An empirical comparison of various representations of Dynamic Systems

Fan Sun

Department of Media & Knowledge Engineering
Delft University of Technology

Dec 16, 2010
Fan Sun

Student Number: 1321668

Man Machine Interaction Group

Media and knowledge Engineering

Faculty of Electrical engineering, Mathematics, and Computer Science

Delft University of Technology

Graduate Committee:

Prof. Drs. Dr. L.J.M. Rothkrantz

Prof. Dr. Ir. M.J. Druzdzel

Ir. H.J.A.M. Geers

Dr. Ir. P. Wiggers
Abstract

There exist several formalisms for representation and reasoning in dynamic systems, for example, Dynamic Influence Diagrams (DID), Influence Diagrams (ID), Dynamic Bayesian Networks (DBN), Bayesian Networks (BN), Hidden Markov Models (HMM), Markov Decision Processes (MDP), and Partially Observable Markov Decision Processes (POMDP). All these formalisms belong to graphical models based on probability theory. It has been shown that all probability models can be seen as variants of one generalization model. The purpose of this thesis is to review these models, to try to propose a unifying representation of these models at some generalization level (assuming DID level), and to test them in practice.
Acknowledgements

This thesis is the result of my graduation project at the University of Pittsburgh, from Jan 2008 to May 2008, and the left time till Dec 2010 at TU Delft. It was a long but good educational experience that will benefit me a lot. The quality of my thesis has been greatly increased under the influence of the following people, to whom I am very thankful:

- Marek Druzdzel. He was my supervisor at the University of Pittsburgh. My memory at DSL was wonderful and Marek was always kind to me. Our weekly meetings were extremely useful for my work in US. “Science is not magic” is the sentence he gave to me which I will keep forever.

- Leon Rothkrantz. He was my supervisor at the TU Delft and helped me going to the University of Pittsburgh. He was very warmhearted and helpful with my life and work.

- Mark Voortman. He was a Ph.D. student in DSL. He helped me to set up the environment of SMILE and gave me lots of good suggestions.

- Tomek Loboda. He was also a Ph.D. student in DSL and introduced me the environment of the lab.

- Milos Hauskrecht. He helped me with some POMDP problems I encountered while implementing the algorithms.

- Pascal Wiggers. He gave me lots of good ideas of unifying the Dynamic models.

- Dragos Datcu. He offered me lots of ideas and facial database to test.

Furthermore, my stay at the University of Pittsburgh and TU Delft would not have been possible without the support of:

- My parents, Sun Tenghuan and Li fang.

- Stimuleringsfonds Internationale Universitaire Samenwerkingsrelaties (STIR), Delft University of Technology.

And my friends:

- Li nuo, Huo feifei, Hu yixiong, Zhao duan, Guan yuxin, Habib, Baqir, Hu hao, Zhu di, Hou rui, Jiang cheng, Zhou yifan, Fu bing jie and Zhou jian
1 Introduction

1.1 Motivation ........................................ 9
1.2 The goals of this thesis .............................. 9
   1.2.1 Theoretical part ............................... 9
   1.2.2 Implementation part ......................... 9
   1.2.3 Experiment part .............................. 9
1.3 Thesis overview ................................... 9

2 Literature Review .................................. 11

2.1 Bayesian Networks ................................. 11
   2.1.1 Network structure ............................. 11
   2.1.2 Conditional probability table ................. 11
   2.1.3 Inference ..................................... 12
2.2 Dynamic Bayesian Networks ....................... 12
   2.2.1 Network structure ............................. 12
   2.2.2 Inference ..................................... 13
2.3 Influence Diagrams ................................ 13
   2.3.1 Network structure ............................. 13
   2.3.2 Evaluation of IDs ............................. 14
2.4 Dynamic Influence Diagrams ....................... 14
   2.4.1 Network structure ............................. 14
   2.4.2 Evaluation of DIDs ........................... 14
2.5 Hidden Markov Models .............................. 14
   2.5.1 Basic framework .............................. 15
   2.5.2 Basic algorithms ............................. 15
   2.5.3 Example ...................................... 15
2.6 Markov Decision Processes ......................... 16
   2.6.1 Markov property .............................. 16
   2.6.2 Basic framework .............................. 16
   2.6.3 Acting in MDPs ............................... 17
   2.6.4 Policy ....................................... 17
   2.6.5 Value function ............................... 18
   2.6.6 Optimal policy ............................... 18
   2.6.7 Basic algorithms ............................. 18
2.6.8 Example ............................................ 19
2.7 Partially Observable Markov Decision Processes ............ 20
  2.7.1 Basic framework ................................ 20
  2.7.2 Belief states .................................... 21
  2.7.3 Acting in POMDPs ................................. 21
  2.7.4 Value function ................................... 22
  2.7.5 Value iteration ................................. 22
  2.7.6 Optimal Policy Computation ...................... 22
  2.7.7 Basic algorithms ................................. 23
  2.7.8 Example ......................................... 23

3 Unifying Model ........................................ 25
  3.1 Introduction ...................................... 25
  3.2 The relation tree ................................ 25
  3.3 DID formalism .................................... 26
  3.4 IDs in DID formalism ............................... 27
  3.5 POMDPs in DID formalism .......................... 28
  3.6 MDPs in DID formalism ............................. 30
  3.7 DBN in DID formalism .............................. 31
  3.8 BN in DID formalism .............................. 32
  3.9 HMMs in DID formalism ............................. 33
  3.10 Conclusion ...................................... 34

4 GeNLe and SMILE ................................... 35
  4.1 Introduction ...................................... 35
  4.2 GeNLe ............................................ 35
  4.3 SMILE ............................................ 36
  4.4 Linear Programming and COIN-OR Linear Programming .... 37
    4.4.1 Linear Programming ............................ 37
    4.4.2 COIN-OR Linear Programming .................. 37

5 Algorithm Description ................................ 39
  5.1 Introduction ...................................... 39
  5.2 Algorithms already exists in GeNLe and SMILE .......... 39
  5.3 Algorithms to be developed in GeNLe and SMILE ........ 40
  5.4 Dynamic Programming ................................ 40
  5.5 The Principle of Optimality ......................... 40
  5.6 Bellman equation ................................ 40
  5.7 Forward algorithm ................................ 41
    5.7.1 Forward variable .............................. 41
    5.7.2 Recursion process ............................. 41
    5.7.3 Complexity of Forward algorithm ............... 42
  5.8 Viterbi algorithm ................................ 42
    5.8.1 Auxiliary variable ............................ 42
    5.8.2 Recursion process ............................. 42
    5.8.3 Complexity of Viterbi algorithm ............... 43
5.9 Value Iteration algorithm ................................... 43
  5.9.1 Bellman equation ...................................... 43
  5.9.2 Recursion process ..................................... 43
  5.9.3 Complexity of Value Iteration algorithm .......... 44
5.10 Policy Iteration algorithm ................................ 44
  5.10.1 Policy evaluation ..................................... 44
  5.10.2 Gaussian elimination ................................. 44
  5.10.3 Policy improvement .................................. 44
  5.10.4 Iteration Process .................................... 44
  5.10.5 Complexity of Policy Iteration algorithm ....... 45
5.11 Witness algorithm .......................................... 45
  5.11.1 Belief state update .................................. 45
  5.11.2 Value functions ...................................... 45
  5.11.3 Value Iteration ...................................... 46
  5.11.4 Stopping criterion .................................. 46
  5.11.5 Policy tree representation ........................... 46
  5.11.6 Witness outer loop .................................. 48
  5.11.7 Witness loop ........................................ 48
  5.11.8 Find the Witness .................................... 48
5.12 Point-Based approximate algorithm ....................... 48
  5.12.1 Traditional solution using vector representation . 49
  5.12.2 Point-Based value backup ............................ 49
  5.12.3 Point-Based Value Iteration ......................... 50
  5.12.4 Belief Point Selection ............................... 50

6 Implementation .................................................. 51
  6.1 DSL_forward_hmm() ....................................... 51
  6.2 DSL_viterbi_hmm() ........................................ 52
  6.3 DSL_ValueIteration_mdp() ............................... 52
  6.4 DSL_PolicyIteration_mdp() ............................. 53
  6.5 DSL_WitnessOuterLoop_pomdp() ......................... 53
  6.6 DSL_witness_pomdp() .................................... 54
  6.7 DSL_besttree_pomdp() ................................... 54
  6.8 DSL_back_pomdp() ....................................... 55
  6.9 DSL_choice_pomdp() ...................................... 55
  6.10 DSL_setchoice_pomdp() ................................. 55
  6.11 DSL_valueP_pomdp() .................................... 56
  6.12 DSL_superior_pomdp() ................................... 56
  6.13 DSL_findb_pomdp() ..................................... 56
  6.14 DSL_Lpfindwitness_pomdp() ............................ 57
  6.15 DSL_purge_pomdp() ..................................... 57
  6.16 DSL_Lpfindpurge_pomdp() ............................... 57
  6.17 DSL_computedifference_pomdp() ....................... 57
  6.18 DSL_Lpfinddif_pomdp() ................................ 58
  6.19 DSL_PBVI_pomdp() ...................................... 58
  6.20 DSL_backup_pomdp() .................................... 59
6.21 DSL_expandedbelief_pomdp() .............................................. 59

7 Evaluation ................................................................. 61
  7.1 MDP experiment ..................................................... 61
    7.1.1 Grid problem ................................................... 61
    7.1.2 Comparative analysis ......................................... 62
  7.2 POMDP experiment .................................................. 63
    7.2.1 Tiger problem .................................................. 63
    7.2.2 Comparative analysis ......................................... 63

8 Summary of Contributions and Open Problems ...................... 65
  8.1 Summary of Contributions ......................................... 65
    8.1.1 Unifying various models ...................................... 65
    8.1.2 Implement Algorithms for HMMs, MPDs and POMDPs .... 65
    8.1.3 Compare the Algorithms for MPDs and POMDPs ............ 65
  8.2 Open Problems ....................................................... 66

A Code of the Algorithms .................................................. 69
Chapter 1

Introduction

1.1 Motivation

There exist several formalisms for dynamic reasoning. The relations among these models and the comparison and the test of different algorithms are still open questions. There is a need to model these algorithms and implement them using the same tool. In our case, we choose SMILE as developer.

1.2 The goals of this thesis

1.2.1 Theoretical part

The work in this part will focus on (1) investigating the existing theory of the models above, (2) representing DID, ID, BN, DBN, HMM, MDP, POMDP in DID formalism, (3) comparing representational power.

1.2.2 Implementation part

This part consists of (1) designing and implementing modules for HMM, MDP, POMDP in SMILE, (2) designing and implementing basic algorithms for HMM, MDP, and POMDP at DID level.

1.2.3 Experiment part

The objects of this part are (1) study and comparison of basic algorithms of all these probability models at different levels including speed, efficiency, etc.

1.3 Thesis overview

Chapter 2 contains the literature review of Bayesian Network, Dynamic Bayesian Network, Influence Diagrams, Dynamic Influence Diagrams, Hidden Markov
Model, Markov Decision Process and Partially Observable Markov Decision Process. The basic description of these models will be given. Chapter 3 contains the discussion about the relationship of these different models. A unifying representation in Dynamic Influence Diagrams form will be given. Chapter 4 contains the introduction of design and implementing environment GeNe and SMILE. Chapter 5 contains the description of the basic algorithms for HMM, MDP, and POMDP in SMILE. Chapter 6 describes the implementation of the algorithms mentioned in the previous chapter. Chapter 7 contains the empirical study and comparison of the algorithms mentioned above. Chapter 8 gives the conclusions and the suggestions of the future work. Finally, the appendix will be given as the last part.
Chapter 2

Literature Review

2.1 Bayesian Networks

A Bayesian Network (BN)[18][11], also named belief network, is a probabilistic graphical model that represents a set of variables and their probabilistic independencies. In a typical Bayesian Network, there is a set of variables $X$ which forms a network structure $S$ that encodes the conditional independence relationships and the local probability distributions about variables in $X$. All of these in the network define the joint probability distribution for $X$. The Bayesian Network is widely used in medical diagnosis, reasoning, and expert system, etc.

2.1.1 Network structure

A Bayesian Network is a directed acyclic graph. Each node in this graph represents a random variable $X_i$. An arc from $X_i$ to $X_j$ means that $X_i$ is a parent of $X_j$ and $X_j$ is a child of $X_i$. The parents of $X_i$ are defined by $Parent(X_i)$. Every node has its own local probability distribution, $P(x_i|Parent(X_i))$. Using these local probability distributions and the conditional independence relations from the directed acyclic graph we get the joint probability distribution representation in a Bayesian Network

$$P(X_1, \ldots, X_i, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|Parent(X_i))$$

2.1.2 Conditional probability table

In a Bayesian Network the conditional probability table (CPT) provides a probability distribution for all the possible states of a child node, for each combination of possible states of the parent nodes. These conditional probability distributions describe the causal influence of its parents. Using CPTs and available evidence, beliefs that represent conclusions can be calculated of the Bayesian Network.
2.1.3 Inference

A Bayesian Network provides a good model for variables and their relationships, so that it can be used to calculate some probabilistic queries. In other words, inference means calculating the posterior probability of some variables \( X \) given some evidence \( e \), \( P(X|e) \).

Inference in a Bayesian Network can be divided into two ways, exact way and approximate way. For exact inference, it includes variable elimination algorithm, junction tree algorithm, Clustering algorithm, etc. For approximate inference, it includes stochastic sampling algorithms, loopy belief propagation algorithm, expectation propagation algorithm, etc.

2.2 Dynamic Bayesian Networks

A Dynamic Bayesian network (DBN)\[16\] is an extended Bayesian network that represents sequences of variables. These sequences often means temporal data or sequential data. The term "Dynamic" means the system is dynamic but the net structure doesn’t change over time.

2.2.1 Network structure

A Dynamic Bayesian Network is also a directed acyclic graph. It can be defined to be a pair, \((B_1, B_{-1})[16]\), where \( B_1 \) is a Bayesian Network which defines the prior probability distribution of the state variables \( P(Z_1) \), and \( B_{-1} \) is two-slice temporal Byesian Network(2TBN)which defines the transition model \( P(Z_t|Z_{t-1}) \) as follows:

\[
P(Z_t|Z_{t-1}) = \prod_{i=1}^{n} P(Z_i|\text{Parent}(Z_i)).
\]

where \( Z_i \) is the \( i \)th node at time \( t \), and \( \text{Parent}(Z_i) \) are the parents of \( Z_i \) in the graph, which can be in the same or the previous time slice of \( Z_i \). A Dynamic Bayesian Network also use CPT to define the probability distribution for the nodes,

\[
P(Z_i|\text{Parent}(Z_i)).
\]

Unlike the way in a Bayesian Network, the parents of a node in a Dynamic Bayesian network, \( \text{Parent}(Z_i) \), can either be in the same time slice or in the previous time slice of the 2TBN. As we mentioned above, the net structure doesn’t change over time. That means the parameters of the CPT are time-invariant. Therefor the model can be fully represented by this 2TBN form. In this way we can get the joint distribution of \( T \) time slice by unrolling the 2TBN network:

\[
P(Z_1:T) = \prod_{t=1}^{T} \prod_{i=1}^{N} P(Z_i|\text{Parent}(Z_i)).
\]
2.2.2 Inference

Inference in a Dynamic Bayesian Network can be represented as:

\[ P(X_i^t | y_{1:τ}). \]

If \( τ = t \), this is called filtering; If \( τ > t \), this is called smoothing; and if \( τ < t \) this is called prediction.

Inference in a Dynamic Bayesian Network can also be divided into exact and approximate way. For exact inference, it includes forwards-backwards algorithm, frontier algorithm, interface algorithm, etc. For approximate inference, it includes Boyen-Koller algorithm, particle filtering, etc.

2.3 Influence Diagrams

Influence Diagrams\cite{19}, also called relevance diagrams or decision network, are acyclic directed graphs which represent decision situations and find a decision alternative that leads to the highest excepted gain.

2.3.1 Network structure

As a probability graph model, Influence Diagrams are made up of four types of nodes and two types of arcs.

- Decision nodes, represented by rectangles in the networks, are used to represent variables that contain different decision alternatives which can be chosen by decision makers.

- Chance nodes, drawn as circles or ovals, are random variables which are used to represent uncertain quantities. Like nodes in Bayesian Networks, these random variables also use conditional probability distributions to encode uncertainty.

- Deterministic nodes, represented by double-circles or double-ovals, are constant values or values that are determined by the states of their parents.

- Values nodes, drawn as diamonds, are used to represent utility that is the measure of gain.

- Influence arcs, ending in Chance nodes, Deterministic nodes and Values nodes, indicate influence between parent nodes and child nodes. Sometimes they imply clear casual meanings between nodes.

- Informational arcs, ending in Decision nodes, have a different meaning. They are used to present temporal information. In other words, Informational arcs indicate the knowledge of the outcome of other decisions or variables before the decision that informational arcs point at.
2.3.2 Evaluation of IDs

Evaluation of Influence Diagrams means to calculate the expected utilities of all
the possible decision alternatives and find the alternative with the highest ex-
pected utility. Olmsted[17] developed the first algorithm for Influence Diagrams,
Influence Diagrams using a Bayesian Network as a middle step.

