SATISFACTION* by appropriately selecting values of *UMBRELLA* for each possible *FORECAST*. In addition to probabilities, the values of *SATISFACTION* for each combination of *UMBRELLA* and *WEATHER* are also given. The objective in an influence diagram is to select values at the decision nodes in order to maximize the values at the utility nodes.

Now, let us define how the optimal actions are calculated by employing the influence diagram theory. Let $A = \{a_1, \ldots, a_n\}$ be set of mutually exclusive actions, and let H be the determining variable. A *utility table* $U(A, H)$ is necessary to yield the utility for each

configuration of action and determining variable in order to decide between the actions in *A*. The problem is solved by calculating the action that maximizes the expected utility:

$$EU(a) = \sum_{H} U(a, H) \cdot P(H \mid a) \tag{2.12}$$

where $U(a, H)$ are the members of the utility table in the value node *U*. The conditional probability $P(H \mid a)$ is an entry in the CPT of the variable *H*, given the action *a* is fired.

Figure 2.5 illustrates a simple influence diagram with one determining variable and one set of actions. An action set is the set of actions in a decision node in an influence diagram. The probability $P(H \mid a)$ is the probability of *H* given that the action *a* is fired. The probability $P(H \mid a)$ can be calculated by facilitating a standard probabilistic inference as in Bayesian network.

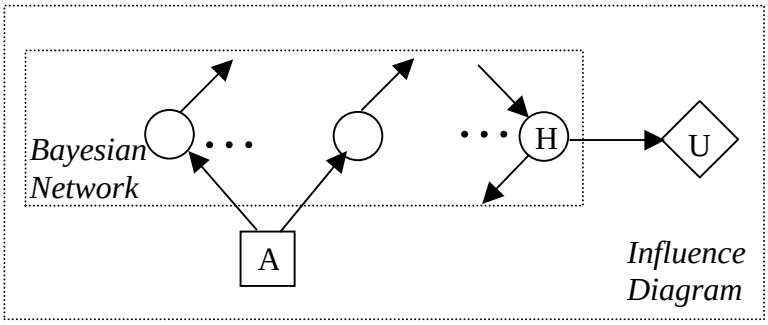


Figure 2.5. An influence diagram with an action set.

Actions are selected by evaluating the decision network for each possible setting of the decision node. Once the decision node is set, it behaves exactly like a chance node that has been set as an evidence variable. The following algorithm illustrates the evaluation of an influence diagram [54].

1. Set the evidence variables for the current state
2. For each possible value of the decision node:

(a) Set the decision node to that value.

(b) Calculate the posterior probabilities for the parent nodes of the utility node, using a standard probabilistic inference algorithm.

(c) Calculate resulting utility function for the action

3. Return the action with the highest utility.

This is a straightforward extension of the Bayesian Network algorithm and will be incorporated into the agent design in the next chapter. An agent that selects rational actions will be designed using the influence diagram theory.

We have explained the causal networks, Bayesian networks, and influence diagrams in detail in this chapter. We have also given theoretical background in Bayesian calculus and reasoning under uncertainty. Since our problem is a learning problem, we need to explain how Bayesian networks learns. The next chapter is devoted for that purpose. Chapter 3 will provide different type of learning situations and different approaches to solve those learning problems.

CHAPTER 3

Learning Bayesian Networks

This chapter is devoted to answering the question: how can Bayesian networks be learned from data? The process of learning Bayesian networks takes different forms in terms of whether the structure of the network is known and whether the variables are all observable. The structure of the network can be *known* or *unknown*, and the variables can be *observable* or *hidden* in all or some of the data points. The latter distinction can also be expressed as *complete* and *incomplete* data. Consequently, there are four cases of learning Bayesian networks from data; known structure and observable variables, unknown structure and observable variables, known structure and unobservable variables, and unknown structure and unobservable variables.

Learning Bayesian networks can also be examined as the combination of *parameter learning* and *structure learning*. Parameter learning is estimation of the conditional probabilities (dependencies) in the network. Structural learning is the estimation of the topology (links) of the network. The four types of learning Bayesian networks cases are discussed in the following paragraphs.

3.1 Known network structure and observable variables (complete data)

This is the easiest and the most studied case of learning Bayesian networks in the literature [31, 32]. The network structure is specified, and the inducer only needs to estimate the parameters. The problem is well understood and the algorithms are computationally efficient. Despite its simplicity, this problem is still extremely useful,

because numbers are very hard to elicit from people. Additionally, it forms the basis for everything else in Bayesian learning.

Because every variable is observable, each data case can be pigeonholed into the CPT entries corresponding to the values of the parent variables at each node. The pigeonhole principle essentially states that if a set consisting of more than $k \cdot n$ objects is partitioned into *n* classes, then some classes receive more than *k* objects [30]. Therefore, estimations will be highly accurate since every variable is observable.

Learning is achieved simply by calculating conditional probability table (CPT) entries using estimation techniques such as Maximum Likelihood Estimation (MLE) and Bayesian Estimation. For simplicity, MLE and Bayesian estimators will be explained by employing parameter learning for a single parameter.

Assume that an experiment was conducted by flipping a thumbtack in the air. The thumbtack comes to land as either heads or tails. As usual, the different tosses are assumed to be independent, and the probability of the thumbtack landing heads is some real number θ. Therefore, the goal is to estimate θ. Assume that we have a set of instances **d**[1],…, **d**[*M*] such that each instance is sampled from the same distribution and independently from the rest. The goal is to find a good value for the parameter θ. A parameter is good if it predicts the data well. In other words, if data are very likely given the parameter, the parameter is a good predictor. The *likelihood function* is defined as

$$L(D \mid \theta) = P(D \mid \theta) = \prod_{m=1}^{M} P(\mathrm{d}[m] \mid \theta). \tag{3.1}$$

Thus, the likelihood for a sequence *H, T, T, H, H* is

$$L(D \mid \theta) = \theta(1-\theta)(1-\theta)\theta\theta \tag{3.2}$$

or $\theta^3(1-\theta)^2$. To calculate the likelihood we need to know number of heads N_h and the number of tails N_t. These are the *sufficient statistics* for this learning problem. A sufficient statistic is a function of the data that summarize the relevant information for computing the likelihood.

The *Maximum Likelihood Estimation* (MLE) principle tells us to choose θ that maximizes the likelihood function. The MLE is one of the most commonly used estimators in statistics. For the above problem, the estimation of the parameter is

$$\hat{\theta} = \frac{N_h}{N_h + N_t} \tag{3.3}$$

as expected.

The MLE estimate seems plausible, but is overly simplistic in many cases. Assume that the experiment with the thumbtack is done and 3 heads out of 10 are recorded. It may be quite reasonable to conclude that the parameter θ is 0.3. On the other hand, what if the same experiment is done with a dime and also 3 heads are recorded. We would be much less likely to jump the conclusion that the parameter of the dime is 0.3 because we have a lot more experience with tossing dimes. Thus, we have a lot more *prior knowledge* about their behavior.

Using MLE, we cannot make the following distinctions: between a thumbtack and a dime, and between 10 tosses and 1,000,000 tosses of a dime. On the other hand, there is another method recommended by Bayesian statistics. The MLE is a frequentist approach since it relies on the frequency in the data. Another approach is the Bayesian approach that assumes that there is unknown but fixed parameter θ. It estimates the parameter

with some confidence, i.e., it calculates a range such that, if the parameter is out of this range, the probability of the data is very low.

The Bayesian approach deals with uncertainty over anything that is unknown by putting a distribution over it. In other words, the parameter θ is treated as a random variable and a distribution $P(\theta)$ is defined over it. Therefore, we can tell how likely the parameter is to take on one value versus another. In other words, we now have a joint probability space that contains both the tosses and the parameter. This joint probability is easy to find given our prior distribution over θ. Let $X[1],\ldots, X[M]$ be our coin tosses. The conditional probabilities $P(X[m] \mid \theta)$ are according to θ, i.e., $P(X[m] = H \mid \theta) = \theta$. Now, the value of the next toss $X[M + 1]$can be predicted by

$$P(X[M+1] \mid X[1],\ldots, X[M]) = \int P(X[M+1] \mid \theta) P(\theta \mid D) d\theta \qquad (3.4)$$

where

$$P(\theta \mid D) = \frac{P(D \mid \theta) P(\theta)}{P(D)}. \qquad (3.5)$$

The first term in the numerator is the likelihood, the second is the prior over parameters, and the third is a normalizing factor, which is the marginal probability of the data.

If we reconsider the thumbtack problem again with a uniform prior over θ in the interval [0, 1], then $P(\theta \mid D)$ is proportional to the likelihood $P(D \mid \theta) = \theta^{N_h} (1-\theta)^{N_t}$. After plugging this into the integral and doing all the math and normalizing, it can be shown that the following equation holds [13].

$$P(X[M+1] \mid D) = \frac{N_h + 1}{N_h + N_t + 2} \qquad (3.6)$$

Clearly, as the number of samples grows, the Bayesian estimator and the MLE estimator converge to each other. This result depends on the use of uniform prior. In the Bayesian networks literature, the most commonly used class of priors are the *Dirichlet* priors [26, 28, 29] because it turns out that most of the interesting calculations can be done in closed form. The conjugacy of the Dirichlet priors allows us to have the posterior probabilities in the same form as prior probabilities. Therefore, we can do sequential updating within the same representations and the closed form solution can be found both for the update and the prediction problem in many cases.

Recall that a multinomial is parameterized via a set of parameters $\theta_1,\ldots,\theta_k$ such that $\sum_i \theta_i = 1$; θ_i corresponds to the probability of *i*th outcome. A Dirichlet distribution over this set of parameters is defined via a set of hyperparameters $\alpha_1,\ldots,\alpha_k$. Then, the generalization can be written as

$$\mathrm{Dir}(\theta \mid \alpha_1,\ldots,\alpha_k) = \frac{\Gamma(\alpha)}{\prod_i \Gamma(\alpha_i)} \prod_i \theta_i^{\alpha_i - 1} . \tag{3.7}$$

All of the results regarding prediction and computing the posterior extend in the obvious way. That is, if θ is distributed as in (3.7), then

$$P(x_i) = \frac{\alpha_i}{\sum_j \alpha_j}$$

and if there is a data set *D* whose sufficient statistics are $N_1,\ldots,N_k$, then

$$P(\theta \mid D) = \mathbf{Dir}(\theta \mid \alpha_1 + N_1,\ldots,\alpha_k + N_k) . \tag{3.8}$$

To generalize these results for a Bayesian network, we need to define the sufficient statistic as $N(x, \mathbf{u})$ for the event $X = x$ and the parents $\mathbf{U} = \mathbf{u}$. In the MLE case, the estimation of the parameters can be calculated as

$$\hat{\theta}_{x|\mathrm{u}} = \frac{N(x,\mathrm{u})}{N(\mathrm{u})}. \tag{3.9}$$

Similarly, in the Bayesian case, the parameter estimation is calculated as

$$\hat{\theta}_{x|\mathrm{u}} = \mathrm{Dir}(\alpha_1 + N(x_1,\mathrm{u}),\ldots,\alpha_{x_k} + N(x_k,\mathrm{u})). \tag{3.10}$$

If the data were actually generated from the given network structure, then both methods converge asymptotically to the correct parameter setting. If not, then they converge to the distribution with the given structure that is closest to the distribution from which the data were generated. Both estimations can be implemented online by accumulating sufficient statistics.

The process above is the method by which Bayesian network parameters are learned when the network topology is known and all variables are fully observable. The next section provides an overview of some proposed methods in the literature if the structure of the network is not known in advance.

3.2 Unknown network structure and observable variables

In this case, the inducer is given the set of variables in the model, and needs to select the arcs between them and estimate the parameters. This problem is very useful for a variety of applications; in general, when we are given a new domain with no available domain expert, and want to get all of the benefits of a BN model. It is also useful for data-mining style applications, where there are masses of data available and we would like to interpret them. In addition to providing a model that will allow us to predict behavior of cases that we have not seen, the structure also gives the expert some indication of what attributes

are correlated. The algorithms for this problem are combinatorially expensive. They basically reduce to a heuristic search over the space of BN structures.

There has been some attention given to the problem of unknown network structure in the literature. The key aspect of the problem is to reconstruct the topology of the network from fully observable variables. In the literature, this is considered as a discrete optimization problem solved by a greedy search algorithm in the space of structures. Some examples of the greedy search algorithm can be found in [34, 35].

A MAP (Maximum a Posterior) analysis of the most likely network structure has been studied in [34] and [35] when the data are fully observable. The resulting algorithms are capable of recovering fairly large networks from large data sets with a high degree of accuracy [16]. On the other hand, they usually adopt a greedy approach to choosing the set of parents for a given node because the problem of finding the best topology is intractable.

There are two main approaches to structure learning in BNs:

- **Constraint based:** Perform tests of conditional independence on the data, and search for a network that is consistent with the observed dependencies and independencies.
- **Score based:** Define a score that evaluates how well the (in)dependencies in a structure match the data, and search for a structure that maximizes the score.

Constraint-based methods are more intuitive. They follow the definition of a BN more closely. They also separate the notion of the independence from the structure construction. The advantage of score-based methods is that they are less sensitive to errors in individual tests. Compromises can be made between the extent to which variables are dependent in the data and the cost of adding the edge [13].

The score-based methods operate on the same principle: a scoring function is defined for each network structure, representing how well it fits the data. The goal is to find the highest-scoring network structure. The space of Bayesian networks is a combinatorial space, consisting of a superexponential number of structures. Thus, it is not clear how one can find the highest-scoring network even with a scoring function. In general, the problem of finding the highest-scoring network structure is NP-hard [13]. On the other hand, the problem of searching a combinatorial space with the goal of optimizing a function is very well studied in AI literature. Consequently, the answer is to define a search space, and then do heuristic search.

In light of the above statements, a BN structure learning algorithm requires the following components be determined:

i) Scoring function for different candidate network structures.

ii) The definition of the search space: operators that take one structure and modify it to produce another.

iii) A search algorithm that does the optimization search.

Each component will be discussed separately. The three main scoring functions commonly used to learn Bayesian networks are the *log-likelihood* [13], the one based on the principle of *minimal description length* (MDL) [11] which is equivalent to Schwarz' *Bayesian information criterion* (BIC) [10], and Bayesian score [3,13].

The log-likelihood function is simply the log of the likelihood function. That is,

$$l(D \mid \mathbf{B}, \theta_{\mathbf{B}}) = \log L(D \mid \mathbf{B}, \theta_{\mathbf{B}}) \tag{3.11}$$

The log-likelihood is easier to analyze than the likelihood, because the logarithm turns all the products into sums. Therefore,

$$L(D \mid \mathbf{B}, \theta_{\mathbf{B}}) = \prod_{m} P(\mathrm{d}[m] \mid \mathbf{B}, \theta_{\mathbf{B}}) \tag{3.12}$$

and, the following equation can be written:

$$l(D \mid \mathbf{B}, \theta_{\mathbf{B}}) = \sum_{m} \log P(\mathrm{d}[m] \mid \mathbf{B}, \theta_{\mathbf{B}}) \tag{3.13}$$

There are a couple of important things to note about the log-likelihood. The log-likelihood increases linearly with the length of data, *M*. The higher scoring networks are those where the node and the parents are highly correlated. Adding a node to the networks always increases the log-likelihood. As a result, the network structure that maximizes the likelihood is often the fully connected network. This is the deficiency of the log-likelihood score and is not desired. Thus, a score that makes it harder to add edges is necessary. In other words, we would like to penalize structures with too many edges.

One possible formulation of this idea is called the *MDL score*. It is defined as:

$$Score_{MDL}(\mathbf{B} : D) = l(D \mid \mathbf{B}, \hat{\theta}_{\mathbf{B}}) - \frac{\log M}{2} Dim(\mathbf{B}) - DL(\mathbf{B}) \tag{3.14}$$

where $Dim(\mathbf{B})$ is the number of independent parameters in $\mathbf{B}$ and $DL(\mathbf{B})$ is the number of bits (the description length) required to represent the structure of $\mathbf{B}$. The abbreviation MDL stands for *minimum description length*. The MDL score is a compromise between fit to data and model complexity. Adding a variable as a parent causes the log-likelihood term to increase, but so does the penalty term. There will be an edge addition if its increase to the likelihood is worth it.

Another commonly used score is called *Bayesian score*. In this case, the network score is evaluated as the probability of the structure given the data. The Bayesian score has the following form:

$$Score_{BDE}(\mathbf{B} : D) = P(\mathbf{B} \mid D) = \frac{P(D \mid \mathbf{B})P(\mathbf{B})}{P(D)} \tag{3.15}$$

As usual $P(D)$ is constant, so it can be ignored when different structures are compared. Therefore, the model maximizes $P(D \mid S)P(S)$, where S represents a structure. The ability to ascribe a prior over structures gives us a way of preferring some structures to others. Here, the probability $P(D \mid \mathbf{B})$ can be calculated as

$$P(D \mid \mathbf{B}) = \int P(D \mid \theta_{\mathbf{B}}, \mathbf{B}) P(\theta_{\mathbf{B}} \mid \mathbf{B}) d\theta_{\mathbf{B}} \,. \tag{3.16}$$

From Equation (3.16), one can see that the more parameters we have the more variables we are integrating over. As a result, each dimension causes the value of the integral go down because the "hill" of the likelihood function is a smaller fraction of the space. Therefore, this idea gives preference to networks with fewer parameters. It can be shown that the Bayesian score is a general form of MDL score. The MDL score can be viewed as an approximation of the Bayesian score. Therefore, the Bayesian score is also a compromise between the model complexity and fit to the data.

Several ways of scoring different Bayesian network structures have been explained. Different scores have been explored in terms of the network complexity and how the network fits to the correlation in the data. Now, the goal is to find the network that has the highest score. In other words, training data *D*, the scoring function, and a set of possible structures are the inputs of the search algorithm while the desired output is a network that maximizes the score. It can be shown that finding maximal scoring network structures where nodes are restricted to having at most *k* parents is NP-hard for any $k > 1$. Therefore, a heuristic search is resorted to for this optimization problem. A search space is defined, where the states in the space are possible structures and the operators denote

the adjacency of structures. This space is traversed looking for high-scoring functions to complete the optimization. The obvious operators in the search spaces are add an edge, delete an edge, and reverse an edge. The search starts with some candidate network, which may be the empty one, or one that some expert has provided as a starting point. Then, applying the operators, the high-scoring network is searched in the space. The parameters of the network are calculated by using training data *D*.

The most commonly used algorithm for optimization search is simple greedy hill climbing, which has the following form:

Greedy BN search

Pick a random network structure $\mathbf{B}$ as starting point
Calculate parameters for each $\mathbf{B}_i$
Compute score for $\mathbf{B}$
Repeat
  Let $\mathbf{B}_1,\ldots, \mathbf{B}_m$ be the successor networks of $\mathbf{B}$ (i.e., operations on $\mathbf{B}$)
  Calculate parameters for each $\mathbf{B}_i$
  Compute score for each $\mathbf{B}_i$
  Let $\mathbf{B}'$ be the highest scoring $\mathbf{B}_i$
  If score ($\mathbf{B}'$) > score ($\mathbf{B}_i$)
    Then let $\mathbf{B}$:= $\mathbf{B}'$
    Else return($\mathbf{B}$)

Even though the hill-climbing method is commonly used, it has several key problems such as local maxima where all one-edge changes reduce the score and plateaux where a large set of neighboring networks that have the same score. There are some clever tricks that avoid some of these problems such as TABU-search, random restart, and simulated annealing. In general, greedy hill climbing with random start works quite well in practice.

We examined methods for learning a Bayesian network from fully observable data. The next sections provide the Bayesian network learning with partially observable data.

Sections 3.3 and 3.4 explore the Bayesian network learning with known network structure and unknown network structure, respectively.

3.3 Known structure and unobservable variables (incomplete data)

The learning of Bayesian networks with known structure and unobservable variables has been studied by Golmard and Mallet [36], Lauritzen [37, 38], Olesen et al. [31], and Spiegelhalter and Cowel [39]. The algorithm that these papers describe is the *expectation maximization* (EM) algorithm [23]. The EM algorithm is an iterative method to calculate maximum likelihood estimates (MLEs) and MAP estimates of the network parameters. The EM algorithm alternates an expectation step with a maximization step. In the expectation step, unknown quantities depending on the missing entries are replaced by their expectations in the likelihood. In the maximization step, the likelihood completed in the expectation step is maximized with respect to the unknown parameters, and the resulting estimates are employed to replace unknown quantities in the next expectation step. The algorithm continues until the difference between successive estimates is smaller than a fixed threshold. [38]. Lauritzen states some difficulties with the use of EM algorithm such as slow convergence rate and local maxima. He then suggests that the gradient descent algorithm can be used as a possible alternative [38].

The third possible approach, introduced by Heckerman [3], is to use *Gibbs sampling* (GS). Gibbs Sampling is one of the most popular Markov Chain Monte Carlo methods for Bayesian inference. The GS algorithm generates a value for the missing data from some conditional distributions and provides stochastic estimations of the posterior probabilities [45]. To illustrate Gibbs sampling, let us approximate the probability

density $p(\theta_s \mid D, S^h)$ for the configuration of parameters θ_s of a particular network S^h, given an incomplete data set $D = \{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$ and a Bayesian network for discrete variables with independent Dirichlet priors. To approximate $p(\theta_s \mid D, S^h)$, we first initialize the states of the unobserved variables in each case somehow (e.g., at random). Therefore, we have a complete random sample D_c. Then, we choose some variable X_{il} (variable X_i in case *l*) that is not observed in the original random sample *D*, and reassign its states according to the probability distribution

$$p(x_{il}' \mid D_c \setminus x_{il}, S^h = \frac{p(x_{il}', D_c \setminus x_{il} \mid S^h)}{\sum_{x_{il}''} p(x_{il}'', D_c \setminus x_{il} \mid S^h)} \tag{3.17}$$

where $D_c \setminus x_{il}$ denotes the data set D_c with observations x_{il} removed, and the sum in the denominator runs over all states of variable X_{il}. Then, this reassignment for all unobservable variables in *D* is repeated producing a new complete random sample D_c'. Using this data set, the posterior density $p(\theta_s \mid D_c', S^h)$ is computed. Finally, the three steps are iterated and the average of $p(\theta_s \mid D_c', S^h)$ is used as our approximation [3].

Both the GS and EM algorithms use a basic strategy called the *missing information principle* [41]: fill in the missing observations on the basis of the available information. Unfortunately, these approximate methods are prone to errors when little and/or biased information is available about the pattern of the missing data [26].

In recent years, an exciting solution to this problem was proposed by Sabestiani and Ramoni [27]. The algorithm is called Bound and Collapse (BC), which is a deterministic method to estimate conditional probabilities from incomplete data. The method *bounds*

the set of possible estimates consistent with the available information by computing the minimum and the maximum estimates that would be gathered from all possible completions of the database. These bounds then *collapse* into a unique value via a convex combination of the extreme points with weights depending on the assumed pattern of missing data [28].

The basic intuition behind BC is that an incomplete database is still able to constrain the possible estimates within a set and that, when exogenous information is available on the pattern of missing data, this can be used to select a point estimate within the set of possible ones. Let X_i be a variable in the set $X = [X_1, \ldots, X_n]$ with parent variable $\mathbf{\Pi}_i$. Sebastiani and Ramoni [25] show that the maximum Bayesian estimate of $p(x_{ik} \mid \pi_{ij})$ is

$$p^{\bullet}(x_{ik} \mid \pi_{ij}, D) = \frac{\alpha_{ijk} + n(x_{ik} \mid \pi_{ij}) + n^{\bullet}(x_{ik} \mid \pi_{ij})}{\alpha_{ij} + n(\pi_{ij}) + n^{\bullet}(x_{ik} \mid \pi_{ij})} \tag{3.18}$$

and the minimum Bayesian estimate is

$$p_{\bullet}(x_{ik} \mid \pi_{ij}, D) = \frac{\alpha_{ijk} + n(x_{ik} \mid \pi_{ij})}{\alpha_{ij} + n(\pi_{ij}) + n_{\bullet}(x_{ik} \mid \pi_{ij})} \tag{3.19}$$

where α_{ijk} are the Dirichlet hyperparameters, $n^{\bullet}(x_{ik} \mid \pi_{ij})$ and $n_{\bullet}(x_{ik} \mid \pi_{ij})$ are maximum and minimum achievable virtual frequencies of $(x_{ik} \mid \pi_{ij})$ in the incomplete data, respectively. The frequency $n(x_{ik} \mid \pi_{ij})$ is the number of occurrences of $(x_{ik} \mid \pi_{ij})$ in the data. The maximum and minimum values of the virtual frequency are calculated by filling the missing entries in order to have maximum and minimum number of occurrences of $(x_{ik} \mid \pi_{ij})$ and counting the number of occurrences of the entry $(x_{ik} \mid \pi_{ij})$, respectively. The probability interval defined by $[p_{\bullet}(x_{ik} \mid \pi_{ij}, D), p^{\bullet}(x_{ik} \mid \pi_{ij}, D)]$ contains

all possible estimates consistent with *D*, therefore it is sound and it is the tightest estimable interval.

The main feature of the BC method is its independence of the distribution of missing data because it does not attempt to infer them: with no information on the missing data mechanism, an incomplete database can only provide bounds on the possible estimates that could be learned [9]. A complete database is just a special case, within available data are enough to constrain the set of possible estimates to a single point. Another advantage of this method is that the width of each interval accounts for the amount of information available in *D* about the parameter to be estimated. Each interval represents a measure of quality of probabilistic information conveyed by the database about a parameter: the wider the interval, the greater the uncertainty due to the incompleteness of the database. In this way, intervals provide an explicit representation of the reliability of the estimates, which can be taken into account when the extracted BN is employed to perform a particular task.

The second step of the BC method collapses the intervals estimated in the bound step into point estimates employing a convex combination of the extreme estimates. This convex combination can be determined either by using external information about the pattern of missing data or by a dynamic estimation of this pattern from the available data.

Assume that some external information is available on the pattern of missing data. One can encode this information as a probability distribution defining, for each datum in the database, the probability of the datum being missing as

$$P(x_{ik} \mid \pi_{ij}, X_i = ?) = \phi_{ijk}$$

where $k = 1,\ldots,c_i$, the number of state in X_i is denoted by c_i, and $\sum_k \phi_{ijk} = 1$. The notation $X_i = ?$ denotes that the state of X_i is missing. The probabilities ϕ_{ijk} can be employed to determine accurate estimates of θ_{ijk}, which is the probability of X_i being in the *k*th state given the parent states π_{ij}. A single probability for each state of the variable X_i given the parent states π_{ij} as

$$p_{k\bullet}(x_{il} \mid \pi_{ij}, D) = \frac{\alpha_{il} + n(x_{il} \mid \pi_{ij})}{\alpha_{ij} + n(\pi_{ij}) + n^{\bullet}(x_{ik} \mid \pi_{ij})} \tag{3.20}$$

for $l \neq k$. Therefore, the local minimum of $E(\theta_{ijk} \mid D)$ can be calculated as

$$p^{l}_{\bullet}(x_{ik} \mid \pi_{ij}, D) = \frac{\alpha_{ijk} + n(x_{ik} \mid \pi_{ij})}{\alpha_{ij} + n(\pi_{ij}) + \max_{h \neq k} n^{\bullet}(x_{ik} \mid \pi_{ij})}, \tag{3.21}$$

which shows that the difference between $p_{\bullet}(x_{ik} \mid \pi_{ij}, D)$ and $p^{l}_{\bullet}(x_{ik} \mid \pi_{ij}, D)$ depends only on the cases in which the state of the child variable is known and the parent configuration is not.

The distribution of missing entries in terms of ϕ_{ijk} can be employed to identify a point estimate within the interval $[p^{l}_{\bullet}(x_{ik} \mid \pi_{ij}, D), p^{\bullet}(x_{ik} \mid \pi_{ij}, D)]$ via convex combination of extreme probabilities:

$$\hat{p}(x_{ik} \mid \pi_{ij}, D, \phi_{ijk}) = \sum_{l \neq k} \phi_{ijk}\, p^{l}_{\bullet}(x_{ik} \mid \pi_{ij}, D) + \phi_{ijk}\, p^{\bullet}(x_{ik} \mid \pi_{ij}, D). \tag{3.22}$$

Finally, if data are missing only on the child variable ($n^{\bullet}(x_{ik} \mid \pi_{ij}) = n^{\bullet}_{ij}$), then we get

$$\hat{p}(x_{ik} \mid \pi_{ij}, D, \phi_{ijk}) = \frac{\alpha_{ijk} + n(x_{ik} \mid \pi_{ij}) + n^{\bullet}_{ij}\phi_{ijk}}{\alpha_{ij} + n(\pi_{ij}) + n^{\bullet}_{ij}} \tag{3.23}$$

so that the incomplete cases are distributed across the states of X_i according to the prior knowledge on the pattern of missing data. Note that Equation (3.23) is the *expected Bayesian estimate* given the assumed pattern of missing data [9].

If there is no external information about the pattern of missing data, the BC method works similar to EM and GS methods due to the use of the pattern of the available data. In this case, $\phi_{ijk} = p(x_{ik} \mid \pi_{ij})$ and it can be estimated from the available data as

$$\hat{\phi}_{ijk} = \frac{\alpha_{ijk} + n(x_{ik} \mid \pi_{ij})}{\alpha_{ij} + n(\pi_{ij})}. \tag{3.24}$$

This estimate can then be employed to compute the convex combination of the extreme probabilities. The estimate of $p(x_{ik} \mid \pi_{ij}, D)$ can be computed as

$$\hat{p}(x_{ik} \mid \pi_{ij}, D) = \frac{\alpha_{ijk} + n(x_{ik} \mid \pi_{ij}) + n_{ij}^{\bullet}\hat{\phi}_{ijk}}{\alpha_{ij} + n(\pi_{ij}) + n_{ij}^{\bullet}} = \frac{\alpha_{ijk} + n(x_{ik} \mid \pi_{ij})}{\alpha_{ij} + n(\pi_{ij})} \tag{3.25}$$

which is a consistent estimate of θ_{ijk} since $\hat{p}(x_{ik} \mid \pi_{ij}, D)$ is a generalized version of the Maximum Likelihood Estimate of θ_{ijk}. If $\alpha_{ijk} = 0$, then the BC estimate becomes the classical MLE of θ_{ijk}. Clearly, the estimates of the conditional probabilities computed by Equation (3.25) are the expected estimates and, as the database increases, they will be the same estimates computed by GS [9].

Sebastiani and Romani compared the accuracy and the efficiency of EM, GS, and BC methods. They found that both EM and GS provide reliable estimates of the parameters and they are currently regarded as the most viable solutions to the missing data [28]. On the other hand, both these iterative methods can be trapped into local minima and the convergence detection can be difficult. Furthermore, they assume that the missing data mechanism is *ignorable*; i.e., within each observed parent configuration, the available

data is a representative sample of the complete database and the distribution of missing data can therefore be inferred from the available entries [41]. When this assumption fails, and the missing data mechanism is *not ignorable* (NI), the accuracy of these methods can drastically decrease. Additionally, Sabestiani and Romani state that the computational cost of these methods depends mainly on the absolute number of missing data, and this dependency can prevent their scalability to large databases [28].

The most important characteristic of BC is its ability to represent the pattern of available data and the assumed pattern of missing data explicitly and separately. The BC algorithm provides probability intervals that can make the analyst aware of the range of possible estimates, and hence of the quality of information on which inference is based. The probability intervals used by BC provide a specific measure of the quality of information conveyed by the database and explicit representation of the impact of the assumption made on the pattern of missing data [9]. Therefore, BC does not depend on the ignorability assumption [28]. Furthermore, BC reduces the cost of estimating each conditional distribution of each variable X_i to the cost of one exact Bayesian updating and one convex combination for each state of X_i in each parent configuration. This deterministic process does not decrease the convergence rate and the convergence detection relative to stochastic processes. Additionally, the BC method's computational complexity is independent of the number of missing data [28].

Consequently, the BC algorithm gives almost the same results as EM and GS when the missing data is ignorable but it gives better results when the missing data mechanism is not ignorable. The convergence rate of BC is also better than EM and GS. Thus, BC learns the network faster than EM and GS methods [28]. The experimental comparison

with EM and GS proves that a substantial equivalence of the estimates provided by these three methods and a dramatic gain in efficiency using BC.

Ramoni and Sebastiani claimed the estimates provided by BC are more robust to departure of the data from the true pattern of missing data. The computational cost of BC is equal to the cost of two exact Bayesian updates−one for each extreme distribution−plus the cost of a convex combination for each parameter in the BN [45].

One may ask what happens if the network structure is unknown in addition to partially observable data. There is no easy answer to this question given in the literature. Some possibilities are explored in the next section.

3.4 Unknown structure and unobservable variables

This is the most difficult case to resolve because the structure of the networks is unknown and the variables are not fully observable. There is no significant amount of research for this case. When some variables are sometimes or always unobserved, the techniques stated in Section 3.2 for recovering the network structure become difficult to apply since they essentially require averaging over all possible combinations of values of the unknown variables [16]. There are two recently developed methods that recover the Bayesian network structure with unobserved variables.

The first algorithm was proposed by Russell [29] and is called *structural* EM (SEM) algorithm. The algorithm combines the standard EM algorithm, which optimizes the network parameters, with structure search for model selection. The main idea of this method is that it attempts to maximize the *expected score* of models instead of their actual scores at each iteration. Russell proves a theorem that the SEM algorithm makes

progress in each iteration on finding the better scoring network. Then, he states that if one chooses a model that maximizes the expected score at each iteration, then a better choice is provably made in terms of the marginal score of the network [29]. The SEM algorithm is exciting since it attempts to directly optimize the true Bayesian score within EM iteration rather than an asymptotic approximation.

The most problematic aspect of SEM is that it might converge to a sub-optimal model. This could happen if the model generates a distribution that causes other models to appear worse when the expected score is examined [29]. This difficulty becomes more obvious when the ratio of missing information is higher. Russell suggests that, in practice, the algorithm needs to be run from several starting points to get a better estimate of the MAP model [29]. Another restriction of the SEM is that it focuses on learning a single model. In practice, several high scoring models is necessary for better prediction. Additional to this deficiency, the algorithm requires large number of computations during learning. This is the main problem in applying this technique to large-scale domains. The following paragraphs provide a computationally cheaper method.

The second algorithm was proposed by Sebastiani and Marino [27]. They were able to show that BC algorithm could also learn the structure of the network with small changes in the algorithm. The algorithm has the following form:

Pick a random network structure $\mathbf{B}$ as starting point
Pick parameters for the network structure $\mathbf{B}$
Compute score for $\mathbf{B}$
Repeat
    Add an edge to the network, the network $\mathbf{B}'$ is created
    Estimate the posterior expectations of parameters of $\mathbf{B}'$ using BC method
    Estimate the posterior values of the network parameters
    Compute score for the network with $\mathbf{B}'$
    If score ($\mathbf{B}'$) > score ($\mathbf{B}$)
        Then let $\mathbf{B}$:= $\mathbf{B}'$

Else return ($\mathbf{B}$)

This method is very similar to the search method described in Section 3.2 where we had fully observed data. The only difference is that, in this case, we have partially observed data or incomplete data. Therefore, the estimation of the parameters of the network is also necessary. The BC method is employed to estimate the parameters of the network. The estimation process is performed in each step, i.e., after adding each edge to the network. Consequently, the method involves both parameter learning and structure learning. However, the main attention was given to the parameter estimation part since it is newly discovered method. The structure learning part can be modified as a greedy search algorithm. In that case, "delete an edge" operator and "reverse an edge" operator have to be incorporated to the algorithm.

There is a slight difference between SEM and BC methods and the problem of self-organizing agents in terms of required data structure. The SEM and BC algorithms require a certain minimum length database. Unfortunately, there will not be a prior database to work with at the beginning of the agents' exploration of the environment. Thus our learning method has to be *online*: estimation of the network structure and parameters will be performed simultaneously with the gathering of new entries in the database. So, our method has to learn the network while the agents are exploring the environment and organizing themselves to manage a common task. Using the current methods this problem cannot be solved because they do not contain an online learning algorithm. In the next chapter we propose a method that allows the agents learn the environment while they are exploring the environment and organizing a common task.

CHAPTER 4

Online Bayesian Network Learning and Multi-agent Organization

This chapter introduces online Bayesian network learning in detail. The structural and parametric learning abilities of the online Bayesian network learning are explored. The chapter starts with revisiting the multi-agent self-organization problem and the proposed solution. Section 4.2 explains the proposed Bayesian network learning.

4.1 Outline of the problem statement and the proposed solution

As stated in the introduction, we attempt to find how a common task can be performed by a multi-agent self-organizing system. The agents are independent in terms of their model of environment and their actions. Each agent explores the environment and decides its actions by itself. Agents will have no information about the environment at the beginning of their exploration of the environment. They will explore the environment, model the environment and take actions to change the environment according to the common task. We attempt to solve these problems by utilizing Bayesian networks and influence diagrams.

Bayesian networks are employed to model the environment. Because the agents have no or limited information about the environment at the beginning of their exploration, an *online* Bayesian network learning method will be used. Influence diagrams will be employed to obtain the agents' actions. Bayesian networks and influence diagrams are combined to produce a decision-theoretic agent [54] in a multi-agent system. Detailed

discussion on the decision-theoretic agent design is presented in Chapter 5. The Bayesian network learning is explored in the next section.

4.2 Online Bayesian network learning

Bayesian network learning is examined broadly in Chapter 3. There are four cases of Bayesian network learning depending on the availability of the network and the data. The unknown structure and incomplete data case is the nearest case to our problem. Our network structure is not defined in advance and the sensor data may not be complete. On the other hand, for simplicity we will assume the data is complete during the simulations. The agents do not have significant amounts of prior knowledge about the environment. Therefore, the BN will be formed during the agents' exploration of the environment. Each new data case will affect the structure of the network.

Online Bayesian network learning consist of two parts, namely parameter learning and structural learning. Parameter learning is the calculation of the conditional probability table elements of each node in a given Bayesian network. In this research, we use a modified version of Maximum Likelihood Expectation method to calculate the network parameters. Maximum likelihood estimation method is modified so that it has a closed form when the probabilities need to be updated. The details of the parameter learning are provided in Section 4.2.1.

Structural learning is the problem of finding the network that represents the data the best. This involves two parameters, complexity of the network and fitness of the network to the data. The structural learning process tries to find the optimal network that provides optimal complexity and fitness. The main building block in structural learning is the search algorithm that generates the network with the highest score. The structural

learning is presented in Section 4.2.2. The following section provides detailed description of the parameter learning in the online Bayesian learning.

4.2.1 The parameter learning

In Chapter 3, we introduced two types of parameter learning techniques used in the literature, MLE and Bayesian estimation. It is stated that with a database having a large number of data cases, these two methods converge to each other. The latter can take prior knowledge if it is available. Also, it is shown that the latter has a closed form. In this section we have redefined the Maximum Likelihood calculation to have a closed form calculation. Because MLE is computationally simpler than Bayesian estimation, it is employed in our parameter learning. The following paragraphs explain how the parameter learning is performed by modified MLE method.

Let $\mathbf{X} = \{X_1, X_2, \ldots X_m\}$ be the discrete variables (nodes) in a Bayesian network, **B**. Assume that we know that the node X_j is the child of the node X_i, which means $X_i \rightarrow X_j$. In this case, the parameter learning has to calculate the values in the conditional probability table in the node X_j. The conditional probability can be calculated by utilizing using the *fundamental formula* for probability calculus as in Equation (4.1)

$$P(X_i \mid X_j) = \frac{P(X_i, X_j)}{P(X_j)} \tag{4.1}$$

Since MLE is employed in parameter learning, the probabilities can be calculated by utilizing the natural frequencies of the data cases. A natural frequency of a data case is calculated by counting the number of occurrences of the data case in the database. For individual probabilities, we count the number of occurrences of a state of a variable in the

database. Let n_{ij} be the number of occurrences of the state j of the ith variable in the database and n is the total number of data cases in the database. Using these frequency values, we can calculate the probabilities in the following way:

$$P(X_i = x_j) = \frac{n(X_i = x_j)}{n} = \frac{n_{ij}}{n} \tag{4.2}$$

Thus, the conditional probabilities can be calculated by using the individual probabilities in Equation (4.1). The conditional probability $P(X_i \rightarrow X_j)$ can be obtained as in the following equations.

$$P(X_i \mid X_j) = \frac{P(X_i, X_j)}{P(X_j)} \tag{4.3}$$

$$P(X_i, X_j) = \frac{n(X_i, X_j)}{n} \tag{4.4}$$

$$P(X_j) = \frac{n(X_j)}{n} \tag{4.5}$$

As can be seen in Equations (4.4) and (4.5), the denominators are the same in the both terms. When we put these two terms into Equation (4.3), the denominators cancel each other as shown in the following equation.

$$P(X_i \mid X_j) = \frac{n(X_i, X_j) \big/ n}{n(X_j) \big/ n} = \frac{n(X_i, X_j)}{n(X_j)} \tag{4.6}$$

In the resulting equation, there are only two natural frequencies. There is no need to involve the number of elements in the database for conditional probability calculations. This technique simplifies the computations in the parameter learning. Equation (4.6) has a closed form because if a new data case is encountered, we can easily update the corresponding natural frequencies accordingly to update the conditional probabilities.

The following example provides practical results to the conditional probability calculation technique. For the cases that have not seen yet, the uniform probability distribution is used to fill the conditional probability tables in the nodes.

Let X_1, X_2, and X_3 be the system variables with two possible states, 0 and 1, in a Bayesian network, **B**. Assume that we know the system dynamics (dependencies), $X_1 \rightarrow X_2$ and $X_2 \rightarrow X_3$. Therefore, we need to calculate the conditional probabilities, $P(X_2 \mid X_1)$ and $P(X_3 \mid X_2)$. Let D be the database of cases to calculate the conditional probabilities, shown in Table 4.1.

Table 4.1. The database to compute the parameters of the BN.