2.4 Dynamic Influence Diagrams

A Dynamic Influence Diagrams (DIDs)[9], also called Dynamic Decision Net-
works, is an extended Influence Diagrams. Not only do they represent how the
world changes over time, but they model general sequential decision making.

2.4.1 Network structure

Like the Influence Diagrams, Dynamic Influence Diagrams are also made up of
four types of nodes Decision nodes, Chance nodes, Deterministic nodes, and
Values nodes and two types of arcs Influence arcs and Informational arcs. The
difference is that the state of Chance Nodes may change over time and the Influ-
ence and Informational arcs can connect nodes from different time slice. In such
way, Dynamic Influence Diagrams can be defined to be a pair. $(I_1, I_{\rightarrow})$, where
$I_1$ is a Influence Diagram which defines the prior attribute of the network, and
$I_{\rightarrow}$ is two-slice temporal Influence Diagrams(2TID)which defines the transition
model.

2.4.2 Evaluation of DIDs

Evaluation in Dynamic Influence Diagrams does not consider the expected util-
ities in one time slice but the whole time process. In other word, we have to
calculate the global expected utilities during the whole time process. Tatman
and Shachter (1990) first introduce an algorithm to solve this problem.

2.5 Hidden Markov Models

A Hidden Markov model (HMM)[21] is a first-order Markov process where each
state generates an observation. In a regular Markov model, the state is directly
visible. In a Hidden Markov models, the state is not directly visible, but ob-
servation variables influenced by the state are visible. Both the hidden and
observation states are modeled by discrete random variables. The value of the
hidden variable at time $t$ only depends on the value of the hidden variable at
time $t - 1$. This is called the Markov property. Similarly, the value of the
observed variable at time $t$ only depends on the value of the hidden variable
at time $t$. Hidden Markov models are widely used in speech recognition, face
recognition and so on.
Inference in HMMs focused on two types of problems:

- Given the parameters of the model, compute the probability of a particular output sequence. This problem is solved by the forward algorithm.

- Given the parameters of the model, find the most likely sequence of hidden states that could have generated a given output sequence. This problem is solved by the Viterbi algorithm.

2.5.1 Basic framework

A Hidden Markov Model (HMM) is a triple (Π, A, B):

- \( \Pi = (\pi_i) \) is the vector of the initial state probabilities;
- \( A = (a_{ij}) \) is the state transition matrix, \( P(x_i | x_{i-1}) \);
- \( B = (b_{ij}) \) is the confusion matrix \( P(y_i | x_j) \).

2.5.2 Basic algorithms

Forward algorithm

The aim is to find the probability of a sequence of observations given a HMM model.

\[ P(\text{observations} | \gamma). \]

Viterbi algorithm

The Viterbi algorithm provides a computationally efficient way of finding the most likely underlying state sequence given observations of HMMs.

2.5.3 Example

Suppose that a man lives with his wife. And his wife has two kinds of mood states, Happiness and Anger. Unfortunately the man cannot know his wife’s mood directly. But he can guess by observing his wife. The observations also have two states, Singing and Chattering. Normally she will sing when she is happy, but it is not always the case. The same happens on Chattering. The Markov assumption promises that his wife today’s mood is always influenced at most by the mood of yesterday. In transition matrix,

\[
\begin{bmatrix}
\text{Happiness(t-1)} & \text{Anger(t-1)} \\
\text{Happiness(t)} & 0.8 & 0.2 \\
\text{Anger(t)} & 0.2 & 0.8 \\
\end{bmatrix}
\]

The probability \( P(\text{Happiness} | \text{Happiness}) = 0.8 \) means the probability of today’s mood being Happiness given yesterday’s mood being Happiness is 0.8.
In confusion matrix,

\[
\begin{bmatrix}
\text{Confusion matrix} & \text{Happiness} & \text{Anger} \\
\text{Singing} & 0.9 & 0.05 \\
\text{Chattering} & 0.1 & 0.95
\end{bmatrix}
\]

The probability \( P(\text{Singing} | \text{Happiness}) = 0.9 \) means the probability of the observation being Singing given the mood being Happiness is 0.9. In this example, the Forward algorithm finds the probability of an observed sequence given the HMM model such as \( P(\text{Singing}, \text{Chattering}, \text{Singing}) \). The Viterbi algorithm finds the most likely sequence of hidden states that generates the observation sequence. In our example, the mood sequence (Happiness, Happiness, Happiness) may be the most likely state sequence to emit the observation sequence (Singing, Singing, Singing). And the Viterbi algorithm is used to find it.

2.6 Markov Decision Processes

A Markov Decision Process (MDP)\(^3\) is a mathematical model for solving decision-making in situations where outcomes are partly random and partly influenced by the control of the decision maker. MDPs were introduced in 1950s by Bellman. And now these models are used in many areas, such as medical decision making, robot navigation, and so on.

Briefly, a MDP is a discrete time stochastic control process characterized by a set of states; in each state there are several actions from which the decision maker must choose. The transition function determines the transition probabilities to from a state to the next state. The decision maker earns a reward by choosing different actions in different states. The goal of the whole process is to maximize some cumulative function of the rewards.

2.6.1 Markov property

2.6.2 Basic framework

A Markov Decision Process can be described as a tuple \(<S,A,T,R>\).

- \( S \) is a finite set of possible states of the world;
- \( A \) is a finite set of possible actions;
- \( T(s,a,s') \) is the state-transition function, giving the probability of ending in state \( s' \), given that the agent starts in state \( s \) and take action \( a \);
- \( R(s,a) \) is the reward-function, giving the expected reward for taking action \( a \) in state \( s \);
### 2.6.3 Acting in MDPs

The goal in MDPs is to act optimally, i.e., to maximize the expected sum of rewards. In finite-horizon case, the agent in MDPs should act in order to maximize the expected sum of rewards from the next k steps. In mathematical form, it can be represented as

$$E\left(\sum_{t=0}^{k-1} r(t)\right).$$

Where $r(t)$ is the expected reward received on step $t$. In infinite-horizon case, the agent should act in order to maximize the expected sum of rewards over the infinite time steps, but with a discount factor $0 < \gamma < 1$. In mathematical form, it can be represented as

$$E\left(\sum_{t=0}^{\infty} \gamma^t r(t)\right).$$

The use of discount factor leads to the fact that rewards received earlier always have more value to the agent. Although the time is infinite, the discount factor used here ensures that the sum is finite.

### 2.6.4 Policy

The solution to MDPs can be described as a policy $\pi$, which represents the behavior of the agent. In other words, a policy is a mapping from states to actions. The next question is how to execute policy in MDPs. The basic steps are the following:

1. Determine the current state;
2. Execute action based on the current state;
3. Go to Step 1 until the terminal states or conditions are satisfied.

Usually policies can be divided into two classes: stationary and non-stationary. A stationary policy is a mapping in which the choice of action depends only on the state and is independent of the time step. In infinite-horizon models, the optimal policy is always stationary. In contrast, a non-stationary policy is a mapping that not only depends on the state but also on the time step. And in the finite-horizon model, the optimal policy is often non-stationary.

### 2.6.5 Value function

In the finite-horizon case, we define $V_{\pi,t}(s)$ to be the expected sum of rewards from starting in state $s$ and executing non-stationary policy $\pi$ for $t$ steps. And then $V_{\pi,t}(s)$ can be calculated recursively as follows:
1. \( V_{\pi,1}(s) = R(s, \pi_1(s)) \). For the last step, the value is just the expected reward for taking the action specified by the policy based on state \( s \).

2. \( V_{\pi,t}(s) = R(s, \pi_t(s)) + \gamma \sum_{s' \in S} T(s, \pi_t(s), s') V_{\pi,t-1}(s') \). The \( t \) step value of starting in the state \( s \) and executing non-stationary policy \( \pi_t(s) \) is the immediate reward, \( R(s, \pi_t(s)) \), plus the discounted expected value of the remaining \( t - 1 \) step. Note that all possible next state \( s' \) should be considered. And the likelihood of their occurrence can be specified by \( T(s, \pi_t(s), s') \), while the next \( t - 1 \) step value is from \( V_{\pi,t-1}(s') \).

In the finite-horizon case, the factor \( t \) is removed. And the value function can be defined and calculated by:

\[
V_{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_{\pi}(s') .
\]

### 2.6.6 Optimal policy

The representation of value function is important to find the optimal policy \( \pi^* \) which will maximize the expected sum of rewards during the whole process. In the finite-horizon case, the last step of action is easy to decide.

\[
\pi^*_1(s) = \arg\max_a R(s, a) .
\]

The optimal \( t \) step mapping for state \( s \) then can be defined by:

\[
\pi^*_t(s) = \arg\max_a [R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V^*_t(s')] .
\]

Here \( V^*_t(s') \) is the optimal \( t - 1 \) step value function. In the infinite-horizon case, there exists a stationary policy \( \pi^* \) that is optimal for every starting state. The mapping can be described as:

\[
\pi^*(s) = \arg\max_a [R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V^*(s')] .
\]

### 2.6.7 Basic algorithms

#### Value Iteration algorithm

The input of value iteration algorithm is the MDPs model, and the output is a mapping from states to optimal actions.

\[
V_1(s) := 0 \text{ for all } s;
\]

\[
t := 1;
\]

Loop

\[
t := t+1;
\]
Loop for all the states and all actions;
\[ V_t = \max_a [R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V_{t-1}(s')] ; \]
keep optimal action for each state;
end loop
until \( |V_t(s) - V_{t-1}(s)| < \epsilon \) for all \( s \in S \)

**Policy Iteration algorithm**

The input of value iteration algorithm is the MDPs model, and the output is a mapping from states to optimal actions.

Pick an initial policy;
Loop
Loop for all states;
Get the linear equations:
\[ G(s) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') G(s') ; \]
Solve all the \( G(s) \);
Loop for all states
\[ Policy(s) = \arg \max_a [R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') G(s')] ; \]
Keep the optimal action for each state;
until Policy(s) stay the same;

2.6.8 Example

We can continue to use the family example in section 2.3.3. Now the man’s wife still has two states, Happiness and Anger. Unlike the previous example, the man does not receive any observations but always knows the state of his wife. And he has to do some actions: Praise, Get angry and Do nothing. His actions will lead to the state change of his wife and he will receive some rewards. The state transition functions are:

\[
\begin{bmatrix}
\text{Praise action} & \text{Happiness}(t-1) & \text{Anger}(t-1) \\
\text{Happiness}(t) & 0.9 & 0.2 \\
\text{Anger}(t) & 0.1 & 0.8
\end{bmatrix}
\]

\[
\begin{bmatrix}
\text{Get angry action} & \text{Happiness}(t-1) & \text{Anger}(t-1) \\
\text{Happiness}(t) & 0.1 & 0.3 \\
\text{Anger}(t) & 0.9 & 0.7
\end{bmatrix}
\]

19
the probability $P(\text{Happiness}|\text{Happiness}, \text{Praise action}) = 0.9$ means the probability of his wife ending in state Happiness is 0.9, given that she starts from Happiness and the man takes Praise action. The reward function is:

$$
\begin{bmatrix}
\text{Reward function} & \text{Happiness} & \text{Anger} \\
\text{Praise} & +10 & -10 \\
\text{Get angry} & -10 & +10 \\
\text{Do nothing} & -5 & -5
\end{bmatrix}
$$

the reward value $R(\text{Happiness}, \text{Praise}) = +10$ means the man will receive +10 when his wife being state Happiness and he choose Praise action.

In MDPs, the man will receive rewards when choosing actions in different states. The goal is to maximize the expected sum of rewards in finite or infinite steps. For one step case it is easy. The man just consider the immediate rewards and choose Praise action when his wife is in Happiness state, Get angry action when his wife is in Anger state. For two or more step even infinite case it is more complex and difficult. Not only the immediate reward will be considered but also the rewards received in the future (often with discounting factor). The transition functions are helpful to consider all possible future states. And the optimal policy can be calculated by the way we mentioned in previous sections.

### 2.7 Partially Observable Markov Decision Processes

A Partially Observable Markov Decision Process (POMDP)\cite{POMDP} is a generalization of a MDP. The states, actions, transitions and immediate rewards are as same as MDPs'. The only difference is that the agent cannot directly observe the current state. Instead, we add observations to provide a hint about what state it is in. The observations can be probabilistic, which tells us the probability of each observation for each state in the POMDPs model. Applications of POMDPs include robot navigation, machine maintenance, and planning under uncertainty in general. The goal of POMDP process is also to maximize some cumulative function of the rewards.

#### 2.7.1 Basic framework

A Partially Observable Markov Decision process can be described as a tuple $<S, A, T, R, \Omega, O>$. 

- $S$ is a finite set of possible states of the world;
- $A$ is a finite set of possible actions;
T(s,a,s') is the state-transition function, giving the probability of ending in state s', given that the agent starts in state s and takes action a;

R(s,a) is the reward-function, giving the expected reward for taking action a in state s;

Ω is a finite set of observations

O(s',a,o) is the observation function, giving the probability of making observation o given that the agent took action a and landed in state s'.

2.7.2 Belief states

In MDPs the goal is to find a mapping from states to actions; in POMDPs the problem is to find a mapping from observation history to actions. Since the agent has finite memory capacity, it is a good idea to compress the observation history into an internal state. And belief state is such an internal state which is a probability distribution over states. This distribution encodes the agent’s subjective probability about the state of the world and provides a basis for choosing the next action.

We will use a simple two state POMDP to introduce how to represent belief states. For a two state POMDP we can represent the belief state with a single number. Since a belief state is a probability distribution, the sum of all probabilities must be 1. With a two state POMDP, if we are given the probability for being in one of the states as being p, then we know that the probability of being in the other state must be 1−p. Thus, the agent’s belief state is [p,1−p].

In POMDPs when the agent receives new observations, the belief state can be updated using Bayes rule:

\[ b'[s'] = P(s'|o,a,b) = (O[s',a,o] \sum_{s \in S} T[s,a,s'][b(s)])/P(o|a,b). \]

Here b' is the new belief state, that results from starting in b, taking action a, and receiving observation o. b'[s'] is the component s' of the belief state b'. And P(o|a,b) normalizes the resulting belief state to add to 1.

Belief state is sufficient for the agent to behave optimally. So now the goal in POMDPs is to specify a mapping from its current belief state to an optimal choice of action.

2.7.3 Acting in POMDPs

The goal in POMDPs remains to act in a way that will lead to maximizing the expected sum of rewards. In mathematical form, it can be represented as

\[ E(\sum_{t=0}^{\infty} \gamma^t r(t)) . \]

The use of discount factor (0 < \gamma < 1) ensures that the sum is finite.
2.7.4 Value function

In POMDPs, the value of a belief state, $b$, is the expected amount of reward that is generated by the optimal policy starting from $b$.

$$V^*(b) = E\left(\sum_{t=0}^{\infty} \gamma^t r(t)\right).$$

The function $V^*$ is called the optimal value function and it is unique. The optimal policy $\pi^*$ is stationary, so the optimal action choice for a belief state is constant over time. And we can represent $V^*(b)$ in this form.

$$V^*(b) = \max_a [\sum_{s \in S} b[s] R(s, a) + \gamma \sum_{o \in O} P(o|a,b) V^*(b')].$$

Here $b'$ denotes the belief state that result from taking action $a$ and observing $o$ from belief state $b$. And $b'$ can be calculated from $b$ by the update rules mentioned above. Because $b'$ will vary with different the starting belief state $b$, actions and observations, we can see clearly here is that POMDPs are really just MDPs running in continuous belief space.

2.7.5 Value iteration

The optimal value function or its approximation can be computed using value iteration. The idea behind value iteration is that by iteratively computing improved estimates of the optimal value function, we can get an arbitrarily good estimate. The key here is that we do not use the optimal value function itself but a step-estimate value function to represent and reach the optimal value function.

$$V(\pi, t)(b) = \max_a [\sum_{s \in S} b[s] R(s, a) + \gamma \sum_{o \in O} P(o|a,b) V(\pi, t-1)(b')].$$

This equation will lead to an improved approximation every step, and reach the optimal value function as close as possible. Besides the $t$ step value function is piecewise linear and convex. Thus the $t$ step value function can be represented by finite means and computable in a finite number of steps.

2.7.6 Optimal Policy Computation

The central task in POMDPs is to find a policy for selecting actions based on belief states. The optimal policy should maximize the expected future discounted cumulative reward:

$$\pi^*_t(b) = \arg\max_a [\sum_{s \in S} b[s] R(s, a) + \gamma \sum_{o \in O} P(o|a,b) V(\pi, t-1)(b')].$$
2.7.7 Basic algorithms

Witness algorithm

The input of witness algorithm algorithm is a POMDPs model, and the output is a mapping from any belief state to optimal actions.

Point-Based approximate algorithm

The input of point-based approximate algorithm is a POMDPs model, and the output is a mapping from a specific belief state to an optimal action.

2.7.8 Example

In POMDP environment, we assume the man lives with his wife again. His wife still has two kinds of mood states, Happiness and Anger. The man still needs observations, Singing and Chattering, to guess his wife’s mood. But he can choose some actions to change his wife’s state. And he will receive some rewards from his actions. There are three possible actions for the man: Praise, Get angry and Do nothing. The state transition functions are:

\[
\begin{pmatrix}
\text{Praise action} & \text{Happiness(t-1)} & \text{Anger(t-1)} \\
\text{Happiness(t)} & 0.9 & 0.2 \\
\text{Anger(t)} & 0.1 & 0.8 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
\text{Get angry action} & \text{Happiness(t-1)} & \text{Anger(t-1)} \\
\text{Happiness(t)} & 0.1 & 0.3 \\
\text{Anger(t)} & 0.9 & 0.7 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
\text{Do nothing action} & \text{Happiness(t-1)} & \text{Anger(t-1)} \\
\text{Happiness(t)} & 1 & 0 \\
\text{Anger(t)} & 0 & 1 \\
\end{pmatrix}
\]

The man can receive two observations: Singing and Chattering. The observation functions are:

\[
\begin{pmatrix}
\text{Observation function(for all actions)} & \text{Happiness} & \text{Anger} \\
\text{Singing} & 0.85 & 0.05 \\
\text{Chattering} & 0.15 & 0.95 \\
\end{pmatrix}
\]

The probability \( P(\text{Singing}|\text{Happiness, Do nothing}) = 0.85 \) means the probability of making observation Singing is 0.85 given that the man takes an action do nothing and his wife is in state Happiness. The reward function is:

\[
\begin{pmatrix}
\text{Reward function} & \text{Happiness} & \text{Anger} \\
\text{Praise} & +10 & -10 \\
\text{Get angry} & -10 & +10 \\
\text{Do nothing} & -5 & -5 \\
\end{pmatrix}
\]
In this POMDPs environment, we assume that the man starts in some belief state. If he takes some action and receives some observation, he will reach a new belief state. Thus, the Maximum expected sum of rewards can be calculated with updating belief state by the methods mentioned in previous sections. And the optimal policy, the mapping from initial belief states to actions will be found.
Chapter 3

Unifying Model

3.1 Introduction

In the previous chapter, some probabilistic models in dynamic systems were reviewed. However, the relations among these models need to be discussed further. In the following chapter, we focus on trying a unifying representations for these models. During the past years a lot of works on relations between some of these models have been done. Tatman and Shachter[9] describe the MDPs in the Influence diagram formalism. Murphy[16] represents HMMs in the Dynamic Bayesian Network formalism. etc. All these previous works imply that we can go further on this way. The Decision Systems Laboratory of the University of Pittsburgh are engaged in developing the reasoning system GeNIe. By far the system have included Dynamic Influence Diagrams (DID), Dynamic Bayesian Networks (DBN), Hidden Markov Models (HMM), Markov Decision Processes (MDP), and Partially Observable Markov Decision Processes (POMDP). We assumed all these models can be tried to rewrite into the most general model Dynamic Influence Diagrams formalism. Then it is not only convenient for our future theory and work in GeNIe and SMILE but also a "bigpicture" for the intelligent readers.

3.2 The relation tree

In Figure 3.1, we present all the different models in a relation tree. Compared with the model in the child node, the model in the parent node is more general. Here the "generality" is defined according to the following factors. If a model A is an extension of a model B, we assume A is more general than B. This also means that model A has more powerful representation ability. The opposite way of the "generality" is the term "specialization". And the more specific model always has less power of expression but more efficient ability not only in representation but in inference and learning.

Among the models we have mentioned above, Dynamic Influence Diagrams
is the most general model and has the most powerful expression. From the general – to – specific order, the following models are Influence Diagrams, Dynamic Bayesian Networks, Bayesian Networks and Hidden Markov Models in one branch, while Partially Observable Markov Decision Processes and Markov Decision Processes in the other branch. It should be possible to represent specific models by the general one. So in the following section, we focus on representing all these models in DID formalism that is a way to unification.

3.3 DID formalism

In figure 3.2, we show the Dynamic Influence Diagrams. And we will describe the DID formalism as follows:

- Decision nodes, representing the set of variables that contain different decision alternatives.
- Decision table, encoding all kinds of decisions.
- Chance nodes, representing the set of uncertain quantities.
- CPT, encoding uncertainty distributions.
- Value nodes, representing utility.
- Value table, encoding the utility-function.
- Influence arcs, indicating influence between parent nodes and child nodes in the same time slice.
- Temporal Influence arcs, indicating temporal influence between parent nodes and child nodes from different time slices.
Dynamic Influence Diagrams have powerful representation ability. In the DID formalism, a dynamic decision situation can be exactly described by the graphical structure. Decision nodes are used to represent the set of variables that contain different decision alternatives. And decision table is used to store these decision alternatives. Chance nodes are used to represent the set of uncertain quantities. Like the Bayesian Network, a DID also use CPT to encode the uncertainty distributions. Value nodes are used to represent the utility and value table are used to store the utility-function. Except the three kinds of nodes mentioned above, a DID have three kinds of arcs to link the different nodes in the same time slice or between the different time slice. Influence arcs indicate the influence between parent nodes and child nodes in the same time slice. Temporal Influence arcs indicate temporal influence between parent nodes and child nodes from different time slices. Informational arcs indicate that the decision at their heads is going to be made with knowing the outcome of the nodes at their tails.

### 3.4 IDs in DID formalism

In figure 3.3, we show an ID in Dynamic Influence Diagrams. And we can rewrite ID in DID formalism as follows:

- Decision nodes, representing the set of variables that contain different decision alternatives.
Figure 3.3: Influence Diagrams.

- Decision table, encoding all kinds of decisions.
- Chance nodes, representing the set of uncertain quantities.
- CPT, encoding uncertainty distributions.
- Value nodes, representing utility.
- Value table, encoding the utility-function.
- Influence arcs, indicating influence between parent nodes and child nodes.
- Temporal Influence arcs, no Temporal Influence arcs needed.
- Informational arcs, representing the knowledge of the outcome of the other decisions or variables before the decision that informational arcs point at.

Influence Diagrams share the most characters with Dynamic Influence Diagrams except the temporal attributes.

### 3.5 POMDPs in DID formalism

In figure 3.4, we show a POMDP in Dynamic Influence Diagrams. And we can rewrite POMDP in DID formalism as follows:

- Decision nodes, representing the set of possible actions $A$.
- Decision table, encoding all kinds of actions $a$. 

Figure 3.4: Partially Observable Markov Decision Process O.

- Chance nodes, representing the set of all the possible states of the world $S$ and the set of observations of the world $\Omega$.

- CPT, encoding the state-transition function $T(s, a, s')$ and the observation function $O(s', a, o)$.

- Value nodes, representing rewards when taking some actions.

- Value table, encoding the reward-function.

- Influence arcs, including arcs from state nodes to reward nodes, from action nodes to reward nodes and from state nodes to observation nodes.

- Temporal Influence arcs, including arcs from state nodes to state nodes, from action nodes to state nodes and from action nodes to observation nodes that belong to different time slice.

- Informational arcs, from observation nodes to action nodes.

In POMDPs the agent doesn’t know the exactly current states of the world, in stead, observations are afforded to give a belief about the current state. When we represent POMDPs by Dynamic Influence Diagrams, decision nodes are used to represent the set of possible actions $A$ and these actions $a$ are stored in the decision table. Chance nodes are needed to represent all the states of world and the set of observations of the world $\Omega$. The CPT table are used to define the the state-transition function $T(s, a, s')$ and the observation function $O(s', a, o)$. Value nodes are used to represent rewards. And the reward-function are stored in the value table. The influence arcs are used to denote the influence from
state nodes to reward nodes, from action nodes to reward nodes and from state nodes to observation nodes. The temporal influence arcs are used to denote the temporal influence including the one from state nodes to state nodes, the one from action nodes to state nodes and the one from action nodes to observation nodes that belong to different time slice. Informational arcs are needed to show the observations in stead of the states themselves before choosing actions.

3.6 MDPs in DID formalism

In figure 3.5, we show a MDP in Dynamic Influence Diagrams. And we can rewrite MDP in DID formalism as follows:

- Decision nodes, representing the set of possible actions $A$.
- Decision table, encoding all kinds of actions $a$.
- Chance nodes, encoding the set of all the possible states of the world $S$.
- CPT, encoding the state-transition function $T(s, a, s')$.
- Value nodes, representing rewards when taking some actions.
- Value table, encoding the reward-function.
- Influence arcs, including arcs from state nodes to reward nodes and from action nodes to reward nodes.
Temporal Influence arcs, including arcs from state nodes to state nodes and from action nodes to state nodes that belong to different time slice.

Informational arcs, from state nodes to action nodes.

Markov Decision Processes can also be represented in Dynamic Influence Diagrams. Decision nodes are used to represent the set of possible actions $A$ and these actions $a$ are stored in the decision table. Chance nodes are needed to represent all the states of world. The state-transition function $T(s,a,s')$ can be defined in the CPT table. Values nodes are used to represent rewards when the agent is taking some actions. And the reward-function are stored in the value table. The influence from state nodes to reward nodes and from action nodes to reward nodes can be denoted as influence arcs. And the influence between the different time slices such as the one from state nodes to state nodes and the one from action nodes to state nodes can be represent as temporal influence arcs. Informational arcs are needed in MDP to show the knowledge about the states before choosing actions.

### 3.7 DBN in DID formalism

In figure 3.6, we show a DBN in Dynamic Influence Diagrams. And we can rewrite DBN in DID formalism as follows:

- Decision nodes, no decision nodes needed in a DBN.
- Decision table, no decision table needed.
3.7 BN in Dynamic Influence Diagrams

In Figure 3.7, we show a BN in Dynamic Influence Diagrams. And we can rewrite BN in DID formalism as follows:

- Decision nodes, no decision nodes needed in a BN.
- Decision table, no decision table needed.
- Chance nodes, representing all kinds of variables $X_i$ of a BN.
- CPT, including $P(X_i|Parent(X_i))$ and initial distribution $P(X_0)$

Dynamic Bayesian Networks can be represented in Dynamic Influence Diagrams. Chance nodes are needed to represent all kinds of variables. The initial and temporal parameters including $P(Z_0)$ and $P(Z_t|Parent(Z_t))$ can be defined in the CPT table. The influence between variables in the same time slice can be denoted as influence arcs. And the influence between variables from the different time slice can be represent as temporal influence arcs.

3.8 BN in DID formalism

In Figure 3.7, we show a BN in Dynamic Influence Diagrams. And we can rewrite BN in DID formalism as follows:

- Decision nodes, no decision nodes needed in a BN.
- Decision table, no decision table needed.
- Chance nodes, representing all kinds of variables $X_i$ of a BN.
- CPT, including local conditional distribution $P(X_i|Parent(X_i))$ and initial distribution $P(X_0)$
Bayesian Networks can be also represented in Dynamic Influence Diagrams. Chance nodes are needed to represent all kinds of variables. The initial and local parameters including $P(X_0)$ and $P(X_i|\text{Parent}(X_i))$ can be defined in the CPT table. The influence between variables can be denoted as influence arcs.

### 3.9 HMMs in DID formalism

In figure 3.8, we represent an HMM in Dynamic Influence Diagrams. And we can rewrite HMM in DID formalism as follows:

- Decision nodes, no decision nodes needed in HMM.
- Decision table, no decision table needed.
- Chance nodes, including state nodes $X$ and observation nodes $Y$
- CPT, including $P(x_0), P(x_i|x_{i-1}), P(y_t|x_i)$
- Value nodes, no values nodes needed.
• Value table, no values table needed.

• Influence arcs, from state nodes to observation nodes.

• Temporal Influence arcs, from state nodes to state nodes between different time slice.

• Informational arcs, no Informational arcs needed.

Hidden Markov Models can be represented in Dynamic Influence Diagrams. Only Chance nodes are needed to represent state and observation nodes of an HMM. The parameters including $P(x_0), P(x_i|x_{i-1}), P(y_i|x_i)$ of an HMM can be defined in the CPT table in Dynamic Influence Diagrams. The links between state nodes and observation nodes can be shown as influence arcs, while the links between state nodes from different time slices can be shown as the temporal influence arcs.

3.10 Conclusion

In this chapter, we describe the Dynamic Influence Diagrams formalism. And we have rewritten IDs, POMDPs, MDPs, DBNs, BNs and HMMs in DID framework. DID formalism show its generality and its power of representation in such a class of probability models.
Chapter 4

GeNIe and SMILE

4.1 Introduction

Graphical Network Interface (GeNIe) and Structural Modeling, Inference, and Learning Engine (SMILE) are the reasoning software developed by the Decision Systems Laboratory (DSL) of the University of Pittsburgh. SMILE is the library and engine while GeNIe is the graphic user interface of SMILE. The software and documents of GeNIe and SMILE are published on the website http://genie.sis.pitt.edu.

4.2 GeNIe

GeNIe is a versatile and user-friendly development environment for graphical decision-theoretic models. It is a graphical click-and-drop interface running on the top of SMILE.

GeNIe1.0 was first released in 1998 which has received a wide acceptance within both academia and industry for decades of years’ development. GeNIe 2.0 is the latest version of GeNIe. Users of the programs have shared their experiences and their suggestions that led to the development of GeNIe 2.0. GeNIe 2.0 has a refreshingly new modern interface, and is even more intuitive and easier to use than GeNIe 1.0. In addition to aesthetics, GeNIe 2.0 has many more new features to offer.

The main features are:

- Graphical editor to create and modify network models.
- Uses the SMILE Engine. You may develop models in GeNIe and create a custom interface for them using SMILE.
- Supports chance nodes with General, Noisy OR/MAX and Noisy AND distribution.
• Open multiple networks and cut and paste sections of models between them.

• Complete integration with MS. Excel, cut and paste data into internal spreadsheet view of GeNIe.

• Cross compatibility with other software. Supports all major file types (e.g. Hugin, Netica, Ergo).

• Support for handling observation costs of nodes.

• Support for diagnostic case management.

4.3 SMILE

SMILE is a fully portable library implemented in C++ supporting all kinds of decision-theoretic methods, such as Bayesian Networks and Influence Diagrams. And we just expanded it to Hidden Markov Models, Markov Decision Process and Partly Observable Markov Decision Processes. By using the API of SMILE, it is convenient for user to create, edit, save and load graphical models, and use them for probabilistic reasoning and decision making under uncertainty.

SMILE supports directly object-oriented methodology. SMILE is implemented in C++ in a platform independent fashion. Individual classes of SMILE are accessible from C++ or (as functions) from C programming languages. As most implementations of programming languages define a C interface, this make SMILE accessible from practically any language on any system. SMILE can be embedded in programs that use graphical probabilistic models as their reasoning engines. Models developed in GeNIe can be equipped with a user interface which utilizes SMILE as the backend engine. SMILE is released as a dynamic link library (DLL). There are also several SMILE wrappers, such as SMILE.NET (.NET interface), SMILEX (Active X), jSMILE (Java interface), etc.

The main features are:

• Graphical editor to create and modify network models.

• Platform independent; versions available for Windows, Unix (Solaris), Linux, Mac, Pocket PC, etc.

• Available for use with .NET framework. Compatible with all .NET languages. May be used to create web-based applications of BNs.

• Thorough and complete documentation.

• Robust and running successfully in the field since 1997.

• Responsive development team support.
4.4 Linear Programming and COIN-OR Linear Programming

4.4.1 Linear Programming

A linear programming problem can be defined as the problem of maximizing or minimizing a linear function subject to linear constraints which are equalities or inequalities. The general form for a Linear Programming problem is as follows:

- **Objective Function**, which is a linear function with variables to express the goal. And it will be maximized or minimized as the result.
  \[ c_1x_1 + c_2x_2 + \ldots + c_nx_n. \]

- **Constraints**, which are series of linear equalities or inequalities to limit the possible solutions.
  \[ a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n \begin{cases} \leq b_i, i = 1, 2, \ldots, m. \\ = b_i, i = 1, 2, \ldots, m. \\ \geq b_i, i = 1, 2, \ldots, m. \end{cases} \]

- **Variable bounds**, which are used to limit the bounds of the variables.
  \[ x_j \geq 0 \text{ or } x_j \leq 0 \text{ or unrestricted.} \]

Linear programming guarantees linearity in the equations in the above structure. In each equation, variables multiplied by other variables and nonlinear functions such as logarithms are not allowed. So linear programming requires the following assumptions:

- **Proportionality**, a change in a variable results in a proportionate change in that variable’s contribution to the value of the function.

- **Additivity**, the value of the function is the sum of the contributions of each term.

- **Divisibility**, the variables can be divided into non-integer values. Integer programming techniques can be used if divisibility assumption does not hold.

- **Certainty**, the coefficients of the variables should be known and constant.

4.4.2 COIN-OR Linear Programming

The COIN Linear Program code or CLP is an open-source simplex solver written in C++. It is primarily meant to be used as a callable library, but a basic, stand-alone executable version is also available.
There are a number of resources available to help new CLP users get started. This document is designed to be used in conjunction with the files in the Samples subdirectory of the main CLP directory (COIN/Clp/Samples). The Samples illustrate how to use CLP and may also serve as useful starting points for user projects. In the rare event that either this document or the available Doxygen content conflicts with the observed behavior of the source code, the comments in the header files, found in COIN/Clp/include, are the ultimate reference.