X_1	X_2	X_3
0	0	0
1	0	1
1	1	0
0	1	1
1	0	0
1	0	0
0	1	0
1	1	1
0	1	0
0	1	1

For example, we can calculate the probabilities of the variable X_1 as in the following equation.

$$P(X_1 = 0) = \frac{n(X_1 = 0)}{n} = \frac{5}{10} \Rightarrow P(X_1 = 0) = 0.5 \tag{4.7}$$

Now, let us calculate the conditional probability of $P(X_2 = 0 \mid X_1 = 0)$ by counting corresponding frequency values, $n(X_2 = 0, X_1 = 0)$ and $n(X_1 = 0)$.

$$P(X_2 = 0 \mid X_1 = 0) = \frac{n(X_2 = 0, X_1 = 0)}{n(X_1 = 0)} = \frac{1}{5} = 0.2 \tag{4.8}$$

$$P(X_2 = 1 \mid X_1 = 0) = 0.8 \tag{4.9}$$

Similarly,

$$P(X_2 = 0 \mid X_1 = 1) = \frac{n(X_2 = 0, X_1 = 1)}{n(X_1 = 1)} = \frac{3}{5} = 0.6 \tag{4.10}$$

$$P(X_2 = 1 \mid X_1 = 1) = 0.4 \tag{4.11}$$

The conditional probability $P(X_3 \mid X_2)$ can be easily calculated by counting the corresponding natural frequencies.

The above technique is also useful to update the probabilities when a new case is introduced to the network. When a new case is encountered, the related frequency counts can be updated to calculate the new probabilities. Jensen introduces a similar updating scheme in [1] as *fractional updating*.

The above technique works if the state of X_3 as well as the states of its parents are known. This could be a problem if the states of the parents are not known when the probability update is being done. In our problem, the network update is done after new data are gather for all the variables in the network. Therefore, the above restriction does not apply to our problem.

For online Bayesian network learning, the parameter learning is not enough because the agents do not know the system dynamics in advance. Thus, the structural learning part is also necessary to discover the system dynamics. The following section presents the details of the structural learning techniques explored in this research.

4.2.2 The structural learning

Structural learning is finding the best network that fits the available data and is optimally complex. This can be accomplished by utilizing a search algorithm over the possible network structures. In this research, a greater importance is given to the search algorithm because we have assumed that the data will be complete. That is, each element of the database is a valid state of a variable. If there are non-applicable entries in the database, then the database is said to be incomplete.

The greedy search algorithm, explained in Chapter 3, is employed to accomplish the structural learning in the online Bayesian network learning. The search algorithm is a score based searching algorithm. The search algorithm is evaluated in terms of the score function used and the technique used to create the candidate networks, such as adding an edge and removing an edge.

The greedy search algorithm is also upgraded to have some online properties such as updating the network parameters and its structure adaptively. The outline of the search algorithm can be given as follows:

1. Collect data
2. Define the variables from the available data
3. Start with a network with no arcs
4. Estimate the parameters (only independent probabilities) of the BN using the MLE method using initial data.
5. Generate candidate networks by adding arcs in a defined fashion (heuristic or exhaustive)

6. Calculate the scores of the candidate networks and choose the network with the highest score.
7. Do step 5 until no arc addition increases the likelihood of the network.
8. Update the network parameters along with new data
9. Update the network structure:
    - If enough new data obtained, go to step 1 and generate a new network structure.
    - If no structural update is necessary go to step 7.

The algorithm above is a generic greedy search algorithm. How the arc addition is done and which scoring method is used are not specified in the above algorithm. In the following section we explore the search algorithms used in this research. In the algorithms, the arcs are added heuristically and exhaustively.

4.2.2.1 Search algorithms

A Bayesian network is not allowed to have a cycle because of the computational difficulties. A cycle in a Bayesian network lead to a "circular reasoning" between the variables. For example, if the dependencies in above network are: $X_1 \rightarrow X_2$, $X_2 \rightarrow X_3$, and $X_3 \rightarrow X_1$, a cycle will be formed. If evidence is entered into the variable X_1, the Bayesian network will run the evidence to X_2, then to X_3. Then, The evidence will travel to X_1 because X_1 depends on X_3. The evidence may run in the network forever because all the variables depend on each other in a circular way.

A heuristic arc addition is employed not to have a cycle in the Bayesian network while generating the Bayesian structure. An exhaustive arc addition is also employed to explore more network possibilities without limitation. In the exhaustive arc addition algorithm, a cycle check is employed before and arc is added. The following section presents the details of heuristic and exhaustive search algorithms.

Heuristic search

In the heuristic search algorithm, the variables of the system have to be ordered in a certain way to prevent cycles from being created. The decision variables should be in the last columns in the database; and, the first columns of the database should be filled with the variables without parents, independent variables. After placing the independent variables in the first columns, the children of the independent variables should be placed in the following columns. The rest of the columns are filled with the children of the previously placed variables. Ordering of the variables is necessary because the heuristic arc addition adds the arcs from the first variables to the last variables. Because of the ordering, we need to have some knowledge about the variables. This does not mean that we need to know the dependencies between the variables. For example, let **B** be a Bayesian network with three variables, $\{X_1, X_2, X_3\}$. If we know the variable X_1 is the first variable and the variable X_2 is the decision node. Then the column order will be $\{X_1, X_3, X_2\}$.

The heuristic search starts with adding and removing arcs from the each variable to the last variable. Let the network have *n* variables. After adding an arc, the algorithm calculates the network score, records the score in a list, and removes the arc. The

algorithm finds the arc that gives the highest increase in the network score. Let us assume that the arc from the *k*th variable to the last variable, *n*, gives the highest increase in network score. Then, the algorithm adds the arc from the *k*th variable to the last variable. After the arc is added, the algorithm adds and removes arcs from the remaining variables to the last variable. Then, the algorithm chooses the arc with the highest score increase and adds the arc to the network. This continues until no increase in the network score can be obtained by adding an arc to the last variable. Then, the algorithm starts adding arcs from the variables $\{1,2,\ldots,n-2\}$ to the (*n*-1)th node. The algorithm adds arcs to (*n*-1)th node until there is no increase in the network score. The algorithm stops when it adds an arc from the first variable to the second variable. The following is the heuristic search algorithm used in this research.

1. Collect data
2. Define the variables from the available data
3. Start with a network with no arc.
4. Estimate the parameters (only independent probabilities) of the BN using the MLE method using initial data
5. Add a new arc from the *i*th variable to the *j*th variable to generate a network candidate and remove the arc. Repeat the process with $i = \{1,2,\ldots, j-1\}$ and generate networks ($\mathbf{B}_1,\ldots, \mathbf{B}_{j-1}$). Start *j* from *n* and decrease *j* by 1.
6. Calculate the scores of the candidate networks and record them in a list.
7. Find the network ($\mathbf{B}'$) with the maximum score and keep it for the next step.
8. Repeat the steps 5, 6, and 7 until there is no increase in the network score.
9. If $j > 1$, then go to step 5.

10. Update the network parameters along with new data

11. Update the network structure:

- If enough new data obtained, go to step 1 and generate a new network structure.
- If no structural update is necessary go to step 10.

Consequently, the heuristic search algorithm adds arcs only in the forward direction because this protects the network from having cycles and complex network structure. On the other hand, there is a price of arranging the variables at the creation of the database in the heuristic algorithm. Since the agents will not have much knowledge about the environmental variables, it is hard to arrange the variables at the beginning. There is a need for a better search algorithm that explores more possibilities in the network. The following paragraph introduces another searching algorithm that eliminates the arranging the variables, namely exhaustive search.

Exhaustive search

The exhaustive search algorithm explores all the possible arcs in the network during its execution. The algorithm starts adding arcs from the *i*th variable to the *j*th variable where $i = \{1,2,\ldots,n\}$, $j = \{1,2,\ldots,n\}$, $i \neq j$. This covers $n \cdot (n-1)$ arcs throughout the network. The algorithm calculates the network score for each arc addition. Then, it chooses the arc with the highest increase in the network score. The algorithm repeats the above steps until there is no increase in the network score.

There are two major drawbacks in the exhaustive search algorithm. First, the number of arcs to be tried might become intractable when the number of variables is large.

Second, during the search, the algorithm might introduce cycles to the network because it can add an arc in any direction. An additional algorithm is incorporated to the search algorithm to keep track of cycles. Using the additional algorithm, the search algorithm checks whether the new arc introduces a cycle or not. If the arc introduces a cycle, the algorithm does not add the arc to the network. The following is the exhaustive search algorithm used in this research.

1 Collect data

2 Define the variables from the available data

3 Start with an empty network

4 Estimate the parameters (only independent probabilities) of the BN using the MLE method using initial data

5 Add a new arc from the *i*th variable to the *j*th variable to create a candidate network and remove the arc. Repeat the process for every value of *i* and *j* where $i = \{1,2,\ldots,n\}$, $j = \{1,2,\ldots,n\}$ and $i \neq j$. This step creates *m* possible networks ($\mathbf{B}_1,\ldots,\mathbf{B}_m$). Algorithm creates $m = n \times (n-1)$ networks in first visit to step 5.

6 Remove the network with cycles from the candidate list.

7 Calculate the scores of the candidate networks and record it in a list.

8 Find the network ($\mathbf{B}'$) with the maximum score and keep it for the next step.

9 Do step 5 through 8 until there is no increase in the network score.

10 Update the network parameters along with new data

11 Update the network structure:

- If enough new data obtained, go to step 1 and generate a new network structure.

- If no structural update is necessary go to step 10.

The search algorithms are explained in detail. There is a need to analyze the complexity of the search algorithm before there are implemented. The following section gives the complexity analysis of both search algorithms.

Complexity analysis for search algorithms

As stated earlier, the heuristic search algorithm needs prior knowledge about the variables in terms of their order in the database. On the other hand, the number of iterations in the heuristic search algorithm may be tractable. In the heuristic search, the algorithm tries $(n-1)$ arcs in the first trip from step 5 to step 7. The algorithm repeats steps 5 through 7 until there is no increase in the network score. Assuming the algorithm adds an arc in every trip, the number of arcs tried will be one less then the previous trip. Algorithm can repeat step 5 through 7 at most $(n-1)$ times. In $(n-1)$ trips, the algorithm generates $(n-1)+(n-2)+\cdots+1$ networks candidates. When the algorithm reaches step 8, the algorithm loops back to step 5 and repeats the same process for the variables $\{X_{n-1}, X_{n-2}, \cdots, X_2\}$. Therefore, after the first loop, the algorithm generates $(n-1)+(n-2)+\cdots+1$ network candidates. The complexity of the heuristic search algorithm is denoted as C_h.

In the following complexity analysis, each loop shows the number of network candidates tried until the algorithm reaches to the step 8. Since the algorithm will repeat itself for $(n-1)$ variables, the analysis has $(n-1)$ loops as the following.

Loop 1 $\quad (n-1)+(n-2)+\cdots+1 = n(n-1)-(1+2+\cdots+(n-1))$

$$\Rightarrow n(n-1)-\frac{n(n-1)}{2}\Rightarrow\frac{n(n-1)}{2}$$

Loop 2 $$(n-2)+(n-3)+\cdots+1=\frac{(n-1)(n-2)}{2}$$

⋮

⋮

Loop (n-1). $$\frac{(n-(n-1))(n-(n-2))}{2}=1$$

If we add the number of candidate networks from each loop, the following can be obtained:

$$C_h=\frac{n(n-1)+(n-1)(n-2)+\cdots\cdots+(n-(n-1))(n-(n-2))}{2}$$

$$C_h=\frac{2(n-1)^2+2(n-3)^2+\cdots\cdots+2(n-(n-2))^2}{2}$$

Then, we can further modify the equation as follows:

$$C_h=(n-1)^2+(n-3)^2+\cdots\cdots+(n-(n-2))^2 \tag{4.12}$$

Since each element in C_h is less than n^2, we can state that

$$C_h<n^2(n-3)<n^3 \tag{4.13}$$

Equation (4.13) illustrates the complexity of the heuristic search. The following paragraphs will explore the complexity of the exhaustive search algorithm.

The exhaustive search algorithm tries every possible arc in the network during its first visit to step 5. In a graph with n nodes, there can be $n(n-1)$ possible directed edges in the graph [30]. Therefore, the algorithm generates $n(n-1)$ network candidates and the complexity of the first visit is $n(n-1)$. Then the algorithm continues until it reaches to step 9 and loops back to step 5 until there is no increase in the network score.

After the first loop, the complexity decreases by 1 in each step because the algorithm will not try the arc added in the previous step. The following presents the complexity analysis of the exhaustive search algorithm. First, the complexity is calculated for each loop. Then, they are added to obtain the complexity of the algorithm.

Loop 1 $n(n-1)$

Loop 2 $n(n-1)-1$

$\vdots$

Loop N $n(n-1)-N+1$

The exhaustive search algorithm does not perform a certain number of loops. The algorithm will continue until there is no increase in the network score. Therefore, we will assume that the algorithm end after N loops for the complexity calculations. If we add the complexities of all the loops together, the complexity of the exhaustive search, C_e, becomes the following.

$$C_e = n(n-1)N - (1+2+\cdots+(N-1)) \tag{4.14}$$

$$C_e = n(n-1)N - \frac{N(N-1)}{2} \tag{4.15}$$

If the network has great number of arcs, then the complexity of the algorithm becomes large. For example, if the algorithm ends in step $N = n$, the complexity becomes

$$C_e = n^2(n-1) - \frac{n(n-1)}{2} = \frac{2n^2(n-1) - n(n-1)}{2} \tag{4.16}$$

$$C_e = \frac{(n-1)n(2n-1)}{2} \qquad \text{for } n = N. \tag{4.17}$$

In general, the number of nodes in a Bayesian network, n, is much larger than 1. Therefore, we can reevaluate the complexity by assuming $n >> 1$. The following

equation represents the computational complexity of the exhaustive search algorithm when the number of steps is equal to the number of variables.

$$C_e \cong \frac{n \cdot n \cdot 2n}{2} = \frac{2n^3}{2} \Rightarrow C_e \cong n^3 \tag{4.18}$$

As can be seen above, the complexity of the exhaustive algorithm is larger than the complexity of the heuristic algorithm when $N = n$.

For the networks with large number of variables (nodes), the algorithm does not stop when $N = n$. Let us calculate the worst case scenario for the exhaustive algorithm. The algorithm might explore all possible arcs in the network, which is equal to $n(n-1)$. This is true because a complete graph with *n* nodes has $n(n-1)$ possible directed edges [30]. Therefore, we will replace *N* with $n(n-1)$ in the complexity analysis. Then, the complexity of the exhaustive search algorithm becomes the following.

$$C_e = n(n-1)N - \frac{N(N-1)}{2} = n(n-1)n(n-1) - \frac{n(n-1)[n(n-1)-1]}{2} \tag{4.19}$$

$$C_e = \frac{2n^2(n-1)^2 - n^2(n-1)^2 - n(n-1)}{2} = \frac{n^2(n-1)^2 - n(n-1)}{2} \tag{4.20}$$

We can simplify the equation above by assuming $n >> 1$. In this case, the complexity of the algorithm becomes the following.

$$C_e \cong \frac{n^2 \cdot n^2 - n^2}{2} = \frac{n^2(n^2-1)}{2} \Rightarrow C_e \cong \frac{n^4}{2} \tag{4.21}$$

Two search algorithms are introduced to learn the structure of a Bayesian network in the previous sections. The heuristic search algorithm is simple and explores a limited number of network structures. On the other hand, the exhaustive search algorithm is complex and explores many possible network structures. The complexity of the

exhaustive algorithm is approximately *n*-fold larger than the complexity of the heuristic search algorithm. Since we calculate the quality (score) of the networks to find the best network, the search algorithm is a score based algorithm. The following section presents the scoring functions explored in this research.

4.2.2.2 Network scoring functions

Three scoring functions are employed in this research, namely Log-Likelihood, Minimum description length (MDL), and Bayesian (BDE) scores. The Log-Likelihood method measures the likelihood of the network given the available data. The MDL also uses likelihood of the network but it includes the measure of the network's complexity. The Bayesian score involves the calculation of the probability of a network given the data. Bayesian scoring method also penalizes complex networks as the MDL scoring. If the length of the database is large enough these two methods converge to each other [54]. The following sections provide the details of the scoring methods used in the research.

Log-Likelihood scoring

The Log-Likelihood score of a network, **B**, is obtained by calculating the likelihood of the data, *D*, given the network, **B,** and the network parameters, $\theta_{\mathbf{B}}$. After calculating the likelihood of the data, a natural logarithm is applied to get the Log-Likelihood of the data. The following formulas explain the details of the Log-Likelihood calculation.

$$Score_L(\mathbf{B} : D) = L(D \mid \mathbf{B}, \theta_{\mathbf{B}}) \tag{4.22}$$

$$L(D \mid \mathbf{B}, \theta_{\mathbf{B}}) = \prod_m P(d[m] \mid \mathbf{B}, \theta_{\mathbf{B}}) \tag{4.23}$$

In the above formula, $d[m]$ represents the mth data case in the database. Let us take the logarithm of the likelihood. The logarithm converts the multiplication in to a summation.

$$l(D \mid \mathbf{B}, \theta_{\mathbf{B}}) = \log L(D \mid \mathbf{B}, \theta_{\mathbf{B}}) \tag{4.24}$$

$$l(D \mid \mathbf{B}, \theta_{\mathbf{B}}) = \sum_{m} \log P(d[m] \mid \mathbf{B}, \theta_{\mathbf{B}}) \tag{4.25}$$

This is basically equal to calculating the probability of each data case in the database, taking their logarithms and adding them together. For example, assume that the network given in the previous section has the relations $X_1 \to X_3$ and $X_3 \to X_2$. Then, we can calculate the log-likelihood of the data with the following equation.

$$\begin{aligned} l(D \mid \mathbf{B}, \theta_B) = & \sum_{m} \log P(X_1[m] \mid \theta_{X_1}) \\ & + \sum_{m} \log P(X_3[m] \mid x_{10}, \theta_{X_2}) + \sum_{m} \log P(X_3[m] \mid x_{11}, \theta_{X_3}) \\ + \sum_{m} \log P(X_2[m] \mid & x_{30}, \theta_{X_2 \mid x_{30}}) + \sum_{m} \log P(X_2[m] \mid x_{31}, \theta_{X_2 \mid x_{31}}) \end{aligned} \tag{4.26}$$

In the log-likelihood approach, the score of the network increases as long as the length of the database and the number of arc in the network increase. Therefore, the search algorithm tries to add as many arcs as possible to the network to get the highest scoring network. At the end of the search, the algorithm ends up with almost a complete network. For the networks with a large number of nodes, this might cause a great increase in complexity of the network. To overcome the complexity problem, we need to find out a way to include the complexity of the network to the scoring function. If the network gets complex, the scoring function should decrease accordingly. The following scoring method handles the complexity problem by introducing the complexity parameter in the scoring function.

Minimum description length scoring

The MDL method combines the likelihood of the data and the complexity of the network to find optimally complex and accurate networks. The MDL method penalizes networks with complex structures. The MDL has two parts, the complexity of the network, $L_{NETWORK}$, and the likelihood of the data, L_{DATA}. Then, the MDL score can be calculated by the following.

$$Score_{MDL} = L_{DATA} - L_{NETWORK} \tag{4.27}$$

The complexity part involves the dimension of the network, $Dim(\mathbf{B})$, and structural complexity of the network, $DL(\mathbf{B})$. The dimension of the network can be calculated using the number of states in each node, S_i. The following equation illustrates the dimension of the network.

$$Dim(\mathbf{B}) = \sum_{i=1}^{N} (S_i - 1) \prod_{j \in pa(x_i)} S_j \tag{4.28}$$

where N is the number of nodes in the network. Let M be the number of data cases in the database. Using the central limit theorem, each parameter has a variance of $\sqrt{M}$. Thus, for each parameter in the network, the number of bits required is given by the following.

$$d = \log \sqrt{M} \Rightarrow d = \frac{\log M}{2} \tag{4.29}$$

The structural complexity of the network depends on the number of parents of the nodes. The following formula calculates the structural complexity.

$$DL(\mathbf{B}) = \sum_{i=1}^{N} k_i \log_2 (N) \tag{4.30}$$

where k_i is the number of parents the node X_i has. Finally, the following formula presents the complexity part of the MDL score by combining the dimension of the network and the structural complexity.

$$L_{NETWORK} = \frac{\log M}{2} Dim(\mathbf{B}) + DL(\mathbf{B}) \tag{4.31}$$

$$L_{NETWORK} = \frac{\log M}{2} \left(\sum_{i=1}^{N} (S_i - 1) \prod_{j \in pa(x_i)} S_j \right) + \sum_{i=1}^{N} k_i \log_2 (N) \tag{4.32}$$

The likelihood of the data needs to be defined after presenting the network complexity part of the MDL score. The likelihood of the data given a network can be calculated by using cross-entropy. The difference between the distribution of the data (*P*) and the estimated distribution (*Q*) from the network. Kullback-Leiber and Euclidean distance are the commonly used cross-entropy methods. Therefore, the likelihood of a data can be calculated by measuring the distance between two distributions. If we use the Kullback-Leiber cross-entropy, the likelihood of the data can be calculated by the following.

$$l(D \mid \mathbf{B}, \theta_{\mathbf{B}}) = \sum_{i=1}^{M} p_i \log \frac{p_i}{q_i}, \tag{4.33}$$

$$L_{DATA} = \sum_{i=1}^{M} p_i \log \frac{p_i}{q_i} \tag{4.34}$$

where p_i is the probability of data case i using the database and q_i is the estimate of the probability of data case i from the network parameters. If Euclidean distance measure is employed to calculate the distance between the distributions, the likelihood of the data is calculated by the following.

$$l(D \mid \mathbf{B}, \hat{\theta}_{\mathbf{B}}) = \sum_{i=1}^{M} (p_i - q_i)^2 \tag{4.35}$$

$$L_{DATA} = \sum_{i=1}^{M} (p_i - q_i)^2 \tag{4.36}$$

After defining the likelihood and complexity parts, the MDL score can be given as

$$Score_{MDL}(\mathbf{B} : D) = l(D \mid \mathbf{B}, \theta_{\mathbf{B}}) - \frac{\log M}{2} Dim(\mathbf{B}) - DL(\mathbf{B}) \tag{4.37}$$

Another commonly used scoring method is Bayesian score as explained in Chapter 3. Now, we will provide the details of the Bayesian scoring technique. Bayesian scoring is calculated by utilizing the Dirichlet parameters of the network.

Bayesian scoring

Bayesian statistics tells us that we should rank a prior probability over anything we are uncertain about. In this case, we put a prior probability both over our parameters and over our structure. The Bayesian score can be evaluated as the probability of the structure given the data:

$$Score_{BDE}(\mathrm{B} : D) = P(\mathrm{B} \mid D) = \frac{P(D \mid \mathrm{B})P(\mathrm{B})}{P(D)} \tag{4.38}$$

The probability $P(D)$ is constant. Therefore, it can be ignored when comparing different structures. Thus, we can choose the model that maximizes $P(D \mid \mathbf{B})P(\mathbf{B})$. Let us assume that we do not have prior over the network structures. Assume that we have uniform prior over the structures. One might ask whether we get back to the maximum likelihood score. The answer is 'no' because the maximum likelihood score for $\mathbf{B}$ was $P(D \mid \mathbf{B}, \theta_{\mathbf{B}})$, i.e. the probability of the data in the most likely parameter instantiation. In Bayesian scoring, we have not given the parameters. Therefore, we have to integrate over all possible parameter vectors:

$$P(D \mid \mathrm{B}) = \int P(D \mid \theta_{\mathrm{B}}, \mathrm{B}) P(\theta_{\mathrm{B}} \mid \mathrm{B}) d\theta_{\mathrm{B}} \tag{4.39}$$

This is, of course, different from the maximum likelihood score.

To understand the Bayesian scoring better, consider two possible structures for a two-node network, where $\mathbf{B}_1 = [A \quad B]$ and $\mathbf{B}_2 = [A \rightarrow B]$. Then, the probability of the data given the network structures can be calculated by the following equations.

$$P(D \mid \mathbf{B}_1) = \int_0^1 P(\theta_A, \theta_B) P(D \mid [\theta_A, \theta_B]) d[\theta_A, \theta_B] \tag{4.40}$$

$$P(D \mid \mathbf{B}_2) = \int_0^1 P(\theta_A, \theta_{B|a_0}, \theta_{B|a_1}) P(D \mid [\theta_A, \theta_{B|a_0}, \theta_{B|a_1}]) d[\theta_A, \theta_{B|a_0}, \theta_{B|a_1}] \tag{4.41}$$

The latter is a higher dimensional integral, and its value is therefore likely to be somewhat lower. This is because there are more numbers less than 1 in the multiplication. Multiplying the numbers less than 1 results in a number smaller than any of the number in the multiplication. For example, multiplying three small numbers (less than 1) is likely to be smaller than the number obtained by multiplying two small numbers (less than 1). Since the probabilities in the integrals are less than 1, the above argument applies to the integrals. Therefore, it can be said that the higher dimensional integral is likely to have lower value that the lower dimensional integral. This idea presents preference to the networks with fewer parameters. This is an automatic control in the complexity of the network.

Let us analyze $P(D \mid \mathbf{B})$ a little more closely to understand the Bayesian score calculations. It is helpful to first consider the single parameter case even though there is no structure learning to learn there. In that case, there is a simple closed form solution for the probability of the data given by the following.

$$P(D) = \frac{\Gamma(\alpha)}{\Gamma(\alpha_0 + \alpha_1)} \cdot \frac{\Gamma(\alpha_0 + n_0) \cdot \Gamma(\alpha_1 + n_1)}{\Gamma(\alpha + n)} \tag{4.42}$$

where $\Gamma[m]$ is equal to $(m-1)!$ for an integer *m*, *n* is the number of data cases in the database, n_0 and n_1 are the number of zeros and ones, respectively, and $\alpha = \alpha_0 + \alpha_1$. Let us assume we have 40 zeros and 60 ones in the database. Assuming that we have uniform priors, $\alpha_0 = \alpha_1 = 3$, the probability of data is

$$P(D) = \frac{\Gamma(6)}{\Gamma(3)\Gamma(3)} \cdot \frac{\Gamma(3+40)\Gamma(3+60)}{\Gamma(6+100)} \tag{4.43}$$

The probability for a structure with several parameters is simply the product of the probabilities for the individual parameters. For example, in our two-node network, if the same priors are used for all three parameters, and we have 45 zeros and 55 ones for the variable *B*, then, the probability of the data for the network $\mathbf{B}_1$ can be calculated as

$$P(D \mid \mathbf{B}_1) = \frac{\Gamma(6)}{\Gamma(3)\Gamma(3)} \frac{\Gamma(43)\Gamma(63)}{\Gamma(106)} \cdot \frac{\Gamma(6)}{\Gamma(3)\Gamma(3)} \frac{\Gamma(48)\Gamma(58)}{\Gamma(106)} \tag{4.44}$$

For the second network, let us assume that $\alpha_{00} = 23$, $\alpha_{01} = 22$, $\alpha_{10} = 29$, and $\alpha_{11} = 26$, where $\alpha_{ij} = n(a_i, b_j)$ is the number of cases with $A = a_i$ and $B = b_j$. Then, we can compute the probability of the data for the network $\mathbf{B}_2$ using the following equation.

$$\begin{aligned} P(D \mid \mathbf{B}_2) = & \frac{\Gamma(6)}{\Gamma(3)\Gamma(3)} \frac{\Gamma(43)\Gamma(63)}{\Gamma(106)} \cdot \frac{\Gamma(6)}{\Gamma(3)\Gamma(3)} \frac{\Gamma(23+3)\Gamma(22+3)}{\Gamma(45+3)} \\ & \cdot \frac{\Gamma(6)}{\Gamma(3)\Gamma(3)} \frac{\Gamma(29+3)\Gamma(26+3)}{\Gamma(55+3)} \end{aligned} \tag{4.45}$$

The intuition is clearer. The analysis shows that we get a higher score by multiplying a smaller number of bigger factorials rather than a larger number of small ones.

It turns out that if we approximate the log posterior probability, and ignore all terms that do not grow with *M*, we can obtain

$$\log P(D \mid \mathbf{B}) \approx l(D \mid \theta_{\mathbf{B}}, \mathbf{B}) - \frac{\log M}{2} Dim(\mathbf{B}) \tag{4.46}$$

i.e, as *M* grows large, the Bayesian score and the MDL score converge to each other using Dirichlet priors. In fact, if we use a good approximation to the Bayesian score, and eliminate all terms that do not grow with *M*, then we are left exactly with MDL score [54]. Therefore, it can be concluded that the Bayesian score gives us, automatically, a tradeoff between network complexity and fit to the data.

The Bayesian score is also decomposable like the MDL score since it can be expressed as a summation of terms that corresponds to individual nodes. In this research, we have decomposed the Bayesian score to make efficient calculations and a uniform distribution is employed for Dirichlet priors. The simulation results will show that the Bayesian score provides optimally complex and accurate network structures.

The online Bayesian network learning is proposed to model the environment for an agent. Online Bayesian network learning has both structural and parametric learning because it can discover the structure of the network and the conditional probabilities in the network. After explaining the proposed Bayesian network learning, there is a need to explain how the proposed Bayesian network learning and influence diagrams can be combined to for an intelligent agent structure. The next chapter describes the design process of the decision-theoretic intelligent agent and how a multi-agent self-organization system can be designed by employing these agents.

CHAPTER 5

Multi-agent self-organization system

As discussed in Chapter 1, in the literature, several methods are employed in multi-agent learning and organization problem such as temporal difference (TD(λ)), genetic algorithms, and learning classifier systems. The advantages and disadvantages of these methods are also examined in Chapter 1. The main disadvantage of these methods is that they perform badly when the data is not fully observable. Additionally, they do not have the desired bi-directional learning property. We proposed Bayesian networks to ease these problems because they can perform well with the partially observable data and, more importantly, Bayesian networks have the bi-directional learning ability. The following paragraphs will illustrate how Bayesian networks can solve the multi-agent self-organization problem with the help of influence diagrams. The next section will explain the structure of an agent, which is designed by a Bayesian network and an influence diagram. Section 5.2 and Section 5.3 will examine a multi-agent organization system and the bi-directional learning feature of the proposed multi-agent self-organizing system. Finally, Section 5.4 presents the system representation of the decision-theoretic intelligent agent design.

5.1 A decision-theoretic intelligent agent design

In Chapter 1, an agent was defined as an entity that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [54]. Therefore, an agent should have sensors and actuators to interact with the environment.

On the other hand, an intelligent agent is an agent that reasons with the sensory information and creates optimal actions to satisfy a goal. Therefore, a reasoning system and a decision support system are necessary elements of an intelligent agent. Bayesian networks and influence diagrams can be considered as reasoning systems and decision support systems respectively.

Communication between the agents is also necessary to establish organizational behaviors in a multi-agent self-organizing system. Therefore, an intelligent agent should have sensors, actuators for actions, a Bayesian network, an influence diagram and a communication system.

An intelligent agent has five levels: sensors, belief, preferences, capabilities and actions. In this design, Shohams' agent oriented programming paradigm is followed. According to this paradigm, the mental state of agents can be represented in terms of their *belief*, *capabilities*, and *preferences* [4]. The belief level consists of a Bayesian network (V_A or V_E) and its nodes represent agent's possibly uncertain beliefs about the world. The nodes in V_A represent variables related to the other agents in the system. The nodes in V_E represent the variables related to the agent itself. The preference level is represented as a utility node (U_A and U_E) that expresses the desirability of a world state. The capability level is represented by decision nodes (V_{DA} and V_{DE}) that contain alternative courses of action, which the agent can execute to interact with the world [42]. This is also called belief, desire, and intention (BDI) architecture in the literature [42].

Each agent models other agents as an influence diagram by modeling other agents' variables (V_A), utility function (U_A), and decision nodes (V_{DA}). Duryadi and Gmytrasiewicz stated that other agents' models could be learned using influence

diagrams [42]. As a modeling representation tool, the influence diagram is able to express an agent's belief, capabilities and preferences, which are required if we want to predict the agent's behavior [42]. Duryadi and Gmytrasiewicz established the learning of other agents' behaviors in the following way: Given an initial model of an agent and a history of its observed behavior, new models can be constructed by refining the parameters of the influence diagram in the initial model. The details of the learning method can be seen in [42].

Agents also need a model of the environment. Bayesian networks can model the environment efficiently, as stated in Chapter 2. The nodes in V_E model the environment and provide beliefs about the environment. Then, these beliefs are dragged into the utility node U_E. The utility node U_E represents the agent's own preference that is defined by the goal of the multi-agent organization system. The utility U_E is a function of the belief about the environment (V_E), the expected actions of the other agents (A_2), its possibly course of actions (A_1). Figure 5.1 presents the proposed intelligent agent model.

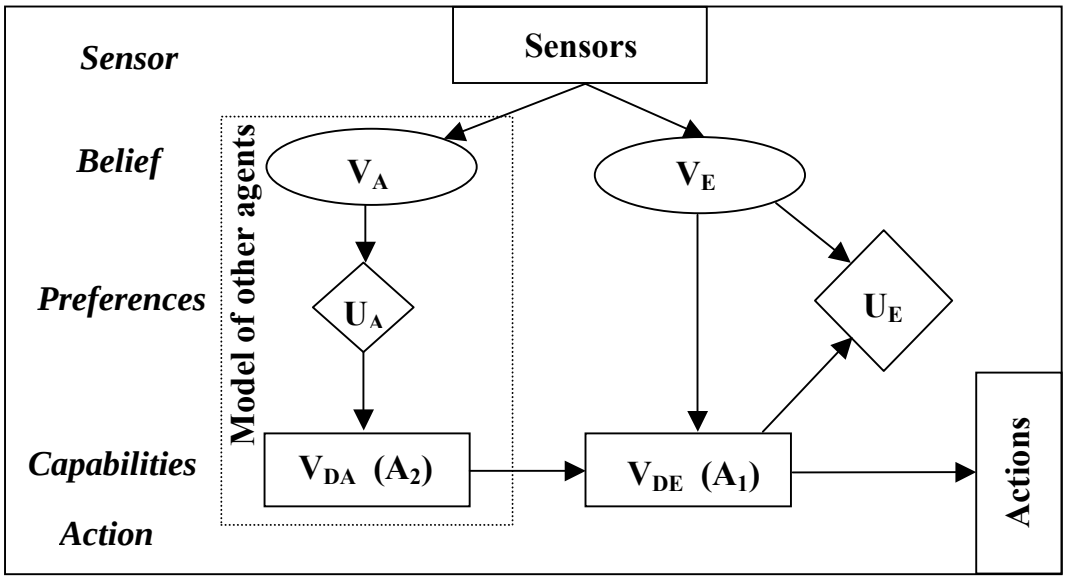


Figure 5.1. The structure of an intelligent agent.

After establishing the world model and the utility function, the agent needs to take an optimal action according to the *principle of maximum expected utility* (PMEU) [54]. The PMEU lets the agents choose the best action from its set of action (A_1), given the belief about the environment (V_E), and other agents' expected behavior (A_2). Formally, it can be expressed as

$$\max_{a_{1i}} U_E = \max_{a_{1i}} f(V_E, A_1, A_2) \tag{5.1}$$

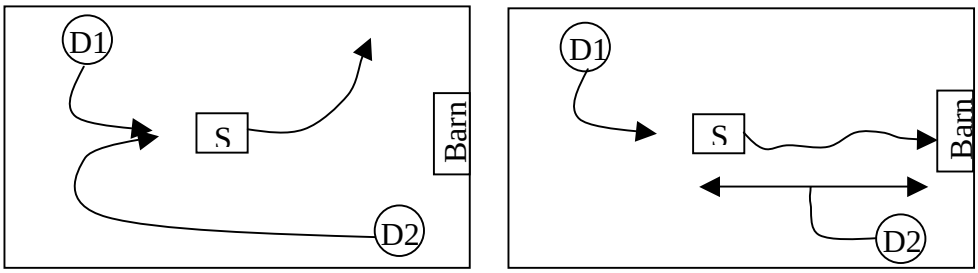
where $V_E = \{X_1, X_2, \ldots, X_n\}$, the variables X_i are the nodes of the Bayesian network V_E, $A_1 = \{a_{11}, a_{12}, \ldots, a_{1k}\}$ is the action set of the agent, $A_2 = \{a_{21}, a_{22}, \ldots, a_{2l}\}$ is the expected action set of the other agents. Therefore, an agent takes its actions after evaluating the environment and the other agents. This property will help to obtain self-organization ability of the system. Each agent first check to see if other agents are performing task before it takes its actions to perform the task.

5.2 Multi-agent self-organizing system.

In the previous section, the structure of an agent is presented. This section will examine the learning problem when we have more than one agent. The agent described in the previous section is specifically designed for multi-agent systems. In a multi-agent environment, coordination requires an agent to recognize the current status and to model the actions of the other agents to decide on its own next behavior [8]. That's why agents model other agents as well as the environment. A computational difficulty may arise if the number of agents is large in the system because agents model the internal structure of other agents in their network. The Bayesian network in the agent may become so large that the calculation of the conditional probabilities might become difficult. The agents

are independent but they take their actions by considering the other agents. Thus, agents take their actions together in coordination. Formally speaking, the agent's utility function U_E depends on the expected actions of other agents (A_1), see Equation (5.1).

We can explain this ability with an example. Suppose we have two dogs and a sheep, as in the sheepdog problem. Dogs are our agents and their goal is to put the sheep into a barn. Dogs will explore the environment and they will model the environment. In this case, the environment contains another dog, a sheep, and a barn. First, the dogs will probably locate the sheep. Then, they will make movements to direct the sheep into the barn. If the dogs do not consider (model) each other, they might not be able to put the sheep into the barn since one's action might hinder the other's action. Thus, they need to cooperate and make movements together. If each dog learns the model of the other dog, then they can make movements together to put the sheep into the barn. If there is no coordination, both dogs will probably go behind the sheep and direct it into the barn. If there is coordination between the dogs, while one of them goes behind the sheep, the other may move back and forth so that the sheep will not escape as shown in Figure 5.2.



(a) (b)

Figure 5.2. Multi –agent behavior without coordination (a) and with coordination (b).

A multi-agent self-organization system with two agents can be seen in Figure 5.3. The multi-agent system is designed by using the agents, shown in Figure 5.1.

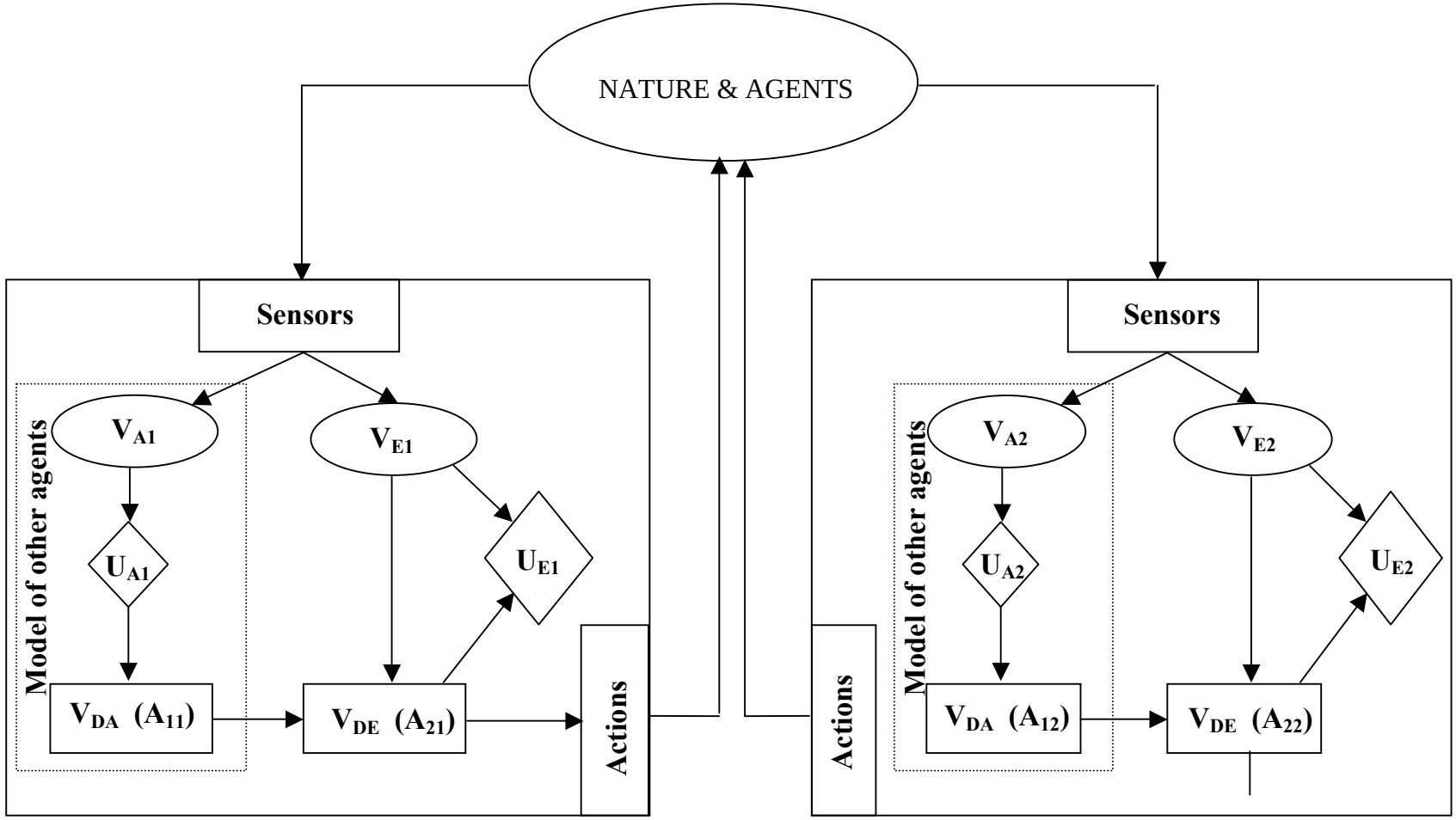


Figure 5.3. Multi-agent self-organizing scheme with two agents.