CLP is written in C++, so it is expected that users of CLP will be writing C++ programs which use CLP as a library. Thus a working knowledge of C++, including basic object-oriented programming terminology is assumed in this document. In addition, the user should be familiar with the fundamental concepts of Linear Programming.
Chapter 5

Algorithm Description

5.1 Introduction

In chapter 3, we discuss the relations of DIDs, IDs, POMDPs, MDPs, DBNs, BNs, and HMMs and use DID formalism to represent other models. In chapter 4, we introduce our software environment GeNIe and SMILE. Our next task is to add some basic inference algorithms into the c++ library SMILE in order to expend the application range for GeNIe and SMILE adapted to more models.

5.2 Algorithms already exists in GeNIe and SMILE

For decades of years development, GeNIe and SMILE already include several decision and probability models such as IDs and BNs as well as the inference algorithms of them. For BNs, GeNIe and SMILE support two kinds of inference algorithms. One is exact algorithms including Clustering algorithm and Poly-tree algorithm. The other one is Stochastic Sampling Algorithms all of them belong to approximate algorithms such as Probabilistic Logic Sampling, Likelihood Sampling Self Importance Sampling, Heuristic Importance Sampling, Backward Sampling, AIS Sampling and EPIS Sampling. Among these algorithms, clustering algorithm is the default algorithm of GeNIe and it can be competent for most use. But sometimes the network is too large and complex, exact inference by clustering algorithm is not fast enough. It is better to use an approximate algorithm like the stochastic sampling algorithms. For IDs, the system afford two kinds of service. Finding the best decision algorithms can give the optimal decision as result. The limitation is the algorithm only finds the best policy for the ‘highest’ decision node in one time computation. To find the best policy for other decision nodes in the network, we have to set the decision for the first decision node and then calculate the network again. Policy evaluation algorithm will calculate all the actual utilities for each decision and therefore will be slower.
5.3 Algorithms to be developed in GeNIe and SMILE

Based on DIDs formalism GeNIe and SMILE has quite a powerful ability to represent all kinds of reasoning models. However, only part of them own their specific algorithms and solve the corresponding problem leaving others still on graphic representation level. To design and implement algorithms for more models in GeNIe and SMILE is one of the goals of this thesis. In the following part, we will describe the algorithms added into SMILE that can be called by the interface GeNIe. These algorithms include Forward algorithm and Viterbi algorithm for HMMS, Value Iteration algorithm and Policy Iteration algorithm for MDPs, Witness algorithm and Point-Based approximate algorithm for POMDPs.

5.4 Dynamic Programming

Dynamic programming is a method of solving complex problems by breaking them down into simpler subproblems in a recursive manner. The key property of this method is that during the process for a solution it avoids full enumeration. To achieve this property Dynamic programming method often solve each subproblem once, and save the solution for later use. And then the strategy is to make use of the relations between the larger problem and the subproblems to find the final solution. Dynamic programming is widely used in computer science, and all the algorithms mentioned above can be referred to this technology.

5.5 The Principle of Optimality

The principle of optimality is the basis of dynamic programming. This principle states: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. \[3\] It means in an optimal sequence of decisions or choices, each subsequence must also be optimal. In other words, the optimal solution is a combination of optimal solutions of its subproblems. Therefore, the large problem can be broken down into subproblems and solved by dynamic programming.

5.6 Bellman equation

Bellman equation is often a key part in dynamic programming, showing how to break a dynamic optimization problem into simple subproblems. Normally the value function of a decision problem at a certain time point is defined. And then the relationship between the value function in one time and the value function in the next time will be described by Bellman equation. It often works in the
following recursive way:

\[ V(x_0) = \max_{a_0} [F(x_0, a_0) + \beta V(x_1)]. \]

Based on the principle of optimality, the equation for the optimal value function is referred to as the Bellman optimal equation:

\[ V^*(x_0) = \max_{a_0} [F(x_0, a_0) + \beta V^*(x_1)]. \]

\( V^* \) is the optimal value function. There exist many methods to solve the Bellman equation, however, it is out of the scope of this paper. The introduction of dynamic programming and Bellman equation here is a help to well understand algorithms in GeNiE and SMILE in the following section.

5.7 Forward algorithm

Forward algorithm is used for finding the probability of an observed sequence \( P(\text{observations}|\gamma) \) given a HMM model. \( \gamma \) is the parameters of a HMM model. Traditional way to calculate this probability would be to find each possible sequence of the hidden states, and sum these probabilities. Calculating the probability in this way is time-consuming, especially with large models or long sequences. The key idea behind the Forward algorithm is to use the time invariance of the probabilities to reduce the complexity of the problem.

5.7.1 Forward variable

The forward variable, defined as the probability of the partial observation sequence, can be used as such kind of time invariance.

\[ \alpha_t(j) = \pi_j . P(o_1, o_2, ..., o_t, x_t = j | \gamma). \]

Here this partial probability terminates at the state \( j \), time \( t \).

5.7.2 Recursion process

After we defined the forward variable, the forward algorithm can run in a recursion way. In the first step when time \( t = 1 \). The partial probability is:

\[ \alpha_1(j) = \pi_j . P(o_1|x_1 = j). \]

Then the following step when time \( t > 1 \). The recursive relationship holds.

\[ \alpha_{t+1}(j) = P(o_{t+1}|x_{t+1} = j) \sum_{i=1}^{n} \alpha_t(i)a_{ij}. \]

So we can calculate \( \alpha_{t+1} \) at time \( t+1 \) using the partial probabilities at time \( t \) and calculate the probability of an observation sequence given a HMM recursively
Finally, the probability of the observed sequence given the HMM is the sum of the partial probabilities at time \( t = T \).

\[
P(o_1, o_2, \ldots, o_T | \gamma) = \sum_{j=1}^{n} \alpha_t(j).
\]

### 5.7.3 Complexity of Forward algorithm

The complexity of forward algorithm is proportional to \( n^2T \), which can be thought as linear in \( T \). Whereas the traditional one has an exponential complexity \( n^T \).

### 5.8 Viterbi algorithm

Viterbi algorithm is used for finding the most likely sequence of hidden states for a given sequence of observations. In other words, this means to calculate the maximal likelihood of the state sequence for the given observation sequence \( \max P(\text{observations}|\text{states}, \gamma) \). We can find this sequence by listing all possible sequences of hidden states and calculating the probability of the given observation sequence for each of the combinations. But it is computationally expensive. As the Forward algorithm, there is a way to use the time invariance of the probabilities to make the problem easy.

#### 5.8.1 Auxiliary variable

Viterbi algorithm define auxiliary variable as:

\[
\delta_t(j) = \max_{x_1, x_2, \ldots, x_{t-1}} P(x_1, x_2, \ldots, x_{t-1}, x_t = j; o_1, o_2, \ldots, o_t | \gamma).
\]

which gives the maximal probability that partial observation sequence and state sequence of the length \( t \) can have, when the current state is \( j \).

#### 5.8.2 Recursion process

The recursion process of Viterbi algorithm is quite the same as the one of the forward algorithm. In addition, we need to keep track of the arguments from \( \delta_t(j) \) for each \( t \) and \( j \), and store them in a matrix \( \psi \).

In the first step when time \( t = 1 \). Viterbi algorithm calculates and stores:

\[
\delta_1(j) = \pi_j P(o_1|x_1 = j).
\]

\[
\psi_1(j) = 0.
\]

Then the following step when time \( t > 1 \). The recursive relationship holds.

\[
\delta_{t+1}(j) = \max_i [\delta_t(i) a_{ij}] P(o_{t+1}|x_{t+1} = j).
\]
\[ \psi_{t+1}(j) = \arg\max_{i} [\delta_{t}(i)a_{ij}] \].

Finally, the algorithm stops
\[ x_{T} = \arg\max_{j} [\delta_{T}(j)] \].

Thus denotes which state is the most probable one when time is over.
\[ x_{t} = \psi_{t+1}(x_{t+1}) \].

This backtracking give the most probable path and is the final result.

5.8.3 Complexity of Viterbi algorithm
The time complexity of Viterbi algorithm is proportional to \( n^2T \) and the space complexity is proportional to \( nT \).

5.9 Value Iteration algorithm

Value Iteration algorithm is one kind of dynamic programming algorithm to solve MDPs finding an optimal mapping from states to actions. Value Iteration calculate the sequence \( V_t \) of discounted optimal value functions. By solving this kind of Bellman equation, the optimal policy will be discovered.

5.9.1 Bellman equation

Value Iteration algorithm define the Bellman equation as:
\[ V_{t} = \max_{a} [R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V_{t-1}] \]

which describe the relations between value function \( V_t \) at time \( t \) and value function \( V_{t-1} \) at time \( t-1 \) for one state.

5.9.2 Recursion process

At the initial step, all the value functions for every state at time \( t = 1 \) are set to 0. \( V_1(s) := 0 \).

When time \( t > 1 \), the recursive process update as the Bellman equation above describes. For all the states and all the actions:
\[ V_{t} = \max_{a} [R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V_{t-1}] \].

At the last, the algorithm should keep the optimal action for each state.

The algorithm stops when
\[ |V_{t}(s) - V_{t-1}(s)| < \epsilon \text{ for all } s \in S \].

If \( |V_{t}(s) - V_{t-1}(s)| < \epsilon \) for all \( s \in S \), then the value of the greedy policy with respect to \( V_t \) does not differ from \( V^* \) by more than \( 2\gamma/(1 - \gamma) \) at any state.[12] It is often the case that \( \pi_{V_t} = \pi^* \) long before \( V_t \) is near \( V^* \).
5.9.3 Complexity of Value Iteration algorithm
The time complexity of Value Iteration algorithm is shown in polynomial time.[14]

5.10 Policy Iteration algorithm
Policy iteration algorithm is a fast approach to solve infinite-horizon MDPs, focusing on improving the current policy to a better one. The whole process alternates between policy evaluation and policy improvement. After each alternation the policy is guaranteed to have an improvement over the previous one unless it is the optimal one. Finally, the algorithm will stop when no changes of policy are made, and the optimal policy will be found.

5.10.1 Policy evaluation
The most tricky part of Policy Iteration is Policy evaluation. In this step the current policy will be evaluated. And this is done by solving the linear system of equations,

$$G(s) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')G(s').$$

where $G(s)$ the cost-to-go for every states are unknown, which will be used to compute a better plan in the next step Policy improvement.

5.10.2 Gaussian elimination
Gaussian elimination is used to find the solution of the linear system of equations mentioned above.

The process of Gaussian elimination is made up of two parts. The first part (Forward Elimination) reduces the matrix to either triangular or echelon form, or indicating the system has no solution by using the elementary row operations (multiplying rows, switching rows, and adding multiples of rows to other rows). The second part uses back substitution to find the solution.

5.10.3 Policy improvement
In the Policy improvement phase, an attempt to improve the current policy is made. In mathematical words,

$$Policy(s) = \arg\max_a [R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')G(s')].$$

5.10.4 Iteration Process
At the initial step, an arbitrary policy is picked. In the policy evaluation phase, given the current policy, all the cost-to-go $G(s)$ of all the states will be calculated by solving the linear system of equations. In the Policy improvement phase,
based on the cost-to-go $G(s)$, a new policy will be acquired. If the policy doesn’t change, then the algorithm stops. Otherwise, the process repeats from the Policy improvement again.

5.10.5 Complexity of Policy Iteration algorithm

The time complexity of Value Iteration algorithm is shown in polynomial time.[14]

5.11 Witness algorithm

Witness algorithm[15] is one kind of value iteration algorithm to solve infinite-horizon POMDPs.

5.11.1 Belief state update

A belief state is a representation of the agent’s current state given its past history of actions and observations.[15] In other word, it is a probability distribution over states. Belief state plays a very important role in POMDPs, normally a POMDP can be seen as a belief state MDP. In POMDPs, the agent doesn’t know the exact current state. Instead, belief state can be calculated from the POMDP environment. We can always update the belief state by the following formula,

$$b'[s'] = P(s'|o,a,b) = \frac{(O[s', a, o] \sum_{s \in S} T[s, a, s']b[s])}{P(o|a,b)}.$$

Here $b'[s']$ is the component $s'$ of the belief state $b'$ which is the new belief state, that results from starting in the previous belief state $b$, taking action $a$, and receiving observation $o$. And $P(o|a,b)$ normalizes the resulting belief state to add to 1, which can be computed as follows:

$$P(o|a,b) = \sum_{s \in S} P(o, s'|a,b) = \sum_{s \in S} O[s', a, o] \sum_{s \in S} T[s, a, s']b[s].$$

Thus, through the previous belief state, observation and transition probabilities, new belief state can be always acquired. This also means when the agent moves about, new information can be incorporated into the belief state.[15] And Astrom [1] shows belief state is sufficient information for the agent to behave optimally. More important, this means policy in POMDPs will be a mapping from the agent’s current belief state to an optimal action.

5.11.2 Value functions

In POMDPs, the optimal policy maps belief states to actions, and it is stationary[4] that means the optimal policy doesn’t change over time. Optimal policy always
means the maximal value on that belief state. In POMDPs, the value of a belief state, is the expected amount of reward that is garnered by the optimal policy when the agent starts from this belief state.

\[ V^*(b) = E\left(\sum_{t=0}^{\infty} \gamma^t r(t)\right). \]

The function \( V^* \) is called the optimal value function and it is unique. We can also represent \( V^*(b) \) in recursion form.

\[ V^*(b) = \max_a \left[ \sum_{s \in S} b(s)R(s,a) + \gamma \sum_{o \in O} P(o|a,b)V^*(b') \right]. \]

Here \( b' \) denotes the belief state that result from taking action \( a \) and observing \( o \) from belief state \( b \). And \( b' \) can be calculated form \( b \) by the update rules mentioned above.

### 5.11.3 Value Iteration

In practice, we use value iteration to approximate the optimal value function.

\[ V_t(b) = \max_a \left[ \sum_{s \in S} b(s)R(s,a) + \gamma \sum_{o \in O} P(o|a,b)V_{t-1}(b') \right]. \]

Here \( V \) is an estimation of the optimal value function. As mentioned in the previous chapter, this equation will lead to an improved approximation every step, and reach the optimal value function arbitrarily.

### 5.11.4 Stopping criterion

There exists a stopping criterion for value iteration of POMDPs: At each iteration \( t \), compare \( V_t \) and \( V_{t-1} \) and stop if the maximum difference is small enough.[15]

### 5.11.5 Policy tree representation

In POMDP world, the agent repeats taking an action and receiving an observation. So a t-step policy can be represented by a t-step policy tree with actions as the nodes and observations as branches like the one in Figure 5.1.

For a finite-horizon POMDP, it’s value functions are piecewise-linear and convex.[20] And the optimal t-horizon value function is the "upper surface" and can be written as [15]:

\[ V_t(b) = \max_{\alpha \in \mathbb{C}_t} \sum_{s \in S} b(s)\alpha[s]. \]

Here \( \alpha \) is a \( |S| \) dimensional vector. In addition, Smallwood and Sondik[20] also showed that any finite-horizon value function can be represented in this
vector way. So for a policy tree $p$, we can always find a vector of values, $value(p)[s]$(the way to calculate this vector will be discussed in the later section), that gives the expected value when executing the policy $p$ from the initial state $s$. And then we can express the value starting form any belief state $b$ and executing the policy $p$ as follows:

$$\sum_{s \in S} b[s] value(p)[s] .$$

In the previous section, we denoted the relations between value vectors and policy trees. And find a way to calculate values for any belief state under a given policy. In value iteration, policy tree and vector representation also play an important part. To solve a POMDP problem, we need to find a mapping from belief state to action. After the introduction of policy trees we can change the problem to finding a mapping from belief state to policy tree, because the root node of the policy tree is exactly the action. Not all the t-step policy trees are needed, because not all of them can lead to the optimal value for a given belief state. Instead of considering the set of all the t-step policy trees, we only need $\mathcal{Y}_t$, which is a minimal set of t-step trees for representing the optimal value function over all the belief states. If a policy tree can not represent the optimal value function, we call it extraneous. To get the minimal representing set, extraneous policy trees should be purged. Policy tree representation give us an important improvement of calculating $value(p)$, the policy tree’s value vector. We can compute the vector of a t-step tree from the vectors from its t-1 step subtrees:

$$value(p)[s] = R[s, a] + \gamma \sum_{o} back(value(choice(p, o)), a, o)[s] .$$

where $back(\alpha, a, o)[s]$ returns a vector that consists of the components of $\alpha$
projected backwards one step by action $a$ and observation $o$. It is used to combine policy trees together. $\text{choice}(p,o)$ is used to choose the $t-1$ step subtree following observation $o$.

### 5.11.6 Witness outer loop

The Witness outer loop is to find $\mathcal{Y}_t$, the minimal set of $t$-step policy trees for representing the optimal $t$-step Value function, $V_t$.