In summary, agents will fire actions to change the environment as well as to organize themselves. Self-organization will happen eventually because each agent takes its actions considering other agents' behaviors in the environment. The simulation of the dog and sheep problem presented the results supporting that the self-organization and the learning ability of the proposed intelligent agent design. This property will make our system a multi-agent self-organizing system. In the proposed learning system, an agent learns the environment using the sensory data, and modifying its world model (Bayesian Network) accordingly. Then, an agent calculates the expected state of the environment using the world model and creates actions to change the environment. Thus, the learning structure is bi-directional because the agent interacts with nature and the world model in both directions.

5.3 Bi-directional learning

As stated earlier, bi-directionality is the most important feature of an intelligent learning system because it combines the supervised learning method and unsupervised learning method and facilitates them at the same time. That is why a Bayesian network is chosen to construct the learning system. Figure 5.4 shows the learning model of the proposed system. The proposed system has four directed edges among nature, the learning system, and the world model: evidence, action, adaptation, and expectation.

The learning system collects evidence through sensors. Then, it creates optimal actions to change the environment according to the objective (utility). These two steps are represented by *Evidence* and *Action* edges in Figure 5.4. On the other hand, the learning system adapts the world model (Bayesian network) using the evidence from the

environment. In other words, adaptation is the parameterization of the BN utilizing the evidence. Then, the learning system calculates the expected state of the environment using the world model. Last two steps are represented by *Adaptation* and *Expectation* edges in the Figure 5.4. Evidence and action edges represent unsupervised learning while adaptation and expectation edges represent supervised learning. This justifies that the proposed learning system is bi-directional since supervised and unsupervised learning schemes are employed simultaneously.

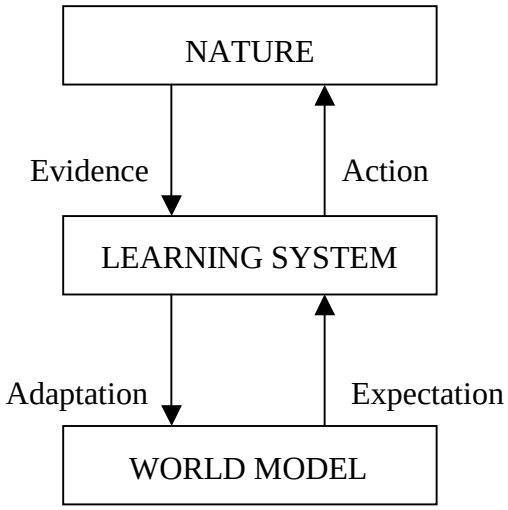


Figure 5.4. The learning model of the proposed system.

The learning system collects evidence through sensors. Then, it creates optimal actions to change the environment according to the objective (utility). On the other hand, the learning system adapts the world model (Bayesian network) using the evidence. Then, it calculates the expected state of the environment using the world model. Adaptation is the parameterization of the BN utilizing the evidence. Evidence and action edges represent unsupervised learning while adaptation and expectation edges represent supervised learning. This justifies that the proposed learning system is bi-directional since it combines supervised and unsupervised learning schemes.

5.4. System representation of the decision-theoretic intelligent agent system

The decision-theoretic intelligent agent system has adaptive learning ability with feedback from the environment. The agent starts with a limited knowledge of the plant (environment), then it explores (samples) the plant to learn the plant's parameters. After it learns about the plant, it takes its actions accordingly. The agent first estimates the plant's behavior using the previous observation, then takes its action according to the estimation. The plant, then, responds to the agent's action with an output. The output of the plant in this stage is used as feedback to update the plant parameters in the predictor (BN). Figure 5.5 shows the decision theoretic-intelligent agent learning system in a block diagram.

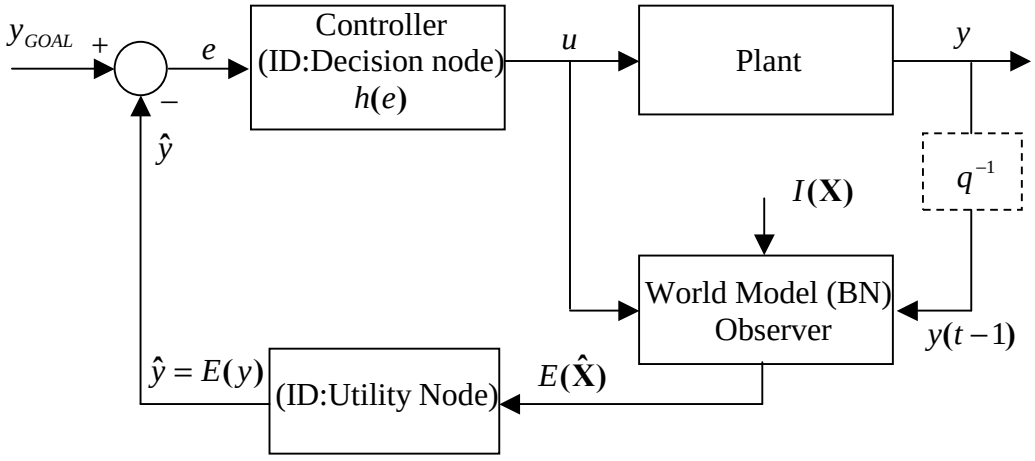


Figure 5.5. System Block representation of the intelligent agent system.

In Figure 5.5, $I(\mathbf{X})$ represents the initial state of the plant, $E(\hat{\mathbf{X}})$ is the expected value of the state, $E(y)$ is the expected value of the plant output, and y_{GOAL} is the desired plant (system) output. The symbol q^{-1} represents one unit delay. The controller (ID) applies controls to the plant to provide a certain plant output because the controller creates the control according to the error between the expected value of the plant output

and the reference. The reference is the desired output to be provided by the plant. The observer (BN) models the plant by using the plant's input/outputs. After a control is applied to the plant, the plant output is used in the next step to update the plant model. Thus, there is a time delay between the control and the output of the plant. The controller creates the control using *a priori* knowledge about the plant (environment).

The decision theoretic intelligent agent system (DTAS) has potential use in feedback control and adaptive control because it uses the plant's output as a feedback and modifies the controller and the observer accordingly. The first part of a DTAS establishes the feedback control; the second part establishes the adaptive control part. The following section presents an analysis to show the feedback and adaptive control ability of the DTAS.

5.4.1 Feedback Control

In the literature, there are two main types of feedback control, namely *output feedback* and *state feedback* [56]. Output feedback is performed by a path (loop) from the output back to the controller as shown in Figure 5.6.

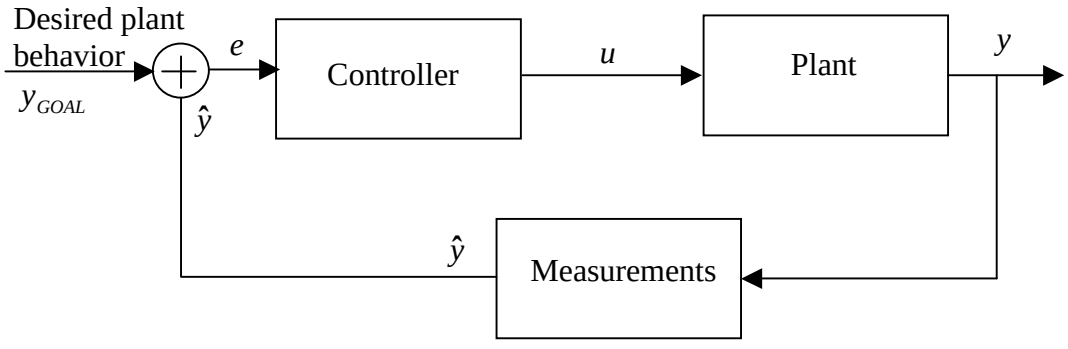


Figure 5.6. Output feedback control

The equations for the system in Figure 5.6 can be given as:

$$e = \hat{y} - y_{GOAL} \tag{5.2}$$

$$u = f(e) \tag{5.3}$$

$$y = g(u) \tag{5.4}$$

Now, let us compare the system equations in the feedback control system and the decision-theoretic intelligent agent system. In the DTAS, the output of the plant, *y*, also depends on the control input, *u*. Let us compare the control signal *u* in both systems.

$$u_{DTAS} = h(e) \Leftrightarrow u_{FEEDBACK} = f(e) \tag{5.5}$$

If we choose the functions *h* and *f* to be equal, then the controllers will give the same control *u* with the same error *e*. Let us compare the errors in both systems. In the DTAS, the error is the difference between the desired output and the expected value of the plant output provided by the predictor. This is very similar to the feedback control system but the expected value of the plant output replaces the measured plant output. These two values are equivalent only if the predictor estimates the output of the plant well enough. In the DTAS, it is shown that the predictor estimates the plant output well enough when there is sufficient data from the plant's input/output. Therefore, the expected value in the DTAS is equivalent to the measured value of the plant output in a feedback control system. The following equations summarize the discussion.

$$e = y_{GOAL} - E(y) \tag{5.6}$$

$$E(y) \cong \hat{y} \tag{5.7}$$

$$e = y_{GOAL} - \hat{y} \tag{5.8}$$

From Equations (5.6), (5.7), and (5.8), we may conclude that the DTAS exhibits feedback control properties.

Another type of feedback control is state feedback control. In state feedback control, the state variables are sensed and fed back to the input through appropriate gains [56]. If there is direct access to the state variables, the state variables can be easily measured and fed back to the input. If there is no direct access to the state variables, then an observer may be employed to perform the estimation of the state variables. Figure 5.7 illustrates a state feedback control system with an observer.

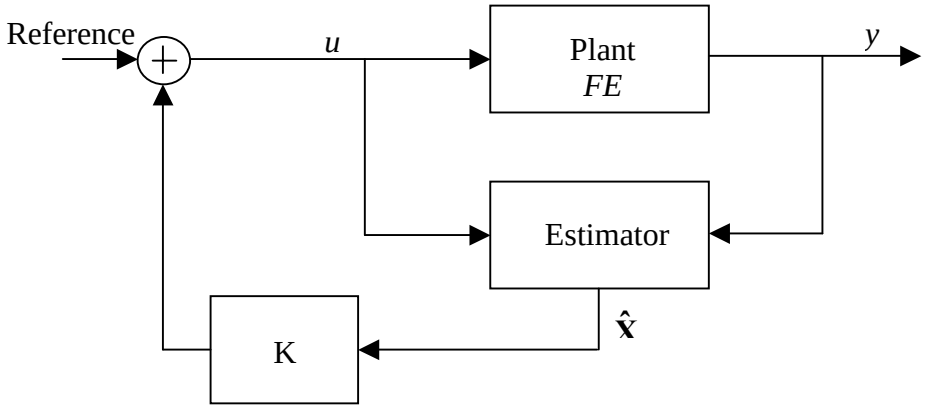


Figure 5.7. A control system with the state feedback.

In Figure 5.7, the block denoted by *FE* is the plant. The estimator predicts the state variables of the plant. The estimated state variables are fed to the input with a gain *K*. Then, the control signal becomes the following:

$$u = r + \mathbf{K}\hat{\mathbf{X}} \tag{5.9}$$

Thus, the control is a function of estimated state variables and the reference input. Let us compare the controls in both systems. In the DTAS, the control is defined as

$$u = f(e) \tag{5.10}$$

where $e = y_{GOAL} - \hat{y}$. The term $\hat{y}$ represents the estimated output of the plant. The term $\hat{y}$ is a function of the estimated state variables because it is calculated by the utility function of the system. Therefore, we can represent $\hat{y}$ with the following equation.

$$\hat{y} = \hat{\mathbf{C}}\hat{\mathbf{X}} \tag{5.11}$$

where the vector $\hat{\mathbf{X}}$ is the estimated state vector and the matrix $\hat{\mathbf{C}}$ is the transformation matrix between the states and the output. Thus, the control can be rewritten as follows:

$$u = f(y_{GOAL} - \hat{y}) \tag{5.12}$$

$$u(X) = f(y_{GOAL} - \hat{C}\hat{X}) \tag{5.13}$$

Let us assume that the function *f* is a linear function with the following form.

$$f(x) = A \cdot x \tag{5.14}$$

$$u = A \cdot (y_{GOAL} - \hat{\mathbf{C}}\hat{\mathbf{X}}) = A \cdot y_{GOAL} - A \cdot \hat{\mathbf{C}}\hat{\mathbf{X}} \tag{5.15}$$

Let $\mathbf{K} = -A \cdot \hat{\mathbf{C}}$, and $r = A \cdot y_{GOAL}$, then the control becomes

$$u = r + \mathbf{K} \cdot \hat{\mathbf{X}} \tag{5.16}$$

As seen in Equation (5.16), the control signal in the DTAS can be interpreted as the control signal in the state feedback control. This concludes the analysis of how the DTAS corresponds to a feedback control system. It can be concluded that the DTAS will have the inherent advantages of feedback control. The following section investigates the adaptive control capabilities of the DTAS.

5.4.2 Adaptive Control

The term *Adaptive Control* covers a set of methods that provide a systematic approach for automatic adjustment of the controllers in real time, in order to achieve or to maintain a

desired level of performance of the control system when the parameters of the plant dynamic model are unknown and/or change in time [57]. A block diagram presenting a basic configuration of an adaptive control system is shown in Figure 5.8.

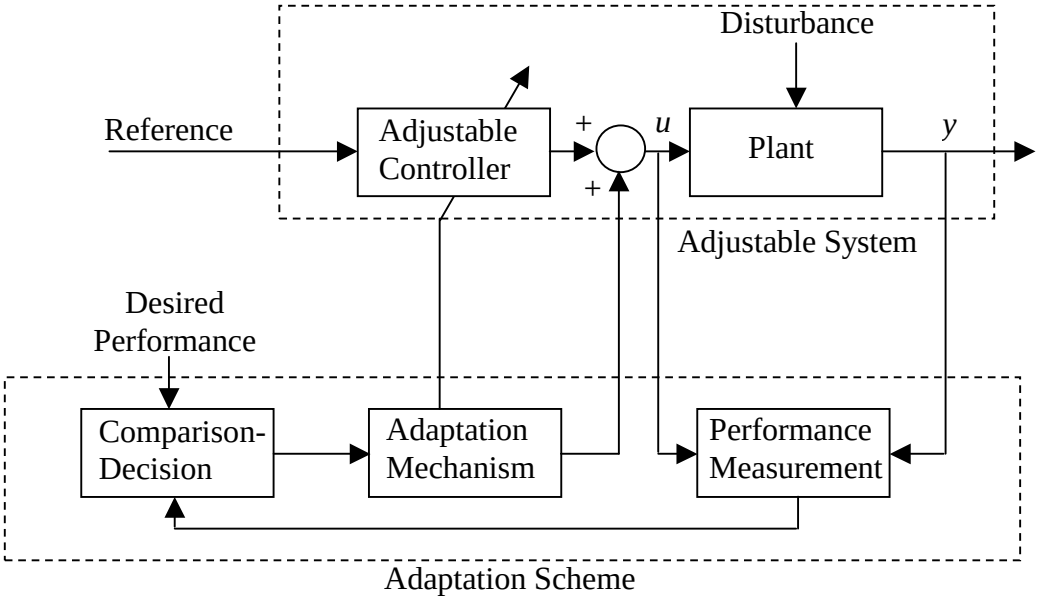


Figure 5.8. A basic adaptive control system.

The following definition provides an adaptive control system given in Figure 5.8.

Definition 5.4.1: An *adaptive control system* calculates a certain performance index (IP) of the control system using the measured inputs, the states, the outputs, and the known disturbances. From the comparison of the performance index and a set of given ones, the adaptation mechanism modifies the parameters of the adjustable controller and/or generates an auxiliary control signal in order to maintain the performance index of the control system close to the set of given ones (i.e., within the set of acceptable ones) [57]

An adaptive control system will monitor the performance of the system in the presence of parameter disturbances in addition to a feedback controller with adjustable

parameters acting as a supplementary loop upon the adjustable parameters of the controller.

There are three types of adaptive control schemes in the literature: open loop adaptive control, direct adaptive control, and indirect adaptive control [57]. In open loop adaptive control, the adaptation mechanism is a simple look-up table stored in the computer that gives the controller parameters for a given set of environment measurements. In the literature, this is also called *gain-scheduling*.

Direct adaptive control is based on the observation that the difference between the output of the plant and the output of the *reference model* (called plant-model error) is a measure of the difference between the real and the *desired performance*. The reference model is a realization of the system with desired performance. This information is used by the *adaptation mechanism* (called *parameter adaptation*) to directly adjust the parameters of the controller in real-time in order to force (asymptotically) the plant model-error to zero. This scheme corresponds to the use of a general concept called *Model Reference Adaptive Systems* (*MRAS*) for the purpose of control [58]. The indirect adaptive control was originally introduced by Kalman [59].

In an indirect adaptive control system, shown in Figure 5.9, the basic idea is that a suitable controller can be designed on line if a model of the plant is estimated on line from the available input-output measurements. The scheme is called *indirect* because the adaptation of the controller parameters is performed in two stages:

1. On-line estimation of the plant parameters (e.g. Bayesian network construction)
2. On-line computation of the controller parameters based on the current estimated plant model (e.g. Influence Diagrams-making decisions)

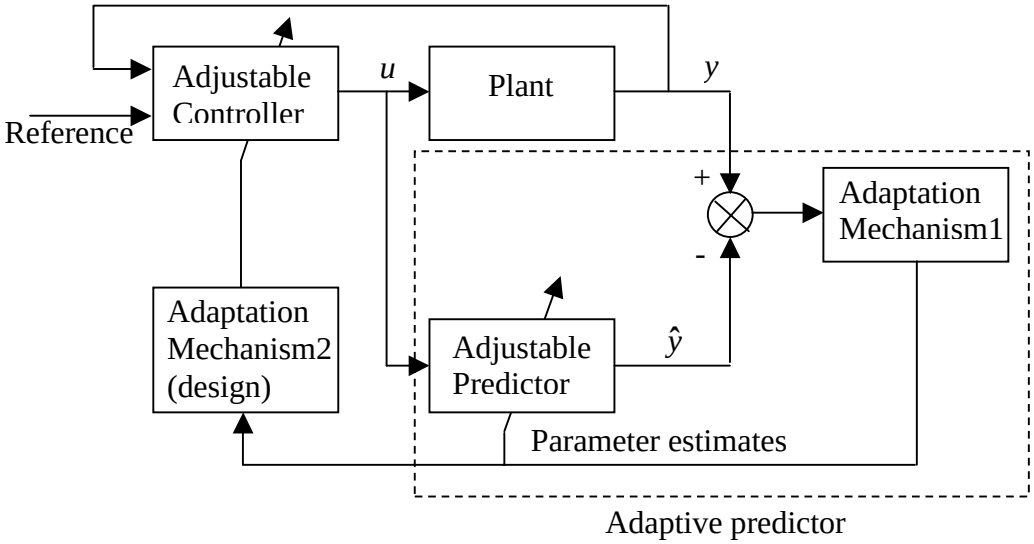


Figure 5.9. Indirect adaptive control system

The main goal is to create an *adjustable predictor* for the plant output and compare the predicted output with the measured output. The error between the plant output and the predicted output (called *prediction error* or *plant-model error*) is used by a *parameter adaptation algorithm* which at each sampling instant will adjust the parameters of the adjustable predictor in order to minimize the prediction error in the sense of a certain criterion.

In [57], there are two options given to effectively implement an indirect adaptive control strategy. The choice is related to a certain extent to the ratio between the computation time and the sampling period.

Strategy 1

1. Sample the plant output.
2. Update the plant model parameters.
3. Compute the controller parameters based on the new plant model parameter estimates.
4. Compute the control signal.
5. Apply the control signal.
6. Wait for the next sample.

In this strategy, there is a delay between $u(t)$ and $y(t)$ that will depend on the time required to achieve (2) and (3). This delay should be smaller than the sampling period.

Strategy 2

1. Sample the plant output.
2. Compute the control signal based on the controller parameters computed during the previous sampling periods.
3. Apply the control signal.
4. Update the plant model parameters.
5. Compute the controller parameters based on the new plant model parameter estimates.
6. Wait for the next sample.

In the second strategy, the delay between $u(t)$ and $y(t)$ is smaller than in the previous case. In this strategy, a *priori* parameter estimation is performed since we apply the control without updating the plant parameters [57].

In the above paragraphs, a general definition of an adaptive control system is provided. A greater importance is given to indirect adaptive control systems because the decision-theoretic agent system (DTAS) has the properties of an indirect adaptive control system. The DTAS has the same steps as the indirect adaptive control system. Additionally, the learning strategy in DTAS is very similar to the second strategy of the indirect adaptive control system.

The first step, the on-line estimation of the plant model parameters, is performed by structuring a Bayesian network and calculating its parameters in the DTAS. As stated in Chapter 4, the online Bayesian network learning is performed to model the plant. The second step, the online computation of the controller parameters, is performed by a decision system (influence diagrams).

As shown in Figure 5.9, there are two adaptation mechanisms in the indirect adaptive control. The first adaptation mechanism corresponds to the online Bayesian network learning in the DTAS. The second adaptation mechanism corresponds to the utility node in the influence diagram part of the decision-theoretic intelligent agent because it determines which action will be fired in the decision node. The adjustable predictor corresponds to the Bayesian network in the DTAS. Finally, the adjustable controller corresponds to the decision nodes in the influence diagram in the DTAS.

Now, the indirect adaptive control system can be redrawn by using the decision-theoretic intelligent agent components, shown in Figure 5.10.

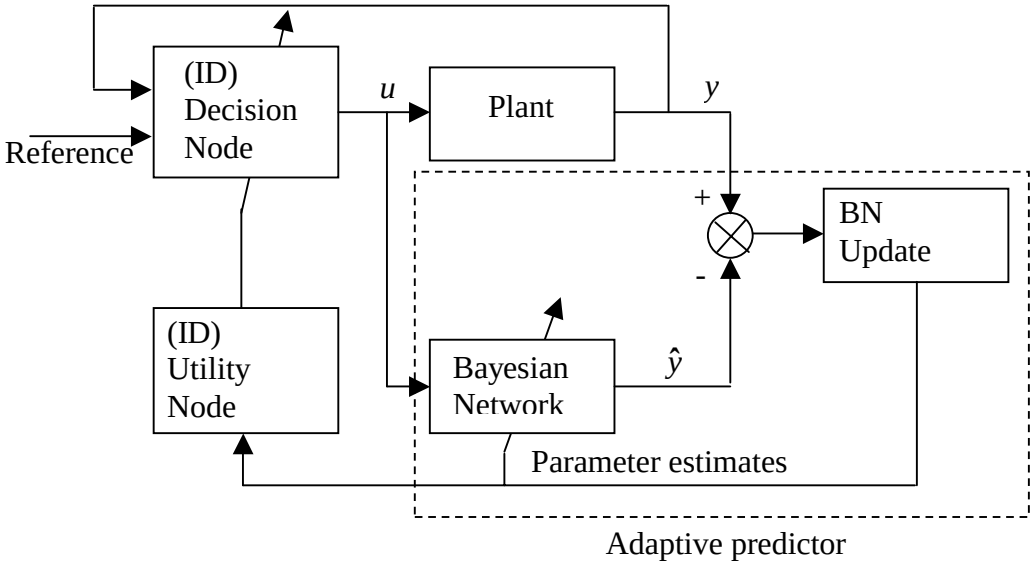


Figure 5.10. Indirect adaptive control representation of the DTAS.

Consequently, the online Bayesian learning determines the plant model structure and parameter estimation; and, the influence diagram determines the controller parameters. Therefore, it can be concluded that the decision-theoretic intelligent agent system implements an indirect adaptive control system.

CHAPTER 6

IntelliAgent Software

This section explores the software created to perform experimental simulation for the decision-theoretic intelligent agent. The IntelliAgent software is created under Visual C++ with for Microsoft Windows NT®. The software is capable of creating intelligent agents by employing Bayesian network and influence diagram structures. As explained in previous chapters, the Bayesian network learning is an online learning since agents continue to learn during their operations.

The IntelliAgent software is presented in three main parts, the user manual, tutorials on Bayesian network creation and knowledge discovery, and the class definitions. The class definitions are presented in Appendix A. The Visual C++ code and the application software is available for the readers on: http://armyant.ee.vt.edu/IntelliAgent. One can contact the author by email, sferat@vt.edu, for further information about the software.

6.1 The user manual for IntelliAgent software

The IntelliAgent software is a single document interface (SDI) visual C++ program. The Microsoft Foundation classes are intensively used to create the software. The software is a Windows application with a menu, a toolbar, and status bar, shown in Figure 6.1. The user manual starts by explaining the menus available. Section 6.1.2 explains the toolbar and the status bar operations. After exploring the menus and the toolbar, the dialog boxes used throughout the program are explored in Section 6.1.3.

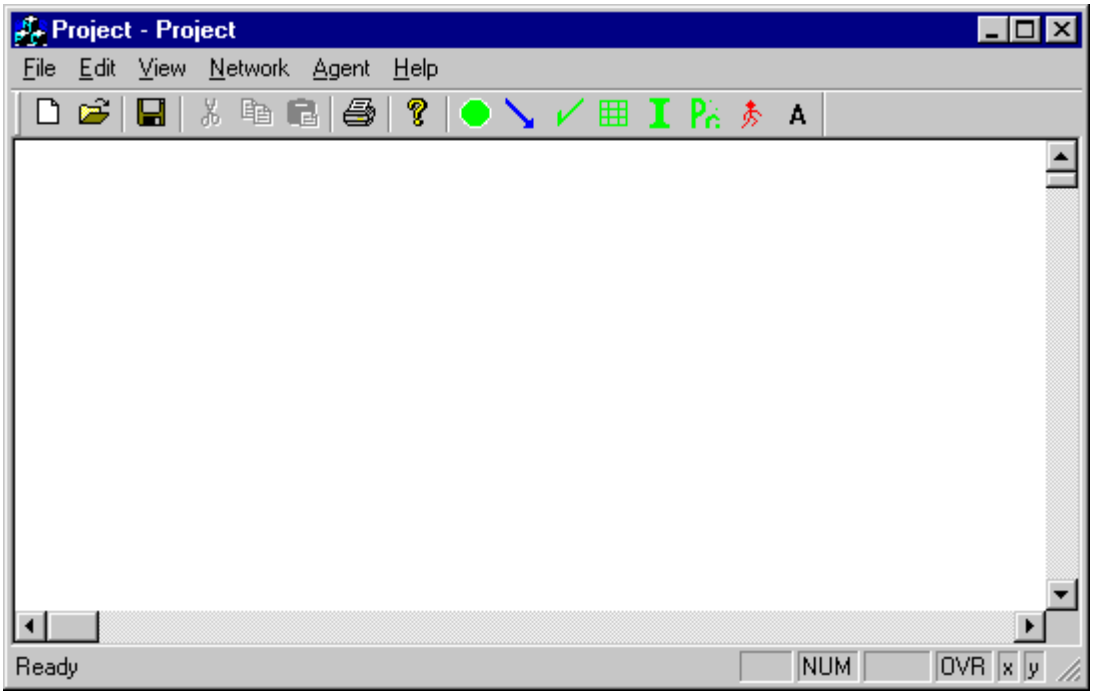


Figure 6.1. The IntelliAgent software (screen shot of the program)

6.1.1 Menus

In the IntelliAgent program, there are five menu items, File, Edit, View, Network, Agent, and Help. The menu items File, Edit, View, and Help are standard Windows application menus. Functions for these menu items are modified for the use of IntelliAgent software. For example, the File menu functions are modified to open and save the files that are specifically defined for the IntelliAgent software. The Network menu item is created for the Bayesian network operations such as network creation, network update and network edit. The Agent menu performs the creation of intelligent agents and the intelligent agent simulation. The following paragraphs explore the menu items with their functionalities.

File

In File menu, there are eight submenus, New, Open, Save, Save As, Print, Print View, Print Setup, and Exit. Figure 6.2 shows the submenus in the File menu.

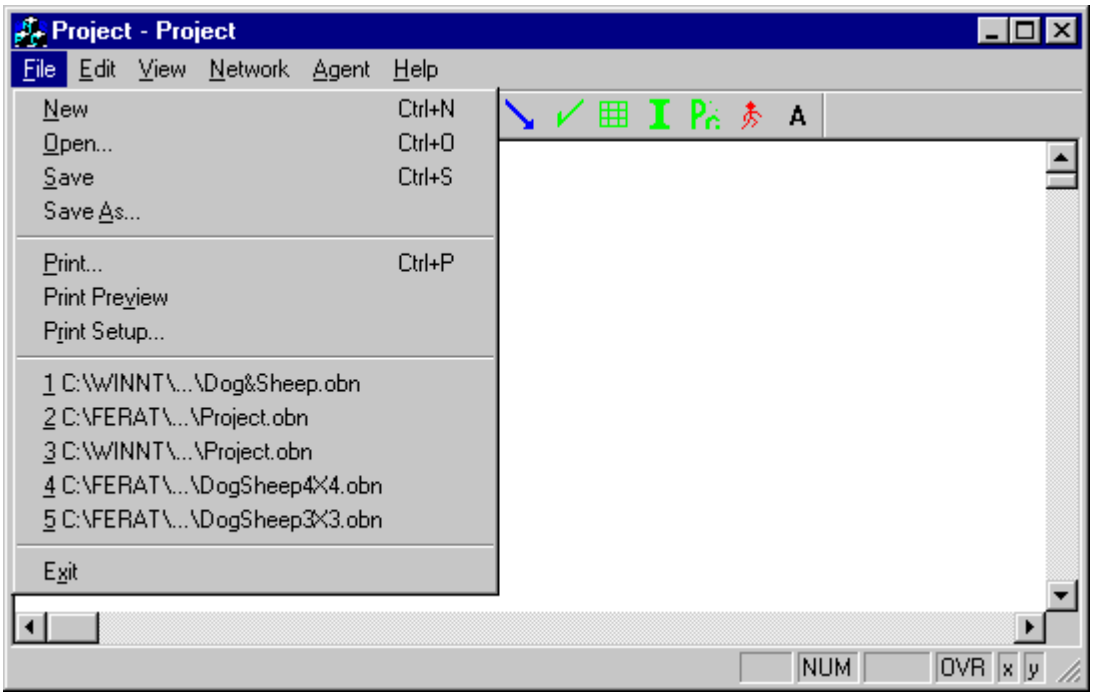


Figure 6.2. The File menu.

The New submenu creates a new online Bayesian network file in "obn" format. The "obn" is online Bayesian network format created for the intelligent agent software. In the format, there are nodes, arcs, and the dependencies in the network. The user chooses this submenu whenever he/she needs to create a new network.

The Open submenu opens a "obn" network that is saved/created previously. The user needs this submenu when there is a need to update or change the previously created network. When the user chooses this submenu, a dialog box appears on the screen, shown in Figure 6.3. This dialog box is a standard dialog box used in Windows

programming. Functions for the dialog box are built-in functions in Microsoft visual C++ but they are edited to be able to open a "obn" file.

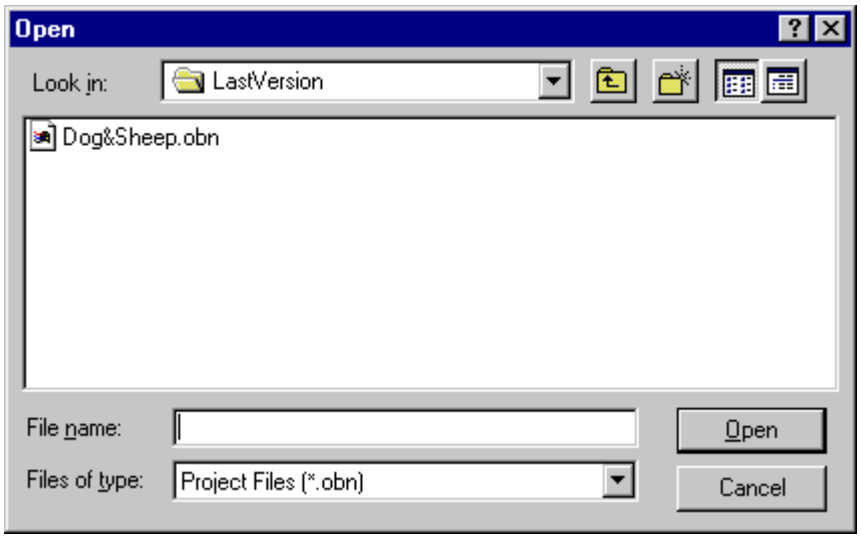


Figure 6.3. Dialog box for the "Open" submenu in File menu.

The Save submenu is to save the "obn" files for future uses. This submenu also creates a standard dialog box, shown in Figure 6.4, if a network is not saved before. If a network is saved before, choosing Save submenu saves the file again without showing any dialog box. The functions in the dialog box are edited to be able to save the online Bayesian networks as an "obn" file. Nodes, arcs, dependencies in the network, and the database are saved to the file. The program asks the user if it should save the newly explored cases, shown in Figure 6.5. The Save As submenu is almost the same as the Save submenu. The only difference is that the user can choose the file type before saving. In Save submenu, the file format is set to "obn" whereas it can be different in Save As.

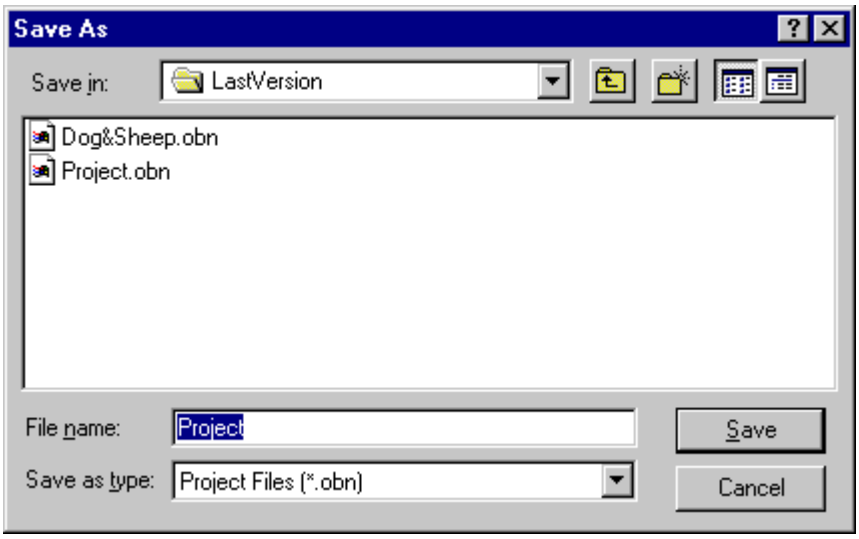


Figure 6.4. Dialog box for "Save" and "Save As" submenus in File menu.

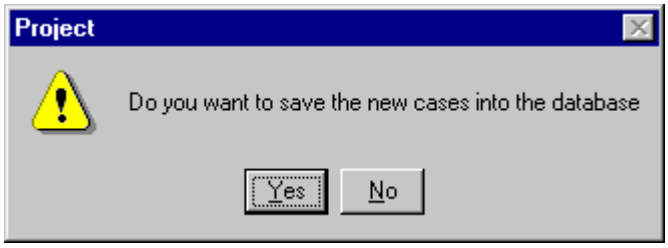


Figure 6.5. Message box to choose saving the new cases into the database.

Print, Print Preview, and Print Setup are printing related submenus. In the IntlliAgent, the users are able to print the networks they create. Print Preview and Print Setup work as in any standard Windows application program. Finally, the Exit submenu is to quit the software. The software asks the user whether to save the network before it quits.

Edit

This menu is kept for cutting, copying and pasting the network components. There are four submenus in the Edit menu; Undo, Cut, Copy, and Paste. None of the submenus are fully functional even though the software has adding and removing functions internally. In the future, these submenus can be made operational by connecting them to the functions in the software.

View

In View menu, there are two submenus; Toolbar and Status Bar. The user can check these submenus by mouse operations. Depending on they are checked or not, the toolbar and the status bar appear on the program window or not. Figure 6.6 show the submenus of View menu in the software.

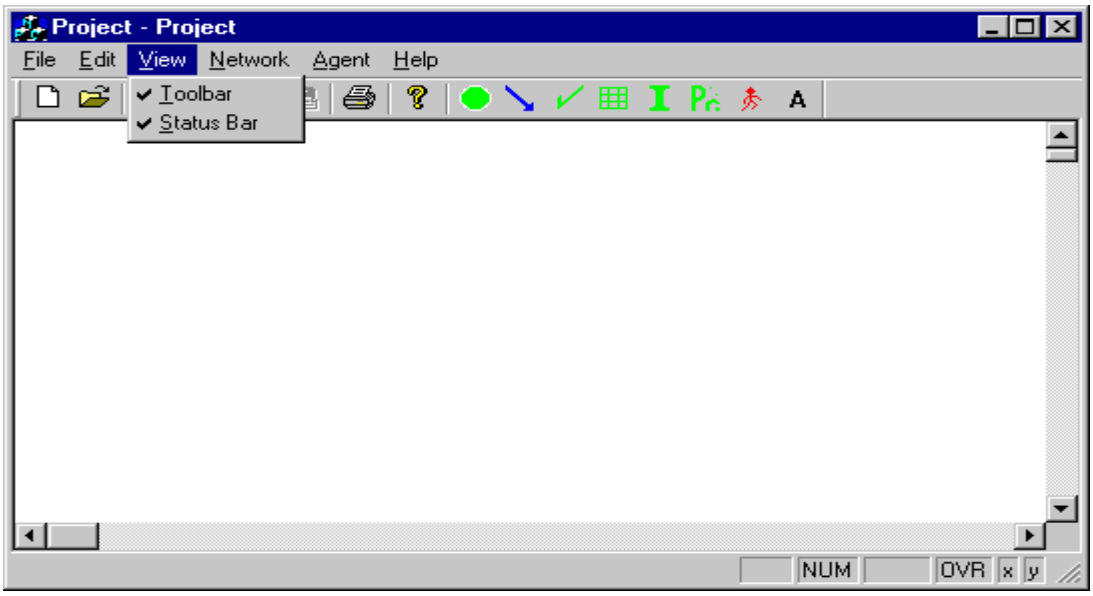


Figure 6.6. The View menu

Network

The Network menu contains core operations in creating online Bayesian network. There are six submenus in Network menu: Node, Arc, Update, Parameters, Load, and Create. Figure 6.7 illustrates the Network menu on the IntelliAgent software.

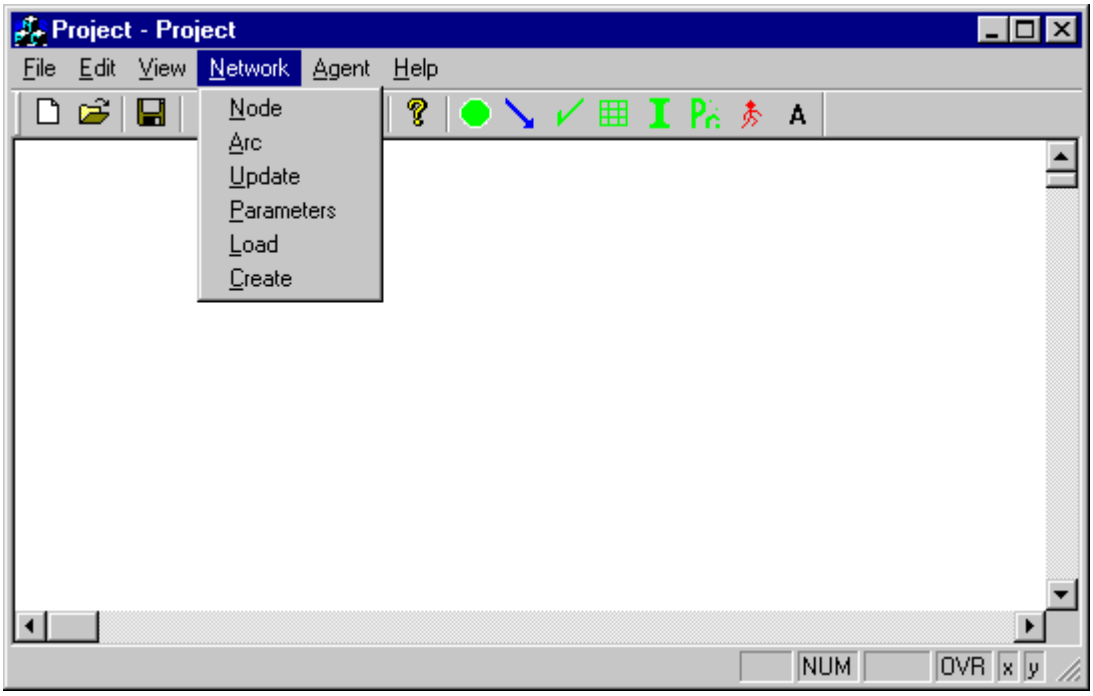


Figure 6.7. The Network menu.

There are three groups of submenus under this menu: manual network creation submenus, presentation submenus, and automatic network creation submenus. The submenus Node and Arc are used to create the Bayesian network manually. The user can create nodes and arcs between the nodes by simple mouse drag and drop operations. After creating the network the user can apply inference by using Update submenu. Parameter submenu is in the presentation group. It displays the parameters of a node in a

network. The submenus Load and Create let the user load a database and construct the Bayesian network using the database.

Node submenu lets the user create nodes of a Bayesian network. To create a node, the user chooses node submenu in the Network menu. Then, the user moves the mouse to a location where the node is going to be created. While the left mouse button is kept pressed, the user draws an ellipsoid on the specified display area by moving the mouse. When the ellipsoid is established, the user releases the left mouse button. With the release of the left mouse button, the software creates a node with the default parameters. There are two states with the values 0.5 by default. Node name is set to NodeX, where X shows the order of the node. A conditional probability table with two rows and one column is filled with 0.5. Figure 6.8 illustrates two nodes created by the user manually.