Instead of building $\mathcal{Y}_t$ directly, a set of policy trees $Q^a_t$, which have the action $a$ as their root node, will be the target first. And $Q^a_t(b)$ represents the expected reward for taking action $a$ from the belief state $b$ and then acting optimally for the remaining $t-1$ step:

$$Q^a_t(b) = \sum_s b[s]R[s,a] + \gamma \sum_o Pr(o|a,b)V_{t-1}(b').$$

And then the union of these sets from all the actions forms a representation of the optimal value function. But there may be lots of extraneous policy trees in the union set, a purging is needed to form $\mathcal{Y}_t$, the optimal set of $t$-step policy trees.

The algorithm will stop when the stopping criterion is satisfied.

### 5.11.7 Witness loop

The witness loop is used to find the minimal set of policy trees for representing $Q^a_t$ for each action $a$. The set of Q-functions for a given action $a$ is a set of non-extraneous policy trees, and a temporal estimate set $U$ starts with a single policy tree that is the best for arbitrary belief state, normally 1, 0, ..., 0. At each iteration, the algorithm try to find if there is some belief state $b$, for which the estimate set $U$ can be expanded. Such a belief state is called a witness because it can show the fact that the set $U$ is not yet perfect enough to represent $Q^a_t(b)$. Once a witness is found, a new policy tree that lead to the best value at that belief state should be added to $U$. The witness loop stops when there are no more witness points found and that means the current $Q^a_t$ is perfect.

### 5.11.8 Find the Witness

In POMDPs, belief states are not discrete but continues, so search a witness from all belief states seems terrible. However, linear programming can solve such kind of problems.

### 5.12 Point-Based approximate algorithm

Point-Based approximate algorithm[10] is one kind of fast approximate approach to solve POMDPs. Exact solutions like witness algorithm mentioned above are
often computationally intractable except the smallest problems. However, approximate algorithm like Point-Based algorithm creates value backups at specific belief points instead of the entire belief space. Therefore, the speed of calculation can be greatly improved.

5.12.1 Traditional solution using vector representation

Sondik has shown that any finite-horizon value function can be represented by a set of vectors \( C_t = \alpha_0, \alpha_1, ..., \alpha_n \). And the optimal value function can be computed as follows:

\[
V_t(b) = \max_{\alpha \in C_t} \sum_{s \in S} b[s] \alpha[s].
\]

And the \( t \)-step solution set \( C_t \) can be founded in such way. First generate intermediate sets \( C_t^{a,*} \) and \( C_t^{a,o} \), \( \forall a \in A, \forall o \in O \):

\[
C_t^{a,*} \leftarrow \alpha^{a,*}(s) = R(s,a),
\]

\[
C_t^{a,o} \leftarrow \alpha_t^{a,o}(s) = \gamma \sum_{s'} O[s',a,o] T[s,a,s'] \alpha_{i}(s'), \forall \alpha_i \in C_{t-1}.
\]

And then generate \( C_t^a \) in the following way:

\[
C_t^a = C_t^{a,*} \oplus C_t^{a,o1} \oplus C_t^{a,o2} \oplus ....
\]

where the symbol \( \oplus \) define the cross-sum operator. For two sets, \( A = \{\alpha_1, \alpha_2, ..., \alpha_m\} \) and \( B = \{\beta_1, \beta_2, ..., \beta_n\} \), \( A \oplus B = \{\alpha_1 + \beta_1, \alpha_1 + \beta_2, ..., \alpha_1 + \beta_n, \alpha_2 + \beta_1, \alpha_2 + \beta_2, ..., \alpha_m + \beta_n\} \). Finally the union of \( C_t^a \) forms \( C_t \):

\[
C_t = \cup_{a \in \alpha} C_t^a.
\]

\( C_t \) is the \( t \)-step solution set and the value function \( V_t \) can be calculated from \( C_t \) in the way mentioned in the previous paragraph.

5.12.2 Point-Based value backup

Traditional exact method updates value function over all beliefs, this always make the problem big. Point-Based approximate algorithm applies value function updates only on specific belief points instead of the whole space. Thus the solution set \( C_t \) will contain at most one vector for each belief point and totally equal to the number of the selected belief points. The key idea of such kind of algorithm is that it assumes the other belief points in a region around a selected belief points have the same value function and choose the same action as the selected one. Therefore the way to generate the solution set \( C_t \) is a little bit different form the traditional one:

First generate intermediate sets \( C_t^{a,*} \) and \( C_t^{a,o} \), \( \forall a \in A, \forall o \in O \) in the same way as the traditional one:

\[
C_t^{a,*} \leftarrow \alpha^{a,*}(s) = R(s,a).
\]
\[ C_{t}^{a,o} \leftarrow \alpha_{t}^{a,o}(s) = \gamma \sum_{s} O[s', a, o] T[s, a, s'] \alpha_{i}(s'), \forall \alpha_{i} \in C_{t-1}. \]

And then generate \( C_{t}^{a} \) by summation instead of cross-sum operation:

\[ C_{t}^{a} = C_{t}^{a,*} + \sum_{o} \text{argmax}_{\alpha_{t} \in C_{t}^{a}} \left( \sum_{s} \alpha(s) b(s) \right), \forall b \in B. \]

Finally choose the best action for each belief point:

\[ \alpha_{b} = \text{argmax}_{\alpha \in A} \left( \sum_{s} C_{t}^{a}(s) b(s) \right), \forall b \in B. \]

\[ C_{t} = \cup_{b} \alpha_{b}. \]

The above process generates the best vector at each selected belief point, an estimate of the value function at any belief state can be represented as follows:

\[ V_{t}(b) = \max_{\alpha \in C_{t}} \sum_{s \in S} b(s) \alpha[s]. \]

### 5.12.3 Point-Based Value Iteration

Point-Based Value Iteration starts with a small initial set of belief points, and then applies the value backup operation to get the solution set \( C_{t} \). However, now the solution set \( C_{t} \) is not good enough, to solve this problem expanding the belief set is needed. Then the new backup operation is applied based on the new belief set. These two processes, backup operation and belief point expansion will repeat until the solution is good enough.

### 5.12.4 Belief Point Selection

There exists many approaches to select belief state. This thesis only take a simple strategy, Random Belief Selection. It’s a way to sample belief points from a uniform distribution over entire belief simplex. The advantage of this method is simple and fast. It doesn’t work badly when the domain is not large (< 100 states).
Chapter 6

Implementation

In this chapter, we will describe the implementation of algorithms mentioned in the previous chapter. All the algorithms are developed as the part of the C++ library SMILE.

6.1 DSL_forward_hmm()

DSL_forward_hmm() is the method to implement forward algorithm of Hidden Markov Models in SMILE. It acquires the input from a Hidden Markov Model represented in GeNiE and then implement in SMILE and gives the result, the probability of an observed sequence.

Algorithm 6.1.1: DSL_FORWARD_MDP(Model)

\begin{verbatim}
comment: forward variable initialization
comment: n is the number of states
for j ← 1 to n
  do
    α₁(j) ← πⱼ.P(o₁|x₁ = j)
for j ← 1 to n
  do
    for t ← 2 to T
      do
        αₜ(j) ← P(oₜ|xₜ = j).∑ₙᵢ=₁ αₜ₋₁(i)aᵢj
        P(o₁, o₂, ...oₜ|γ) ← ∑ₙⱼ=₁ αₜ(j)
return (P(o₁, o₂, ...oₜ|γ))
\end{verbatim}
6.2 DSL_viterbi_hmm()

DSL_viterbi_hmm() is the method for implementing Viterbi algorithm of Hidden Markov Models in SMILE. It gets the input from a Hidden Markov Model represented in GeNiE and then gives the output, the most likely sequence of hidden states for a given sequence of observations.

Algorithm 6.2.1: DSL_VITERBI_MDP(Model)

\textbf{comment:} auxiliary variable initialization
\textbf{comment:} \(n\) is the number of states
\begin{verbatim}
for \(j \leftarrow 1\) to \(n\)
do
\{ \(\delta_1(j) \leftarrow \pi_j.P(o_1|x_1 = j)\)
\(\psi_1(j) \leftarrow 0\)
\}
for \(j \leftarrow 1\) to \(n\)
do
for \(t \leftarrow 2\) to \(T\)
do
\{ \(\delta_t(j) = \max_i(\delta_{t-1}(i)a_{ij})P(o_t|x_t = j)\)
\(\psi_t(j) = \arg\max_i(\delta_{t-1}(i)a_{ij})\)
\}
x_T \leftarrow \arg\max_j[\delta_T(j)]
for \(t \leftarrow T\) to \(2\)
do
x_{t-1} \leftarrow \psi_t(x_t)
\return\ (x_1,x_2...x_T)
\end{verbatim}

6.3 DSL_ValueIteration_mdp()

DSL_ValueIteration_mdp() defines the method to implement Value Iteration algorithm of Markov Decision Processes. It gets the input from a Markov Decision Processes Model represented in GeNiE and then gives the output, the optimal mapping from states to actions.
Algorithm 6.3.1: DSL_VALUE_ITERATION_mdp(Model)

for each $s \in S$
do
$V_1(s) \leftarrow 0$
t $\leftarrow 1$
repeat
\begin{align*}
& t \leftarrow t + 1 \\
& \text{for each } s \in S \\
& \text{do} \\
& \text{for each } a \in A \\
& \text{do} \\
& \quad V_t(s) = \max_a [R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V_{t-1}(s')] \\
& \text{comment: keep optimal action for each state}
\end{align*}
until $|V_t(s) - V_{t-1}(s)| < \epsilon$ for all $s \in S$

6.4 DSL_PolicyIteration_mdp()

DSL_PolicyIteration_mdp() is the method to implement Policy Iteration algorithm of Markov Decision Processes. It gets the input from a Markov Decision Processes Model represented in GeNiE and gives the output, the optimal policy.

Algorithm 6.4.1: DSL_PolicyIteration_mdp(Model)

comment: Choose an initial policy
repeat
\begin{align*}
& \text{comment: Get the linear equations} \\
& \text{for each } s \in S \\
& \text{do} \\
& \quad G(s) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') G(s') \\
& \text{comment: Solve all the } G(s) \text{ from the linear equations} \\
& \text{for each } s \in S \\
& \text{do} \\
& \quad Policy(s) \leftarrow \arg\max_a [R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') G(s')] \\
& \text{comment: Keep the optimal action for each state}
\end{align*}
until Policy(s) stay the same

6.5 DSL_WitnessOuterLoop_pomdp()

DSL_WitnessOuterLoop_pomdp() is the method to implement Witness algorithm of Partially Observable Markov Decision Processes. It takes the input
from a Partially Observable Markov Decision Processes Model in GeNIe and output the result as the optimal mapping from belief state to actions.

Algorithm 6.5.1: DSLWitnessOuterLoopPomdp($\epsilon$, Model)

\[
\begin{align*}
\Upsilon_0 & \leftarrow \{\} \\
\mathbf{t} & \leftarrow 1 \\
\text{repeat} & \\
\quad \text{for each } a \in A & \\
\quad \quad Q_a^\mathbf{t} & \leftarrow DSL\_witness\_pomdp(\Upsilon_{\mathbf{t}-1}, a, Model) \\
\quad \Upsilon_{\mathbf{t}} & \leftarrow DSL\_purge\_pomdp(\cup_a Q_a^\mathbf{t}) \\
\quad \mathbf{t} & \leftarrow \mathbf{t} + 1 \\
\text{until} & (DSL\_computedifference\_pomdp(\Upsilon_{\mathbf{t}-1}, \Upsilon_{\mathbf{t}}) \leq \epsilon) \\
\text{return} & (\Upsilon_{\mathbf{t}})
\end{align*}
\]

6.6 DSL\_witness\_pomdp()

DSL\_witness\_pomdp() is the method to find a representation for Q-function. Its inputs are the POMDP model, a given action $a$ and the minimal set of $t-1$ step policy trees for representing $V_{t-1}$. Its output is a minimal set of $t$ step policy trees for representing Q-function given a specified action.

Algorithm 6.6.1: DSL\_witness\_pomdp($\Upsilon_{t-1}, a, Model$)

\[
\begin{align*}
Q_a^\mathbf{t} & \leftarrow \{DSL\_besttree\_pomdp([1, 0, 0, ..., 0], a, \Upsilon_{t-1}, Model)\} \\
b & \leftarrow DSL\_findb\_pomdp(a, \Upsilon_{t-1}, Q_a^\mathbf{t}, Model) \\
\text{while} & (b \neq \text{nil}) \\
\quad \text{do} & \quad \{Q_a^\mathbf{t} \leftarrow Q_a^\mathbf{t} \cup \{DSL\_besttree\_pomdp(b, a, \Upsilon_{t-1}, Model)\}\} \\
\quad \text{return} & \quad (Q_a^\mathbf{t})
\end{align*}
\]

6.7 DSL\_besttree\_pomdp()

DSL\_besttree\_pomdp() is the method to find the best policy tree. The inputs are a given belief state $b$, a given action, the minimal set of $t-1$ step policy trees for representing $V_{t-1}$ and the POMDP model. The output returns the the best policy tree beginning with a given action for a belief state.
Algorithm 6.7.1: DSL_besttree_pomdp(b, a, Υ_{t-1}, Model)

comment: set p' to be a new policy tree

\(\text{action}(p') \leftarrow a\)

for each \(o \in O\)

\[
\begin{align*}
\text{bestpol}' & \leftarrow \text{nil} \\
\text{bestvec'} & \leftarrow [-\infty, -\infty, ..., -\infty] \\
\text{bestval} & \leftarrow -\infty
\end{align*}
\]

for each \(\text{pol}' \in \text{Υ}_{t-1}\)

\[
\begin{align*}
\text{vec'} & \leftarrow \text{DSL_back_pomdp(DSL_valueP_pomdp(pol', Model), a, Model)} \\
\text{val} & \leftarrow \text{vec'}
\end{align*}
\]

if \((\text{val} > \text{bestval})\) or \((\text{val} = \text{bestval}\) and \(\text{DSL_superior_pomdp(vec', bestvec')}\))

\[
\begin{align*}
\text{bestpol}' & \leftarrow \text{pol}' \\
\text{bestvec'} & \leftarrow \text{vec'} \\
\text{bestval} & \leftarrow \text{val}
\end{align*}
\]

DSL_choice_pomdp(p', o) \leftarrow bestpol'

return \(p'\)

6.8 DSL_back_pomdp()

DSL_back_pomdp() is the method to generate a vector. The inputs are a given vector \(\alpha\), a given action \(a\) and a given observation \(o\). The output returns a vector that consists of the components of \(\alpha\) projected backwards one step by action \(a\) and observation \(o\).

Algorithm 6.8.1: DSL_BACK_POMDP(\(\alpha, a, o, Model\))

for each \(s \in S\)

\[
\begin{align*}
\text{a}'[s'] & \leftarrow \sum_{s} \alpha[s]T[s, a, s']O[s', a, o]
\end{align*}
\]

return \(\text{a}'\)

6.9 DSL_choice_pomdp()

DSL_choice_pomdp() is the method to get a policy tree. The inputs are a given \(t\)-step policy tree \(p\) and a given observation \(o\). The output returns a \(t-1\) step sub policy tree that follows the branch observation \(o\).

6.10 DSL_setchoice_pomdp()

DSL_setchoice_pomdp() is the method to set a policy tree. The inputs are a given \(t\)-step policy tree \(p\), a given \(t-1\) step policy tree \(p'\) and a given observation \(o\). The method sets the \(t-1\) step policy tree \(p'\) as a sub-tree of \(p\) on the branch observation \(o\).
6.11 DSL_valueP_pomdp()

DSL_valueP_pomdp() is the method to generate a vector. The inputs are a given policy tree $p$ and the POMDP model. The output returns a vector that stores the expected value for following policy $p$.

Algorithm 6.11.1: DSL_VALUEP_POMDP($p$, Model)

```plaintext
if (value($p$) undefined)
    \( a \leftarrow \text{action}(p) \)
    for each $s \in S$
        do \( \text{value}(p) \leftarrow R[s, a] + r \sum_{o} \text{DSL\_back\_pomdp(} DSL\_valueP\_pomdp(\text{DSL\_choice\_pomdp}(p, o), Model), a, o, Model)\[s]\)
    return (value($p$))
```

6.12 DSL_superior_pomdp()

DSL_superior_pomdp() is the method to compare two vectors. The inputs are two given vectors. The output returns the result which vector is lexicographically largest. Here we define lexicographically largest as follows. Vector $\alpha_1$ is lexicographically larger than vector $\alpha_2$ if $\alpha_1$’s first component is bigger than $\alpha_2$’s first component. If the first components are tied, the result will be decided by the second components, and so on.

6.13 DSL_findb_pomdp()

DSL_findb_pomdp() is the method to find a witness point. The inputs are a given action $a$, the minimal set of $t - 1$ step policy trees for representing $V_{t-1}$, the estimate set of policy trees for Q-function and the POMDP model. In the method, the linear programming will be used. The output returns the witness belief state.