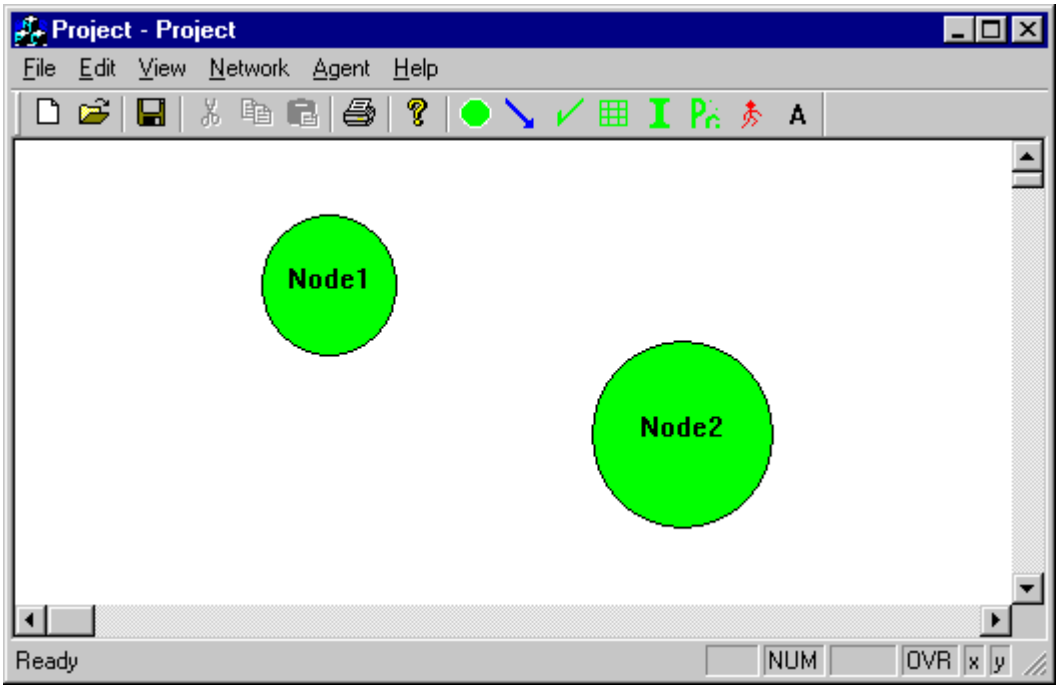


Figure 6.8. Creation of network nodes by mouse operations.

After creating the nodes, the user can create arcs between the nodes by mouse operations. The user, first, chooses Arc submenu in the Network menu. Second, the user moves the mouse over a node that the arc is going to start from. Then, while the left mouse button is pressed, the user draws an arc between the nodes by moving the mouse on the node that the arc is going to point. When the user releases the left mouse button, the software draws an arrow between the two nodes. The user can start drawing in any part of the node because the software adjusts the starting and ending points of the arc according to the nodes' relative positions. Figure 6.9 illustrates a network with two nodes and an arc.

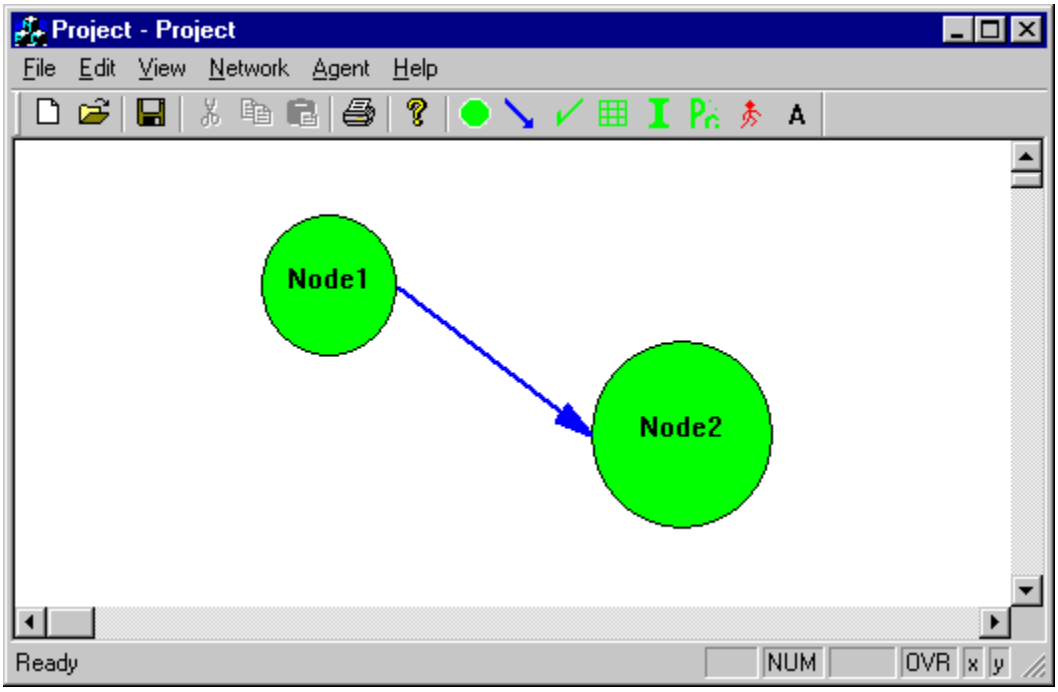


Figure 6.9. Creation of an arc between the nodes by mouse operations.

The parameters submenu takes care of presenting and changing the nodes' parameters. When the user wants to see the parameters of a node, first, the node has to be selected by

clicking the left mouse button on the node. Then, the user can choose the Parameter submenu in the Network menu. After the submenu is chosen, a dialog box appear on the screen as shown in Figure 6.10.

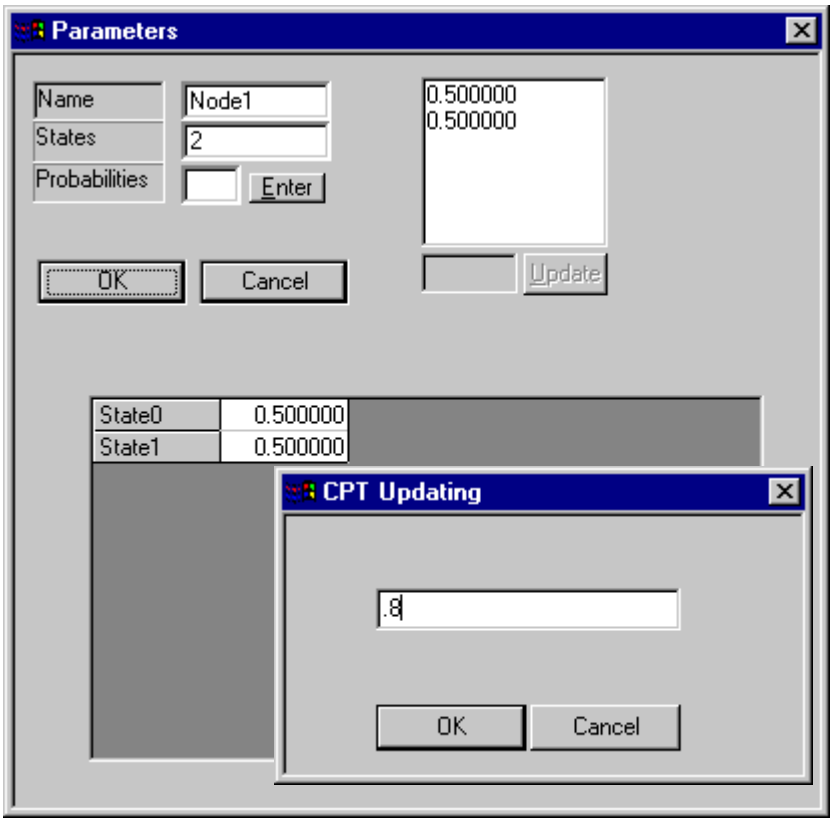


Figure 6.10. Dialog boxes for presenting and changing node attributes.

On the dialog box shown above, the user can change the name, the number of state, the state values (probabilities), and the conditional probability table of the node. The detailed description of the dialog boxes is provided in Section 6.1.3. To change the conditional probability table of a node, the user needs to move the mouse on to a desired element of the table and double clicks the left mouse button. Then, a dialog box, shown in Figure 6.10, appears on the parameter dialog box. The user needs to enter the new

probability value into the CPT updating dialog box. Finally, the user clicks "OK" button on the CPT updating dialog box to put the new value into the CPT.

After creating nodes and arcs in a Bayesian network, the user can change the node parameters by dialog boxes shown in Figure 6.10. Then, the user can perform inference in the network by activating Update submenu. This button updates the network parameters if evidence is entered to a node or a change has been made on a node. The software performs the inference by employing the technique defined in [1,2].

Let us take the network given in Figure 6.9 and change the CPT of Node2 as in Figure 6.11.

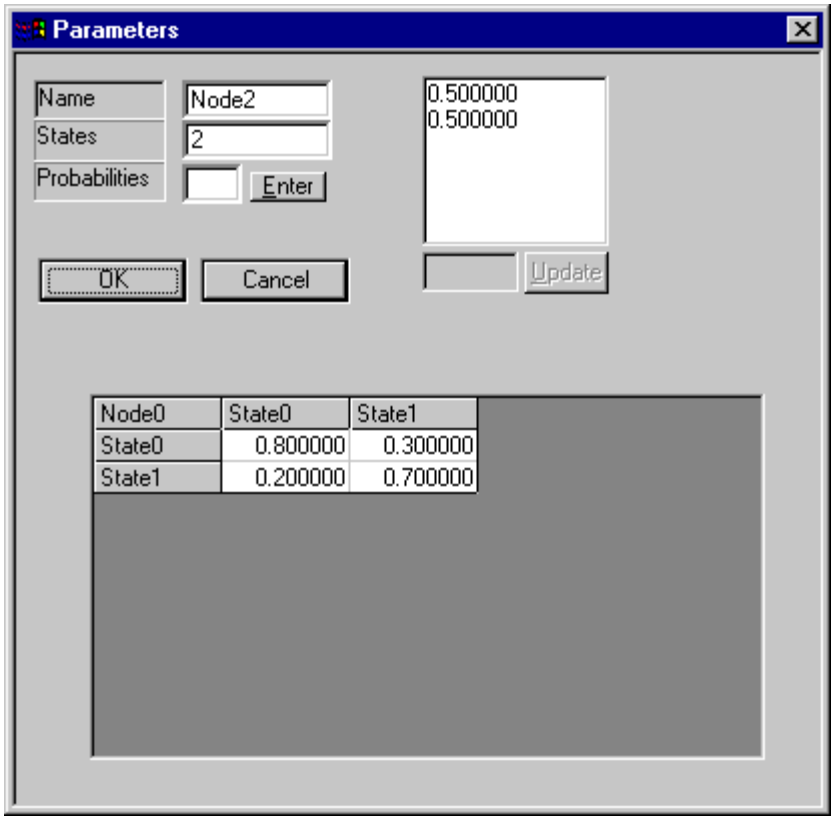


Figure 6.11. Changing the CPT of Node2.

After changing the CPT values in Node2, the user can choose the Update submenu in the Network menu. Choosing the Update submenu let the software calculate the other parameters of the nodes accordingly. After choosing the Update submenu, the user can choose the Parameters submenu to see the new values of Node2, shown in Figure 6.112.

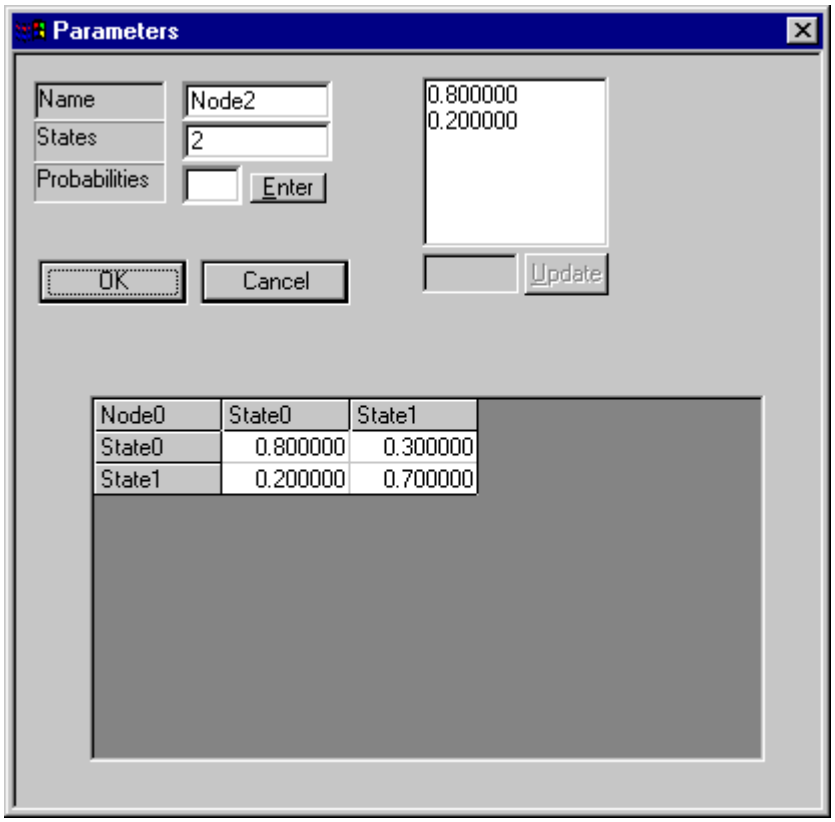


Figure 6.12. Parameters of Node2 after the Update command.

As can be seen above, the probabilities of the node have changed according to the new CPT values. The software checks every node in the network whether they need updating or not. If the user makes changes on a node, the software sets a flag for the node.. The user can change parameters in many nodes. Then, the Update submenu will update all flagged nodes and related nodes. For example, if a parent node is modified,

the Update submenu needs to update the child nodes of the node as well because child nodes are dependent on the parent nodes.

As stated earlier, the user can create the Bayesian network using a database. Load and Create submenus let the user create a Bayesian network from a database. When the user chooses Load, a dialog box appears for loading a file as shown in Figure 6.13.

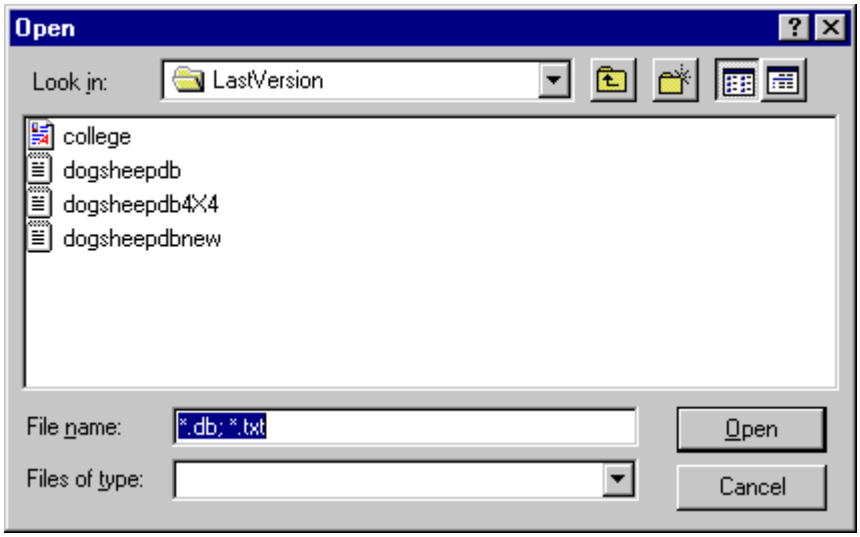


Figure 6.13. Loading a database to automatically construct a Bayesian network.

By double clicking the left mouse button on a database file, the dialog box loads a database into the program. The database file is a text file with a specific format. It could be a plain text file or ".db" file. The extension "db" stands for database and its is a standard Bayesian network database used in the literature [28]. The first line of the database file contains the name of the variables. The rest of the rows in the database are the data cases recorded over time. Entries in a row are delimited by a space. There is a "end of line" character after the last entry in each row. After loading the database file, the software creates nodes by reading the first line. Then, it calculates independent probabilities for the states of each node. Assume that the user has chosen the database

file "college.db". Then, the software creates the nodes and calculates their parameters as shown in Figure 6.14. The software draws the nodes on the screen in a line. The user can move the nodes to the desired places by mouse drag and drop operations.

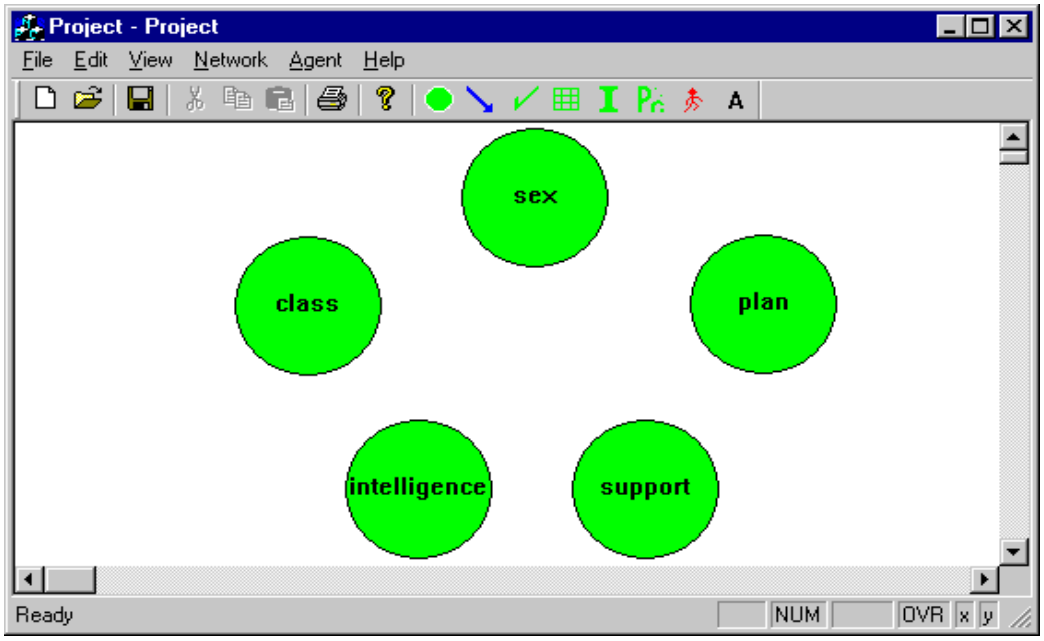


Figure 6.14. Bayesian network nodes created by a database file.

After the software has created the nodes of the network, the user can choose submenu Create in Network menu to construct the Bayesian network automatically. When the user chooses the Create submenu, a dialog box appears on the screen to specify how the network search is going to be performed. Figure 6.15 shows the search dialog box.

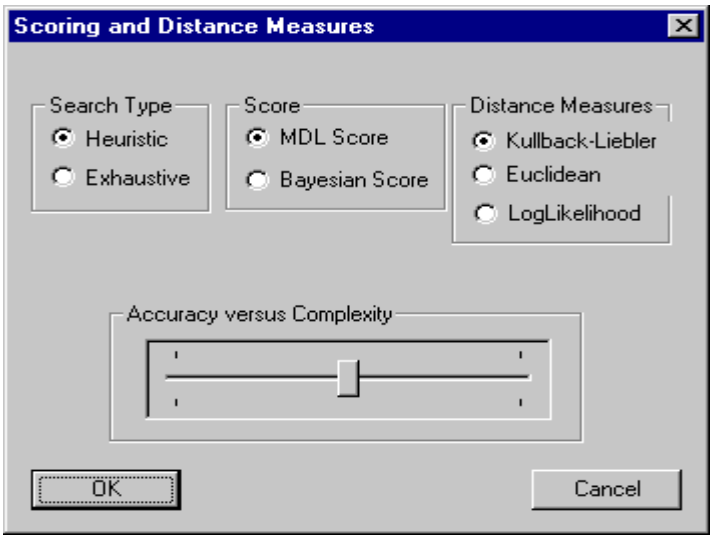


Figure 6.15. Dialog box for specifying the type of network search.

There are two search methods available in IntelliAgent software, heuristic and exhaustive, as stated in Section 4. There are three scoring types, MDL, Bayesian scoring, and Log-Likelihood. In the dialog box, Log-Likelihood is placed in the distance measures group because Log-Likelihood score involves only the distance between the distributions from the database and the network. Bayesian scoring and MDL use both distance measure and complexity of the network. If the user chooses the MDL scoring, a distance measure has to be chosen also. There are two distance measures for MDL scoring, Kullback-Leiber and Euclidean. If Bayesian scoring is chosen, there is no need to specify the distance measure because Bayesian scoring combines distance measure and complexity as stated in Section 4.

Search types, score types and distance measure types are grouped in three sections. The user can click on radio buttons besides the items to specify the search algorithm. For example, for a heuristic search with MDL score and Euclidean distance, the user can click the radio buttons in front of heuristic, Bayesian score and Euclidean. The default

search type is a heuristic search with MDL score and Kullback-Leiber distance measure. Let us assume that the user has chosen the heuristic search with Bayesian scoring. Figure 6.16 shows the resulting Bayesian network.

We have covered the submenus in Network menu. The user can create a Bayesian network either manually or using a database. This part of the IntelliAgent software can be used as knowledge discovery tool. For example, the network shown in Figure 6.16 is created by employing a database. The database includes information about college plans for number of students. The aim is to find out the relationship between the variables and how they affect the decision to go to college. After loading the database, we have searched a network that fits the database. The network shown in Figure 6.16 is a resulting Bayesian network after the search.

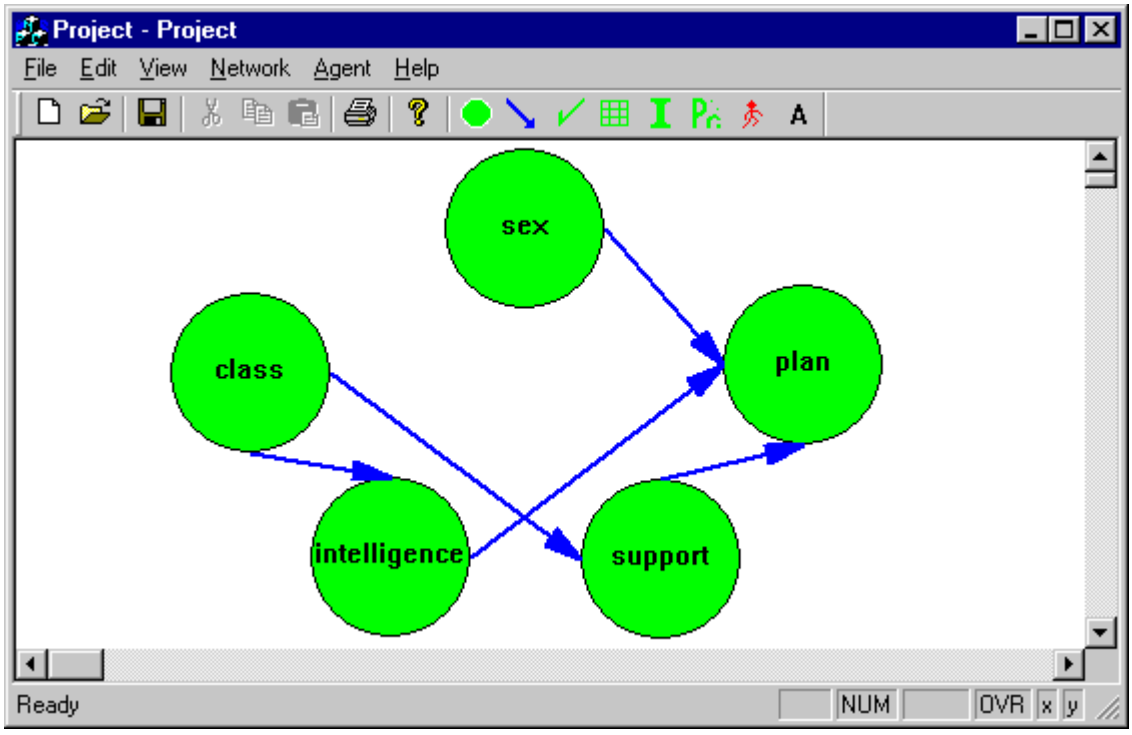


Figure 6.16. A Bayesian network created by a heuristic search with Bayesian scoring.

At this stage, the user can find out the probabilities for the nodes and their relationships. Additionally, by specifying certain variables, the user can find out the probability of making a college plan for a given student. To do that, the user will need to set the variables with specific parameters, then choose Update submenu in Network menu to run the inference to other nodes. Tutorials on inference in Bayesian network and knowledge discovery with Bayesian networks are presented in Section 6.2.

Agent

The Agent menu contains submenus for intelligent agent simulations. There are two submenus in Network menu, Create Agent and Simulate, shown in Figure 6.17.

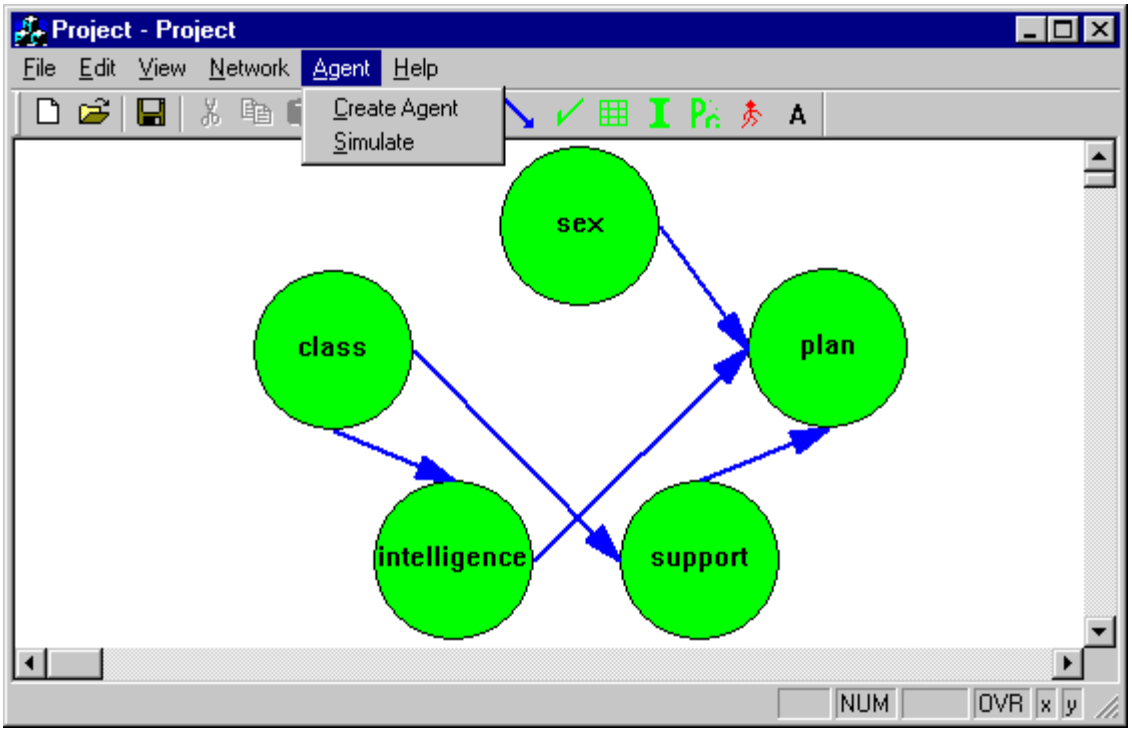


Figure 6.17. The Agent menu in the IntelliAgent software.

Create Agent submenu is designed for creating agents. When the user chooses Create Agent, the software shows a dialog box shown in figure 6.18 to specify agent's parameters. The Create Agent submenu and the Simulate submenu is designed for a specific problem, the Dog & Sheep problem. In the dialog box, the user can enter the name of the agent and its *X* and *Y* coordinates. The dialog box has also designed to specify what type of simulation will be run. The user can choose step by step or continuous simulation by pressing Step or Continuous button, respectively.

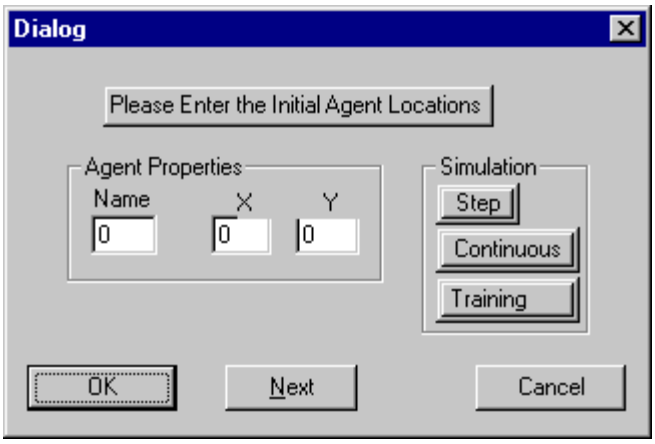


Figure 6.18. Dialog box for agent creation and simulation attributes.

There is one more push button on the dialog box, Training. When the user presses Training button, an edit box appears on the dialog box to enter the number of training steps. Figure 6.19 illustrates the dialog box after the training button is pushed. The software simulates the system with random starting locations for the agents until the number of training step is reached. The agent may not get enough information about the environment by only using the initial database. With the training, the agents can modify their conditional probability tables according to other agent's behavior.

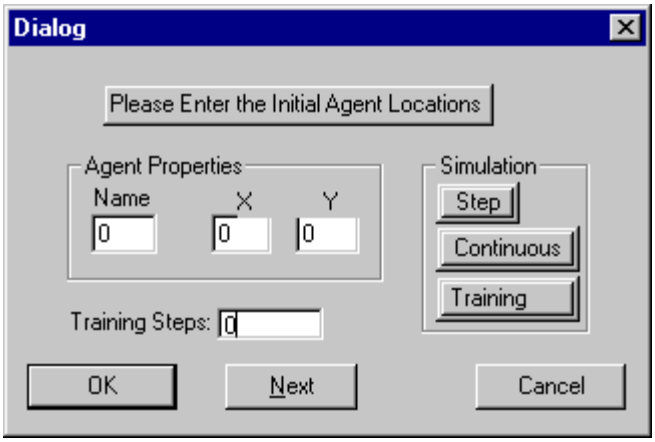


Figure 6.19. Dialog box for agent creation and simulation with training steps.

At this stage of the IntelliAgent software, Simulate submenu works for only our Dog&Sheep problem. The reason is that the utility node has huge number of elements in it because of the problem dimensionality. Therefore, in the software, the utility of an agent is a function rather than a table. If the utility node is made visual, the user has to enter too many elements in the utility table. In the future, a function editor can be placed into the software so that the user can edit the utility function by typing the function in a text box.

Help

There is no help for the IntelliAgent at this stage. This manual will be put into the software in the future. In Help menu, there is only one submenu, About Project. The About Project submenu presents the version and the icon of the software. Figure 6.20 shows the Help menu and About Project dialog box.

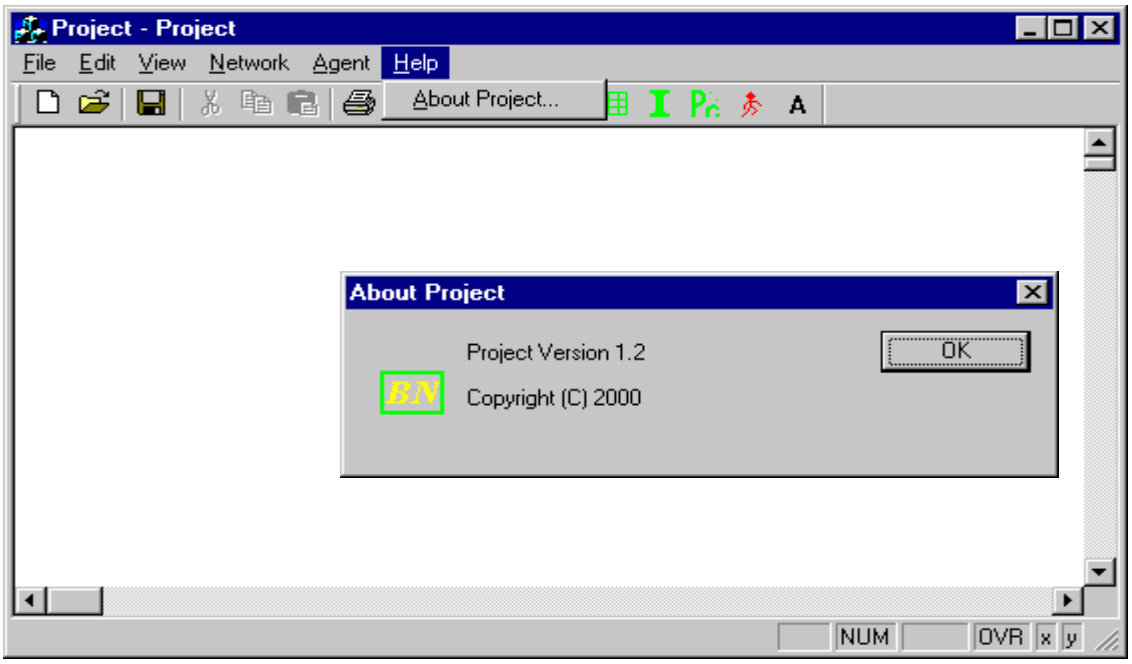


Figure 6.20. About project dialog box and Help menu.

6.1.2 Context menus

There are two types of context menus. The first one appears when the user presses the right mouse button on an empty space in the device context. The first context menu is called network context menu. The second context menu is called node context and appears when the user presses the right mouse button on a node.

Network context menu

Network context menu appears on the screen when the user clicks the right mouse button when the mouse is on an empty space on the device context, as shown in Figure 6.21. The network context menu contains the same submenus as the Network menu. The user can choose the network submenus without moving the mouse to Network menu. Context menus speed up the menu process in Windows applications.

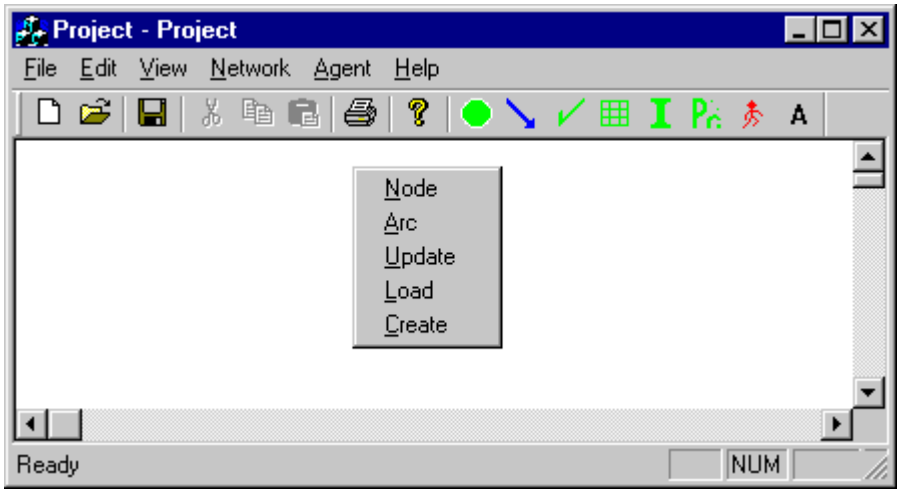


Figure 6.21. Context menu for the network submenus.

Node context menu

The IntelliAgent software has another context menu for node operation. The Node context menu appears when the user clicks the right mouse button on a node. There are two submenus in this menu, Set Evidence and Parameters, as shown in Figure 6.22.

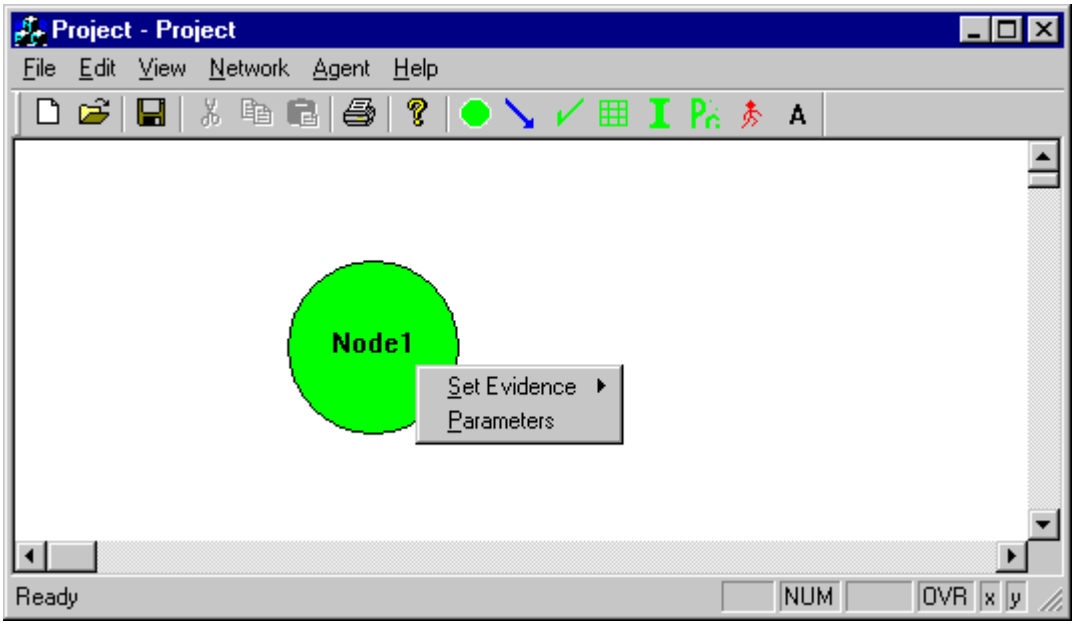


Figure 6.22. Context menu for the node operations.

Set Evidence submenu is used for instantiating the node. When the user chooses Set Evidence, another menu opens from the Set Evidence submenu to determine which state will be instantiated. Figure 6.23 illustrates how a node can be instantiated by the node context menu.

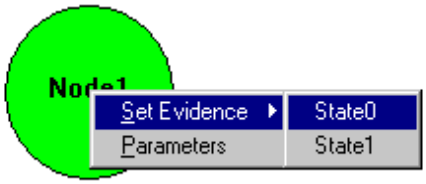


Figure 6.23. Instantiation of a node by node context menu.

In Figure 6.23, there are two possible selections in the Set Evidence submenu because the node has two states. When the number of states is more than two, the second menu shows more selections. For different number of states, the software has a node context menu assigned for them. For example, if the node has three states, the software shows another node context menu with three states as shown in Figure 6.24.

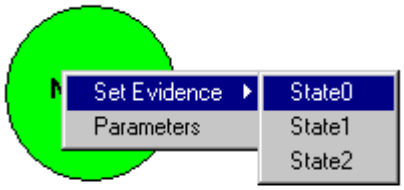


Figure 6.24. Node context menu for a node with three states.

The user can use the node context menus until the node has eight states. The program can handle nodes with more than eight states but the node context menu cannot. Instead of using context menu, the user can set evidence on a node by choosing Parameters

submenus and setting the state probabilities in the dialog box. The Parameters submenu in the node context menu actives the same function as Parameters submenu in the Network menu actives. Therefore, a dialog box appears to change the attributes of the node as shown in Figure 6.10. If the user sets the desired state to 1 and other states to zero, then the node becomes instantiated. We have not added context menu for handling nodes with more than eight states because the context menu gets too long and hard to use.

6.1.2 Toolbar

The program toolbar consists of the buttons that performs the same operations with the menu items. In the toolbar, there are 16 buttons. Figure 6.25 illustrates the toolbar of the IntelliAgent software.

Figure 6.25. The toolbar of the IntelliAgent software.

First eight buttons are standard Windows toolbar buttons. They will not be explained here. There are eight more buttons on the toolbar. They are used for Bayesian network operations and intelligent agent simulations.

Node

This button has the same functionality as the Node submenu in the Network menu. The user pushes this button if a node is going to be created as shown in Figure 6.8.

Arc

The arc button is the short cut for the Arc submenu in Network menu. The user can create arcs after pushing this button as shown in Figure 6.9.

Update

This button works as a short cut for the Update submenu in the Network menu. The user can update the network or apply inference by pushing this toolbar button instead of using menu.

Parameters

The parameters toolbar button is the short cut for the Parameters submenu in the Network menu. The user first moves the mouse on a node and clicks the left mouse button to choose the node. Then, the user moves the button to the toolbar and presses the Parameters button. Then, the software displays the dialog box shown in Figure 6.10. Changing node parameters is explained in the previous section.

Load

The load button works as the same as the Load submenu in the Network menu. The user can load a database by simply pushing the Load button on the toolbar. After the user

pushes the Load button, the program displays the dialog box shown in Figure 6.13. After the user chooses a database on the dialog box, the software creates the nodes and their parameters as shown in Figure 6.14.

Calculate

The calculate button is used for setting the structure of the search algorithm that creates the Bayesian network. After pushing this button, a dialog box appears on the screen as shown in Figure 6.15. Then, the user chooses the structure of the search algorithm by clicking corresponding radio buttons on the dialog box. After the user sets the search algorithm, the software creates the Bayesian network as shown in Figure 6.16.

Agent

The agent button is the short cut for the Create Agent submenu in the Agent menu. When the user pushes this button, the software displays the dialog box shown in Figure 6.18. The user can create by specifying agent's parameters such as name and location. In the dialog box, there are additional parameters for the simulation. The user can set the type of simulation and whether the agent will be trained in advance or not.

Simulate

This button is the short cut for the Simulation submenu in the Agent menu. The software simulates the Dog&Sheep problem, after the user clicks on this button. The program displays the simulation on the device context in action.

6.1.3 Dialog boxes

Excluding the MFC's built-in dialog boxes such as printing and saving dialog boxes, there are four dialog boxes for presenting the parameters of the nodes, updating conditional probability table (CPT) in the nodes, generating Bayesian network, and agent creating and training dialog boxes. In this section, the dialog boxes are introduced in terms of their functionality and their operation. Detailed class definitions is given in Appendix A.

Parameter Presentation

The Parameters dialog box is created for presenting and editing the parameters of a node. Figure 6.26 illustrates the Parameters dialog box. The name of the node can be edited on the dialog box by moving the mouse on the edit box in front of name. Similarly, the number of states in the node can be entered from the second edit box. As soon as the values are entered from the edit boxes, the dialog box actives corresponding functions to update the values. If a user increases the number of states, the new probabilities for the new states have to be entered. The user can enter the new probabilities by typing the values in the edit box in front of Probabilities static text. Then, the user has to push the enter push button to enter the new probabilities to the scroll box. The scroll box shows the probabilities of the states of a node. The user can update the state probabilities by

clicking the left mouse button on the probability that needs to be changed. Then, the dialog box activates an edit box and a push button under the scroll box. The user can enter the new value into the edit box and push the Update push button to enter the value into the scroll box.

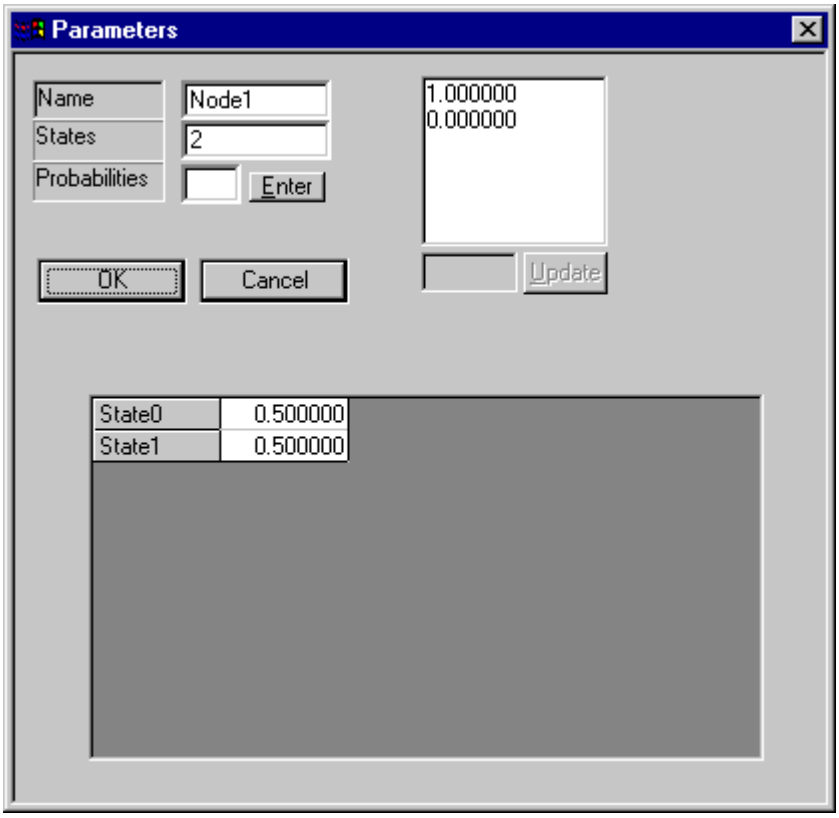


Figure 6.26. Dialog box for parameter presentation.

We have mentioned that the CPT of a node can also be edited by the user. To edit the CPT values, the user double clicks the left mouse button on the value that needs to be changed. Then, the software displays a dialog box for updating the CPT value.

CPT Updating

As stated above, the user can change the values in the CPT with the help of a dialog box. Figure 6.27 shows the dialog box for CPT updating. As soon as the user double clicks the left mouse button on a CPT value, the CPT updating dialog box appears on the Parameters dialog box. The user enters the new value into the edit box in the dialog box. Then, the value is entered to the CPT as soon as the user pushes the "OK" button.

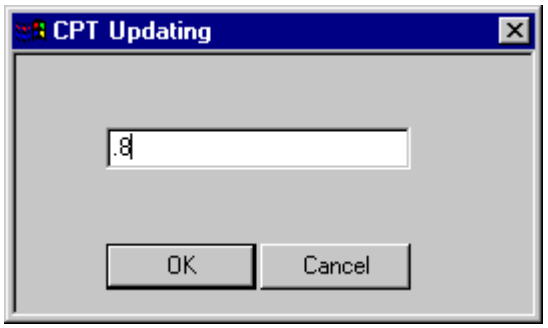


Figure 6.27. Dialog box for the CPT updating.

Bayesian network generation

The third dialog box used in the software is designed for Bayesian network generation. After the user creates the nodes and the independent probabilities by evaluating a database, a Bayesian network can be constructed by the help of a search algorithm. As stated in Section 3, there are several search algorithms in the literature. The search algorithms used in this research are introduced in Section 4. The Dialog box shown in Figure 6.28 is designed for specifying the properties of the search algorithm to be used.

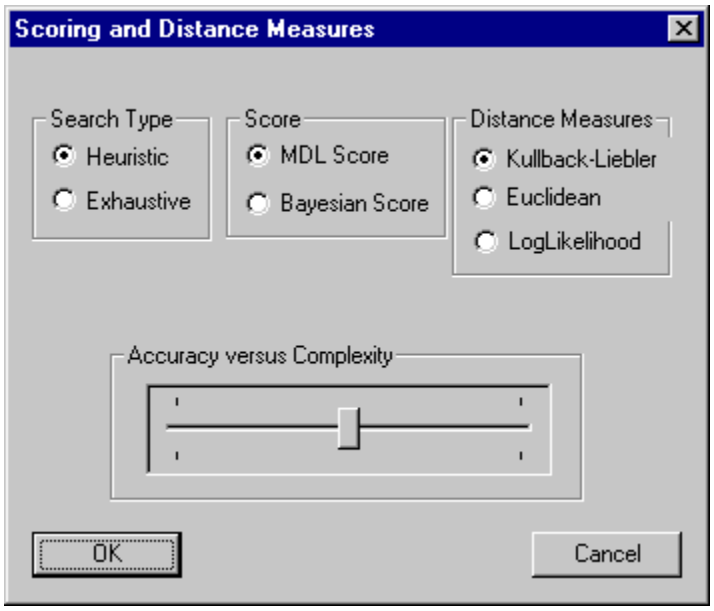


Figure 6.28. Dialog box for setting submenu for Bayesian network generation.

The dialog box consists of three groups of radio buttons for the search type, the score type, and the distance measure type, respectively. There are two radio buttons for the type of search algorithm, Heuristic and Exhaustive. There are three types of score type, MDL, Bayasian, and Log-Likelihood. Log-Likelihood is grouped in the distance measure group because it is also a distance measure type. Log-Likelihood scoring is modified to have complexity parameter in the score equation. Because of this modification, it works as MDL scoring with the Log-Likelihood distance measure. The user can choose the search type, the score type, and the distance measure type by clicking the left mouse button on the desired radio buttons.

In the dialog box, there is a sliding bar to adjust complexity and the accuracy of the search algorithm. The sliding bar is not functional in Bayesian scoring since Bayesian scoring handles the complexity and the accuracy internally as explained in Section 4. The sliding bar defines the weights for the accuracy and the complexity parts of the score.

If the user slides the bar towards the complexity, the software decreases the penalty for the complexity of the network. Therefore, the software ends up with a network with more arcs. If the sliding bar is moved towards the accuracy, the software penalizes the complexity completely. In this case, the software may end up with a network with no arcs because having an arc may be more costly than not having an arc. The software starts with the default complexity and the accuracy values. The default values weigh the complexity and the accuracy equally.

Agent creation and training

The last dialog box designed for the software appears when the user would like to create an intelligent agent for the simulation. Figure 6.29 shows the agent creation dialog box.

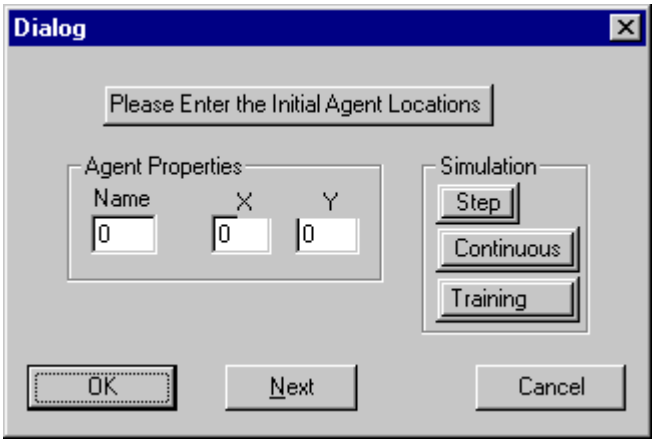


Figure 6.29. Dialog box for agent creation and training.

In the dialog box, the user can set the name and the location of the agent. In the dialog box, the user can also set the properties of the simulation. The user can determine whether the simulation will be performed step by step or continuous. If the user pushes the Step button, the simulation runs step by step. The user has to click the Simulate button on the toolbar for each step. If the Continuous button is pushed, the simulation

runs continuously until either the number of maximum steps is reached or the goal of the agent is established.

In most cases, the agents may not have enough information about the environment by only evaluating the initial database. The user may choose to train the agents before the actual simulation starts. The user can push the Training button to train the agents. As shown in Figure 6.30, an edit box appears on the dialog box after the Training button is pushed. The user can enter the number of training steps into the edit box. Then, the software starts the simulation with random initial locations for the agents until the number of training steps is reached. if the agents establish their goal and stop, then the software starts the simulation again with random agent locations.

Figure 6.30. Training abilities of the agent creation dialog box.

6.2 Tutorials on Bayesian network creation and knowledge discovery

This section presents tutorials on how to create a Bayesian network learning system. Bayesian networks can be created in two ways in the IntelliAgent software. First, they can be created manually by mouse operations. This is the case where the user knows the dependencies in the Bayesian network. It can be used for inference only. Second, a

database can be utilized to create a Bayesian network. This is the case where a knowledge discovery performed on a database.

6.2.1 Inference in a Bayesian network

This is the case where the user creates the Bayesian network by using the knowledge of dependencies in the network. Let us use the same example defined in Section 4.

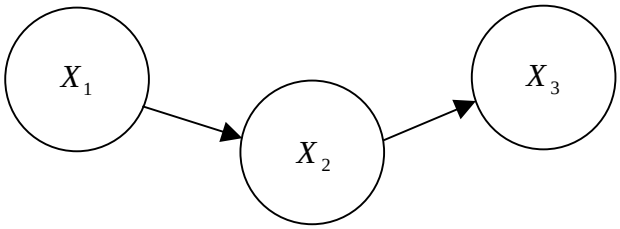


Figure 6.31. Example Bayesian network for manual network creation.

In the network, there are three variables, X_1, X_2, X_3. In the network, dependencies are given as $X_1 \rightarrow X_2$ and $X_2 \rightarrow X_3$. Figure 6.31 illustrates the Bayesian network to be created. The independent probability for the first variable is $P(X_1) = [0.5 \;\; 0.5]$ as stated in Equation 4.7. Similarly, the conditional probabilities $P(X_2 \mid X_1)$ and $P(X_3 \mid X_2)$ are given as:

$$P(X_2 \mid X_1) = \overbrace{\begin{bmatrix} 0.2 & 0.6 \\ 0.8 & 0.4 \end{bmatrix}}^{X_1} \tag{6.1}$$

$$P(X_3 \mid X_2) = \overbrace{\begin{bmatrix} 0.75 & 0.5 \\ 0.25 & 0.5 \end{bmatrix}}^{X_2} \tag{6.2}$$

Now, the above Bayesian network can be created by the IntelliAgent software. First, the user needs to create the nodes of the network. There will be three nodes with two states

each. The user moves the mouse on the Node button on the toolbar and clicks the left mouse button for node creation. Then, the user can create the nodes by keeping the left mouse button pressed and moving the mouse in a circular motion, shown in Figure 6. 32.

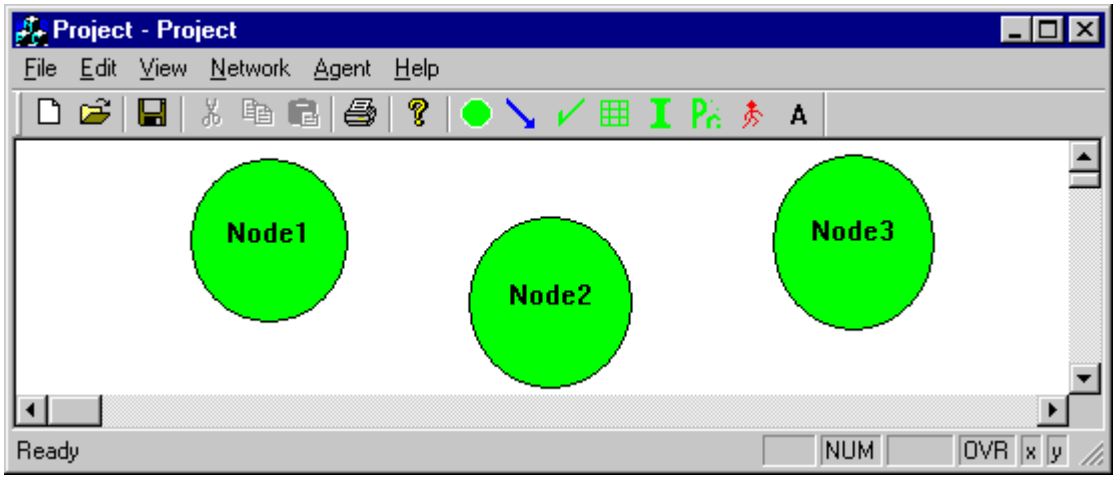


Figure 6.32. Creation on the network nodes.

In above figure, the nodes have the default parameters; two states with the probabilities 0.5 and 0.5, 2x1 conditional probability table filled with 0.5, and a default name. The user can change these values by double clicking the left mouse button on the nodes or clicking the right mouse button and choosing the Parameters submenu in the node context menu. Let us change the node names and put independent probabilities into the first variable. Figure 6.33 shows the Bayesian network with the new node names.

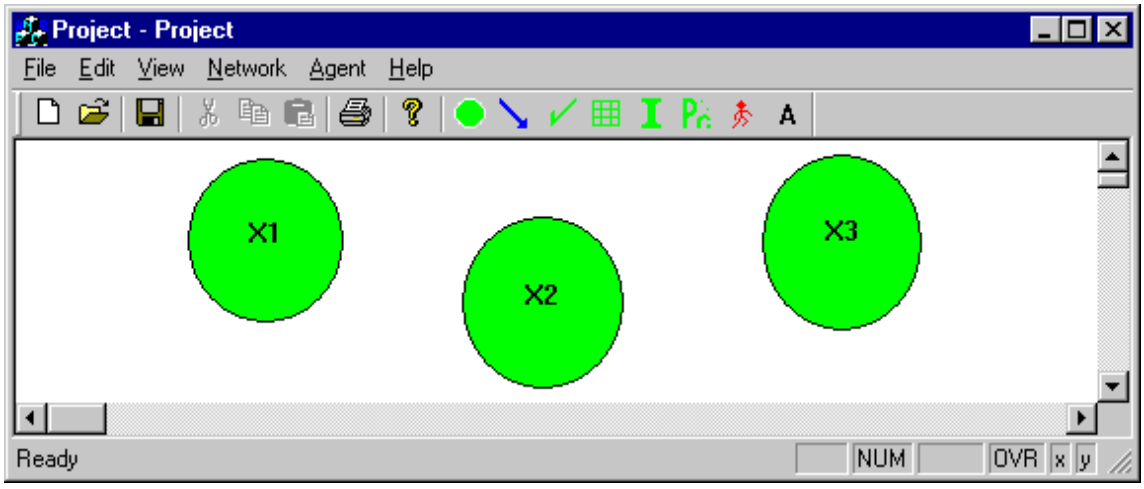


Figure 6.33. Changing node names and editing the independent probabilities.

After changing the names and placing the independent probabilities, the user can add the dependencies by drawing an arc between the variables. To draw an arc, the user clicks the left mouse button on the Arc button on the toolbar. Then, the user presses and holds the left mouse button on the node where the arc starts. While keeping the left mouse button pressed, the user moves the mouse to the node where the arc ends and releases the left mouse button. Then, the software draws an arc between the nodes. The user can start and end the arc anywhere on the nodes because the software calculates the best place to start and end the arc according to the relative positions of the nodes. Figure 6.34 shows the creation of an arc between X_1 and X_2 before the mouse is released.

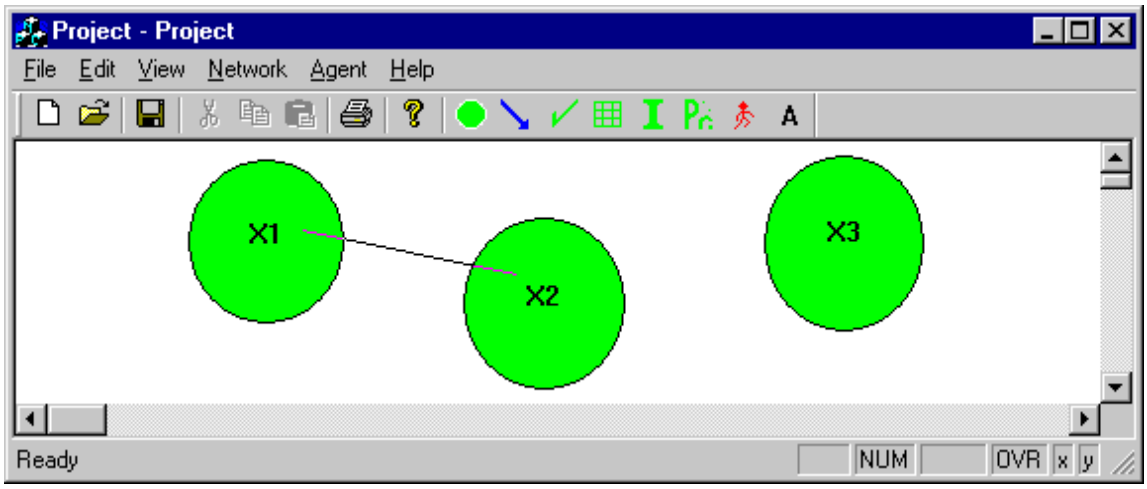


Figure 6.34. Arc creation before the left mouse button is released.

As soon as the left mouse button is released, the software displays a message box stating the creation of the arc as shown in Figure 6.35.

Figure 6.35. Message box stating the arc creation.

When the user clicks on OK button on the dialog box, the software draws an arc between X_1 and X_2. The second arc can be created by following the same procedure. Figure 6.36 illustrates the network with two arcs.

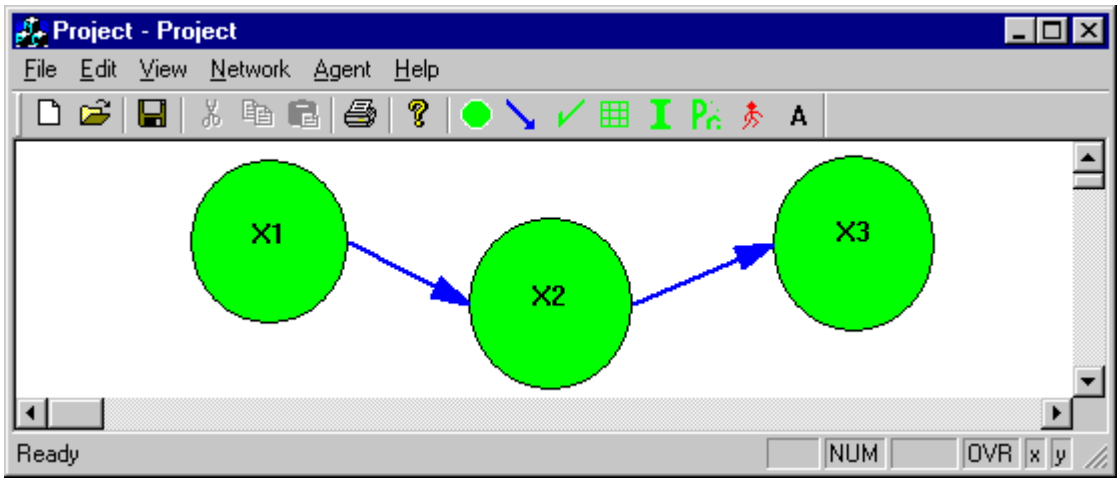


Figure 6.36. Creating an arc in a network.

When the user adds arcs from one to another node, the software automatically adjusts the dimension of the CPT of the child node. A child node is the node to which an arc points. In the network, X_2 is the child node of X_1. After the arc creation, the software expands the CPT of node X_2 to 2x2. The software puts the same values into the new column as in the first column. If the user increases the number of state in a node, the software also expands the CPT by adding a row with zero probabilities.

The structural creation of the network is completed by adding the arcs. Now, the CPTs can be edited according to Equations (6.1) and (6.2). As stated in the previous section, the user can edit the CPTs by double clicking the left mouse button on the CPT values. Then, the software displays a dialog box for CPT updating. Let us update the CPT values of node X_2.

First, the user double clicks on the node X_2 to get the Parameters dialog box, shown in Figure 6.37. Then, the user can double click the left mouse button on the value corresponding to $X_1 = state0$ and $X_2 = state0$. After the double clicking, the CPT updating dialog box appears on the screen as shown in Figure 6.37.

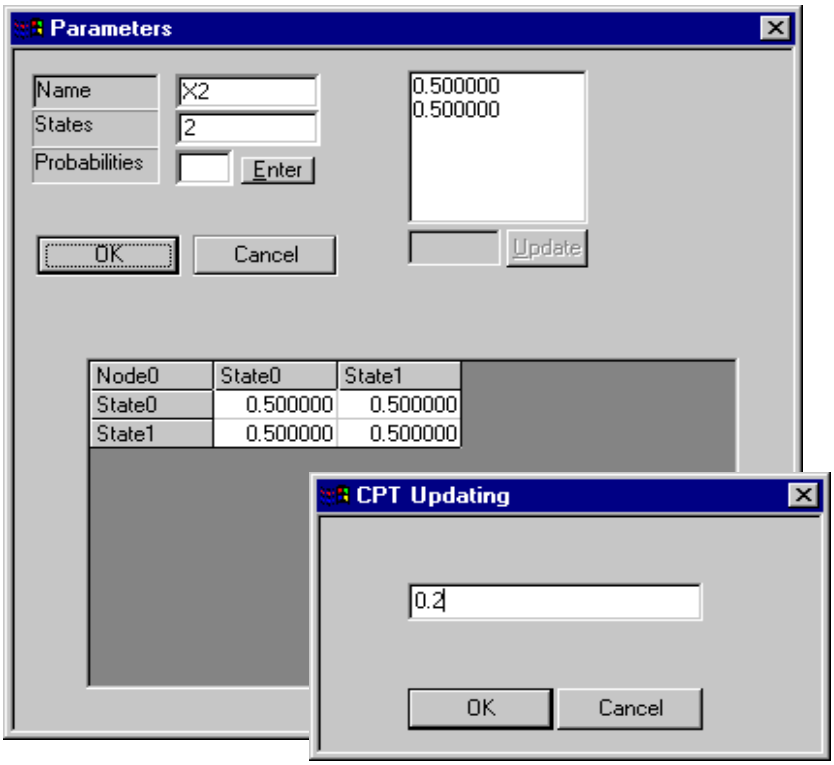


Figure 6.37. Updating the CPT table with CPT updating dialog box.

After the CPT updating dialog box appears, the user can enter the new value into the text box in the dialog box as shown in Figure 6.37. Then, the software puts the new value into the corresponding location in the CPT. The same procedure can be followed to put all values of the CPT in X_2 and X_3 using the values in Equation (6.1) and (6.2).

After the CPTs are updated, the user is ready to update the network. The user can move the mouse on Update button on the toolbar and click the left mouse button to activate the network update. The network update will produce probability values for X_2

and X_3. An inference technique defined in [1] is used to calculate the probabilities. For example, to calculate $P(X_2)$, the software uses the probabilities $P(X_1)$ and $P(X_2 \mid X_1)$ and computes the following equation:

$$P(X_2) = P(X_2 \mid X_1) \cdot P(X_1) \tag{6.3}$$

The probability $P(X_3)$ is calculated by the similar equation.

After the network update, the manual creation of a Bayesian network is completed. Now, the user can perform inference calculations by entering evidence to the network and updating network. For example, the user can set the node X_1 to *state*0 and click Update button on the toolbar to forward the evidence to the network. Figure 6.38 illustrates how to set evidence on the node X_1.

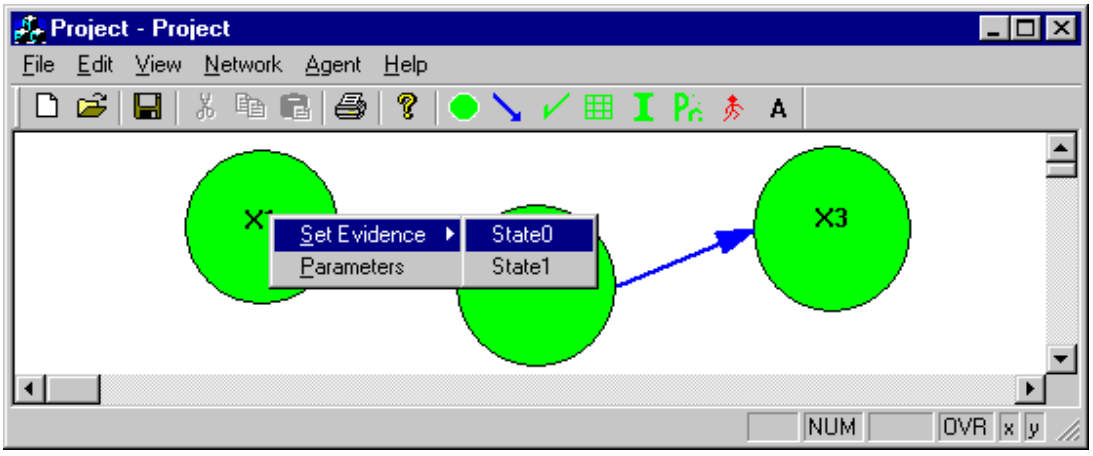


Figure 6.38. Setting node X_1 to *state*0.

After clicking the Update button, the user can double click the left mouse button on the other nodes to see the new probabilities. Figure 6.39 shows the parameters of the node X_2 before the inference. Figure 6.40 shows the network after the inference.

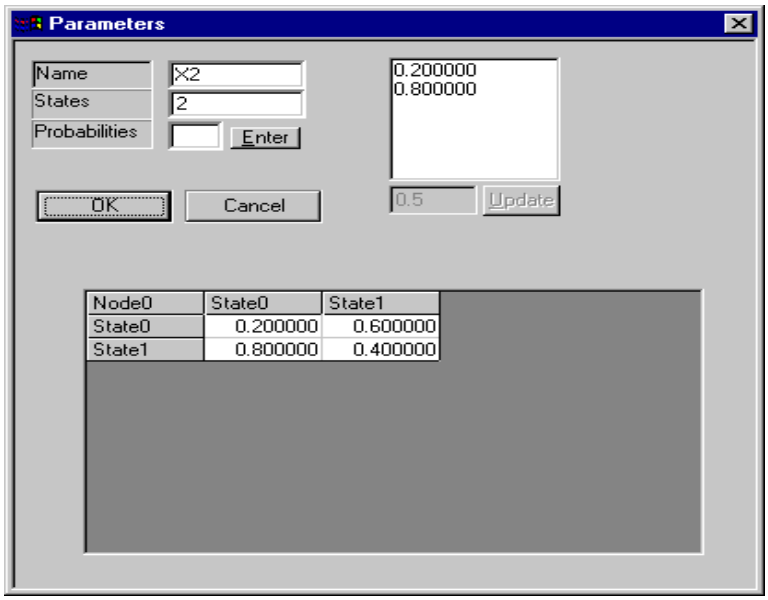


Figure 6.39. Parameters of the node X_2 before inference is applied.

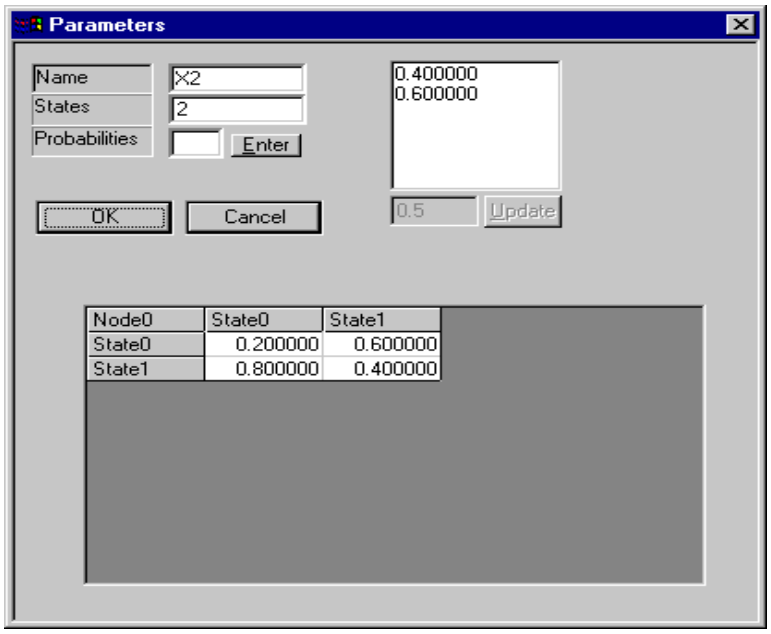


Figure 6.40. Parameters of the node X_2 after inference is applied.

As can be seen the probabilities of the node X_2 have changed with the inference. The software also updates the probabilities of X_3 according to the probabilities of the node X_2. In short, the inference travels through the network until it reaches an end node. An end node is the node that has no child. As can be seen in Figure 6.40, the CPT values are the same as the CPT values given in Equation (6.1).

The IntelliAgent software can also be used as a knowledge discovery tool because of its ability to create a Bayesian network from a database.

6.2.1 Knowledge discovery with IntelliAgent

This is the case where the user exploits a database to generate the Bayesian network that fits the data best. The user can employ several Bayesian structural learning algorithms in the IntelliAgent such as heuristic search and exhaustive search as defined in Section 4. To explain the knowledge discovery with IntelliAgent software, we will present an example in this section.

Let us take the example about the college student as defined in the previous section. The database for the problem is gathered by surveying number of college students about their college plan, sex, intelligence, family support, and social class. The IntelliAgent software will be used to create a Bayesian network that fits the database the best.

First, the user needs to load the database into the software by clicking the Load button on the toolbar. After clicking this button, the software displays a dialog box as shown in Figure 6.13. Let the user choose the database "college.db". Then, the software automatically generates the nodes and the independent probabilities for these nodes as shown in Figure 6.41.

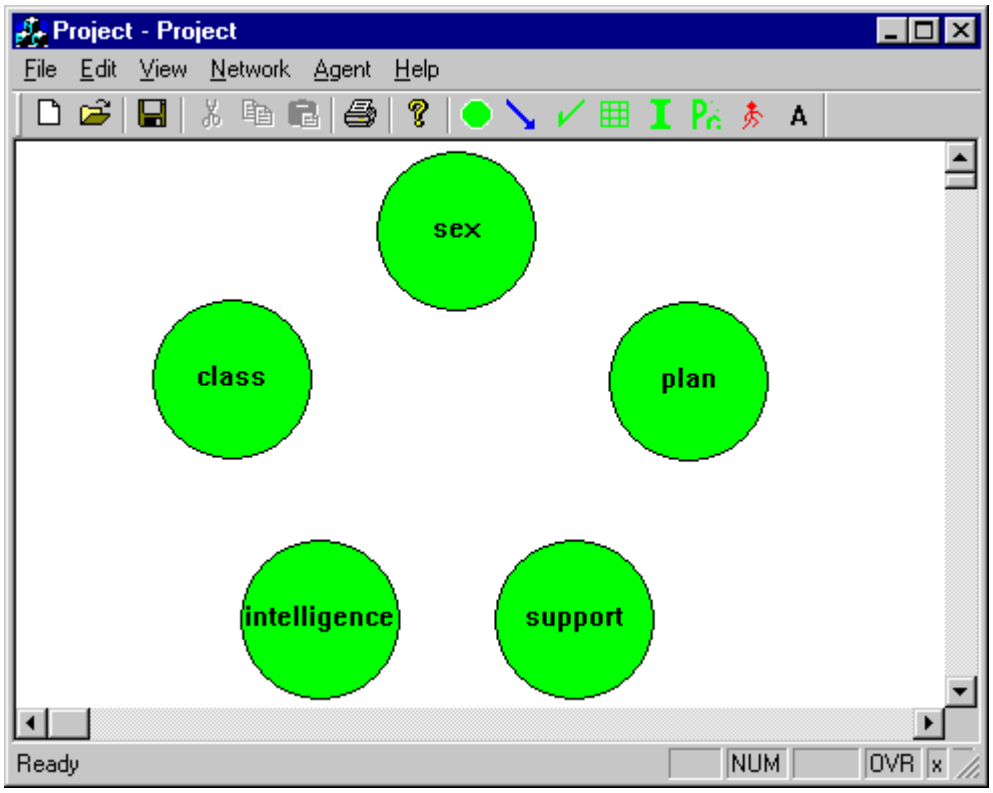


Figure 3.41. Nodes of the Bayesian network after loading "college.db".

After the software creates the nodes from the database, the user can click on Calculate button on the toolbar or choose Create submenu in the Network menu to start the search for the best Bayesian network that fits to the database. After clicking the Calculate button, the software displays the dialog box shown in Figure 6.15. This is a dialog box for setting the properties of the search algorithm as explained in the previous section. Using this dialog box, the user can choose the type of search algorithm, the score type, and the distance measure type. Let us assume that the user clicked the Heuristic radio button for the search type, the MDL score for the score type and the Kullback-Lieber distance measure for the distance measure type. Therefore, the software will search for a Bayesian network using a heuristic MDL score based algorithm with Kullback-Lieber distance measure. As soon as the user clicks OK button on the dialog box, the software starts constructing the Bayesian network. Figure 6.42 shows the final Bayesian network.

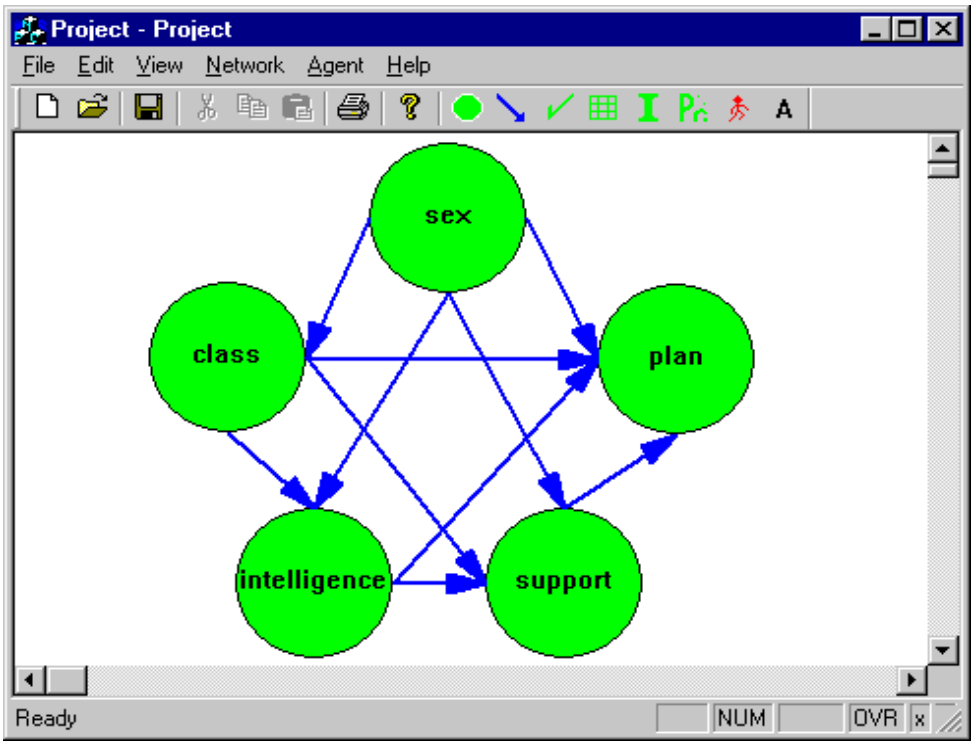


Figure 6.42. Bayesian network created by the search algorithm.

As can be seen above, the algorithm put all the possible arcs into the network. This is computationally okay for this network because the number of nodes in the network is only six. In any case, the user can decrease the complexity by sliding the complexity bar in the dialog box. Let us assume that the user would like to have simpler network. First user clicks the Calculate button on the toolbar again to get the dialog box. Then, the user needs to slide the complexity bar towards the accuracy as shown in Figure 6.43. Finally,

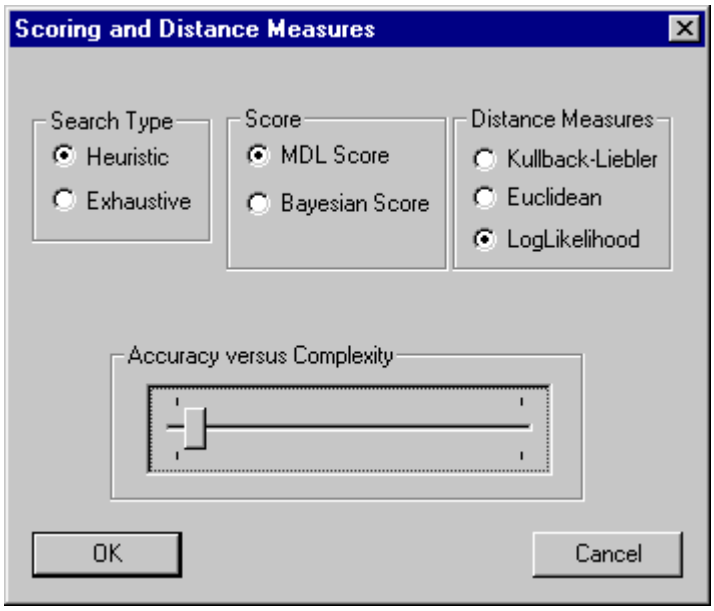


Figure 6.43. Decreasing the complexity of the network with sliding bar.

Finally, the user can click the OK button on the dialog box to start the search. Figure 6.44 shows the resulting Bayesian network.

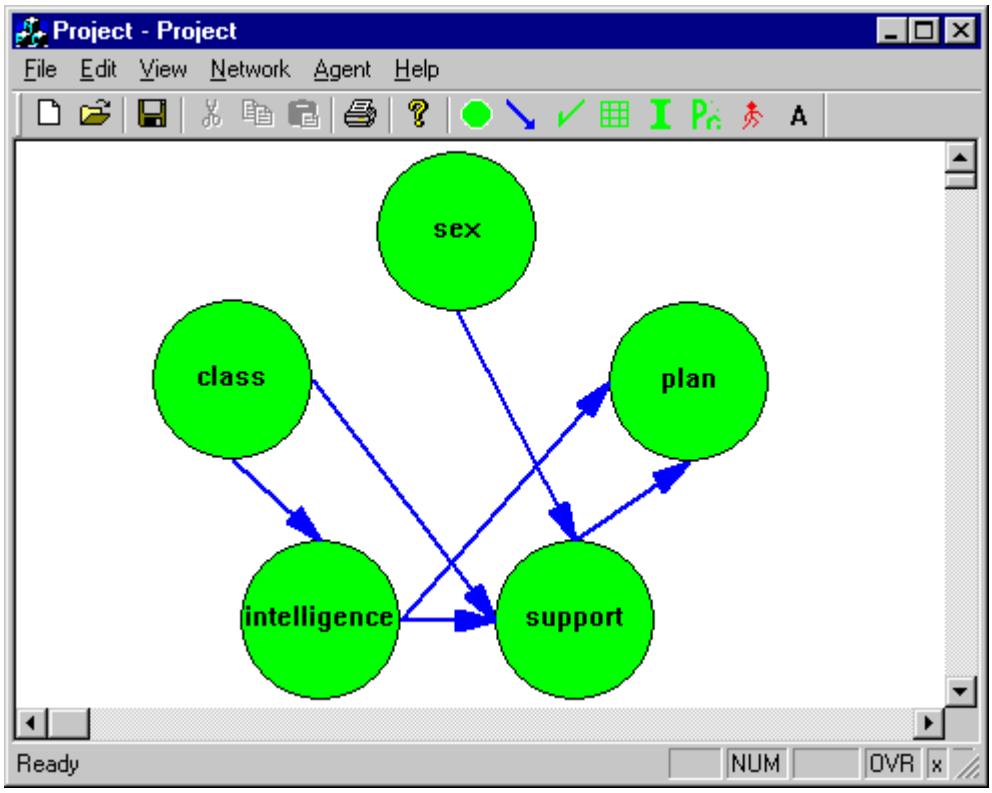


Figure 6.44. Bayesian network after decreasing the complexity.

As seen in Figure 6.44, the complexity of the network decreased noticeably. Let us assume that the user thinks that the resulting network is reasonable. Then, the user can do the knowledge discovery by observing the parameters of the network such as independent probabilities and conditional probability tables. Conditional probability tables help us to discover the dependencies between the variables. For example, we can find out how a variable effects another variable. More generally, we can find out the college plan for a given college student. This is exactly the inference explained in Section 6.2.1.

Let us assume that we have a student who is male (*state*0), with average intelligence (*state*1), in a high class (*state*0), and with family support (*state*0). To find out the probability of him making a college plan, the user needs to enter above evidence to the network and apply inference by clicking on the Update button on the toolbar. The evidence can be entered by clicking the right mouse button on the nodes and choosing desired state in the Evidence submenu as shown in Figure 6.45

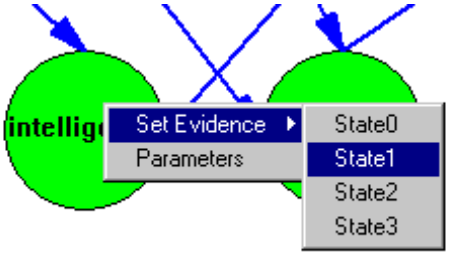


Figure 6.45. Setting the evidence for the "intelligence" node.

After updating the network, the user now can double click the left mouse button on the node for college plan to see its probabilities. Figure 6.46 illustrates the parameters of the "plan" node.