Algorithm 6.13.1: DSL_FINDB_POMDP($a$, $\Upsilon_{t-1}$, $Q_t^p$, Model)

```plaintext
\begin{align*}
\text{repeat} & \quad \{(p, a, p') \leftarrow \text{checkme} \\
& \quad \beta \leftarrow \text{back(value}(p', \text{Model}), a, o, \text{Model}) - \text{back(value(choice}(p, o), \text{Model}), a, o, \text{Model}) \\
& \quad \text{comment: Linear Programming with } Q^p_t, \beta, p \\
& \quad (\text{objective}, b) \leftarrow \text{DSL\_lpfindwitness\_pomdp()} \\
& \quad \text{if (objective > 0)} \\
& \quad \quad \text{then return (b)} \\
& \quad \quad \text{checkme} \leftarrow \text{nextcheckme()} \\
& \quad \quad \text{if (checkme = nil)} \\
& \quad \quad \quad \text{then return (nil)} \\
\text{until}
\end{align*}
```
6.14 **DSL_lpfindwitnes_s_pomdp()**

DSL_lpfindwitnes_s_pomdp() is the linear programming method to find the witness belief state. It returns the conclusion if there is a witness belief state.

6.15 **DSL_purge_pomdp()**

DSL_purge_pomdp() is the method to purge extraneous policy trees from a set. The input is the set of policy trees $X$. In the method, the linear programming will be used. The output returns the policy tree set without extraneous policy trees.

**Algorithm 6.15.1: DSL_PURGE_POMDP($X$)**

\[
\begin{align*}
\Upsilon &\leftarrow \{\} \\
\text{for each } p \in X &\quad \text{do} \\
&\quad \text{(comment: Linear Programming with } p, X) \\
&\quad (\delta, b) \leftarrow DSL_lpfindpurge_pomdp() \\
&\quad \text{if } (\delta \leq) \\
&\quad \quad \text{then } X \leftarrow X - p \\
&\quad \quad \text{else } \Upsilon \leftarrow \Upsilon \cup p \\
\text{return } (\Upsilon)
\end{align*}
\]

6.16 **DSL_lpfindpurge_pomdp()**

DSL_lpfindpurge_pomdp() is the linear programming method to decide whether policy tree $p$ is extraneous in a given set. It returns a maximum value.

6.17 **DSL_computedifference_pomdp()**

DSL_computedifference_pomdp() is the method to compute the Bellman residual. The inputs are the minimal set of $t$ step policy trees for representing $V_t$ and the minimal set of $t-1$ step policy trees for representing $V_{t-1}$. In the method, the linear programming will be used. The output returns maximal difference between the two input set.

57
Algorithm 6.17.1: DSL\_COMPUTEDIFFERENCE\_POMDP($\mathcal{Y}_{t-1}, \mathcal{Y}_t$)

maxdifference $\leftarrow -\infty$

for each $p_t \in \mathcal{Y}_t$
  do
  for each $p_{t-1} \in \mathcal{Y}_{t-1}$
    do
      comment: Linear Programming with $p_t, \mathcal{Y}_t, p_{t-1}, \mathcal{Y}_{t-1}$
      $(\text{objective}_1, b) \leftarrow DSL\_lpfinddif\_pomdp()$
      if $(\text{objective}_1 = \text{nil})$
        then $\text{objective}_1 \leftarrow -\infty$
      do
      comment: Linear Programming with $p_{t-1}, \mathcal{Y}_{t-1}, p_t, \mathcal{Y}_t$
      $(\text{objective}_2, b) \leftarrow DSL\_lpfinddif\_pomdp()$
      if $(\text{objective}_2 = \text{nil})$
        then $\text{objective}_2 \leftarrow -\infty$
      do
      maxdifference $\leftarrow \max(\text{maxdifference, objective}_1, \text{objective}_2)$
  do

return $(\text{maxdifference})$

6.18 DSL\_lpfinddif\_pomdp()

DSL\_lpfinddif\_pomdp() is the linear programming method to compute the maximum value difference between two policy trees. It returns the maximum value.

6.19 DSL\_PBVI\_pomdp()

DSL\_PBVI\_pomdp() is the main function for Point-Based value iteration. The inputs are the initial set of belief points $B_{\text{init}}$, initial solution set $C_0$, the fixed number of belief point expansions $N$ and the planning horizon $T$. $T$ can be decided by $\gamma^T[R_{\text{max}} - R_{\text{min}}] < \epsilon$. The output returns the solution vector set $C$.  

Algorithm 6.19.1: DSL\_PBVI\_POMDP($B_{\text{init}}, C_0, N, T$)

$B \leftarrow B_{\text{init}}$
$C \leftarrow C_0$

for $i \leftarrow 0$ to $N$
  do
    for $j \leftarrow 0$ to $T$
      do
        $C \leftarrow DSL\_backup\_pomdp(B, C)$
        $B_{\text{new}} \leftarrow DSL\_expandbelief\_pomdp(B, C)$
        $B \leftarrow B \cup B_{\text{new}}$

return $(C)$
6.20 DSL_backup_pomdp()

DSL_backup_pomdp() is the method for Point-Based value backup. The inputs are the selected set of belief points $B$ and the $t-1$ step solution set $C_{t-1}$. The output returns the $t$ step solution set $C_t$.

Algorithm 6.20.1: DSL_BACKUP_POMDP($B, C_{t-1}$)

```plaintext
for each $a \in A$
  do
    for each $o \in O$
      do
        for each $\alpha_i \in C_{t-1}$
          do
            $\alpha_{i,a,o} = \gamma \sum_{s} T(s,a,s')O(s',a,o)\alpha_i(s'), \forall s \in S$
            $C_{t,a,o} = \cup_i \alpha_{i,a,o}$
        end
      end
    end
  end
end

for each $b \in B$
  do
    $\alpha_b = \arg\max_{a \in A} (\sum_s R(s,a)b(s) + \sum_o \max_{\alpha \in C_{t,a,o}} [\sum_s \alpha(s)b(s)])$
  end
end

if ($\alpha_b \notin C_t$)
  then $C_t \leftarrow C_t \cup \alpha_b$
end
return $(C_t)$
```

6.21 DSL_expandbelief_pomdp()

DSL_expandbelief_pomdp() is the method for belief expansion with random selection. The inputs are the selected set of belief points $B$ and the solution set $C$. The output returns the new set of belief points $B_{new}$.

Algorithm 6.21.1: DSL_EXPANDBELIEF_POMDP($B, C$)

```plaintext
$B_{new} \leftarrow B$
for each $b \in B$
  do
    comment: $S$ is the number of states
    for $i \leftarrow 0 \text{ to } S$
      do
        $b_{temp}[i] = \text{rand.uniform(0,1)}$
        comment: sort $b_{temp}$ in ascending order
    end
    for $i \leftarrow 1 \text{ to } S-1$
      do
        $b_{new}[i] = b_{temp}[i + 1] - b_{temp}[i]$
    end
$B_{new} \leftarrow B_{new} \cup b_{new}$
return $(B_{new})$
```
Chapter 7

Evaluation

7.1 MDP experiment

We compare the performance of these algorithms on a motion planning problem described in (Russel 1995)[22]. The goal of these experiment is to compare the Value Iteration (VI) and the Policy Iteration (PI) algorithms to show their difference in time.

7.1.1 Grid problem

The environment is described by a grid (see example in Table 7.1) of size $N \times M$. Squares are correspond to the states of the world. There is a start state an agent is initially in, and a set of terminal states. An agent is moving from one square to another until a terminal state is reached. In each location, there are four available actions, North, South, East and West, that supposedly move an agent one square in the intended direction, but achieving the desired direction only with some probability $p$. The rest of the time, the action moves the agent at right angles to the intended direction. For example, if the agent chooses the action North, it moves North with probability $p$, and with probability $(1-p)/2$ it moves East or West. If the agent cannot move in the intended direction because of an obstacle such as a wall, it stays in the current location. There is a reward function defined on all states, and the goal is to find an optimal policy.

Table 7.1: An example of an environment with reward function defined on states

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-1</td>
</tr>
<tr>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
</tr>
<tr>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
</tr>
<tr>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
</tr>
<tr>
<td>+1</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
</tr>
</tbody>
</table>
Table 7.2: The result of 5\times5 world

<table>
<thead>
<tr>
<th></th>
<th>-0.01(South)</th>
<th>-0.01(South)</th>
<th>-0.01(South)</th>
<th>-0.01(West)</th>
<th>-1(West)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.01(South)</td>
<td>-0.01(South)</td>
<td>-0.01(South)</td>
<td>-0.01(West)</td>
<td>-0.01(South)</td>
<td></td>
</tr>
<tr>
<td>-0.01(South)</td>
<td>-0.01(South)</td>
<td>-0.01(West)</td>
<td>-0.01(West)</td>
<td>-0.01(South)</td>
<td></td>
</tr>
<tr>
<td>-0.01(South)</td>
<td>-0.01(West)</td>
<td>-0.01(West)</td>
<td>-0.01(West)</td>
<td>-0.01(West)</td>
<td></td>
</tr>
<tr>
<td>+1(Stay)</td>
<td>-0.01(West)</td>
<td>-0.01(West)</td>
<td>-0.01(West)</td>
<td>-0.01(West)</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: Size of the world 5\times5

<table>
<thead>
<tr>
<th>Threshold</th>
<th>VI</th>
<th>PI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.03s</td>
<td>1.61s</td>
</tr>
<tr>
<td>0.5</td>
<td>0.05s</td>
<td></td>
</tr>
<tr>
<td>0.01</td>
<td>0.11s</td>
<td></td>
</tr>
<tr>
<td>0.005</td>
<td>0.16s</td>
<td></td>
</tr>
<tr>
<td>0.001</td>
<td>0.22s</td>
<td></td>
</tr>
</tbody>
</table>

7.1.2 Comparative analysis

In the experiments we varied the size of domain (the number of states) and the threshold of VI. Some of the results are shown in the table below. The following input data were used: two goal states, one with reward 1, the other with reward -1 (penalty), and reward -0.01 for all non-goal states; probability p = 0.7 of successfully performed action.

We have experimented with 2 different algorithms for computing an optimal policy in MDPs (Table 7.2, 7.3 and 7.4). For VI, the execution time increases dramatically when threshold changes to small. When the world size increases from 5\times5 to 10\times10, both the VI and PI algorithms spend more time and the complexity might increase dramatically. Besides, our experiments show that VI is slightly more efficient than PI on the grid problems we experimented with. And we have to note the behavior of all 2 algorithms depends on other parameters of the problem, such as the probability of success of actions.

Table 7.4: Size of the world 10\times10

<table>
<thead>
<tr>
<th>Threshold</th>
<th>VI</th>
<th>PI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.25s</td>
<td>3.95s</td>
</tr>
<tr>
<td>0.5</td>
<td>0.39s</td>
<td></td>
</tr>
<tr>
<td>0.01</td>
<td>1.08s</td>
<td></td>
</tr>
<tr>
<td>0.005</td>
<td>1.21s</td>
<td></td>
</tr>
<tr>
<td>0.001</td>
<td>1.50s</td>
<td></td>
</tr>
</tbody>
</table>
7.2 POMDP experiment

In this section, we look at some simulated POMDP domain to evaluate the empirical performance of our algorithms. The domain – tiger – is selected from the established POMDP literature[12]. The first goal of these experiment is to show that these algorithms can successfully solve the problems. We also compare the approximate and the exact algorithms to show their difference in time and accuracy.

7.2.1 Tiger problem

Imagine an agent standing in front of two closed doors. Behind one of the doors is a tiger and behind the other is a large reward. If the agent opens the door with the tiger, then a large penalty is received (presumably in the form of some amount of bodily injury) Instead of opening one of the two doors, the agent can listen, in order to gain some information about the location of the tiger. Unfortunately, listening is not free; in addition, it is also not entirely accurate. There is a chance that the agent will hear a tiger behind the left-hand door when the tiger is really behind the right-hand door, and vice versa.

We refer to the state of the world when the tiger is on the left as $S_l$ and when it is on the right as $S_r$. The actions are left, right, and listen. The reward for opening the correct door is $+10$ and the penalty for choosing the door with the tiger behind it is $-100$. The cost of listening is $-1$. There are only two possible observations: to hear the tiger on the left (TL) or to hear the tiger on the right (TR). Immediately after the agent opens a door and receives a reward or penalty, the problem resets, randomly relocating the tiger behind one of the two doors.

The transition and observation models can be described in detail as follows. The listen action does not change the state of the world. The left and right actions cause a transition to world state $S_l$ with probability $0.5$ and to state $S_r$ with probability $0.5$ (essentially resetting the problem). When the world is in state $S_l$, the listen action results in observation TL with probability $0.85$ and the observation TR with probability $0.15$; conversely for world state $S_r$. No matter what state the world is in, the left and right actions result in either observation with probability $0.5$.

7.2.2 Comparative analysis

We ran each of the two methods on a POMDP benchmark problem selected from the literature. The Figure 7.1 shows the non stationary solution policy for time $t = 4$. For the optimal infinite-horizon solution, the value function has a finite but large number of linear segments. The Figure 7.2 shows the solution vectors of Point-Based Value Iteration (PBVI) starts method. As the methods only consider some specific belief states, the result is not the optimal one but the time difference is enormous. The witness method(Table 7.5) always spends
Table 7.5: Approximate running times for Tiger Problem

<table>
<thead>
<tr>
<th>Name</th>
<th>Witness</th>
<th>PBVI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiger</td>
<td>3000s</td>
<td>15s</td>
</tr>
</tbody>
</table>

Figure 7.1: The optimal non-stationary policy for $t = 4$.

Figure 7.2: The solution vectors of Point-Based Value Iteration.

30 to 100 minutes, by contrast, the PBVI method always achieve the result no more than 15 seconds.
Chapter 8

Summary of Contributions and Open Problems

In this chapter, we will conclude the work and discuss the future work.

8.1 Summary of Contributions

8.1.1 Unifying various models

In the thesis, the relations of various probabilistic models in dynamic systems have been discussed. We try to give unifying representations for these models including DIDs, IDs, DBN, BN, HMMs, MDPs and POMDPs. We try to rewrite these models into the most general model Dynamic Influence Diagrams formalism. So it is convenient for the future theory and work in GeNIe and SMILE.

8.1.2 Implement Algorithms for HMMs, MPDs and POMDPs

After giving an unifying descriptions of all kinds of dynamic models, we choose to design and implement some algorithms not included in GeNIe and SMILE. These algorithms include Forward algorithm and Viterbi algorithm for HMMs, Value Iteration algorithm and Policy Iteration algorithm for MDPs, Witness algorithm and Point-Based approximate algorithm for POMDPs.

8.1.3 Compare the Algorithms for MPDs and POMDPs

We look at some simulated MDP and POMDP domain to evaluate the empirical performance of our algorithms. The goal of these experiment is to show the results of these algorithms and the difference between different algorithms in time and accuracy.
8.2 Open Problems

The most imagined future work is to extend the algorithms of GeNié and SMILE. We have developed some basic algorithms for MDPS and POMDPs, however, more algorithms are needed to solve more problems such as factored MDP models or Interactive POMDP Models. GeNié also need to be changed to accurately represent the new MDP and POMDP models.
Bibliography


67


Appendix A

Code of the Algorithms

\include{test3.cpp}

// test 3.cpp : Defines the entry point for the console application.
//
// #include "stdafx.h"

#include "stdafx.h"
#include <afx.h>

// #include "lp.lib.h"

#include "smile.h"
#include "dbnimpl.h"
#include "setoftree.h"

#include "ClpSimplex.hpp" // linear programming

// file op
#include <fstream>
#include <iostream>
#include <sstream>

#include <string>

#include <tchar.h>
#include <math.h>
#include <stdio.h>
#include <cstdlib>
//#include <time.h>
#define epsilon 0.001
#define N 100

using namespace std;

//#include <iostream.h>

// void CreateNetwork(void);
// void InferenceWithBayesNet(void);
// int demo();
// int demo2();
void demoCoin(void);
void mdpValueIteration(void);
void mdpPolicyIteration(void);
// void InferenceWithInfluenceDiagram(void);
// void ComputeValueOfInformation(void);
// Vector<DSL_rNetworks> *Subnets_Filter(Vector<
  DSL_rNetworks> *rnet_vector);
void pomdpWitness(void);
void solvepomdp(void);
void witness(SetOfTree&, int , SetOfTree&, int , int , int ,
  DSL_Dmatrix *, DSL_Dmatrix *, DSL_Dmatrix *, double);
void besttree(DSL_intArray&, int , DSL_doubleArray&,
  SetOfTree&, int , int , int , DSL_Dmatrix *, DSL_Dmatrix *,
  DSL_Dmatrix *, double);
void back (DSL_doubleArray&, DSL_doubleArray&, int , int ,
  int , int , DSL_Dmatrix *, DSL_Dmatrix *,
  DSL_Dmatrix *, double);
void choice (DSL_intArray&, DSL_intArray&, int , int ,
  DSL_Dmatrix *, doubleArray&);
void setchoice (DSL_intArray&, DSL_intArray&, int , int ,
  doubleArray&);
void valueP (DSL_doubleArray&, int , DSL_doubleArray&,
  SetOfTree & , int , int , DSL_Dmatrix *, DSL_Dmatrix *
  , double);
bool superior (DSL_doubleArray&, DSL_doubleArray&);
void findb(DSL_doubleArray&, int , SetOfTree&, int , SetOfTree & ,
  int , int , DSL_Dmatrix *, DSL_Dmatrix *
  , double);
double lpf indifferent witness ( DSL_doubleArray&, DSL_doubleArray&, 
DSL_intArray&, SetOfTree&, int, SetOfTree&, int, int, 
int, DSL_Dmatrix*, DSL_Dmatrix*, DSL_Dmatrix*, double );
void purge ( SetOfTree&, SetOfTree&, SetOfTree&, 
int, int, int, int, SetOfTree&, int, SetOfTree&, 
int, int, int, SetOfTree&, int, int, SetOfTree&, 
int, int, DSL_Dmatrix*, DSL_Dmatrix*, DSL_Dmatrix*, double );
double lpf indifferent purge ( DSL_intArray&, SetOfTree&, SetOfTree& 
, int, int, int, int, DSL_Dmatrix*, DSL_Dmatrix*, DSL_Dmatrix*, 
double );
double computedifference ( SetOfTree&, SetOfTree&, int, int, 
int, int, SetOfTree&, int, SetOfTree&, 
int, int, SetOfTree&, int, int, SetOfTree&, 
int, int, DSL_Dmatrix*, DSL_Dmatrix*, int, int, double );
double lpf indifferent difference ( DSL_intArray&, SetOfTree&, SetOfTree& 
, double );