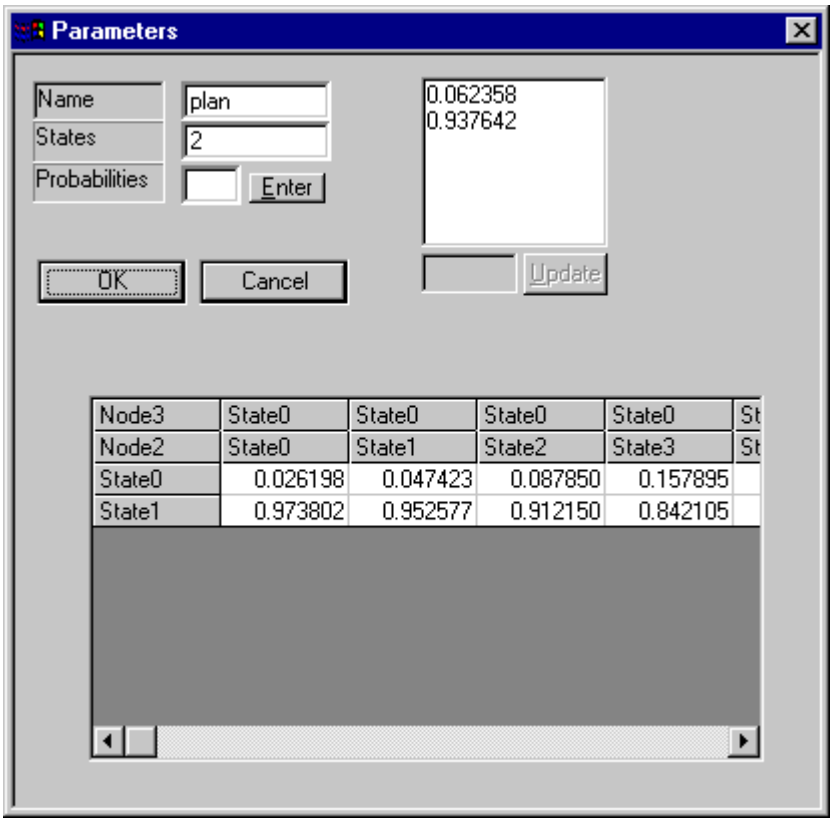


Figure 6.46. The parameters of the "plan".

In above Figure, the user can find out the probabilities of the college plan of the student. State 1 of the plan means no college plan. Therefore, the probability that student will go to college is 0.937642. This result is meaningful because the student has family support and high intelligence. Additionally, he is from a high class so he can afford the college easily.

The user can also find out how the variables effect each other. For example, how much does being a male influence the parents' support? Do families support their son more that they support their daughter? These questions can be answered by setting the "sex" node to *state*0 and *state*1 and observe the probabilities of the "support" node. In

short, knowledge discovery can be performed on a database using the IntelliAgent software.

In IntelliAgent software, when the network creation is completed, a message box appears on the screen as shown in Figure 6.47.

Figure 6.47. Message box informing the end of the network generation.

After the user clicks OK button on the message box, the software displays another message box that says the user should set the initial values of the dog and the sheep as shown in Figure 6.48. This part of the software is dedicated for the Dog&Sheep problem.

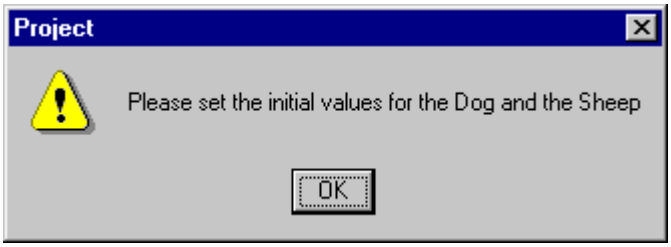


Figure 6.48. Message box for initializing the dog and the sheep agents.

At this stage of the IntelliAgent software, only Dog&Sheep problem can be simulated because the utility function is a function in the software rather than an editable table or function. In the future, this function can be made an editable function by the user. Details of the Dog & Sheep problem and its simulation results are presented in Section 6.

CHAPTER 7

Experimental Results

In this section, the decision-theoretic intelligent agent model is employed to solve a herding problem. Intelligent agent software is written to realize the proposed intelligent agent model. The same software is then used to simulate the herding problem with one sheep and one dog. Simulation results show that the proposed intelligent agent is successful in establishing a goal (herding) and learning other agents behaviors.

In the herding problem, a dog (our intelligent agent) has to herd a sheep to a desired location (i.e., a pen). The details of the herding problem are provided in Section 7.1. The simulation results are presented in Section 7.2. Finally, Section 7.3 explores the effectiveness of the online Bayesian network learning in intelligent agent system.

7.1 The Dog & Sheep Problem

The Dog & Sheep problem is considered in a rectangular $n \times m$ grid as shown in Figure 7.1. The goal of the dog is to herd the sheep into the pen. In other words, the dog is trying to minimize the distance between the sheep and the pen. The pen is at (0,0).

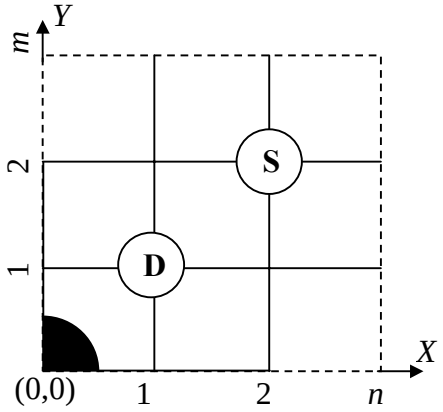


Figure 7.1. The 4-by-4 Grid Dog & Sheep problem.

There are six system variables in the problem; the *X* and *Y* coordinates of the dog, the *X* and *Y* coordinates of the sheep, the next action of the dog, and the next action of the sheep. The following illustrates the system variables and their possible values.

D_x: X coordinate of the dog; takes values form 0 to *n*.

D_y: Y coordinate of the dog, takes values from 0 to *m*.

S_x: X coordinate of the sheep; takes values form 0 to *n*.

S_y: Y coordinate of the sheep; takes values form 0 to *m*.

D_N:Next action of the dog, takes values from 0 to 4.

S_N: Next action of the sheep, takes values from 0 to 4.

The coordinates of the dog can take values between 0 and *n*. The coordinates of the sheep can take values between 0 and *m*. Therefore, the number of states in the variables D_x and S_x is *n*. Similarly, the number of states in the variables D_y and S_y is *m*. The number of states in the coordinate variables changes depending on the dimension of the problem. Agents have five possible actions; “don’t move”, “move right”, “move left”, “move down”, and “move up”. The states of the variables D_N and S_N are "don't move", move right (*x* direction), left (-*x* direction), down (-*y* direction), and up (*y* direction) with the state identifiers from 0 to 4 respectively, shown in Figure 7.2. Thus, the variables D_N and S_N have 5 states.

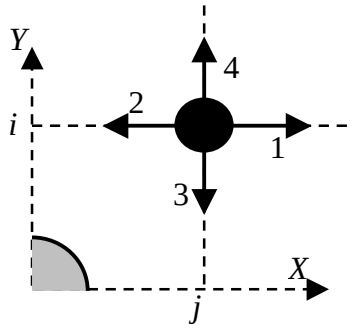


Figure 7.2. Possible moves (states) for the sheep and the dog.

After defining the system variables, we need to define the node types in the influence diagram. For the specific problem, the coordinate variables are chance nodes since they show the environmental state. Therefore, they constitute the Bayesian network (world model) of the agent. The variables D_N and S_N are decision nodes since their values can change the environmental state. The variable D_N is the decision node for the decision-theoretic intelligent agent (the dog). The variable S_N is the decision node for the other agent. The dog observes the other agent's actions (S_N) to make its decisions accordingly. Figure 7.3 illustrates the nodes type in the intelligent agent (the dog).

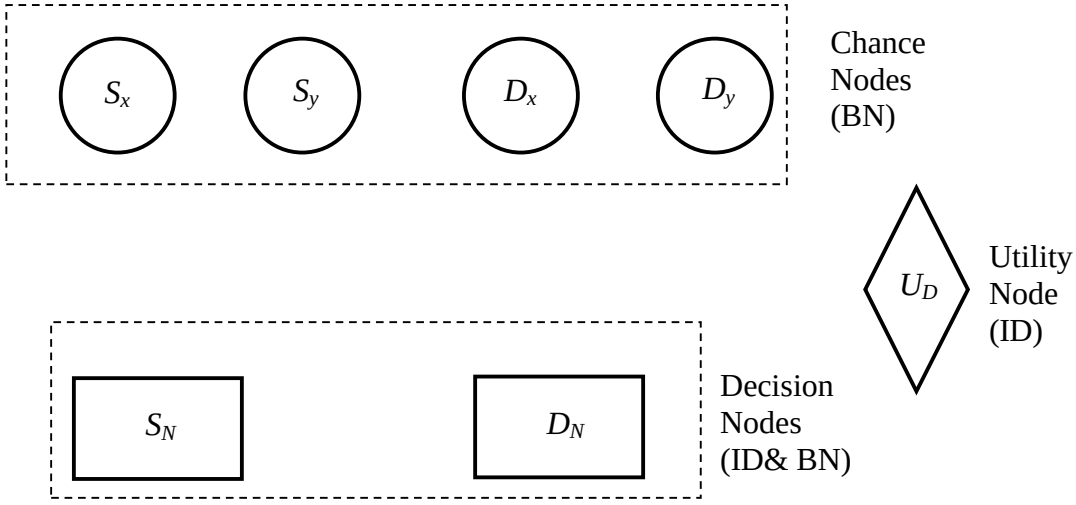


Figure 7.3. The node types in the intelligent agent for the Dog & Sheep problem.

Finally, we need to define the utility node in the influence diagram. The goal of the dog is to make the sheep go to the pen and/or to stay close to the sheep. Therefore, the utility function for the dog includes the distance between the dog and the sheep and the distance between the sheep and the dog. The utility function can be defined as:

$$U_D = f_u = \frac{1}{\sqrt{S_x^2 + S_y^2} + \sqrt{(S_x - D_x)^2 + (S_y - D_y)^2}} \tag{7.1}$$

The Euclidean distance is employed to calculate the distances. Since the maximum utility is established when the distances zero, the utility function is set to be the inverse of the sum of the distances.

After defining the Bayesian network part and the influence diagram part, the dependencies between the variables (the system dynamics) have to be established. If the system dynamics are known, the dependencies are entered to the system by inserting arcs between the variables using the agent software. If the system dynamics are not known, the agent software uses network structuring algorithms defined in Section 4 to establish the best network. The software needs a small database to generate the Bayesian network of the agent. The details of the network search algorithms are explained in Section 4. Section 6 explained how these algorithms are employed and how they can be modified.

Let us analyze the problem when the system dynamics are known. That is, we know the conditional dependencies between variables in the Bayesian network. From the nature of the Dog & Sheep problem, it is obvious that the sheep's next action is dependent on the position variables (D_x, S_x, D_y, S_y) and the dog's next action (D_N). Figure 7.4 illustrates the structure of the agent with the system dynamics.

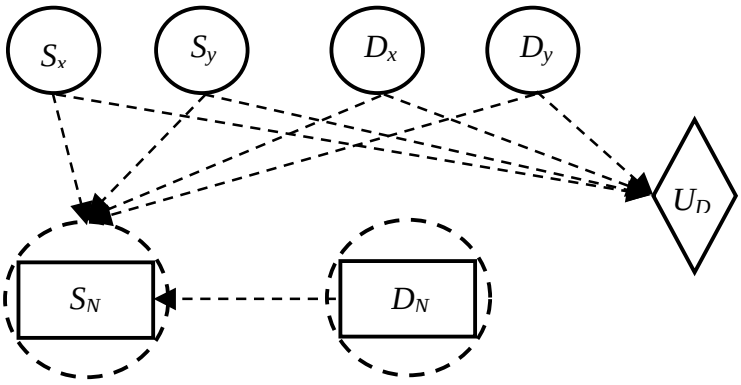


Figure 7.4. The structure of the intelligent agent with the known system dynamics.

In the Bayesian network shown in Figure 7.4, the following represents the conditional probabilities to be calculated.

$$P(D_x), P(D_y), P(S_x), P(S_y) \tag{7.2}$$

$$P(D_N) \tag{7.3}$$

$$P(S_N \mid D_x, D_y, S_x, S_y, D_N) \tag{7.4}$$

Equations (7.2), (7.3), and (7.4) define the dynamics of the system. The positions and the sheep's next action are independent but the dog's next action is dependent on the other five variables. The decision variables D_N and S_N are analyzed as chance nodes because a decision node becomes a chance node once it is instantiated. The dog takes its action before the sheep takes an action. The dog takes actions on the fly and estimates the next sheep action by updating the network. Then, the program calculates the utility function using estimated sheep position and the dog position. Finally, it fires the action that creates the maximum utility. Note that the positions of the dog and the sheep do not directly affect the dog's action. They affect the sheep's action directly. Since the dog decides its actions according to the sheep's expected action, the positions affect the dog's action.

As stated earlier, if the system dynamics are not known, the agent software generates a network from the available data. The search algorithms defined in Section 4 are used to find the network that fits to the available data. As stated in Section 4, the properties of the algorithm used can be adjusted by the user.

Let us consider the network created by using the heuristic search with Bayesian scoring. A small database is provided for the search algorithm. The resulting Bayesian network is shown in Figure 7.5. In the generated Bayesian network, additional arcs

between the variables are added by the algorithm. The algorithm showed that the positions of the sheep and the dog affects not only the dog's next action but also the sheep's next action. The dog models the sheep's dynamics with the arcs between the sheep's next action and the position variables. The additional arcs complicate the network but it also makes more sense to model the sheep's behavior.

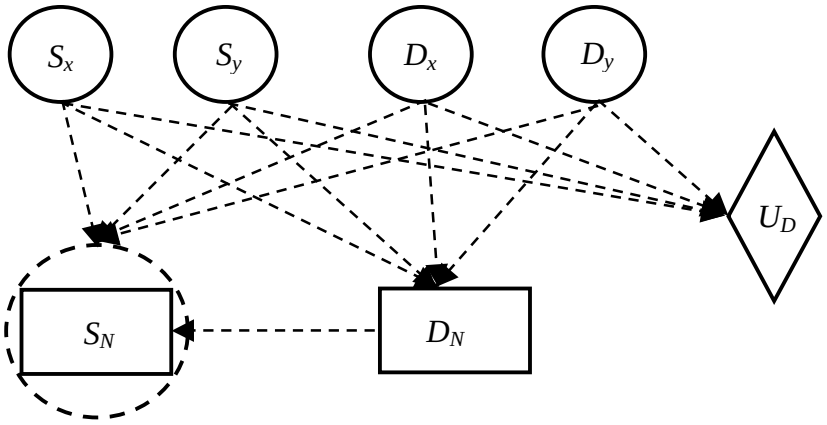


Figure 7.5. The structure of the agent with BN created by the search algorithm.

The structure in Figure 7.5 adds one more conditional probability to the calculations because the sheep's next action depends on the positions of the dog and the sheep. Therefore, the equations necessary for the inference calculations become the following.

$$P(D_x), P(D_y), P(S_x), P(S_y) \tag{7.5}$$

$$P(S_N) \tag{7.6}$$

$$P(S_N \mid D_x, D_y, S_x, S_y, D_N) \tag{7.7}$$

$$P(D_N \mid D_x, D_y, S_x, S_y) \tag{7.8}$$

The equation (7.8) is added to the calculations. Section 7.3 provides more detailed simulation results for the Dog & Sheep problem in a specific domain.

The creation of the decision theoretic intelligent agent is completed after generating the structure of the Bayesian network. Now, the agent can start exploring the environment to establish its goal. The agent exploits the environment during its exploration by updating itself with the new information about the environment. The following summarizes the exploration and exploitation processes of an agent.

The agent (the dog) takes its actions in order to maximize the utility function in equation (7.1). First, the agent fires the actions on the fly and calculates the probabilities of the states in the sheep's next action node. Second, the value of the utility function is calculated for each possible action (state) of the variable S_N. Then, using Equation (2.12), the expected utility of the dog's (agent's) action, d_i. The following formula presents how the expected utility is calculated for the action d_i.

$$\overline{U}(D_N = d_i) = \sum_{s_j} U(S_N = s_j, D_N = d_i) \cdot P(S_N = s_j \mid D_N = d_i) \tag{7.9}$$

The same formula is applied to calculate the expected utility of each action in the agent's action set. The utility functions in the summation are calculated by the using Equation (7.1). In the formulation, the positions of the dog and the sheep are not shown because they are updated by the action d_i and s_j. Let us denote the updated positions with a bar in the formulation. The following equation presents the utility function for the action pair, d_i and s_j.

$$U(S_N = s_j, D_N = d_i) = \frac{1}{\sqrt{\overline{S}_x^2 + \overline{S}_y^2} + \sqrt{\left(\overline{S}_x - \overline{D}_x\right)^2 + \left(\overline{S}_y - \overline{D}_y\right)^2}} \tag{7.10}$$

where $\overline{S}_x, \overline{S}_y, \overline{D}_x,$ and $\overline{D}_y$ are the updated (expected) positions of the dog and the sheep. The utilities for all possible action pairs are calculated by Equation (7.10). The

conditional probability, $P(S_N = s_j \mid D_N = d_i)$, is calculated by the inference algorithm defined in [1].

Finally, the agent chooses the action with the highest expected utility by using the PMEU as in equation (5.1).

$$d = \max_{D_N = d_i} \overline{U}(D_N = d_i) = \max_i \left\{ \sum_{s_j} U(S_N = s_j, D_N = d_i) \cdot P(S_N = s_j \mid D_N = d_i) \right\} \quad (7.11)$$

where d represents the action with the highest expected utility.

To sum up, first the agent fires it actions on the fly. Second, the inference is run through the BN to calculate the corresponding probabilities of the sheep's possible next actions. Third, the utilities are calculated for each possible action of the sheep and the dog's action using Equation (7.10). Then, the probabilities and the utilities are placed into Equation (7.9) to calculate the expected utility of the dog's action. The process is repeated for each possible action of the dog. Finally, the agent (the dog) chooses the action with the maximum expected utility by employing the formula in (7.11).

After deciding which action will be fired, the dog takes the action and observes the sheep's next action. The dog records the current states of the system variable after the sheep moves. The agent updates its BN (the world model) using the current states of the system variables according to the algorithm defined in Section 4. The dog (the agent) continues to take actions in the same way until the sheep is in the pen. The following section explains the Dog & Sheep simulation performed by the IntelliAgent software.

7.2 The 4-by-4 Grid Dog & Sheep Simulation

In Figure 7.1, the Dog&Sheep problem is presented on a *n*-by-*m* grid. This section presents the simulation results for a 4x4 grid, $n = m = 3$. The section explores both known dynamics case and unknown dynamics case. We have run the simulations by placing the dog and the sheep in several different locations. For all the locations, the dog herded the sheep to the pen successfully. The simulation results were satisfactory for both known and unknown dynamics cases. Let us start with the simulations performed with known system dynamics.

7.2.1 Simulation results for known system dynamics

Let us use the same system dynamics shown in Figure 7.4. The nodes in the Bayesian networks can be created manually with mouse moves or with a database file. A database is used to generate the initial network parameters. The database is created with 19 data cases. Each data cases consist of the dog and the sheep locations and the corresponding actions of the sheep and the dog.

The IntelliAgent software is used to create the Bayesian network for the intelligent agent. The utility function is placed in the software as a function with the form of Equation (7.1). Thus, the user does not have access to the utility function of the intelligent agent in the software. The IntelliAgent software can only simulate the Dog & Sheep problem. To simulate other intelligent agent problem, the utility function for the agent has to be edited accordingly in the source code.

Let us create the intelligent agent's Bayesian network by using IntelliAgent software. First we need to create the network nodes. To create the network nodes, the database in Table 7.1 is loaded to the software.

Table 7.1. Initial database for the Dog&Sheep problem

D_x	D_y	S_x	S_y	D_d	D_s
3	0	3	1	0	4
3	1	3	0	0	2
2	1	1	1	4	3
1	2	1	0	3	2
2	1	1	1	2	3
0	2	1	1	1	3
1	2	0	1	2	3
3	0	1	0	2	2
2	1	1	0	3	2
1	2	1	1	0	3
2	0	1	0	0	2
0	1	1	0	1	2
0	2	1	0	1	0
1	0	0	1	1	0
1	1	0	1	4	3
0	3	1	3	0	1
2	2	3	3	1	3
3	2	2	3	4	1
3	2	2	3	4	3

The data cases are created manually. We put the dog and the sheep into random locations and we have chosen the actions of the dog and the sheep. Then, we have put those six values into a row in the database. The number of data cases in the database is long enough to calculate the initial parameters of the Bayesian network. The agents update their network parameters while they are exploring the environment.

The above database is edited into a text file, called dogsheepdb.txt for the IntelliAgent software. Then, we have loaded the database into the software as shown in Figure 7.6.

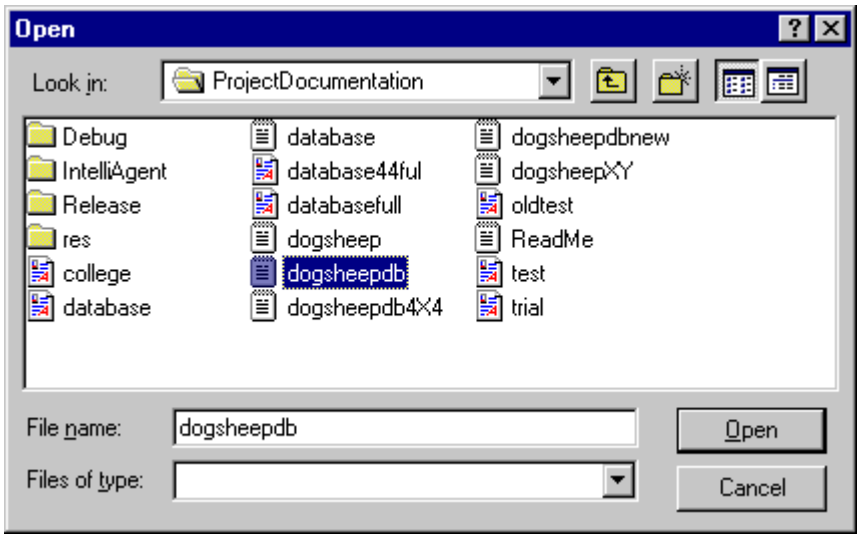


Figure 7.6. Loading the initial database.

After loading the database, the IntelliAgent software creates the network nodes and calculates the independent probabilities for each variable. Now, we have the network nodes with their parameters. The software determines the number of states in the nodes, their probabilities, and their names by evaluating the database.

After the software creates the network nodes, we need to define the dependencies. As stated in Section 6, the dependencies (the arcs) between the nodes can be established by mouse operations. A tutorial on how to create a Bayesian network is also presented in Section 6.2. We have created a Bayesian network with same dependencies as the network shown in Figure 7.1. Figure 7.7 presents the Bayesian network created in the IntelliAgent software.

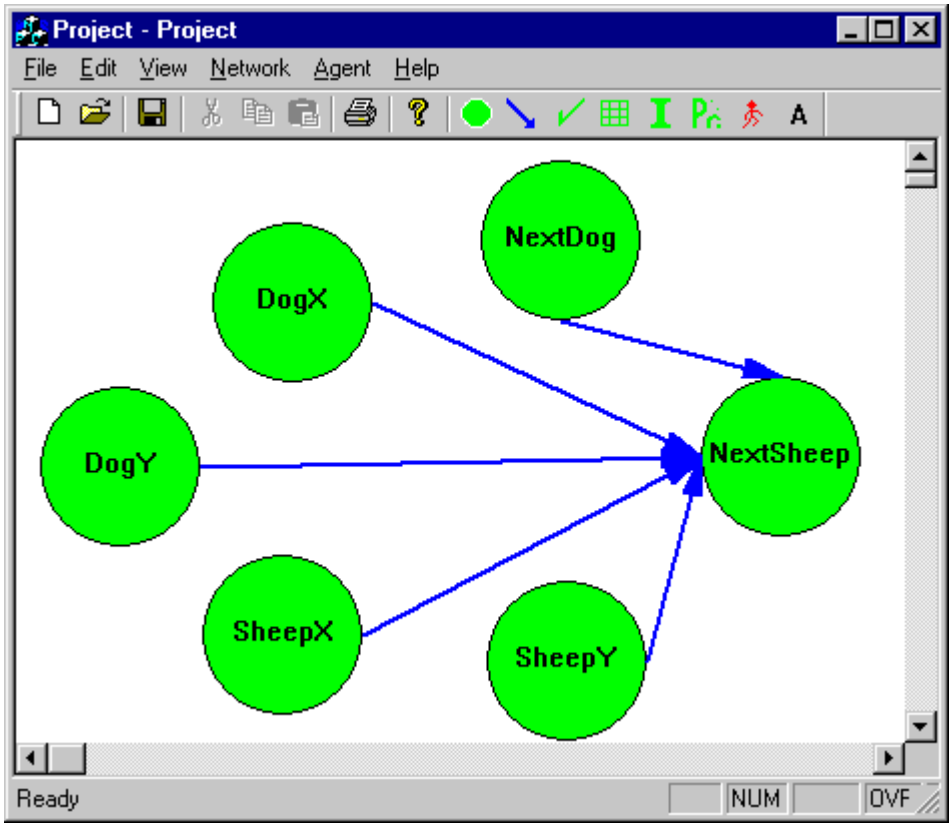


Figure 7.7. Bayesian network with known dependencies.

After the arcs are drawn in the network, the software adjusts the CPTs and the probabilities of the nodes by running inference in the network. Now, we can start running the simulation.

To simulate the problem, we need to create the intelligent agent. As described in Section 7, agents are created by using a dialog box. The user can give the name and the location of the agents using this dialog box. In the IntelliAgent simulation the dog agent and the sheep agent are named 1 and 2, respectively. Thus, the user should enter either 1 or 2 as the name of the agent during the creation of the agents. The software knows which agent is the dog or the sheep by checking the name of the agent. Let us create the

dog and the sheep at (0,0) and (3,3), respectively. This is the hardest case for the dog to herd the sheep into the pen. As soon as an agent is created, the software draws a grid on the screen to display the simulation. Figure 7.8 shows the simulation grid created by the IntelliAgent software.

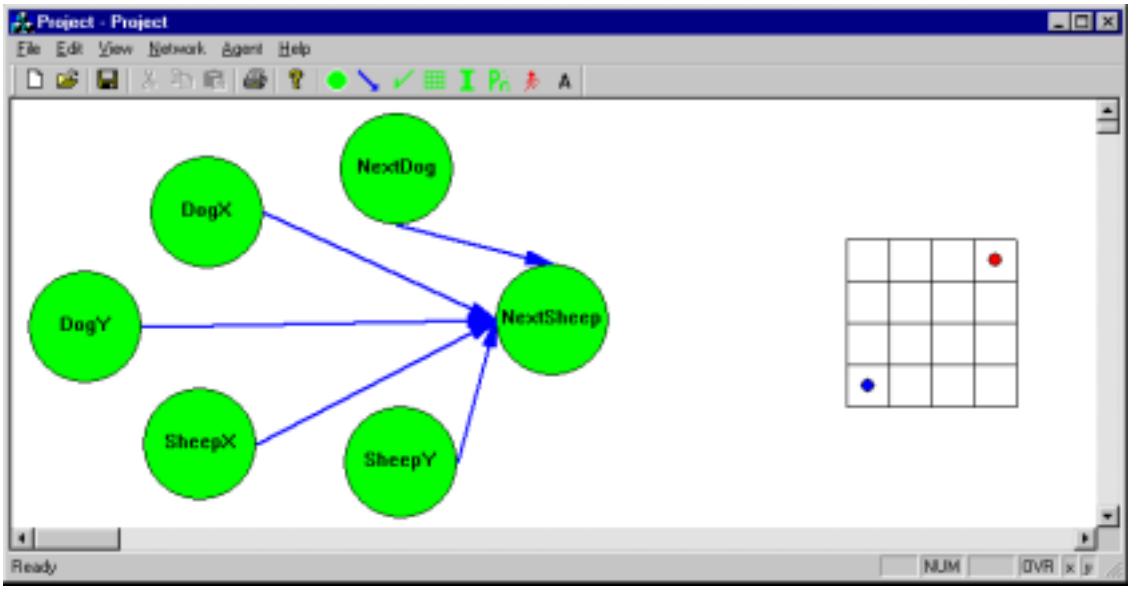


Figure 7.8. Bayesian network and the simulation grid.

Let us assume that we have chosen continuous simulation on the agent creation dialog box. The simulation results were successful for different placements of the agents. We will only present simulation results for the hardest case in this section. Figure 7.9 shows the paths that the sheep and the dog have taken during the simulation. The dog was able to establish its goal by herding the sheep to the pen. The simulation ended when the sheep was at (0,0) and the dog was at (1,1).

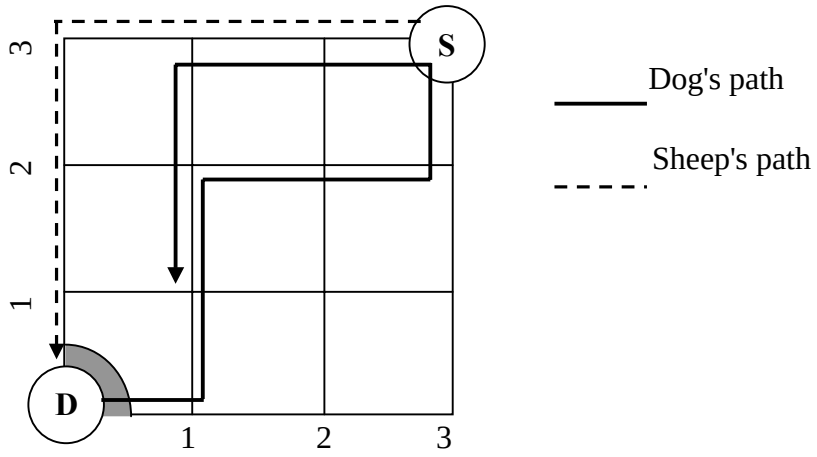
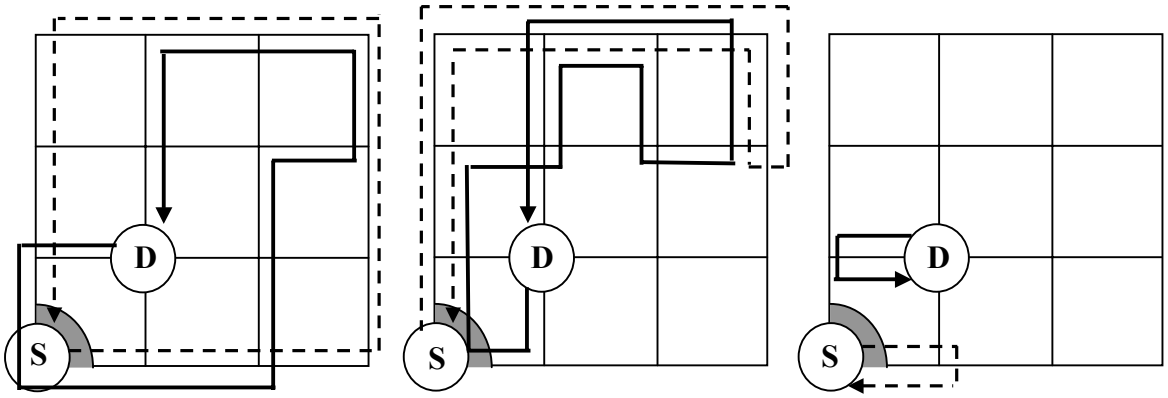


Figure 7.9. The paths taken by the dog and the sheep.

In Figure 7.9, the sheep does not move away from the corner until the dog is near the sheep. When the dog comes closer to the sheep, the sheep moves away from the corner and the dog. Then, the dog follows the sheep trying to herd it to the pen. The dog chases the sheep until the sheep is in the pen.

We have also run three consecutive simulations without changing their positions, shown in Figure 7.10. The goal of these consecutive simulations is to see whether the dog learns from its experience. In the first run, the sheep has escaped from the pen by moving to the right. The dog then followed the sheep and put the sheep into the pen. In the second run, the sheep managed to escaped from the pen by moving up because the dog moved down to stop the sheep moving to the right. The dog learned from its previous experience that the sheep will move to the right. In the third simulation, the dog first moved to the left to stop the sheep moving up and the sheep moved to the right. Then, the dog moved to the right to stop the sheep moving to the right. Finally, the sheep moved back into the pen in response to the dog's movement. The sheep could not escape in the third run because the dog learned the sheep's behavior by experiencing previous escapes. The dog takes its actions according to the knowledge it gets from its experience.



(a) (b) (c)

Figure 7.10. Learning from the experience.

As stated earlier, the dog agent updates the parameter of the Bayesian network while it explores the environment. The simulation starts with the initialization of the positions of the agent. Using Equations (7.9), (7.10), and (7.11), the dog calculates the expected utility for its actions and finds the action with the highest expected utility.

The dog fires the action with the maximum expected utility. Then, it waits for the sheep's next action. The sheep has its own dynamics and tries to avoid the dog. The sheep's dynamics are bunch of rules that determine the next action of the sheep. The rules are defined so that the sheep is moving away from the dog.

After the sheep takes its action, the dog records the current positions and actions of the sheep and the dog into the database as a data case. Since the database is modified, the software modifies the parameters of the Bayesian network according to the new data case. The modification of the parameters does in fact establish the learning. The next time the same setting is faced, the agent will take its actions according to the modified parameters of the network. The following paragraph presents how learning occurs in the decision-theoretic intelligent agent.

In Figure 7.10, the simulation starts with specified agent locations; the dog is at (1,1) and the sheep is at (0,0). Let us go through the learning process for the dog by analyzing it possible actions and their expected utilities. The sheep has three possible actions in this setting; "don't move", "move left", and "move up". The sheep cannot move down or move left because it is at (0,0). To calculate the expected utilities for the dog's actions, we need to calculate the probabilities of each state action and corresponding utility value. Since the agent has limited information about the environment, the probabilities of the sheep's actions will be uniformly distributed. The software places uniform priors if a case has never seen before. The software can show the probabilities of the sheep's decision node S_N by simply double clicking on the node. The following is the probabilities of the sheep's next action states.

$$P(S_N \mid D_N = d_i) = \{0.2, 0.2, 0.2, 0.2, 0.2\} \tag{7.12}$$

As can be seen above, probabilities are uniformly distributed for all the action of the dog. Now, let us calculate the expected utilities for each the dog's action d_i and the possible actions of the sheep (s_0, s_1, s_3). The software only calculates the utility for the sheep's possible actions. The utilities for the sheep's impossible actions are equal to zero. The following equations present the calculation of the utilities for the sheep's possible actions.

$$U(S_N = s_0, D_N = d_0) = \frac{1}{\sqrt{0+0} + \sqrt{1+1}} = \frac{1}{\sqrt{2}} \tag{7.13}$$

$$U(S_N = s_1, D_N = d_0) = \frac{1}{\sqrt{1+0} + \sqrt{0+1}} = \frac{1}{2} \tag{7.14}$$

$$U(S_N = s_2, D_N = d_0) = 0 \tag{7.16}$$

$$U(S_N = s_3, D_N = d_0) = \frac{1}{\sqrt{0+1} + \sqrt{1+0}} = \frac{1}{2} \tag{7.15}$$

$$U(S_N = s_4, D_N = d_0) = 0 \tag{7.17}$$

Using Equations from (7.12) to (7.17), we can calculate the expected utility of the dog's action d_0.

$$\begin{aligned} \overline{U}(D_N = d_0) &= \sum_{i=0}^{4} U(S_N = s_i, D_N = d_0) \cdot P(S_N = s_i \mid D_N = d_0) \\ &= \frac{1}{\sqrt{2}} \cdot 0.2 + \frac{1}{2} \cdot 0.2 + \frac{1}{2} \cdot 0.2 \cong 0.3404 \end{aligned} \tag{7.18}$$

The expected utilities for the other actions of the dog are calculated in the same way. The following equation presents the expected utilities for the dog's all actions.

$$\overline{U}(D_N) = \begin{bmatrix} 0.3404 \\ 0.2385 \\ 0.4788 \\ 0.4788 \\ 0.2385 \end{bmatrix} \tag{7.19}$$

After the expected utility for each action is calculated, the agent (the dog) fires the action that generates the maximum expected utility. Therefore, the dog fires the action d_2, which is "move left". Thus, the dog moves to (0,1) on the grid. The dog waits for the sheep's next action after it fires its best action.

The sheep has move to the right because it is trying to get away from the dog. Therefore, the new positions are (0,1) and (1,0) for the dog and the sheep, respectively. The simulation is run until the sheep is in the pen again. When the simulation is ended, the sheep was in the pen and the dog was at (1,1) as shown in Figure 7.10 (b).

Let us check the state probabilities of the sheep's action (decision) node S_N and the expected utilities for the dog's actions.

$$P(S_N \mid D_N = d_0) = \{0.2,0.2,0.2,0.2,0.2\} \tag{7.20}$$

$$P(S_N \mid D_N = d_1) = \{0.2,0.2,0.2,0.2,0.2\} \tag{7.21}$$

$$P(S_N \mid D_N = d_2) = \{0,1,0,0,0\} \tag{7.22}$$

$$P(S_N \mid D_N = d_3) = \{0.2,0.2,0.2,0.2,0.2\} \tag{7.23}$$

$$P(S_N \mid D_N = d_4) = \{0.2,0.2,0.2,0.2,0.2\} \tag{7.24}$$

$$\overline{U}(D_N) = \begin{bmatrix} 0.3404 \\ 0.2385 \\ 0.4141 \\ 0.4788 \\ 0.2385 \end{bmatrix} \tag{7.25}$$

As can be seen in Equation (7.22), the conditional probability of S_N given the action d_2 is changed after the first run. This is because the sheep moved to the right in the first simulation. The agent then updated this particular conditional probability accordingly. This shows that the decision-theoretic agent can learn from its experience. The change in the conditional probability is also changed the expected utility of the action d_2. The expected utility of the action d_2 is reduced because the sheep went away from the pen in the previous run. Now, the dog knows that if it fires the action d_2 again, the sheep will move to the right.

Since the expected utilities are changed, the expected utility of the action d_3 became the maximum. Therefore, the dog moves down to stop the sheep going to the right. After the dog moved down, the sheep went up (s_4) to avoid the dog. We have run the simulation until the sheep is in the pen. One should keep in mind that the agent updates

its model of the environment in every step. The simulation is ended when the sheep is in the pen and the dog is at (1,1) as shown in Figure 7.10 (c).

One might guess that the conditional probabilities and the expected utilities will be different than that of the previous run. The following equations present the conditional probabilities and the expected utilities.

$$P(S_N \mid D_N = d_0) = \{0.2,0.2,0.2,0.2,0.2\} \tag{7.26}$$

$$P(S_N \mid D_N = d_1) = \{0.2,0.2,0.2,0.2,0.2\} \tag{7.27}$$

$$P(S_N \mid D_N = d_2) = \{0,1,0,0,0\} \tag{7.28}$$

$$P(S_N \mid D_N = d_3) = \{0,0,0,0,1\} \tag{7.29}$$

$$P(S_N \mid D_N = d_4) = \{0.2,0.2,0.2,0.2,0.2\} \tag{7.30}$$

$$\overline{U}(D_N) = \begin{bmatrix} 0.3404 \\ 0.2385 \\ 0.4141 \\ 0.4141 \\ 0.2385 \end{bmatrix} \tag{7.31}$$

The agent continues to learn from its experience because the conditional probabilities of S_N given the action d_3. Therefore, the expected utility for the action d_3 is also changed. The utilities of the action d_2 and d_3 are equal. The agent chooses the action with the lower indices if there is equality. Therefore, the dog fires the action d_2.

After the dog moved to the left by firing the action d_2, the sheep moved to the right to avoid the dog. Then, the dog moved to the right by firing the action 1 because it knows that in the first run the sheep moved to the right and escaped. To keep the sheep around the pen, the dog moved to the right instead of going down and following the

sheep. This shows that the dog learns the sheep's behavior in time and acts accordingly. In fact, after the dog moved to the right, the sheep moved to the left and went into the pen as shown in Figure 10 (c).

We have covered the case where the dynamics of the system is known. The simulation results were satisfactory. This part of the research will be presented in IEEE SMC2000 conference. In above simulations, the next action of the dog is not directly dependent on the positions of the dog and the sheep. In reality, the dog's next action is also dependent on the positions of the dog. The simulations worked well because the dog had the exact knowledge of the relationship between the positions and the sheep's next action. If the dog does not have that information, it cannot make its decisions only depending on the sheep's next action. In that case, the agent has to create its Bayesian network and find out the dependencies in the network. The next section explores the case when the system dynamics are not known.

7.2.2 System dynamics are not known.

In real life, an intelligent agent may not have the knowledge of the system dynamics. For example, if a mobile robot is placed in a room to do certain tasks, the robot will not have the exact knowledge of the room at the beginning. Furthermore, if there are more than one robot, the robots will not know the dynamics of other robots. In this kind of problem settings, the robots have to explore the environment and exploit (learn) the data that they have gathered. In our agent design, we have placed an online Bayesian network learning ability to our decision-theoretic intelligent agents. In previous sections, the online

Bayesian network learning and software are explained in detail. In this section, we will simulate the Dog & Sheep problem with unknown system dynamics.

In the previous case, the agent learned the parameters of the network using a database because the system dynamics were known. Now, the agent has to learn both structure and parameters of the Bayesian network using the database. In Section 4, structural learning and parameter learning in the online Bayesian network learning are presented.

Let us start the simulation by loading a database to the software. Loading a database and creating the nodes are explained above. Since the agent will also learn the structure of the network a longer database might be needed. A database of cases is created simulating the problem for each position set and for each action in the dog's action set. We have entered those five values into the sheep's dynamics and recorded the sheep's action with other five values. For the following simulation, this database is used.