// approximate for pomdp
void pomdpPBVI ( void );
void backup ( SetOfTree&, SetOfTree&, SetOfTree&, int, int, 
int, int, DSL_Dmatrix*, DSL_Dmatrix*, DSL_Dmatrix*, 
double );
void expandbelief ( SetOfTree&, SetOfTree&, SetOfTree&, int, 
int, int, DSL_Dmatrix*, DSL_Dmatrix*, DSL_Dmatrix*, 
double );
void forwardhmm ();
void viterbihmm ();
void testSetOfTree ();
void changeSetOfTree ( SetOfTree&, DSL_intArray& );
void computeAverageImage ();
void computeFeature ();

int main()
{

clock_t start, finish; // calculate time
double duration;
start = clock();

//CreateNetwork();
//InferenceWithBayesNet();
//mdpValueIteration();
//mdpPolicyIteration();
//InferenceWithInfluenceDiagram();
//ComputeValueOfInformation();
//pomdpWitness();
//demoCoin();
pomdpPBVI();
//forwardhmm();
//viterbihmm();
//testSetOfTree();
//cvexample();
//computeAverageImage();
//computeFeature();

finish = clock(); // calculate time
duration = (double)(finish - start) /
          CLOCKS_PER_SEC;
printf( "The run time is %f seconds \n", duration );

return (DSL_OKAY);
};

/* void CreateNetwork(void) {
 */
void mdpValueIteration(void) {

DSL_network theNet;
theNet.ReadFile("mdp55.xdsl");

// Vector<DSL_rNetwork*> *net_list = Relevant(
  theNet);

// test information of the Net
int numberOfNet =theNet.GetNumberOfNodes();
//cout<<"the node number of net is"<<numberOfNet
  <<endl;
printf ("the numberOfNet is: %i \n", numberOfNet) ;
int numberOfSample =theNet.GetNumberOfSamples();
printf ("the numberOfSample is: %i \n",
           numberOfSample);
int numberOfTarget =theNet.GetNumberOfTargets();
printf ("the numberOfTarget is: %i \n",
           numberOfTarget);
int NumberOfSlices =theNet.GetNumberOfSlices();
printf("the NumberOfSlices is: %i \n", NumberOfSlices);
int MaxTemporalOrder =theNet.GetMaxTemporalOrder();
printf("the MaxTemporalOrder is: %i \n", MaxTemporalOrder);

int state = theNet.FindNode("State");
int action = theNet.FindNode("Action");
int rewards = theNet.FindNode("Rewards");
int sizeOfState = theNet.GetNode(state)->
Definition()->GetSize();
printf("the size of State is: %i \n", sizeOfState);
int sizeOfAction = theNet.GetNode(action)->
Definition()->GetSize();
printf("the size of Action is: %i \n", sizeOfAction);

// initialize the value array;
double minValue = -100000.0;

DSL_doubleArray ValueOfState(sizeOfState);
for (int i=0; i<ValueOfState.GetSize(); i++)
{
    ValueOfState[i]=0.0;
    // printf("the value array is: %i %f\n", i , ValueOfState[i]);
}

DSL_doubleArray PreValueOfState(sizeOfState);
for (int i=0; i<PreValueOfState.GetSize(); i++)
{
    PreValueOfState[i]=0.0;
    // printf("the prevalue array is: %i %f\n" , i , PreValueOfState[i]);
}

DSL_intArray ActionOfState(sizeOfState);
for (int i=0; i<ActionOfState.GetSize(); i++)
{
    ActionOfState[i]=-1;
    // printf("the ActionOfState is: %i \n", i , ActionOfState[i]);
}
DSL_nodeDefinition *temporalStateDef = theNet.
GetDbn() -> GetDefinition(state, 1);
DSL_Dmatrix *probsOfState = temporalStateDef ->
GetMatrix();
for (int i = 0; i < probsOfState -> GetSize(); i++)
{
    // printf(" probOfState \[ \] is : %i %f \n", i,
            (*probsOfState)[i]);
}

DSL_nodeDefinition *RewardsDef = theNet. GetNode(
    rewards) -> Definition();
DSL_Dmatrix *probsOfRewards = RewardsDef ->
    GetMatrix();
for (int i = 0; i < probsOfRewards -> GetSize(); i++)
{
    // printf(" probOfRewards \[ \] is : %i %f \n", i,
            (*probsOfRewards)[i]);
}

double Q = minValue;
double maxQ = Q;
double R = 0.0;
double T = 0.0;
DSL_intArray odmdpT(3);
DSL_intArray odmdpR(2);
int logR = 0;
int logT = 0;
int flagAction = -1;
bool flagStop = TRUE;
int itime = 0;

do{
    itime++;
    for (int istate = 0; istate < sizeOfState;
         istate++)
    {
        // ...
    }
}
maxQ = minValue;
flagAction = -1;
for (int iaction=0;iaction<
    sizeOfAction; iaction++)
{
    //odmdpR[0]=istate;
    //odmdpR[1]=iaction;
    //odmdpT[0]=istate;
    //odmdpT[1]=iaction;
    //R = (*probsOfRewards)[
        odmdpR];
    logR = istate* 
        sizeOfAction+iaction;
    R = (*probsOfRewards)[
        logR];
    //debug
    //printf(" probOfRewards []
        is : S %i A %i %f\n", 
            istate, iaction ,R);
    double sumOfPreValue
        =0.0;
    for (int istate1=0;
        istate1<sizeOfState;
        istate1++)
    {
        logT = istate* 
            sizeOfAction* 
            sizeOfState+ 
            iaction* 
            sizeOfState+ 
            istate1;
        T = (* 
            probsOfState)[
            logT];
        //debug
        //printf(" 
            probOfState []
            is : %f\n",T)
            ;
        sumOfPreValue = 
            sumOfPreValue+ 
            T* 
            PreValueOfState
            [istate1];
        //debug
// printf("sumOfPreValue is: \%f\n", sumOfPreValue);

} Q = R + 0.95 * sumOfPreValue; // discount very important
// debug
// printf("Q is: \%f\n", Q);
if (Q >= maxQ)
{
    maxQ = Q;
    flagAction = iAction;
}
// debug
// printf("maxQ is: \%f\n", maxQ);

}
// for istate
ValueOfState[istate] = maxQ;
ActionOfState[istate] = flagAction;

}

// for istate
ValueOfState[istate] = maxQ;
ActionOfState[istate] = flagAction;

double diffMax = 0;
for (int i=0;i<ValueOfState.GetSize();i++)
{
    // ValueOfState[i]=0.0;
    if (fabs(PreValueOfState[i] - ValueOfState[i])>diffMax)
    {
        diffMax = fabs(PreValueOfState[i] - ValueOfState[i])
        // ValueOfState[i]=0.0;
        if (fabs(PreValueOfState[i] - ValueOfState[i])>diffMax)
        {
            diffMax = fabs(PreValueOfState[i] - ValueOfState[i])
ValueOfState[i]);

    }
    PreValueOfState[i] = ValueOfState[i];
    // printf("the diff is: %f\n", diffMax);
    // printf("the value, array is:
    // time %i state %i %f\n", itime, i, ValueOfState[i]);
    }
    if (diffMax<epsion)
    {
        flagStop = FALSE;
    }
    /*
    for (int i=0;i<ActionOfState.GetSize();i++)
    {
        printf("the Action is: time %i state %i %i
", itime, i, ValueOfState[i]);
    }
    */
} while(flagStop);

for (int i=0;i<ActionOfState.GetSize();i++)
{
    printf("the Action is: time %i state %i %i
", itime, i, ActionOfState[i]);
}

    */

DSL_sysCoordinates theCoordinates (*theNet.
    GetNode(state)->Definition());
for (int i=0;i<3;i++)
{
    printf("prob[] is : %f\n", theCoordinates.
    UncheckedValue());
    theCoordinates.Next();
77
/
DSL_doubleArray *arr:
theNet.GetNode(state)->Definition()->GetDefinition(&arr);
printf(“prob[arr0] is : %f\n”, arr[0]);
printf(“prob[arr1] is : %f\n”, arr[1]);
printf(“prob[arr2] is : %f\n”, arr[2]);
//const DSL_doubleArray* GetTemporalDefinition(
int order);
/*
const DSL_doubleArray* theProbs = theNet.GetNode(
state)->Definition()->GetTemporalDefinition(1);
printf(“the size of temporal prob[] is : %i\n”,
theProbs->GetSize());
for (int i=0;i<theProbs->GetSize();i++)
{
printf(“prob[] is : %f\n”, theProbs[i]);
}*/

/*
DSL_nodeDefinition *temporalDef = theNet.GetDbn()
->GetDefinition(state, 1);
DSL_Dmatrix *probs = temporalDef->GetMatrix();
//printf(“prob[] is : %f\n”, (*probs)[i]);
for (int i=0;i<probs->GetSize();i++)
{
printf(“prob[] is : %i %f\n”, i, (*probs)[
i]);
}*/

//theNet.writeFile(“mdp.xdsl”);
*/

void InferenceWithInfluenceDiagram(void) {
}

void ComputeValueOfInformation(void) {
}
};*/

78
void mdpPolicyIteration(void) {
    DSL_network theNet;
    theNet.ReadFile("mdp55.xdsl");

    // Vector<DSL_rNetwork*> *net_list = Relevant(
    // theNet);

    // test information of the Net
    int numberOfNet = theNet.GetNumberOfNodes();
    // cout<<"the node number of net is"<<numberOfNet
    // <<endl;
    printf("the number of Net is: %i 
", numberOfNet);
    int numberOfSample = theNet.GetNumberOfSamples();
    printf("the number of Sample is: %i \n", numberOfSample);
    int numberOfTarget = theNet.GetNumberOfTargets();
    printf("the number of Target is: %i \n", numberOfTarget);
    int NumberOfSlices = theNet.GetNumberOfSlices();
    printf("the Number of Slices is: %i \n", NumberOfSlices);
    int MaxTemporalOrder = theNet.GetMaxTemporalOrder();
    printf("the MaxTemporalOrder is: %i \n", MaxTemporalOrder);

    int state = theNet.FindNode("State");
    int action = theNet.FindNode("Action");
    int rewards = theNet.FindNode("Rewards");
    int sizeOfState = theNet.GetNode(state)->
    Definition()->GetSize();
    printf("the size of State is: %i \n", sizeOfState);
    int sizeOfAction = theNet.GetNode(action)->
    Definition()->GetSize();
    printf("the size of Action is: %i \n", sizeOfAction);

    const double minValue = -100000.0;

    DSL_doubleArray ValueOfState(sizeOfState);
    for (int i = 0; i < ValueOfState.GetSize(); i++)
    {
        ValueOfState[i] = 0.0;
    }
}
DSL doubleArray PreValueOfState(sizeOfState);
for (int i=0; i<PreValueOfState.GetSize(); i++)
{
    PreValueOfState[i] = 0.0;
    // printf("the prevalue array is: %i %f\n", i, PreValueOfState[i]);
}

DSL_nodeDefinition *temporalStateDef = theNet.GetDbn()->GetDefinition(state, 1);
DSL_Dmatrix *probsOfState = temporalStateDef->GetMatrix();
for (int i=0; i<probsOfState->GetSize(); i++)
{
    // printf("probOfState[] is: %i %f\n", i, (*probsOfState)[i]);
}

DSL_nodeDefinition *RewardsDef = theNet.GetNode(rewards)->Definition();
DSL_Dmatrix *probsOfRewards = RewardsDef->GetMatrix();
for (int i=0; i<probsOfRewards->GetSize(); i++)
{
    // printf("probOfRewards[] is: %i %f\n", i, (*probsOfRewards)[i]);
}

DSL_intArray ActionOfState(sizeOfState);// store policy
DSL_intArray PreActionOfState(sizeOfState);// store policy before
//initialize ActionOfState[];
for (int i=0; i<sizeOfState; i++)
{
    ActionOfState[i] = -1;// use action 0 as initial action;
    printf("the initial ActionOfState is: %i %i \n", i, ActionOfState[i]);
}
for (int i=0; i<sizeOfState; i++)
PreActionOfState[i] = -2;// use action 0 as initial action;
printf("the initial ActionOfState is: \%i \n", i, PreActionOfState[i]);

double Q = minValue;
double maxQ = Q;
double R = 0.0;
double T = 0.0;
DSL_intArray odmdpT(3);
DSL_intArray odmdpR(2);
int logR = 0;
int logT = 0;
int flagAction = -1;
int row = sizeOfState;
int column = sizeOfState + 1;
DSL_doubleArray paraMatrix(row * column);

// give random policy
for (int i = 0; i < sizeOfState; i++)
{
    double MaxR = minValue;
    for (int j = 0; j < sizeOfAction; j++)
    {
        logR = i * sizeOfAction + j;
        R = (*probsOfRewards)[logR];
        if (R >= MaxR)
        {
            MaxR = R;
            ActionOfState[i] = j;
        }
    }
    printf("the initial ActionOfState is: \%i \n", i, ActionOfState[i]);
}
do{

PreActionOfState = ActionOfState;
for (int istate=0; istate<sizeOfState; istate++)
{
    int iaction = ActionOfState[istate];
    logR = istate * sizeOfAction + iaction;
    R = (*probsOfRewards)[logR];
    //debug
    // printf("probOfRewards[] is : %i A %i %f\n", istate, iaction, R);

    DSL_doubleArray ParameterOfT(sizeOfState);
    for (int istate1=0; istate1<sizeOfState; istate1++)
    {
        logT = istate * sizeOfAction * sizeOfState + iaction * sizeOfState + istate1;
        T = (*probsOfState)[logT];
        //debug
        // printf("probOfState[] is : %f\n", T);
        ParameterOfT[istate1]=T;
        //debug
        // printf("ParameterOfT[%i]: %f\n", istate1, T);
    }
}

82
\begin{verbatim}

 istate1,
 ParameterOfT[
  istate1]);

) } 

DSL_doubleArray 
    ParameterOfEquation ( 
        sizeOfState+1); 
    for (int istate1 =0; 
        istate1<sizeOfState; 
        istate1++) 
    { 
        if (istate1 == 
            istate) 
            { 
                ParameterOfEquation 
                    [ 
                        istate1 
                    ] = 
                ParameterOfT 
                    [ 
                        istate1 
                    ] -1;
                    
            } 
        else 
        { 
            ParameterOfEquation 
                [ 
                    istate1 
                ] = 
            ParameterOfT 
                [ 
                    istate1 
                ];
            
        
    } 
    ParameterOfEquation [ 
        sizeOfState] = -R;

for (int i =0; i< 
    ParameterOfEquation. 
    GetSize(); i++) 
{ 
    paraMatrix [istate 
        *column+i] = 

83
\end{verbatim}
ParameterOfEquation
[i];
printf("parameterOfEquation
[i]: %f ", i,
ParameterOfEquation
[i]);
}
printf("\n");
}
// time
for (int i = 0; i < row; i++)
{
    for (int j = 0; j < column; j++)
    {
        printf("%f ", paraMatrix[i *
column + j]);
    }
    printf("\n");
}
DSL_doubleArray ResultOfEquation(s
izeOfState);
DSL_doubleArray TempOfEquation(s
izeOfState + 1);
for (int i = row - 1; i >= 0; i --)
{
    if (paraMatrix[i * column + i] == 0)
    {
        row = row - 1;
        ResultOfEquation[i] = 0;
    }
}
// use gaussian to solve Equations
int flagOfrow = -1;
for (int r = 0; r < row; r++)
{
    double temp = 0.0;
    for (int i = r; i < row; i++)
    