After the nodes are created, we can generate the Bayesian network from the database. To generate the Bayesian network, we can either click on the Create button on the toolbar or choose the submenu Create in the Network menu. Then, the software displays a dialog box to specify the search algorithm. In the dialog box, the user can choose the search type, the score type and the distance measure type. The details of the dialog box and how to specify the search algorithm are given in knowledge discovery tutorial in Section 6.2.

In this section, we will give some simulation results obtained by applying different search algorithms. There are eight possible search algorithms in the software as shown in Table 7.2.

Table 7.2. Possible search algorithms in the IntelliAgent software

Algorithm	Search Type	Score Type	Distance Measure
1	Heuristic	MDL	Kullback-Leiber
2	Heuristic	MDL	Euclidean
3	Heuristic	MDL	LogLikelihood
4	Heuristic	Bayesian	-
5	Exhaustive	MDL	Kullback-Leiber
6	Exhaustive	MDL	Euclidean
7	Exhaustive	MDL	LogLikelihood
8	Exhaustive	Bayesian	-

Analyses of the search algorithms are presented in Section 4. In this section, we will not repeat the analysis of the search algorithms. We will present two simulations. The first one is a heuristic search with Bayesian scoring since it creates the network shown in Figure 7.5. The second search algorithm will be an exhaustive search with MDL score using Kullback-Lieber distance measure.

To create the first search algorithm, we have clicked heuristic and Bayesian score radio buttons on the dialog box. Then, the software started to generate a Bayesian network. We have run the search algorithm with the default complexity and accuracy but the resulting network had only three arcs. Then, we have increased the complexity by moving the sliding bar to the complexity. Finally, we have established a network with reasonable amount of arcs. Figure 7.11 shows the resulting Bayesian network.

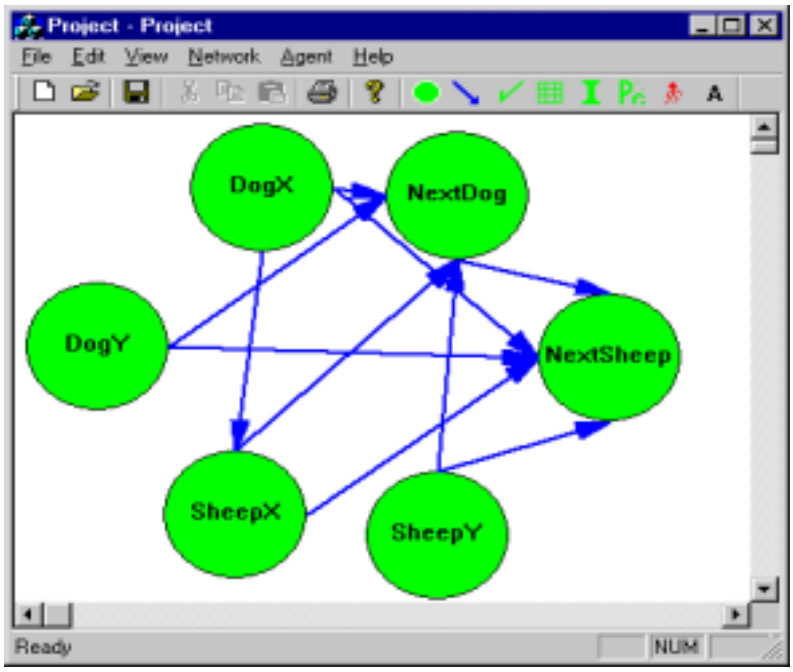


Figure 7.11. Bayesian network generated by heuristic search with Bayesian score.

As can be seen above, the number of arcs in the network is higher than the network created with known dynamics. The search algorithm discovered additional dependencies in the network along with the known ones. For example, in the previous case, there was no arc from the positions to the dog's next action. We have pointed out that there should be some relationship between the positions and the dog's action. Since the previous simulations were successful one might ask what benefit we will get by having more arcs in the network. We can answer the question by running the simulation with the network structure given in Figure 7.11.

We have started the simulation with same positions, (0,0) and (1,1) for the dog and the sheep, respectively. After the first run, the dog managed to herd the sheep to the pen by following the same paths shown in Figure 7.9. In the second run, we discovered that the sheep could not move out of the pen because the dog was not letting it go. In the first

case, the dog learned the same thing after three runs but with the network it learned the sheep's behavior from the database by putting additional dependencies in the network. In fact, the network shown in Figure 7.11 should be closer to the ideal system dynamics because it has connection between the dog's action and the positions. In short, the additional dependencies enabled faster learning for the dog.

As stated in Section 4, the heuristic search algorithm requires the ordering of the network nodes in the database. The exhaustive algorithm lifts this requirement by visiting more network structures during the search. In fact, it tries every possible arc in the network to improve the network score. Let us perform an exhaustive search with MDL score using Kullback-Lieber distance measure. Figure 7.12 illustrates the resulting Bayesian network.

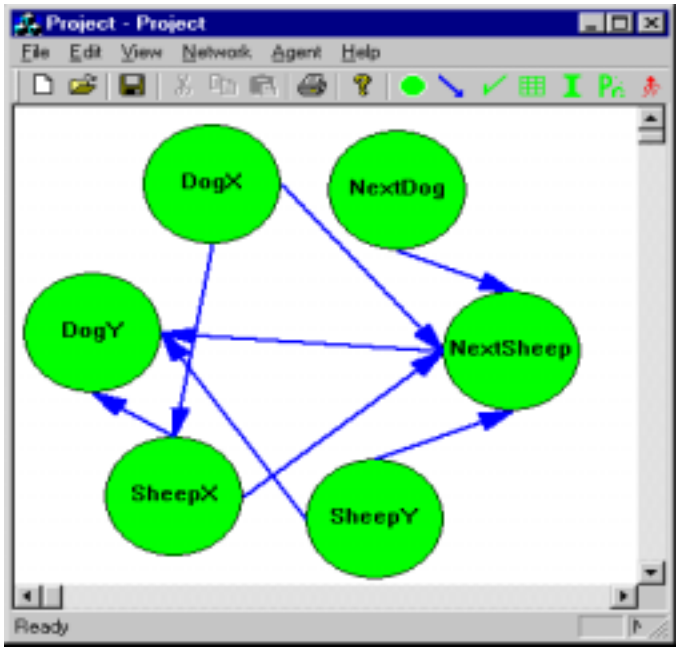


Figure 7.12. Bayesian network generated by exhaustive search with MDL score.

As can be seen above, the exhaustive search generated quite different network from the network created by heuristic search. Even though the network is quite different, it has

necessary dependencies representing the system dynamics in the known system dynamics case. The directions of some arcs are in opposite direction in the network. This does not cause any problem because inference can also travel in a backward direction. There are arcs between the sheep's position and the dog's position. The arcs from the sheep's positions to the dog's positions are logical because the sheep moves before the dog. The arc from DogX to DogY is in the opposite direction and may not be necessary. Since they will not increase the computational complexity too much, we can keep these arcs in the network.

We have run several simulations starting with the same positions. After the first simulation, the dog herded the sheep to the pen successfully. The simulation ended when the sheep is in the pen and the dog is at (1,1). The paths for the agents are shown in Figure 7.13.

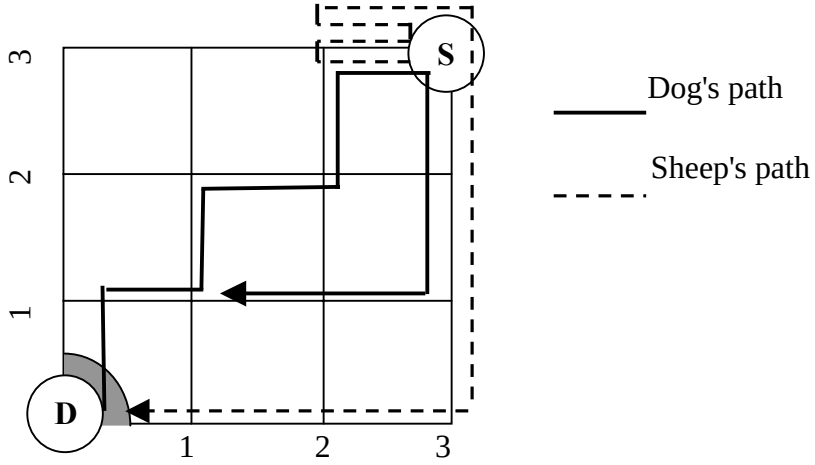


Figure 7.13. Paths of the agents for the first simulation.

As can be seen above, the sheep is going forward and backward until the dog is close enough to force the sheep out of the corner. The sheep tries to escape, but the dog moves diagonally to the sheep to keep the sheep at the corner while the dog gets closer to the sheep. When the dog is close enough to the sheep, the sheep has no choice but to move

out of the corner. The dog follows the sheep until the sheep is at (0,3) and the dog is at (1,3). Then, the sheep moves towards the pen. The dog does not go down to be just behind the sheep because the sheep may then go up and get away from the pen. Thus, the dog moves parallel to the sheep to keep the sheep down and move it to the pen. The sheep moves towards the pen until it is in the pen. When the sheep is in the pen, the simulation stops.

There are two important behaviors in the simulations. First, the sheep does not move away from the corner until the dog gets close. Second, the dog does not try to go behind the sheep when the sheep is at bottom of the area. The dog does not go behind the sheep any more because it estimates that the sheep may go up and get away from the pen. These two behaviors make it clear that the dog can estimate the sheep's behavior and act accordingly.

We have run couple of simulations to get a feeling about the dog's behavior. In one of the simulations, the dog and the sheep were caught in a loop where they repeat the same action for certain number of times. Then, the dog was able to break the loop and herd the sheep successfully. During the loop, the dog updates its network parameters with each action. After a certain amount of time, it reaches the knowledge of the loop and takes action to break it. This can be explained as forgetting or changing the agent beliefs. The dog had a certain knowledge about the sheep before the looping. When they start looping, the dog sees that the sheep is not doing what the dog expects. Therefore, after each step in the loop, the dog updates its belief about the sheep's behavior. When the number of steps in the loop reaches a certain value, the dog's belief about the sheep completely changes and the dog takes a different action to force the sheep out of the loop.

That is, the conditional probability of the sheep's action (S_N) and the expected utilities of the dog's actions are changed by experiencing the loop. When the expected utilities of the dog's actions are changed, the dog takes a different action and breaks the loop. Figure 7.14 illustrates how the agents changes its belief about the environment and takes actions accordingly.

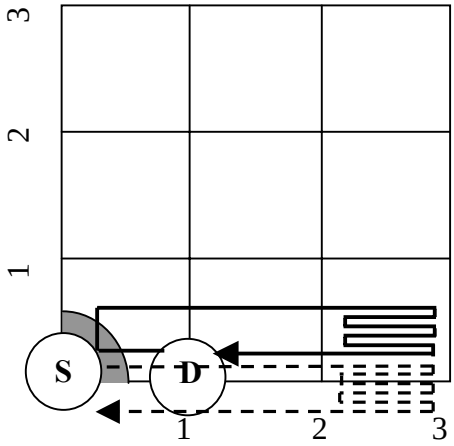


Figure 7.14. Changing belief of an intelligent agent.

In Figure 7.14, the follows the sheep to the corner. Then, they go back and forward between (3,0) and (2,0) for a while. Finally, the dog stops and waits for sheep to move to the pen. The dog learns the behavior of the sheep in time and fires a different action after certain amount of experience.

Let us explain the loop in terms of the dog's belief about the sheep and the expected utilities of the dog's actions. When the simulation is started, the dog moved towards the sheep by firing the action d_2. Then, the sheep moved away from the dog by taking the action s_1. Now, the dog is at (0,0) and the sheep is at (1,0). In the next step, the dog moves to the right by firing the action d_1. Then, the sheep also moves to the right to get away from the dog. Now, the dog is at (1,0) and the sheep is at (2,0). The following

equations present the conditional probability of the states of the sheep's decision node S_N given the dog's actions d_i and the expected utilities for the dog's actions.

$$P(S_N \mid D_N = d_0) = \{0,0,1,0,0\} \tag{7.32}$$

$$P(S_N \mid D_N = d_1) = \{0,0.25,0.72,0,0\} \tag{7.33}$$

$$P(S_N \mid D_N = d_2) = \{0,1,0,0,0\} \tag{7.34}$$

$$P(S_N \mid D_N = d_4) = \{0,0,1,0,0\} \tag{7.35}$$

$$\overline{U}(D_N) = \begin{bmatrix} 0.1999 \\ 0.4374 \\ 0.1666 \\ 0.0 \\ 0.1909 \end{bmatrix} \tag{7.36}$$

In above equations, we did not show the conditional probabilities for the action d_3 because it is physically impossible for the dog to move down. Thus, the action d_3 is not a possible action for the dog and the utility for this action is set to zero. As can be seen in Equation (7.36), the maximum expected utility is provided by the action d_1. Therefore, the dog fires the action d_1 and moves to the right. Then, the sheep also moves to the right to get away from the dog by taking the action s_1.

Now, the dog is at (2,0) and the sheep is at (3,0). The following equations present the conditional probabilities and the expected utilities for this setting.

$$P(S_N \mid D_N = d_0) = \{0,0,0,0,1\} \tag{7.37}$$

$$P(S_N \mid D_N = d_1) = \{0,0,1,0,0\} \tag{7.38}$$

$$P(S_N \mid D_N = d_2) = \{0,0,0,0,1\} \tag{7.38}$$

$$P(S_N \mid D_N = d_4) = \{0,0,0,0,1\} \tag{7.40}$$

$$\overline{U}(D_N) = \begin{bmatrix} 0.2185 \\ 0.3333 \\ 0.1852 \\ 0.0 \\ 0.2402 \end{bmatrix} \tag{7.41}$$

Using the equation (7.41), the dog fires the action d_1 and moves to the right because it provides the highest expected utility. Then, the sheep moves to the left by firing the action s_2 because it is physically impossible for the sheep to move to the right.

Now, the sheep is at (2,0) and the dog is at (3,0). Let us calculate the conditional probabilities and the expected utilities for this setting.

$$P(S_N \mid D_N = d_0) = \{0,0,1,0,0\} \tag{7.42}$$

$$P(S_N \mid D_N = d_2) = \{0,0.2,0.8,0,0\} \tag{7.43}$$

$$P(S_N \mid D_N = d_4) = \{0,0,1,0,0\} \tag{7.44}$$

$$\overline{U}(D_N) = \begin{bmatrix} 0.3333 \\ 0.0 \\ 0.4499 \\ 0.0 \\ 0.3090 \end{bmatrix} \tag{7.45}$$

As shown in Equation (7.45), the expected utilities for the actions d_1 and d_3 are zero because they are not physically possible dog actions. Therefore, only the conditional probabilities corresponding to the possible actions are shown above. After the expected utilities are calculated, the dog fires the action d_2 since it provides the maximum expected utility. Then, the sheep moves to the right again by firing the action s_1. This is basically where the loop starts in the simulation. The sheep and the dog moved back to the same locations after two actions.

The current locations for the dog and the sheep are (2,0) and (3,0), respectively. Let us present the conditional probabilities and the expected utilities for this setting one more time.

$$P(S_N \mid D_N = d_0) = \{0,0,0,0,1\} \tag{7.46}$$

$$P(S_N \mid D_N = d_1) = \{0,0,1,0,0\} \tag{7.47}$$

$$P(S_N \mid D_N = d_2) = \{0,0,0,0,1\} \tag{7.48}$$

$$P(S_N \mid D_N = d_4) = \{0,0,0,0,1\} \tag{7.49}$$

$$\overline{U}(D_N) = \begin{bmatrix} 0.2185 \\ 0.3333 \\ 0.1852 \\ 0.0 \\ 0.2402 \end{bmatrix} \tag{7.50}$$

These values are the same as the values shown two actions ago. Therefore, the dog takes the action d_1 and moves to the right. Then, the sheep fires the action s_2 and moves to the left.

Now, the dog is at (3,0) and the sheep is at (2,0). Thus, the dog and the sheep went back to the same location after two firing two actions. Let us examine the conditional probabilities and the expected utilities for this setting one more time.

$$P(S_N \mid D_N = d_0) = \{0,0,1,0,0\} \tag{7.51}$$

$$P(S_N \mid D_N = d_2) = \{0,0.36,0.64,0,0\} \tag{7.52}$$

$$P(S_N \mid D_N = d_4) = \{0,0,1,0,0\} \tag{7.53}$$

$$\overline{U}(D_N) = \begin{bmatrix} 0.3333 \\ 0.0 \\ 0.4099 \\ 0.0 \\ 0.3090 \end{bmatrix} \tag{7.54}$$

In Equation (7.52), the conditional probabilities are different than that of Equation (7.43). This shows that after firing two actions, the dog updated its belief about the sheep. The change in the conditional probability is reflected on the expected utility of the action d_2. The expected utility of the action d_2 is decreased from 0.4499 to 0.4099. Although the expected utility of the action d_2 is decreased, it is still the maximum. Therefore, the dog fires the action d_2. Then, the sheep fires the action s_1.

Now, the dog is at (2,0) and the sheep is at (3,0). The conditional probabilities and the expected utilities of this setting are the same as the values obtained two actions ago. Therefore, the dog fires the action d_1. Then, the sheep fires the action s_2. Now, the dog is at (3,0) and the sheep is at (2,0). During the simulation, the dog and the sheep comes to this setting three more times. The dog has to fire different action from the action d_2 to break the loop. Thus, we will examine only the conditional probability for the action d_2 and the expected utilities. The following equations show the conditional probabilities and the expected utilities for these three visits.

$$P(S_N \mid D_N = d_2) = \{0, 0.488, 0.512, 0, 0\} \tag{7.55}$$

$$\overline{U}(D_N) = \begin{bmatrix} 0.3333 \\ 0.0 \\ 0.3779 \\ 0.0 \\ 0.3090 \end{bmatrix} \tag{7.56}$$

$$P(S_N \mid D_N = d_2) = \{0,0.5634,0.4366,0,0\} \tag{7.57}$$

$$\overline{U}(D_N) = \begin{bmatrix} 0.3333 \\ 0.0 \\ 0.3523 \\ 0.0 \\ 0.3090 \end{bmatrix} \tag{7.58}$$

$$P(S_N \mid D_N = d_2) = \{0,0.67232,0.32768,0,0\} \tag{7.59}$$

$$\overline{U}(D_N) = \begin{bmatrix} 0.3333 \\ 0.0 \\ 0.3319 \\ 0.0 \\ 0.3090 \end{bmatrix} \tag{7.60}$$

As can be seen in Equations (7.56), (7.58), and (7.60), the expected utility for the action d_2 decreases after every visit. This is because the conditional probability for the action d_2 (the dog's belief about the sheep) changes after every visit. Finally, the expected utility of the action d_2 becomes lower that the expected utility of the action d_0. This is the point where the dog breaks the loop by firing the action d_0 and staying at the same location. Then, the sheep fires the action s_2 and gets away from the dog. The sheep gets closer to the pen. In the next step, the dog moves to the left by firing the action d_2. Then, the sheep moves to the left and gets into the pen as shown in Figure 7.14.

Let us examine how the conditional probability and the expected utility for the action d_2 change over time by summarizing the results shown above. The following equations summarize the conditional probability $P(S_N \mid D_N = d_2)$ for each visit to the locations (3,0) and (2,0) for the dog and the sheep, respectively.

$$P(S_N \mid D_N = d_2) = \{0,0.2,0.8,0,0\} \quad (7.61)$$

$$P(S_N \mid D_N = d_2) = \{0,0.36,0.64,0,0\} \quad (7.62)$$

$$P(S_N \mid D_N = d_2) = \{0,0.488,0.512,0,0\} \quad (7.63)$$

$$P(S_N \mid D_N = d_2) = \{0,0.5634,0.4366,0,0\} \quad (7.64)$$

$$P(S_N \mid D_N = d_2) = \{0,0.67232,0.32768,0,0\} \quad (7.65)$$

The conditional probability $P(S_N = s_1 \mid D_N = d_2)$ changes from 0.2 to 0.67232. Similarly, the conditional probability $P(S_N = s_2 \mid D_N = d_2)$ changes from 0.8 to 0.32768. This can be interpreted as the dog changes its belief about the next action of the sheep. At the beginning, it believes that sheep is most likely to fire the action s_2 because $P(S_N = s_2 \mid D_N = d_2)$ is higher than $P(S_N = s_1 \mid D_N = d_2)$. After five visits to the same positions, the conditional probabilities are changed drastically. Then, the probability $P(S_N = s_1 \mid D_N = d_2)$ became larger than $P(S_N = s_2 \mid D_N = d_2)$.

The change in the conditional probability has an affect on the expected utility for the dog's actions. The expected utility for the action d_2 was 0.4499 at the beginning of the loop. The expected utility of the action d_0 was 0.3333. After five visits to the same location, the expected utility of the action d_2 became 0.3319 while the expected utility of the action d_0 stayed the same. After the fifth visit, the utility of the action d_2 became smaller than the expected utility of the action d_0. As a result, the dog has fired the action d_0 and broken the loop after the fifth visit. The following graph shows how the expected utility of the actions d_2 and d_0 change over time.

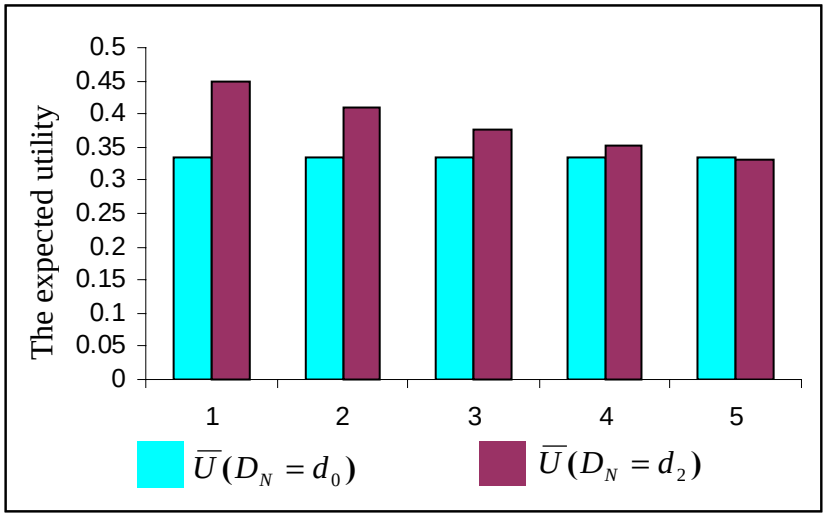


Figure 7.15. The expected utilities of the actions d_2 and d_0.

Breaking a loop is another example of the learning capability of the decision-theoretic intelligent agent design. The intelligent agent updates its world model according to its experience over time. After certain amount of time, the intelligent agent changes its behavior according to its experience. This is seen as similar to human belief. People do not change their beliefs suddenly. They tend to wait a certain amount of time before they change their mind. This is normal because if the agent changes its belief quickly, then, it will not have any memory or belief about the environment. It will take its actions according to the very latest experience, which could be a random one. The agents exhibit a humanoid belief process in the simulation. The details of the biological aspects of the agents are explained in Section 1.

In the simulations, the agents became stuck in a loop partly because their network structure is not good enough to take better actions and partly because the length of the database is not enough to provide accurate network parameters. After looping for a

while, the agent updates the network parameters and takes actions to move the sheep out of the loop. The agent records its experience into the database during its exploration. The agent updates the network parameters with its experience, but it cannot change the network structure automatically. The user can run the network creation algorithm to regenerate the network with the modified database. We have run the network search algorithm with the modified database and the new network structure has been generated. Figure 7.16 shows the resulting network.

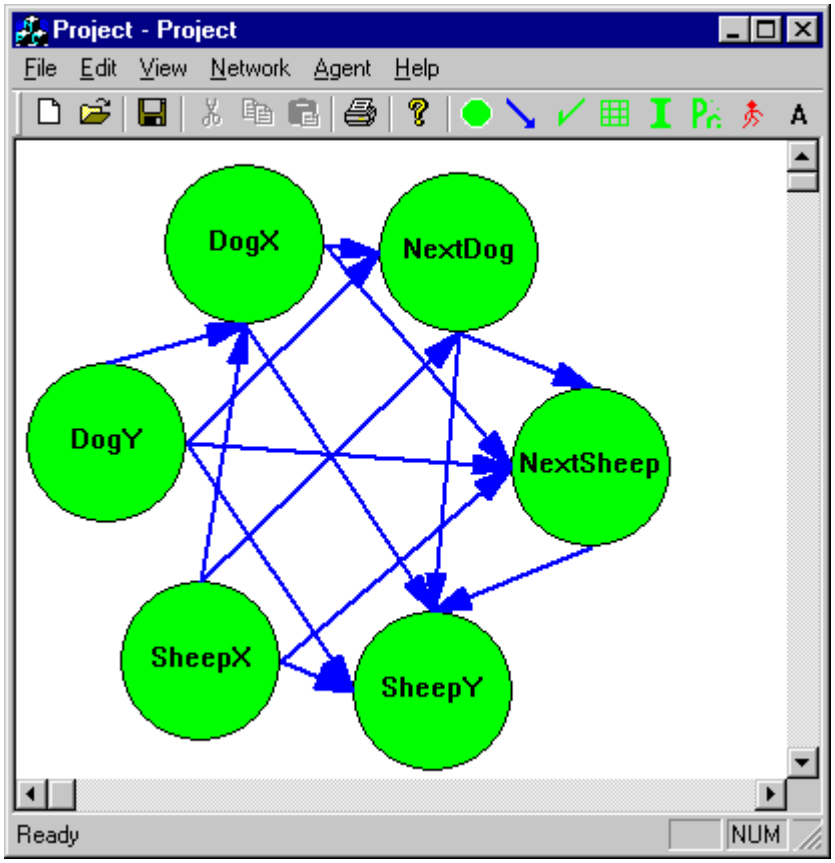


Figure 7.16. Network generated after the agent explored the environment.

In Figure 7.16, there are new arcs and some of the arcs have opposite directions if we compare the network with the network shown in Figure 7.12. Additionally, the network

has more arcs than the previous one. Since the network creation is not done by the agent there is no way of knowing the score of the network. Therefore, we do not know that this is a better network than the previous one. Simulations are performed with the new network to find out its behavior. There was no looping after running 10 simulations. The new network, in fact, is a better network than the previous one. Generating the network with the modified database has improved the performance of the network and the agent. In the future, the agent can automatically regenerate the network to see whether it can create better networks using its experience.

We have shown that the learning from experience causes intelligent agents to take better actions in time. After the learning, the intelligent agent establishes the task in a shorter time or in fewer steps. The next section will present the effectiveness of the proposed online Bayesian network learning by simulating the problem without learning.

In this section, we have performed simulations with different search algorithms and databases. In all simulations, the dog herded the sheep successfully. In some simulations, the agent had to explore the environment and learn more about the environment to correct its behavior. In short, we can conclude that if the Bayesian network structure is accurate enough, the agent can be successful with a limited initial knowledge. On the other hand, if the network structure is not accurate enough, then, the agent has to explore and learn the environment. In some cases, the agent may even need to regenerate the network structure using its experience.

Simulation results show that the online Bayesian network learning provides the learning from the experience and the self-organization in the intelligent agent model. The next section presents the effectiveness of the online Bayesian network learning by

simulating the problem without using the online Bayesian network learning in the proposed intelligent agent model.

7.3 The effectiveness of the online Bayesian network learning.

The online Bayesian network learning is the most important feature of the proposed intelligent agent model because the intelligent agents change their behavior after they learn from their experience. The more they learn about the environment and the other agents, the better they perform their task. For example, as shown in Figure 7.10, the intelligent agent (the dog) learns to keep the sheep in the pen by only taking two actions after two simulations. Before the learning, the dog put the sheep into the pen after several steps.

The proposed intelligent agent model learns its environment continuously. The learning causes the intelligent agents to change their belief about the environment and the other agents. The change in the belief causes a change in the agent's behavior because the agent takes different actions after the learning. Because the agents change their behavior according to the other agent's behavior and the environmental changes, we can claim that the agents take actions in coordination. The coordination between the agents provides the self-organization of the agents in a multi-agent system.

The following simulation results show that if the online Bayesian Learning is removed from the proposed agent design, the learning and self-organization capabilities of the agents diminish. The agent act according to its knowledge from the initial data. If initial data are not available, then the agent acts by assuming the uniform probability in

the Bayesian network. We will repeat the three simulations in the previous section without the online learning in the Bayesian network.

Let us simulate the dog & sheep problem by using the network in Figure 7.7 without the online learning. In this case, the system dynamics are known. As stated in the previous section, the intelligent agent learned to keep the sheep in the pen after two simulations. Figure 7.17 shows the simulation results without the online learning.

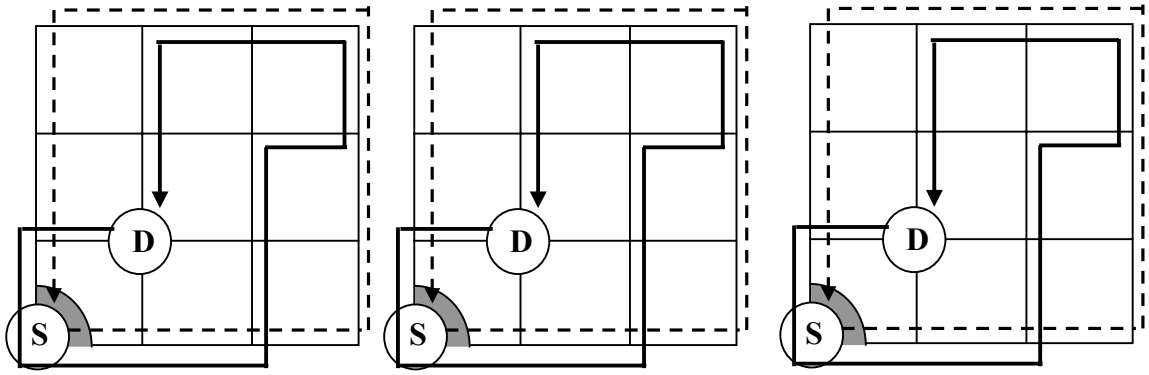


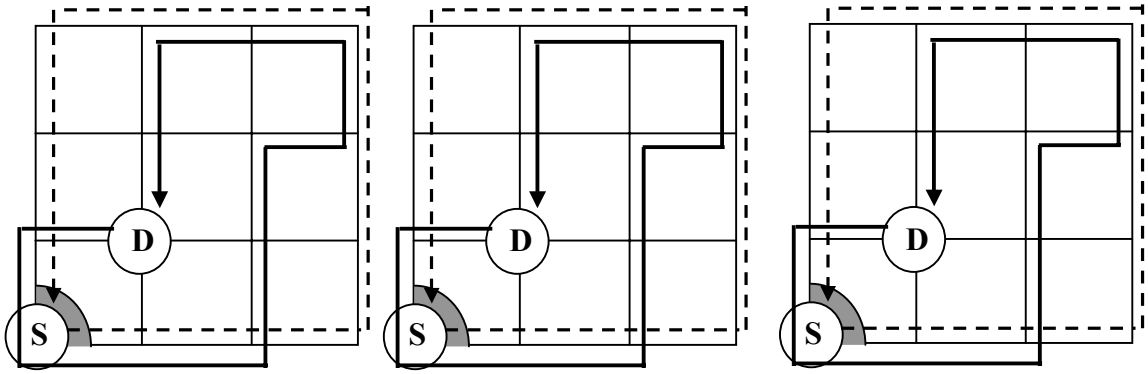
Figure 7.17. (a) is the first run, (b) is the second run, and (c) is the 10^{th} run.

As can be seen in Figure 7.17, the agents do not change their behavior over time. The dog always takes the same route to put the sheep in to the pen. It does not try other actions to do put the sheep into the pen in fewer steps. This is because the dog (intelligent agent) does not adapt its belief about the sheep. In other words, the dog does not learn from its experience over time. Therefore, we can conclude that the intelligent agent performs poorly when it does not learn from its experience.

Although the dog performs poorly, it still puts the sheep into the pen successfully. This is mostly because the expected utilities of the actions are determined by the utility function, which involves the distance between the sheep and the pen and the distance

between the dog and the sheep. Additionally, the structure of the network is good enough that the agents do not get in a loop during the simulations. When the network structure is not good enough, the agents will probably get in a loop and stay there forever.

Let us simuate the system by constructing the network from the data as in Figure 7.11. In this case, the heuristic search and the Bayesian score is employed to generate the network. Figure 7.18 shows the results for the first, the second, and the 10^{th} runs.



(a) (b) (c)

Figure 7.18. Simulations for unknown network structure and no online BN learning.

Figure 7.18 supports our claim that if the structure is good enough the dog can still put the sheep into the pen but the dog has to take several actions. The behavior of the dog is the same in each run since it does not learn during the simulation.

Finally, we will simulate the case where the network is not good enough as shown in Figure 7.12. In this case, the exhaustive search and the MDL score with Kullback-Leiber is employed to generate the Bayesian network. As shown in Figure 7.14, the agents get into a loop during the simulation. In Figure 7.14, the dog breaks the loop after certain number of steps because it continues to learn during the simulation. Figure 7.19 presents the simulation results obtained by canceling the online BN learning in the agent model.

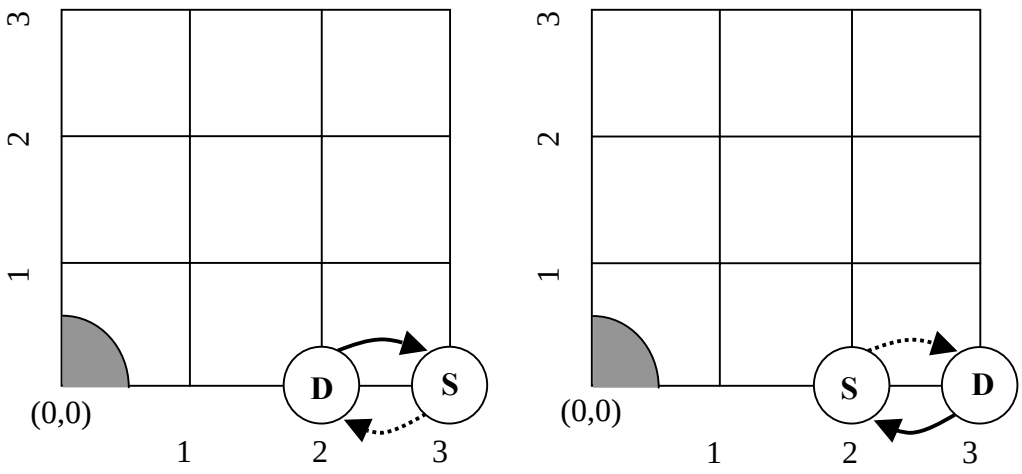


Figure 7.19. Looping in the simulations when the online BN learning is not applied

In Figure 7.19, only two steps are shown because the agents loop between (2,0) and (3,0). The simulation ends when the maximum number of steps is reached. If we do not limit the number of steps in the simulation, the simulation continues forever. The agents stay in the loop until the maximum number of step is reached. This is because the dog takes its actions according to its initial belief about the sheep and the environment. Since it does not learn the sheep's behavior and change its behavior accordingly, the dog takes the same action for the same setting. The sheep also moves to avoid the dog and tries to get away from the dog. In real life, the sheep will not take the same action forever because it will definitely be tired after certain amount of time. When the sheep is tired, it will behave differently and may break the loop. This is also valid for the dog. Even though the loop can be broken after certain amount of time, it will take more steps than that of the case where the intelligent agent learns from its experience.

In summary, the online BN learning provides the learning from the experience and the self-organization of the multi-agent system. If the online BN learning is not applied,

the intelligent agent cannot improve its behavior according to the other agent's behavior and the environmental changes. Since they cannot change their belief about the other agents, the self-organization of the agents cannot be accomplished without applying the online Bayesian network learning.

Simulation results show that if the online Bayesian network learning is not applied in the intelligent agent design, the self-organization ability of the intelligent agents cannot be accomplished. Additionally, the intelligent agents cannot adapt their behavior if the environment changes over time. In the next section, we will conclude this research and present possible future work.

CHAPTER 8

Conclusions

A decision-theoretic intelligent agent model has been proposed and applied to a real world problem. Bayesian networks and influence diagrams are combined with the help of utility theory to define the decision-theoretic intelligent agent. Learning in the agent is accomplished by introducing an online Bayesian network learning. An intelligent agent software, IntelliAgent, is written using Visual C++ and a C++ class library for the decision-theoretic intelligent agent design. Finally, The herding problem was successfully simulated by the help of the intelligent agent software.

Bayesian network learning is explored in Section 3. Design of the proposed online Bayesian network learning is explored in Section 4. The online Bayesian network learning has the following properties:

- Bi-directional learning (Bottom-up, Top-down)
- Combines supervised and unsupervised learning
- Online; learning is continuous
- Adaptive; network structure and parameters are updated by the new information
- Biologically inspired by the usage of Bayesian networks.

The online Bayesian network is combined with influence diagrams to create an intelligent agent as described in Section 4. Shoham's agent design is employed to design the decision-theoretic intelligent agent. An agent consists of *belief* (BN), *preference* (Utility - ID) and *capabilities* (action set - ID). In the decision-theoretic intelligent agent

design, two more levels, *sensors* and *actions*, are added to the agent design for practical purposes. The "sensors" level is responsible for gathering sensory information and passing it to the BN. The "actions" level is responsible for carrying the actions fired by the agent.

After designing the decision-theoretic intelligent agent, the IntelliAgent software is utilized to perform simulations of a real life problem. The IntelliAgent software user manual and tutorials are presented in Section 6. The software is Windows application software created by C++ class libraries written for the decision-theoretic intelligent agent design. Manual and automatic agent creation is possible in the IntelliAgent software.

As stated earlier, the decision-theoretic intelligent agent model is applied to a real time problem, a herding problem. The herding problem, also called Dog&Sheep problem, is analyzed for one sheep and one dog. The goal of the dog is to herd the sheep to the pen. The goal of the sheep is to avoid the dog. The simulations are performed on a $n \times n$ grid. The user can set the dimensions of the grid in the IntelliAgent software. Simulation results for 4×4 grid is presented in Section 7. The following is concluded after analyzing the simulation results:

- The dog (intelligent agent) herds the sheep to the pen successfully in every simulation. Simulations are run with different positions of the dog and the sheep.
- The intelligent agent shows learning capability by presenting behavioral change by observing the sheep (other agent and the environment). This is also defined as 'learning from experience". The dog takes its actions according to sheep's behavior. This is the self-organization property of the proposed decision-theoretic

intelligent agent model. Each agent is independent but takes its actions according to other agents' behaviors.

- The dog has human-like belief about its environment. The dog changes its belief about the environment including other agents in a humanoid way. For example, if the sheep and the dog are stuck in a behavioral loop, the dog does not change its behavior immediately. As shown in Section 7, the dog does not change it's believe about the sheep immediately. It waits for a couple of steps, then it changes its behavior by taking a different action for the same situation. This type of behavior is a standard human behavior. People do not change their belief abruptly after they have encountered an unusual event on a specific subject. They would like to experience the event several times. Then, they modify their believe on the subject. The decision-theoretic agent also modifies its belief by a certain amount of experience on an unusual environmental state.

After concluding the simulation results, the general properties of the decision-theoretic intelligent agent model are presented. The system analysis of the model is presented in Section 5. The following are the properties of the decision-theoretic intelligent agent system:

- The agent has bi-directional learning capability. It starts with an initial world model. It takes its actions according to the initial world model and its utility function (goal). The agent explores the environment and gathers information after taking actions. Then, it uses the information to modify its world model. As

explained in Section 5, it has bottom-up and top-down learning. In the literature, there are only a few learning methods that can claim to be bi-directional learning.

- The decision-theoretic intelligent agent also combines supervised and unsupervised learning. It takes its actions according to the initial world model and the utility function - unsupervised learning. Then, it modifies its world model by responses it gets from the world. While it explores the environment, it also exploits the environment by generating a world model - supervised learning.
- As stated in the context of system analysis, the decision-theoretic intelligent agent system can be seen as adaptive and as a feedback control system. The agent system combines feedback and adaptive control properties. There is a feedback loop because the agent observes the current environmental state, compares it with its goal state, and takes actions according to the difference between the current state and the goal state. The agent system is an indirect adaptive control system because the agent modifies the plant (world) model and the controller with the actual responses from the environment. Section 5.2 presents the details of the system analysis of the decision-theoretic intelligent agent system.
- As learning is biologically inspired, the decision-theoretic intelligent agent system is also biologically inspired. Each agent has sensors, belief, preference (goal), and capabilities, actuators as people do. People have sensors such as eyes, ears, and skin. They also have belief about environment, i.e., there is a college in town. They have a goal/goals, i.e., a college degree. They have capabilities such as walking, studying, and reading. Finally, they have actuators such as arms, legs, and brain. Therefore, a student is going to "walk" to the college, "get" an

application, "fill" the application, and "hand in" the application, "get in" to the college, "study" four years, and finally "get" a college degree. While the student is taking his/her actions, he/she is updating his/her world model. For example, when he/she applies to the college he/she sees the requirements and updates his/her knowledge about the college. Then, the student plans his college career and takes actions accordingly. The decision-theoretic intelligent agent takes its decision using the same decision structure as humans. That is why the decision-theoretic intelligent agent is said to be a biologically inspired agent model.

Hardware implementation of the problem is studied by using a mobile robot. An advanced mobile robot is purchased from Real World Interface, Inc. for the research. The robot has its own PC and a CORBA based software package, Mobility™. A C++ program is written using the CORBA based Mobility software to let the robot take actions. The program takes a text file that contains dog's action commands. Then, the commands are used to control the robots "rotate" and "translate" movements. The closed loop control algorithm is obtained by using the odometer of the robot. The control system steers the robot to a target x-y coordinate. The target coordinate is calculated by using the current position of the robot and the next command (next action) in the text file. The IntelliAgent software is modified to record the dog's actions into a text file.

In summary, the decision-theoretic intelligent agent model is successfully applied to a real life problem. The herding problem is simulated by the IntelliAgent software. Simulation results clearly reflect the behavior of the decision-theoretic intelligent agent. The next section presents possible future work.

CHAPTER 9

Future Work

Even though the results of the simulations are successful, additional work can be done in the IntelliAgent software and practical implementation of the proposed agent design. This section presents the future work in two parts; software and hardware.

The following is the list of possible improvement in the IntelliAgent software:

- The online Bayesian network learning in the IntelliAgent software can only handle complete databases. The proposed Bayesian network learning method can handle the cases where the network structure is known and the system variables are observable and where the network structure is unknown and the system variables are observable. The learning algorithm in the IntelliAgent software can be modified to handle the unobservable system variables. In other words, the software should handle databases with unknown values. Methods for learning from incomplete databases are explored in Section 3. For example, expectation maximization (EM) algorithm can be added to the software.
- The IntelliAgent software can be designed as Multiple Document Interface programs so that the user can run different simulations at the same time. That is, a problem can be run with different Bayesian network structures and a choice made for the best one.
- Bayesian network creation can be done by mouse operations or by using a database. As stated earlier, the user cannot edit the utility function in the agent since the utility nodes cannot be created visually. The reason is that the utility

node has many elements because it is dependent on four variables; *X* and *Y* coordinates of the sheep and the dog. On the other hand, a visual function editor can be placed into the program so that the user can edit the utility function of the agents.

- Similarly, the decision nodes are not also created as a rectangle in the software. They are shown as ellipsoidal because they can be treated as chance nodes after they are instantiated. A radio button can be placed into the parameters dialog box to specify the decision nodes after they are created as a chance node. This may help the user to understand the network better for complex network structures.
- Finally, the Edit menu can be activated by creating functions for copying, cutting, and pasting nodes and arcs. Adding a node and an arc is already in the C++ class library. One can easily incorporate those functions to the Edit menu elements.

After visiting the future work for software development, the following is the future work for the hardware aspect of the research.

- As explained in the conclusions, an offline hardware implementation is performed where the robot gets all the actions necessary during the simulation. There is no real time interaction between the robot and the sheep in this hardware implementation since the robot moving according to the simulations results. To make the system real-time. The IntelliAgent software can be recompiled with a CORBA interface to communicate with the robot. Then, the IntelliAgent software can tell the robot what to do. Similarly, the robot can send sensory information to the IntelliAgent software. Another mobile robot can be designed

to be a sheep with limited capabilities. Finally, the herding problem can be performed with these two robots.

- Since the second robot is not ready at the moment, an alternative can be to use the IntelliAgent for the sheep's behavior. The IntelliAgent knows the dynamics of the sheep and easily determined its actions after the dog's actions. In this case, the IntelliAgent software simulates the problem and sends "translate" and "rotate" commands to the mobile robot through Internet.
- Experimental CORBA interface is written for communication of two Windows programs over the net. Satisfactory results are obtained for the CORBA interface. The next step will be to establish an interface between a Linux program to a Windows program using CORBA. Since the robot's program is a CORBA based program, connecting that program to IntelliAgent software should not be difficult.
- A program is written for the robot to move to a certain location. The program takes three inputs; speed of translation, speed of rotation and the length of operation. By choosing the right values for these parameters, the program can move the robot to a certain location. The results obtained from this program are not accurate because the distance is calculated by the speed and the time. The robot may not obtain the same speed all the time because the surface friction may not be constant. Therefore, there is a need to find out whether the program can read translation values for the wheels. A study is being performed to find out how the robot can be moved accurately.

APPENDIX

A. Classes of the IntelliAgent Software

Four types of classes are used to create the IntelliAgent software, namely MFC classes, helper classes, visual C++ project classes, and ActiveX classes. The MFC classes will not be discussed here since there are standard classes in Microsoft Visual C++.

A.1 Helper classes

Helper classes can be presented in two categories, Bayesian network related classes and intelligent agent related.

A.1.1 Bayesian network related classes

There are six classes related to the Bayesian network creation; CNode, CArrow, CMatrix, CCptDialog, CParamDialog, and CNetGenerationDlg.

A.1.1.1 CNode

This class consists of the definition of a node and its functionality. The application programmer creates nodes in a network by creating an object of this class. The class has two constructors, CNode() and CNode(CRect nodeLocation). The second constructor creates a node in a desired location whereas the first one creates a default node with default parameters. Let us explore the functions in CNode class briefly.

AddParentOnCPT()

As stated earlier, the software has the ability of expanding the CPT of a node when a new arc is added or removed from the node. The AddParentOnCPT() function automatically expands the CPT matrix of a node, whenever the number of child or parent is changed by adding or removing an arc to the node.

Inference()

This function performs forward and backward inference after the network update is done.

BackwardInference()

BackwardInference() function performs backward inference by transmitting the evidence to its parents. This function also calls Inference() function on its parents. Thus, the inference travels through the network until a first level node or an end node is reached.

ForwardInference()

BackwardInference() works very similar to the BackwardInference() function. It performs forward inference by changing its children's probabilities and calling Inference() function on them.

OnCalculateBayesScore()

This function calculates the Bayesian score for the node. It uses the technique defined in Section 4. To calculate the score, the function either uses the node's probabilities or conditional probability table depending on the parents of the node.

OnCalculateLikelihood(int r)

This function works similar to OnCalculateBayesScore() except it calculates the likelihood score of the node given a data case. The resulting score value is used in MDL and LogLikelihood score calculations.

OnCalNodeLength()

This function calculates the length of the node. The length of the node is the number of element in the CPT. This value is then used to calculate the complexity of the network.

CreateNodeCPT()

This function creates the initial CPTs in the nodes when the software first creates the node. It can be considered as initial creation of the CPTs in the nodes.

OnUpdateCPT()

This function updates the CPT similar to AddParentOnCPT() function. This function is called when the user would like to update the network.

OnVisit()

OnVisit() functionrecords whether the node is visited on a path. This function is used to determine whether there is a cycle in the network or not. If the node is visited twice on a path, then the program decides there is a cycle.

OnDraw()

This function draws the nodes on the device context whenever the creation of the node is completed. The function draws an ellipsoid and fills the ellipsoid with green.

Serialize(CArchive &ar)

This is a serialization function for the node objects in the program. Whenever the user chooses to save the work, this function determines what needs to be saved in the node.

CNode class variables

int	m_InstantiatedState
int	m_EvidenceFlag
int	m_NumOfStates
int	m_NodeNumber
CMatrix	m_NodeCPTnum
CMatrix	m_NodeCPTdenum
CMatrix	m_Prob
CMatrix	m_NodeCPT
CUIntArray	m_Child
CUIntArray	m_Parent
CRect	m_NodeLocation
CString	m_NodeName
BOOL	m_VisitPass
BOOL	m_IsNodeVisited
BOOL	m_Modified

A.1.1.2 CArrow

CArrow class is designed to create arrow (arc) objects in the network. It has two constructor, CArrow() and CArrow(CPoint tail, CPoint head). The first one is the default constructor. The second constructor is designed to create the arrows mouse operations. The constructor takes two points as input and creates an arrow between the corresponding

nodes. It first finds in which nodes the points are. Then, it draws an arrow between those nodes. Let us explore the functions in the class.

*Draw(CDC *pDC)*

This is the function for drawing the arrow on the device context of the software. The function gets the pointer (*pDC*) to the device context (*CDC*).

Serialize(Archive &ar)

This function performs the serialization of the arrows in the network when the user saves the network.

CArrow class varables

CPoint	m_Head
CPoint	m_Tail
int	m_HeadNode
int	m_TailNode
CPoint	m_Arrow[3]

A.1.1.3 CMatrix

CMatrix class is designed to perform matrix operations in the inference calculations. The are also used as a value type. For example, the variable m_Prob in a node is a one-column matrix. Similarly, a CPT of a node can also be represented as a matrix, i.e. m_NodeCPT. There are four constructors for the class;

CMatrix(): Default constructor.

CMatrix(int row, int row): Creates a matrix "row" rows and "col" columns.

CMatrix(int row, int col, char Iden): Creates identity matrix.

There are 14 functions in the CMatrix class. The functions is presented with their brief functionalities.

AddColumn(int i)

This function adds "i" number of columns to a matrix. The function fills the new column with 0.5 because 0.5 is the initial probability for every variable.

AddRow(int i)

This function adds "i" number of row to a matrix. It also fills the new row with 0.5.

GetElement(int i, int j)

An element of a matrix can be obtained by this function. The function returns the element in *i*th row and *j*th column.

MaxElement()

This function finds the maximum element in the matrix. It returns the row of the maximum element as an integer.

OnZero()

All elements of the matrix becomes zero after this function is applied to a matrix.

operator()(int i, int j)

This function works as the same as GetElement(int i, int j) function. It returns the value in the *i*th row and *j*th column.

*operator *(const CMatrix & rhs)*

This is an override function of "*" operator for matrix multiplication. It multiples two matrix and returns the resulting maxtrix.

operator =(const CMatrix &rhs)

This is an override function of "=" operator. It replaces the matrix on the left with the matrix on the right.

SetElement(int row, int col, float x)

An element of a matrix can be replace with a new value. The value in row "row" and column "col" is replaced by *x*.

Supermultiply(Cmatrix &)

This is a special multiplication designed for handling multiplication in inference calculations. In inference calculations, multiplying two CPT is not equal to multiplying two matrices. Supermultiply function multiplies two CPT according to the inference calculation techniques.

Transpose()

This function takes the transpose of a matrix. It returns a matrix.

NumOfStates()

This is not a standard matrix operation. It is designed for determining the number of states in the nodes by going through a database. It looks for the maximum value in each column and put in a row matrix with the same number of columns.

CalculateJP(Cmatrix &test, int m)

This is also a special function for calculation joint probability of a data case in a database. It takes a database matrix and a row number (*m*), then, returns the joint probability of the data case in the *m*th row of the database. This function is used in probability calculations of the network variables.

CMatrix class variables

CArray <float, float>	m_CPT
int	m_col
int	m_row

A.1.1.4 CCptDialog

This is a dialog box class. It handles the CPT updating dialog box. The user enters the new values into this dialog. When the user clicks the OK button on the dialog the new value is placed into the CPT. There are two main functions in the class:

OnInitDialog()

This function handles the initialization of the dialog box. It displays the default parameters of the dialog box.

OnOK()

This is the main function in the dialog box. Whenever the OK button is clicked by a user, this function is called. The function puts the new value entered from the edit box into the CPT.

CCptDialog class variables

CString m_dEditCPT: Handles the edit box in the dialog box.

A.1.1.5 CParamDialog

This is the class that handles the Parameters dialog box. The user edits and updates node parameters using the functions in this class. The following paragraphs present the main functions in the CparamDialog class.

OnOK()

This is the function for OK button. This function wraps up all the changes the user made on the parameters dialog box. This function finalize the changes on the node parameters.

OnCheckProbSum(double initial)

This function checks whether the new probabilities have legal values or not. It checks whether the summation of the probabilities is 1 or not. It returns a Boolean value after the check.

OnInitDialog()

This is the initialization function. It determines the values in the dialog box when the dialog box appears.

OnListEnter()

This is the function for enter button on the dialog box. It takes the value in the Probabilities edit box and puts the value into the state probability list. It is used for entering the value of a state after increasing the number of states in the node.

OnSelchangeProbList()

This function is activated if the user clicks the left mouse button on one of the state probabilities. The function enables an edit text box and a push button (Update) under the probability list box. Then user can change the value in the state probability list.

OnListUpdateselitem()

This function is called whenever the user clicks the left mouse button on the Update push button. The function takes the value in the edit box and places it in the selected line in the probability list.

SetModifiedFlag()

This fucntion sets a flag after the parameters of the node are updated. Then, the software knows which nodes are updated. Finally, when the user clicks the network update button, the software update the network according to these flag values.

SetParameters(int states, CMatrix prob, CString name, int nodeNumber, CUIntArray &parent, CUIntArray &child, CMatrix cpt)

This function is designed to update the node parameters with the new values before the dialog box is closed.

OnDblClickMsflexgridCpt()

This is the function for editing the CPTs in the nodes. When the user double clicks the left button on a CPT value, this function is called. The function first gets the row and the column of the CPT value. Then, it activates the CPT updating dialog box. Finally, the value in the CPT updating dialog box is entered to the CPT table on the parameter dialog box.

UpdateDialogCPT()

This function puts the CPT table on the parameter dialog box into the node's CPT table. It also updates the CPT if the user has changed the number of states in the node. Increasing the state number increases the number of row in the CPT.

CParamDialog class variables

int	m_States
int	m_NodeNumber
CString	m_Probabilities
CString	m_Name
CString	m_dChangeListItem
CStringArray	m_ProbabilityList
CMatrix	m_dCPT
CUIntArray	m_ChildList
CUIntArray	m_ParentList
CListBox	m_ListControl
CMSFlexGrid	m_dMSFlexGridCPT
CCptDialog	m_EditCPTDialog

A.1.1.6 CNetGenerationDlg

This is also a dialog box class. It handles the network generation dialog box. The user can specify the properties of the network search algorithm using the function of this class. As stated earlier, there are seven radio buttons concerning the choices the user can make in the dialog box. Each radio button has a function attached to it.

OnRadioHeuristic()

This function is called when the user chooses the Heuristic radio button. The function sets the type of search algorithm by setting SEARCH_ALGORITHM to HEURISTIC constant integer. The variable SEARCH_ALGORITHM is a global variable in the document class. When the search algorithm starts, the software checks this value and decides which search algorithm needs to be used.

OnRadioExhaustive()

This function works as the same as OnRadioHeuristic() except it sets the variable SEARCH_ALGORITHM to EXHAUSTIVE constant integer.

OnRadioMdl()

This function is called when the MDL radio button is clicked. The fucntion sets the document global variable SCORE_TYPE to 0. The software checks the variable SCORE_TYPE to decide the score type. Score type can take three values 0, and 1 for MDL and Bayesian scores respectively.

OnRadioBayesian()

This function sets the variable SCORE_TYPE to 1 to choose the Bayesian scoring.

OnRadioKl()

This function sets the document global variable DISTANCE_TYPE to 0 to choose Kullback-Lieber distance measure for the score calculations.

OnRadioEuclidean()

Similarly, this function sets the document global variable DISTANCE_TYPE to 1 to choose Euclidean distance measure for the score calculations.

OnRadioLoglikelihood()

This function sets the distance measure type to Log-Likelihood by setting the variable DISTANCE_TYPE to 2.

OnInitDialog()

This is the initialization function for the network generation dialog box. It sets the default values on the dialog box.

CNetGenerationDlg class variables

CSliderCtrl	m_SliderCtrl
int	m_SliderValue

A.1.2 Agent related classes

There are two agent-related classes; CAgent and CAgentDlg. First one handles the agent object creation. The second handles the agent creation dialog box.

A.1.2.1 CAgent

This class has only one constructor, CAgent(). It creates an agent at (0,0) location with a NULL name. There is only one function in the class, Draw (CDC *pDC). It draws an agent on the screen at a specified location.

A.1.2.1 CAgentDlg

The locations and names of the agents can be entered form the agent creation dialog box. Additionally, simulation properties can be set with this dialog box. There are five main functions in the class. Names and the locations of the agents can be entered into the corresponding edit boxes on the dialog box. The dialog box automatically sets the variables of the class using those values.

OnInitDialog()

This function handles the initialization of the dialog box.

OnOK()

This function finalizes the parameters edited in the dialog box. It closes the dialog box.

OnRadioStepsim()

This function is called when the user clicks on the Step push button on the dialog box. The function sets a variable in the document class to run the simulation step by step.

OnRadioContsim()

Similar to the previous function, it is called by pushing the Continuous button on the dialog box. The function sets a variable in the document class to run the simulation continuously.

OnButtonTraining()

This is the function for Training push button. The function enables an edit box and a static text on the dialog box to let the user enter the number of training steps.

CAgentDlg class variables

int	m_dAgentLocX
int	m_dAgentLocY
int	m_dAgent
int	m_dTrainingStep

We have completed the helper classes used in the IntelliAgent software creation. The following section explores the visual C++ project classes, CProjectDoc and CProjectView. These two classes are responsible for network calculations, simulation and visual parts of the software.

A.2 Visual C++ project classes

When an application program is written in visual C++, the program creates four classes automatically, mainframe class (CMainFrame), application class (CprojectApp), document class (CProjectDoc), and view class (CProjectView). Usually, the programmer does not edit the mainframe and the application classes. Thus, they are not discussed here. The document class contains all the data handling and the calculations of the program. Finally, the view class handles the visualization of the program. In this section, the focus will be on the functions added into the document class and the view class.

A.2.1 Document class

The document class has member functions and member variables to perform necessary calculations in the decision-theoretic intelligent agent systems. The functions and variables will be discussed in terms of their functionality in the program. Programming details will not be presented here.

A.2.1.1 Document class member functions

The following list is the functions in the document class with their brief definitions. Since the actual source code can be obtained from the author, the details of the functions are not presented here.

BOOL OnIsNetworkCyclic()

The function returns TRUE if the Bayesian network has cycles.

void OnCreateDatabase()

The function reads a database into the program and puts the database in a matrix form. The program uses this matrix for the network calculations.

*void OnCreateDatabase(CStdioFile *f, CMatrix dataMatrix)*

The function prepares the database for saving. This is important especially if new data is collected from the environment. The function saves the database along with the Bayesian network. The matrix *dataMatrix* represents the database for the system. It is a global matrix called *testTable* throughout the program.

void OnRenewOrUpdateNetwork()

After a change is made on the network, this function updates or renews the network according to the changes.

CMatrix CreateNodeProbability(int i)

This function calculates the probabilities of the node *i*. It returns a matrix containing the probabilities of the node.

long double Gamma(unsigned int i)

Gamma functions are necessary to calculate the Bayesian score of a network. This function calculates the gamma function for a given integer and returns the results as long double.

void RemoveAllArrows()

This function removes all the arcs (arrows) in a Bayesian network. It also updates the network after the arcs are removed.

void OnNetworkGenerate()

This is the main function for the network generation. It generates a Bayesian network according to the network creation parameters such as the search type, the score type and the distance measure type.

float OnCalculateActLikelihood(int i)

This function calculates the likelihood of the conditional probabilities in a node. The function returns a *float*. The results produced by this function is then added together to calculate the over all likelihood of the network.

float OnCalNetworkScore()

This function calculates the score of a Bayesian network depending on the distance measure type such as Kullback-Lieber and Euclidean.

void OnPositionAgentsRandomly()

This function is used in the simulation of the intelligent agent system, namely Dog&Sheep. It is used to train the agent by locating agents randomly and running the simulations. This function places the agents on the environment randomly.

int createRandomNumber(int i)

This function generates a random integer between 0 and *i*. It is used in above function to place the agent randomly.

CAgent GetAgent(int i)*

The function returns a pointer to the agent object at the specified location (*i*) in the agent object array (m_*oaAgents*).

*CAgent * AddAgent(int X, int Y)*

This function creates and agent located at *X*, *Y*. Then, it adds the agent to agent object array and returns a pointer to the agent.

void UpdateDogSheepPos()

This function updates the locations of the dog and the sheep after they make a move.

void OnCreateNextPosTable()

This function creates a table for the next position for the sheep and the dog. The table consist of the changes in the *x* and *y* direction for all possible actions of the dog and the sheep. UpdateDogSheepPos() function uses this table to determine the new coordinates of the dog and the sheep.

BOOL OnLegalMove(int x, int y, int m)

The function checks whether the actions of the agents are legal by comparing their coordinates with the problem dimension (the dimension of the grid). It returns TRUE if the action is legal and returns FALSE otherwise.

float OnDogSheepUtility(int i)

This function calculates the expected utility of a specified action. The function takes an integer denoting the sheep's next move after the dog's next move. Then, the function calculates the new position of the sheep and calculates the corresponding utility.

void OnSetEvidence(int node, int state)

This function set evidence on a node. It takes the node number and the state to be instantiated. Then, it sets the specified state value to 1 and the rest of the state values to zero.

void OnRecordNewEntry()

This function records a new entry into the database after the sheep and the dog completed one action. The function also updates the network parameters with the new data.

void OnCalculateNewSheepPos(int choice)

This function calculates the sheep's next coordinates after the sheep moves. The integer *choice* presents the sheep's next action. The function uses the next position table created by the OnCreateNextPosTable() function.

int OnDecision(CMatrix &values)

This function determines which action the dog will take after the expected utilities are calculated for each action. The matrix *values* consists of the expected utilities of the dog's actions. The function finds the maximum expected utility in the matrix and returns its index. The index illustrates the action with the highest expected utility.

CMatrix OnValues(int dnode, int unode)

This function calculates the expected utilities for the actions in the node *dnode*. The function fires each action in *dnode* and calculates the state probabilities of the node

unode. Then, it calculates the expected utility of the system using these probabilities and the action fired. The process is repeated for each action and the expected utilities are placed into a matrix. Finally, the function returns the expected utility matrix.

void CreateJPT()

This function calculates the joint probability distribution from the database.

CMatrix CreateJPT(CUIntArray &list)

This function calculates the joint probability for a given data case. For example, it can calculate $p(A = 0, B = 0, C = 1)$ for a database with three variables.

CMatrix CreateCPT(int node, CUIntArray &list)

This function creates a conditional probability table for a node with a specified parents. The function takes an integer for the node number and an integer array for the numbers of the parent nodes.

CMatrix CreateCPT(int i, int j)

This function calculates a conditional probability table for given variables. It takes two integers for node numbers for the variables. For example, in $P(A \mid B)$, the integers *i* and *j* represent the variable *A* and *B*, respectively.

CMatrix CreateCPMatrix(CUIntArray &list)

This function is similar to the previous function. It can calculate the CPT for more than two variables. It takes an array of integers for the node numbers. In the array, the first element represents the first variable in a conditional probability equation. For $P(A \mid B, C)$, the first element of the integer array is filled with the node number of the variable *A*. Then, the node numbers of the variables *B* and *C* are placed into the array.

void CalFirstLevelProbs()

This function calculates independent probabilities for the first level nodes. A first level node is a node without any parents. These nodes do not have a conditional probability table.

*CNode * AddNode(CRect nodeLocation)*

This function adds a node to the network at the location determined by the *nodeLocation* variable. The function creates a node, adds it to the node object array (*m_oaNodes*), and returns its pointer.

void SetNode(int nodePos, CString name, int states, CStringArray &prob, CMatrix cpt)

This function sets the name, the position, number of states, the state probabilities, and the conditional probability table of a node. The *nodePos* represents the position of the node in the node object array. The function sets the parameters of the node using the variables *name*, *states*, *prob*, and *cpt*.

*CNode * GetNode(int nIndex)*

This function returns a pointer to a specified node. The function takes an integer as an index to get the corresponding pointer value from the node object array.

int GetNodeCount()

This function calculates the number of nodes in the network and returns the results as an integer.

BOOL AddArrow(int i, int j)

This function adds and arrow (arc) to a Bayesian network. It takes the node numbers of the parent node (*i*) and the child node (*j*). If the function is successfully adds the arc to the network it returns TRUE. Otherwise, it returns FALSE and does not modify the arc object array (*m_oaArrows*).

void RemoveArrow(int i, int j)

This function removes the arc from the node *i* to the node *j*. It also removes the arc from the arc object array.

*CArrow * GetArrow(int nIndex)*

This function returns a pointer to a specified arc (arrow). The function takes an integer *nIndex* as the index of the specified arrow in the arrow object array.

int GetArrowCount()

This function calculates the number of arrow in the network and returns the results as an integer.

*CArrow * AddArrow(CPoint tail, CPoint head)*

This function creates an arrow by using two points; *tail* and *head*. The program first finds the nodes by comparing whether the points on a node or not. After determining the corresponding nodes for the points, the function calls *AddArrow(int i, int j)* function to create the corresponding arrow (arc).

void UpdateView()

This function updates the device context of the program after a modification is made in the network.

void CreateNodes()

This function creates nodes after a database is read into the program. It generates the names of the nodes from the first line of the database. The function also creates the independent probabilities for the nodes.

CMatrix CreateNodeProb(int i)

This function calculates the probabilities of a specified node. It takes an integer for as the node number and returns a matrix with the node probabilities.

void GenerateNetwork()

This function is called when the user clicks the Create button on the toolbar. It calls the OnNetworkGenerate() to generate the network.

void CreateTestTable()

This function creates a table from the database. It finds the number of states of each node. Then, it creates a table that contains all the possible combinations of the states. This table is then used in the network calculations as a reference.

CMatrix Parents(int x)

This function finds the parents of the node *x*. Then, it creates a vector using the probabilities of the parent nodes. The function returns this vector as a matrix.

void UpdateNodeCPT()

This function updates the CPTs tables of the nodes in the network. If a change is made to the network, this function is called and the CPTs are updated accordingly.

void ModifiedFlagChild(int x)

This is a function for specifying the nodes that need update after a change is made on a node. The function sets a flag in its child nodes. When the update network command is called, the program checks the flags in each node before it updates the node parameters.

A.2.1.2 Document class member variables

The following list illustrates the member variables of the document class. These are also called global variables since they can be reached from any function in the document class.

```
int             SEARCH_METHOD;
int             DISTANCE_TYPE;
int             TRAINING_STEP;
int             COMPLEXITY;
int             ACCURACY;
int             SCORE_TYPE;
int             caseCounter;
BOOL            IsCyclic;
BOOL            m_Continue;
CMatrix         NextPosTable;
CMatrix         g_JPT;
CMatrix         testTable;
CMatrix         cumStates;
CMatrix         States;
CMatrix         caseTable;
CObArray        m_oaNodes;
CObArray        m_oaArrows;
CObArray        m_oaAgents;
CUIntArray      lastCase;
CStringArray    NodeNames;
```

A.2.2 View class

The view class handles the visualization of the software such as updating the workspace, mouse operations, drawings, painting, and brushing. The following sections explore the member functions and the member variables of the class.

A.2.2.1 View class member functions

The following is the list of view class member functions and their brief definitions.

void OnDrawAgentRegion()

This function draws the problem domain for the Dog & Sheep simulation. It draws an $n \times m$ grid depending on the number of states in the variables DogX, Dog Y, SheepX, and SheepY. If the user increases the number of states in the variable, the software updates the problem domain accordingly.

BOOL OnNoRelation(unsigned int node1, unsigned int node2)

This function returns TRUE if there is an arc between the nodes; *node1* and *node2*.

The function is used during the arc additions to the network. The purpose of the function is to avoid the creation of the same arc twice.

int OnInANode(CPoint point)

This function takes a point and finds whether the point on a node or not. If the point is on node, the function returns the number of the node. Otherwise, it returns -1.

void OnShowParam(int x)

This function is called by the Parameters toolbar button. It displays the parameters of a certain node. The node number is entered to the function as an integer.

afx_msg void OnNetworkArc()

This function is called when the user clicks on the Arc toolbar button. The function sets the *drawingElement* member variable to ARC. The ARC is a constant integer set to 2.

afx_msg void OnNetworkNode()

This is similar to the OnNetworkArc() functions. The function sets the *drawingElement* variable to NODE. The NODE is defined as a constant integer set to 1. When the user creates a network with mouse drag and drop operations, the program checks the value of the *drawingElement* variable to determine what to draw.

afx_msg void OnLButtonDown(UINT nFlags, CPoint point)

This function is called when the user clicks the left mouse button. Many visual operations is done by this function such as drawing and object, moving a node, choosing a node or an arc, choosing a toolbar button operation.

afx_msg void OnLButtonUp(UINT nFlags, CPoint point)

This function is called when the user releases the left mouse button. This function is also used in many operations, i.e., dropping a selected node to a desired location.

afx_msg void OnMouseMove(UINT nFlags, CPoint point)

This function is called when the mouse is moved around. Most of the time, this function and the previous mouse operation functions work together. For example, To move a node to a certain location, the OnLButtonDown(UINT nFlags, CPoint point) function selects the node, this function moves the node, and finally the OnLButtonUp(UINT nFlags, CPoint point) function releases the node on a desired location.

afx_msg void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar pScrollBar)*

This is the horizontal scrolling function. It updates the coordinates of the screen when the user scrolls horizontally.

afx_msg void OnVScroll(UINT nSBCode, UINT nPos, CScrollBar pScrollBar)*

This is the vertical scrolling function. It updates the coordinates of the screen when the user scrolls vertically.

afx_msg void OnContextMenu(CWnd pWnd, CPoint point)*

This function is called when the right mouse button is clicked. It displays the context menu on the screen. It lets the user choose the submenu items Set Evidence and Parameters.

afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point)

This function is called when the left mouse button is clicked twice. If the left mouse button is double clicked on a node, the function displays the parameters of the node.

afx_msg void OnNetworkParameters()

This function is called when the user clicks on the Parameter toolbar button. It also displays the parameters of a specified node.

afx_msg void OnRButtonDown(UINT nFlags, CPoint point)

This function is called when the right mouse button is clicked. It displays the context menu by calling the *OnContextMenu(CWnd* pWnd, CPoint point).*

afx_msg void OnSetevidenceState0()

To invoke this function, the user first chooses the Set Evidence menu item in the context menu. Then, the user chooses the State0 submenu item. The function sets the probability of the state0 to 1. It also sets the probability of the other states to zero.

The software has seven more functions similar to the above function to handle the instantiation of a node with at most eight states. For the nodes with more states, the user can display the parameters of the node and change the probabilities from the parameters dialog box.

afx_msg void OnNetworkAgentLoc()

This function is called when the user clicks the Create Agent toolbar button. The function displays a dialog box for agent creation. The user sets the parameters of the agent creation dialog box. As soon as the user clicks OK button on the dialog box, the function creates an agent at a specified location.

afx_msg void OnNetworkCreate()

This function is called when the Create toolbar button is clicked. The function displays the network creation dialog box as presented in Chapter 6. The user can the parameters

of the dialog box. As soon as the user clicks OK button on the dialog box, the function transfers the parameters to the document class and call the *OnNetworkGenerate()* function from the document class.

A.2.2.2 View class member varibles

```
HCURSOR          cross;
HCURSOR          arrow;
CPoint           start, old;
BOOL             started;
int              moveNode;
int              drawingElement;
CRect            movingNodeLoc;
CNetGenerationDlg    m_dNetGenerationDlg;
CPoint           prevPoint;
CParamDialog     m_dParamDialog;
CAgentDlg        m_dAgents;
int              hScrollPos,
int              vScrollPos,
int              lineSize,
int              vPageSize,
int              hPageSize,
int              maxPos;
```

A.3 ActiveX classes

ActiveX classes are helper classes created by various programmers under Microsoft visual C++. In IntelliAgent software, the ActiveX control "MSFlexGrid" is employed to present the conditional probability tables in the nodes. A general definition of the ActiveX is presented here. The MSFlexGrid ActiveX has four classes for row and column operations (CRowCursor), picture operations (CPicture), fonts (COleFont), and the main class (CMSFlexGrid).

References

[1] F. V. Jensen, *An Introduction to Bayesian Networks*. London, UK: University College London Press, 1996.

[2] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann, 1988.

[3] D. Heckerman, "A tutorial on learning Bayesian networks," Technical Report MSR-TR-95-06, Microsoft Research, 1995.

[4] Y. Shoham, Agent-oriented programming," *Artificial intelligence*, vol. 60(1), pp. 51-92, 1993.

[5] J. Pearl, "Bayesian networks", in M. Arbib (Ed.), *Handbook of Brain Theory and Neural Networks*, MIT Press, pp. 149-153, 1995

[6] J. Pearl, "Bayesian networks," Technical Report R-246, MIT Encyclapedia of the Cognitive Science, October 1997.

[7] F.V. Jensen, "Bayesian network basics," *AISB Quarterly*, vol. 94, pp. 9-22, 1996.

[8] S. Noh and P. J. Gmytrasiewic, "Coordination and belief update in a distributed anti-air environment," in *Proceedings of the 31st Hawaii International Conference on System Sciences*, vol. V, pp. 142-145, Los Alamitos, CA: IEEE Computer Society, January 1998.

[9] M. Ramoni and P. Sebastiani, "Parameter estimation in Bayesian networks from incomplete databases," Technical Report KMi-TR-57, Knowledge Median Institute, The Open University, November 1997.

[10] G. Schwarz, "Estimation the dimension of a model," *Annals of Statistics*, vol. 6, pp. 462-464, 1978.

[11] W. Lam and F. Bacchus, “Learning Bayesian belief networks: an approach based on the MDL principle,” *Computational Intelligence*, vol. 10, pp. 269-293, 1994.

[12] N. Friedman, M. Goldszmidt, D. Heckerman, and S. Russell, “Challenge: Where is the impact of the Bayesian networks in learning?” In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, pp.10-15, 1997.

[13] D. Koller, *Artificial intelligence: Knowledge representation and reasoning under uncertainty*, Course material (CS 288), winter 1999, available at http://www.stanford.edu/class/cs228/index.html.

[14] J. H. Kim and J. Pearl, “A computational model for combined causal and diagnostic reasoning in inference systems,” in *Proceedings IJCAI-83*, Karlsruhe, Germany, pp. 190-193, 1983.

[15] S. Russell, “Learning agents for uncertain environments (Extended abstract),” in *Proceedings of the COLT-98*, Wisconsin: ACM Press, pp. 101-103, 1998.

[16] S. Russell, J. Binder, and D. Koller, “Adaptive probabilistic networks,” Technical Report UCB//CSD-94-824, July 1994.

[17] D. Nilsson and F. V. Jensen, “Probabilities of future decisions” Research Report R-97-2007, Dept. of Mathematics, Aalborg University, Denmark, June 1997.

[18] J. Pearl, “Reverend bayes on inference engines: A distributed hierarchical approach,” in *Proceedings AAAI National Conference on AI*, Pittsburgh, PA, pp. 133-136, 1982.

[19] N. Friedman, K. Murphy, and S. Russell, “Learning the structure of dynamic probabilistic networks,” in G.F. Cooper and S. Moral (Eds.), *Proceedings of*

Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI '98), San Francisco, CA: Morgan Kaufmann, 1998.

[20] T. S. Verna, "Causal networks: Semantics and expressiveness, *Proceedings of the Third Workshop on Uncertainty in Artificial Intelligence*, pp. 352-359, 1987.

[21] G. F. Cooper and E. Herskovits, "A Bayesian method for constructing Bayesian belief networks from databases," in *Proceedings of the Conference on Uncertainty in AI*, pp. 86-94, 1990.

[22] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Journal of the Royal Statistical Society, Series B*, vol. 50(2), pp. 157-224, 1988.

[23] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society, Series B*, vol. 39, pp. 1-38, 1977.

[24] B. Theisson, C. Meek, and D. M. Chickering, and D. Heckerman, "Learning mixtures of Bayesian networks," in G.F. Cooper and S. Moral (Eds.), *Proceedings of Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI '98)*, San Francisco, CA: Morgan Kaufmann, 1998.

[25] M. Ramoni and P. Sebastiani, "Efficient parameter learning in Bayesian networks from incomplete databases," Technical Report KMi-TR-41, Knowledge Median Institute, The Open University, January 1997.

[26] M. Ramoni and P. Sebastiani, "Discovering Bayesian networks in incomplete databases," Technical Report KMi-TR-46, Knowledge Median Institute, The Open University, March 1997.

[27] P. Sebastiani and M. Ramoni, “Bayesian inference with missing data using bound and collapse,” Technical Report KMi-TR-58, Knowledge Median Institute, The Open University, November 1997.

[28] M. Ramoni and P. Sebastiani, “Learning conditional probabilities from incomplete data: An experimental comparison,” Technical Report KMi-TR-64, Knowledge Median Institute, The Open University, July 1998.

[29] N. Friedman, “The Bayesian structural EM algorithm,” in G.F. Cooper and S. Moral (Eds.), *Proceedings of Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI ’98)*, San Francisco, CA: Morgan Kaufmann, 1998.

[30] D. B. West, *Graph Theory*. New Jersey: Prentice Hall, 1996.

[31] K. G. Olesen, S. L. Lauritzen and F. V. Jensen, “aHUGIN: A system for creating adaptive causal probabilistic networks,” in *Proceedings of the Eighth Conference on Uncertainty in AI (UAI ’92)*, Stanford, CA: Morgan Kaufmann, 1992.

[32] D. Spiegelhalter, P. Dawid, S. L. Lauritzen, and R. Cowell, “Bayesian analysis in expert systems,” *Statistical Science*, vol. 8, pp. 219-282, 1993.

[33] J. Pearl, “Constraint-propagation approach to probabilistic reasoning,” in L. M. Kanal and J. Lemmer (Eds.), *Uncertainty in Artificial Intelligence*, North-Holland, Amsterdam, pp. 357-288, 1986.

[34] G. Cooper and E. Herskovits, “A Bayesian method for induction of probabilistic networks from data,” *Machine Learning*, vol. 9, pp. 309-347, 1992.

[35] D. Heckerman, D. Gieger, and M. Chickering, “Learning Bayesian networks: The combination of knowledge and statistical data,” Technical Report MSR-TR-94-09, Microsoft Research, Redmond, WA, 1994.

[36] J. L. Golmard and A. Mallet, " Learning probabilities in causal trees from incomplete databases," *Revue d'Intelligence Artificielle*, vol. 5, pp. 93-106, 1991.

[37] S. L. Lauritzen, "The EM algorithm for graphical association models with missing data," Technical Report TR-91-05, Department of Statistics, Aalborg University, 1991.

[38] S. L. Lauritzen, "The EM algorithm for graphical association models with missing data," *Computational Statistics and Data Analysis*, vol. 19, pp. 191-201, 1995.

[39] D. J. Spiegelhalter and R.G. Cowell, "Learning in probabilistic expert systems," in J.M. Bernardo, J. O. Berger, A. P. Dawid, and A. F. M. Smith (Eds.), *Bayesian Statistics 4*, 1992.

[40] P. Dayan, "The convergence of TD(λ) for general λ," *Machine Learning*, vol. 8, pp. 341-362, 1992.

[41] R. J. A. Little and D. B. Rubin, *Statistical Analysis with Missing Data*, Wiley, New York, 1987.

[42] D. Suryadi and P. J. Gmytrasiewicz, "Learning models of other agents using influence diagrams," in *Proceedings of User Modeling: The Seventh International Conference*, Springer Wien, New York, 1999, to apper.

[43] C. Gerber and C. Jung, "Resource management for boundedly optimal agent societies," in *Proceedings of the ECAI'98 Workshop on Monitoring and Control of Real-Time Intelligent Systems*, Brighton, 1998.

[44] T. L. Dean and M. P. Wellman, *Planning and Control*. San Mateo, CA: Morgan Kaufmann, 1991.

[45] M. Ramoni and P. Sebastiani, “Learning Bayesian networks from incomplete data,” Technical Report KMi-TR-43, Knowledge Median Institute, The Open University, February 1997.

[46] J. Pearl, “A probabilistic calculus of actions,” in *Proceedings of the Tenth Conference on Uncertainty in AI (UAI-94)*, San Mateo, CA: Morgan Kaufmann, 1994.

[47] G. Shafer and J. Pearl, *Readings in Uncertain Reasoning*. San Mateo, CA: Morgan Kaufmann, 1990.

[48] M. I. Jordan and R. A. Jacobs, “Hierarchical mixtures of experts and the EM algorithm,” *Neural Computation*, vol. 6, pp. 181-214, 1994.

[49] P. Maes, “How to do the right thing,” *Connection Science*, vol. 1, no.3, 1989.

[50] C. Claus, “Dynamics of multi-agent reinforcement learning in Cooperative multi-agent systems,” Ph.D. Dissertation, Univ. of British Colombia, Canada, 1997.

[51] C. Gerber, “Evolution-based self-adaption as an expression for the autonomy degree in multi-agent societies,” in *Proceedings of the IEEE Joint Conference on the Science and Technology of Intelligent Systems*, Gaithersburg, MD, pp. 741-746, September 1998.

[52] S. Sen and M. Sekaran, “Multi-agent coordination with learning classifier systems,” in *Proceedings of the IJCAI Workshop on Adaptation and Learning in Multi-agent Systems*, Montreal, pp. 84-89, 1995.

[53] C. Boutilier, “Planning, learning and coordination in multi-agent decision processes,” in *Sixth conference on Theoretical Aspects of Rationality and Knowledge (TARK ‘96)*, The Netherlands, 1996.

[54] S. Russell and P. Norvig, *Artificial Intelligence: A modern Approach*, New Jersey: Prentice Hall, 1995.

[55] T. Malsch and I. Schulz-Schafer, "Generalized media of interaction and interagent coordination," in *Socially Intelligent Agents – Papers from the 1997 AAAI Fall Symposium*, Technical Report FS-97-02, AAAI, 1997.

[56] G.H. Hostetter, C.J. Savant, and R.T. Stefani, *Design of Feedback Control Systems*, New York: CBS College Publishing, 1982.

[57] I.D. Landau, R. Lozano, and M. M'Saad, *Adaptive Control*, London: Springer, 1998.

[58] I. Landau, *Adaptive Control: The Modal Reference Approach*, New York: Marcel Dekker, 1979.

[59] R. Kalman, "Design of self-optimizing control systems," *Transactions of ASME, J. Basic Eng*., vol. 80, pp. 468-478, 1958.

VITA

Ferat Sahin was born in Istanbul, the most beautiful city in Turkey, on October 12, 1971. He expressed a very special interest in Electronics at very young age, and decided to become an Electrical Engineer after an experiment about the electricity in Physics class in the last year of secondary school. He continued his education in a technical high school majoring in electronics. He blew his first capacitor when he was trying to repair a hand radio in his senior year in high school. He received his Bachelor of Science degree in Electronics and Telecommunications Engineering in October 1992, at Istanbul Technical University and went on to pursue a Master of Science degree in the same field. After one year, he decided to continue his study in the U.S. and came to Virginia Tech. He received his Master of Science degree in Electrical Engineering in May 1997, at Virginia Tech. His thesis topic was a Radial Basis Function Network solution to an image classification problem is a real-time industrial setting. He is pursuing his Ph.D. in Electrical Engineering at Virginia Tech. His dissertation topic is a Bayesian Network approach to the self-organization and learning in intelligent agents. He will be a faculty member at Rochester Institute of Technology starting September 2000. His extracurricular interests include soccer, basketball, photography, saz (a Turkish musical instrument), and social organizations. He was the president of Turkish Student Association at Virginia Tech during 1996-1997 academic year. He also served in Council of International Student Organizations (CISO) at Virginia Tech as Member at Large.