    84
if (fabs(paraMatrix[i*column+r]) > temp)
{
    temp = fabs(paraMatrix[i*column+r]);
    flagOfRow = i;
    printf(" exchange r:%i flagOfRow : %i\n", r, flagOfRow);
}

double maxOfRow = temp;

if (maxOfRow == 0)
    //
{
    printf(" invalid equations ");
}
else if (flagOfRow != r) // exchange tow rows
{
    printf(" exchanged exchanged\n");
    for (int j=0; j<column; j++)
        TempOfEquation[j] = paraMatrix[r*column+j];
    for (int j=0; j<column; j++)
        paraMatrix[r*column+j] = paraMatrix[flagOfRow*column+j];
    for (int j=0; j<column; j++)
        paraMatrix[flagOfRow*column+j] = TempOfEquation[j];
}
for (int i=r+1; i<row; i++)
{
    double local = paraMatrix[i*column+r]/
                   paraMatrix[r*column+r];
    for (int j=r; j<column; j++)
    {
        paraMatrix[i*column+j]=
                    paraMatrix[i*column+j] -
                    paraMatrix[r*column+j]*
                    local;
    }
}
for (int i=0; i<row; i++)
{
    for (int j=0; j<column; j++)
    {
        printf("%f ",
               paraMatrix[i*column+j]);
    }
    printf("\n");
}

ResultOfEquation[row-1]= paraMatrix[(row
-1)*column+(column-1)]/ paraMatrix[(row
-1)*column+(row-1)];
if (row>=2)
{
    for ( int r=row-2; r>=0; r--)
    {
        double local=0.0;
        for (int j=r+1; j<row; j++)
        {
            local=local+paraMatrix[r*column+j]*
            ResultOfEquation[j];
        }
    }
}


```c
for (int i=0; i<ResultOfEquation.GetSize(); i++) {
    printf("%f ", ResultOfEquation[i]);
}
printf("\n");

// in the time
// find a better policy
for (int istate=0; istate<row; istate++) {
    double localMax=minValue;
    for (int iaction=0; iaction<sizeOfAction; iaction++) {
        logR = istate * sizeOfAction + iaction;
        R = (*probsOfRewards)[logR];
        double local=0.0;
        for (int istate1=0; istate1<sizeOfState; istate1++) {
            logT = istate * sizeOfAction * sizeOfState + iaction * sizeOfState +
        }  
```
\text{istate1};
T = (*probsOfState)[logT];
//debug
//printf(" probOfState [] is : %f\n", T);
local = local + T * ResultOfEquation[istate1];
//debug
//printf(" ParameterOfT[%i]: %f\n", istate1, ParameterOfT[istate1]);
}
local = local + R;
if (local > localMax)
{
localMax = local;
ActionOfState[istate] = iaction;
}
}
printf(" better action policy are:");
for (int i = 0; i < ActionOfState.GetSize(); i++)
{
printf("%i", ActionOfState[i]);
}
printf("\n");
while (! (PreActionOfState == ActionOfState));
void pomdpWitness(void) {
    // test my set of tree
    /*
    int num = 6;
    DSL.intArray int0(num);
    for (int i = 0; i < 6; i++)
        int0[i] = -1;
    */
    DSL.intArray int1(num);
    for (int i = 0; i < 6; i++)
        int1[i] = 1;
    DSL.intArray int2(num);
    for (int i = 0; i < 6; i++)
        int2[i] = 2;
    DSL.intArray int3(num);
    for (int i = 0; i < 6; i++)
        int3[i] = 3;
    SetOfTree tree(int0);
    tree.Insert(int1);
    tree.Insert(int2);
    tree.Insert(int3);
    tree.Print();
    tree.Find(int2);
    tree.Remove();
    tree.Print();
    */
    // end test setoftree
}
solvepomdp();

void solvepomdp()
{
    printf("we are going to solve pomdp with witness\n") ;
    DSL_network theNet ;
    theNet.ReadFile("pomdp.xdsl");

    //Vector<DSL_rNetwork*> *net_list = Relevant(
        theNet);

    //test information of the Net
    int numberOfNet =theNet.GetNumberOfNodes();
    //cout<<"the node number of net is"<<numberOfNet
    <<endl;
    printf("the number Of nodes in the net is: %i\n",
        numberOfNet);;
    int numberOfSample =theNet.GetNumberOfSamples();
    printf("the numberOfSample is: %i\n",
        numberOfSample);;
    int numberOfTarget =theNet.GetNumberOfTargets();
    printf("the numberOfTarget is: %i\n",
        numberOfTarget);;
    int NumberOfSlices =theNet.GetNumberOfSlices();
    printf("the NumberOfSlices is: %i\n",
        NumberOfSlices);;
    int MaxTemporalOrder =theNet.GetMaxTemporalOrder()
        ();
    printf("the MaxTemporalOrder is: %i\n",
        MaxTemporalOrder);
    //initialize the value array;
    double minValue = -100000.0;

    //get the data from the model
    int state = theNet.FindNode("State");
    int action = theNet.FindNode("Action");
    int rewards = theNet.FindNode("Rewards");
    int observation = theNet.FindNode("Observation");

}
int sizeOfState = theNet.GetNode(state)->
   Definition()->GetSize();
printf(" the size of State is: %i \n",
   sizeOfState);
int sizeOfAction = theNet.GetNode(action)->
   Definition()->GetSize();
printf(" the size of Action is: %i \n",
   sizeOfAction);
int tempsizeOfObservation = theNet.GetNode(
   observation)->Definition()->GetSize();
int sizeOfObservation = tempsizeOfObservation/
   sizeOfState;
printf(" the size of Observation is: %i \n",
   sizeOfObservation);

DSL_nodeDefinition *temporalStateDef = theNet.
   GetDbn()->GetDefinition(state, 1);
DSL_Dmatrix *probsOfState = temporalStateDef->
   GetMatrix();
for (int i=0;i<probsOfState->GetSize();i++)
{
   printf(" probOfState [] is: %i %f\n",i, (*
      probsOfState)[i]);
}

DSL_nodeDefinition *temporalObservationDef =
   theNet.GetDbn()->GetDefinition(observation, 1);
DSL_Dmatrix *probsOfObservation =
   temporalObservationDef->GetMatrix();
for (int i=0;i<probsOfObservation->GetSize();i++)
{
   printf(" probsOfObservation [] is: %i %f\n",
      i, (*probsOfObservation)[i]);
}

DSL_nodeDefinition *RewardsDef = theNet.GetNode(
   rewards)->Definition();
DSL_Dmatrix *probsOfRewards = RewardsDef->
   GetMatrix();
for (int i=0;i<probsOfRewards->GetSize();i++)
{

printf("probOfRewards[] is: %i %f\n", i, (*probsOfRewards)[i]);
}

int initialNum = 1;
DSL_intArray initialHead(initialNum);
for (int i = 0; i < initialNum; i++)
{
    initialHead[i] = -1;
    printf("head nod %i\n", initialHead[i]);
}

DSL_doubleArray initialHeadValue(initialNum);
for (int i = 0; i < initialNum; i++)
{
    initialHeadValue[i] = 0.0;
    printf("head nod value %f\n", initialHeadValue[i]);
}

SetOfTree vPrevious(initialHead, initialHeadValue);

double discount = 0.75;
double valueOfDifference = 1;

int t = 1;
do {

    SetOfTree unionQta(initialHead, initialHeadValue);
    unionQta.First();

    int localSize = 1;
    for (int i = 0; i < t; i++)
    {
        localSize = localSize * sizeOfObservation;
    }
    int sizeOfTree = (localSize - 1)/(sizeOfObservation - 1);

    92
for (int iaction = 0; iaction < sizeOfAction; iaction++)
{
    // SetOfTree qTI(initialHead, initialHeadValue);
    SetOfTree Qta(initialHead, initialHeadValue);
    witness(Qta, t, vPrevious, iaction,
            sizeOfState, sizeOfAction,
            sizeOfObservation, probsOfState
            , probsOfObservation
            , probsOfRewards, discount);

    /*
    Qta.Print();
    printf("let me focus on Qta: Qta length is %i\n", Qta.GetLength());
    */
    Qta.First();
    // printf("Here!!!!\n");
    for (int i=0; i<Qta.GetLength(); i++)
    {
        DSL_intArray polTree(sizeOfTree);
        polTree = Qta.GetData();
        // printf("Here!!!!\n");
        /*
        printf("the Qta.GetData() is:");
        for (int i = 0; i<Qta.GetData().GetSize(); i++)
        {
            printf(" %i", Qta.GetData()[i]);
        }
        printf("\n");
    */
}
printf("the pol Tree is ":");
for (int i = 0; i<polTree
 .GetSize(); i++)
{
    printf(" %i ",
polTree[i]);
}
printf("\n");
/*

DSL_doubleArray
    valueOfPolTree{
        sizeOfState;
    valueP(valueOfPolTree , t,
        polTree , vPrevious,
        sizeOfState,
        sizeOfAction,
        sizeOfObservation,
        probsOfState,
        probsOfObservation,
        probsOfRewards,
        discount);

    unionQta . Insert (polTree ,
        valueOfPolTree);// get
    the unionQta
    Qta . Next() ;
}
/*
 printf("in loop the unionQta set
 is: ");
 unionQta . Print();*/
}

//print unionQta
printf("the step is : %i\n", t);
printf("the unionQta set is:");
unionQta . Print();
printf(" finish unionQta\n");

//now purge

94
SetOfTree vCurrent (initialHead, initialHeadValue);
//vCurrent.Print();

purge(vCurrent, unionQta, vPrevious, t,
    sizeOfState, sizeOfAction,
    sizeOfObservation, probsOfState,
    probsOfObservation, probsOfRewards,
    discount);

valueOfDifference = computedifference(
    vCurrent, vPrevious, t, sizeOfState,
    sizeOfAction, sizeOfObservation,
    probsOfState, probsOfObservation,
    probsOfRewards, discount);

//now transfer vCurrent to vPrevious
//clean vPrevious
printf("the step is %i
", t);
printf("before clean the previous set is 
");
vPrevious.Print();
printf("\n");

vPrevious.First();

int lengthOfSet = vPrevious.GetLength();
for (int ipol = 0; ipol < lengthOfSet; ipol++)
{
    vPrevious.Remove();
    //vPrevious.Next(); do not need
    //next();
    //printf("in clean the previous set is %i
", ipol);
    //vPrevious.Print();
    //printf("\n");
}
printf("the step is %i
", t);

printf("after clean the previous set is:
");
vPrevious.Print();
printf("\n");

95
vCurrent.First();
vPrevious.First();
for (int ipol = 0; ipol < vCurrent.GetLength(); ipol++) {
    vPrevious.Insert(vCurrent.GetData(), vCurrent.GetValue());
    vPrevious.m_current->SetFlag();
    vCurrent.Next();
}

} while (valueOfdifference < 30);
// t < 5
// printf("the step is : %i \n", t);
printf("the result of POMDPs is: \n");
vPrevious.Print();

// only for testing 2 states world
printf("the belief world is: ");

vPrevious.First();
for (int i = 0; i < vPrevious.GetLength() - 1; i++) {
    //
    TreeNode* localpvPrevious = vPrevious.m_current;
    double x = 0.0;
    double maxY = -100000.0;
    double v0 = vPrevious.GetValue()[0];
    double v1 = vPrevious.GetValue()[1];
    vPrevious.First();
    for (int j = 0; j < vPrevious.GetLength(); j++) {
        if (i != j) {
            double t0 = vPrevious.Getvalue()[0];
        }
double t1 = vPrevious.Getvalue()[1];

double localx = (v0 - t0) / (t1 - t0 - v1 + v0);
double localy = (v1 - v0) * localx + v0;
// printf("v0= %f, v1= %f; 
  t0= %f, t1= %f; x= %f, 
  y=%f\n", v0, v1, t0, t1,
  localx, localy);
if (localy >= maxY)
{
  maxY = localy;
  // printf(" 
    localmaxY %f\n 
  ", localy);
  // printf("localx 
    %f\n", localx)
  ;
  x = localx;
}

vPrevious.Next();

printf(" %f\n", x);
vPrevious.m_current = localpvPrevious;
vPrevious.Next();

}
printf("\n");
// valueOfDifference > epsilon

void witness(SetOfTree& Qta, int t, SetOfTree& vPrevious, 
  int interaction, int sizeOfState, int sizeOfAction, int 
  sizeOfObservation, DSL_Dmatrix* probsOfState, 
  DSL_Dmatrix* probsOfObservation, DSL_Dmatrix* 
  probsOfRewards, double discount)
{
  // save the pointer of Qta and vPrevious;
  // treeNode* punionQta = Qta.m_current;
  treeNode* punionQta = Qta.m_current;
  treeNode* pvPrevious = vPrevious.m_current;
//test only

/*
 printf(“Just in witness, Qta: \n”);
 Qta.Print();*/

/*
for (int i=0;i<probsOfState->GetSize();i++)
{
    printf(“in probOfState[] is: %i %f
”,i, (*probsOfState)[i]);
}

for (int i=0;i<probsOfObservation->GetSize();i++)
{
    printf(“in probsOfObservation[] is: %i %f
”,i, (*probsOfObservation)[i]);
}

for (int i=0;i<probsOfRewards->GetSize();i++)
{
    printf(“in probOfRewards[] is: %i %f
”,i, (*probsOfRewards)[i]);
}*/

//printf(“Now in witness!!!\n”);
DSL_doubleArray belief(sizeOfState);
for (int i=0;i<belief.GetSize();i++)
{
    belief[i]=0.0;
    //printf(“the value_array is: %i %f\n”, i, ValueOfState[i]);
}
bias[0]=1.0;

int localSize = 1;
for (int i =0; i<t; i++)
{
    localSize = localSize*sizeOfObservation;
}
int sizeOftree = (localSize - 1)/(sizeOfObservation - 1);
// witness information
/*
printf(" Begin in witness t: %i o: %i localSize: %i the size of tree is %i action is %i \n", t, sizeOfObservation, localSize, sizeOftree, action);
vPrevious.Print();
*/
DSLIntArray newPol(sizeOftree);

besttree(newPol, t, belief, vPrevious, action, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState, probsOfObservation, probsOfRewards, discount);

DSL_doubleArray newValueOfPol(sizeOfState);
valueP(newValueOfPol, t, newPol, vPrevious, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState, probsOfObservation, probsOfRewards, discount);
Qta.Insert(newPol, newValueOfPol);

DSL_doubleArray newbelief(sizeOfState);
newbelief[0]=-1;// set a flag

findb( newbelief, action, vPrevious, t, Qta, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState, probsOfObservation, probsOfRewards, discount);

while (newbelief[0]!=-1)
{
    besttree(newPol, t, newbelief, vPrevious, action, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState,
}
probsOfObservation, probsOfRewards, 
discount);

// DSL_doubleArray newValueOfPol( 
sizeOfState);
valueP(newValueOfPol, t, newPol, vPrevious, 
sizeOfState, sizeOfAction, 
sizeOfObservation, probsOfState, 
probsOfObservation, probsOfRewards, 
discount);
Qta.Insert(newPol, newValueOfPol);
newbelief[0] = -1; // set a flag

findb( newbelief, iaction, vPrevious, t, 
Qta, sizeOfState, sizeOfAction, 
sizeOfObservation, probsOfState, 
probsOfObservation, probsOfRewards, 
discount);

}

// for test
/*

printf("Qta: ");
Qta.Print();
printf("End of witness \n");*/

// give back the pointer of Qta and vPrevious;
//Qta.m_current= punionQta;
vPrevious.m_current=pvPrevious;

}

void besttree(DSL_intArray& newPol, int t, DSL_doubleArray& 
belief, SetOfTree& vPrevious, int iaction, int 
sizeOfState, int sizeOfAction, int sizeOfObservation, 
DSL_Dmatrix* probsOfState, DSL_Dmatrix* 
probsOfObservation, DSL_Dmatrix* probsOfRewards, double 
discount)
{

// save the pointer of Qta and vPrevious;
treeNode* pvPrevious = vPrevious.m_current;


/*
  printf("Now in besttree!!!\n");
  printf("the belief state is :["");
  for (int i =0; i<belief.GetSize(); i++)
  {
    printf(" %f ", belief[i]);
  }
  printf("\n");
*/

int localSize = 1;
for (int i =0; i<t; i++)
{
  localSize = localSize*sizeOfObservation;
}

int sizeOftree =(localSize -1)/(sizeOfObservation -1);
printf("t: %i o: %i localSize: %i the size of
tree is %i \n",t,sizeOfObservation,localSize ,
sizeOftree);
DSLIntArray pNewTree(sizeOftree);
*/

int localSize1 = 1;
int sizeOftree1 = 0;
for (int i =0; i<t-1; i++)
{
  localSize1 = localSize1*sizeOfObservation
  ;
}
if ( t == 1)
{
  sizeOftree1 = 0;
}
else
{

101
sizeOfTree1 = (localSize1 - 1) / (sizeOfObservation - 1);

// compute the size of t-1 policy tree;
// printf("Here!!!\n");

newPol[0] = iaction;
double minvalue = -10000;
for (int iobservation = 0; iobservation < sizeOfObservation; iobservation++)
{
    DSL_intArray bestpol(sizeOfTree1);
    DSL_doubleArray bestvec(sizeOfState);
    for (int i = 0; i < sizeOfState; i++)
    {
        bestvec[i] = minvalue;
    }
    double bestval = minvalue;
    vPrevious.First(); // point to the first node which has data;
    DSL_doubleArray valueOfPol(sizeOfState);
    for (int ipol = 0; ipol < vPrevious.GetLength(); ipol++)
    {
        valueP(valueOfPol, t, vPrevious, Getdata(), vPrevious, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState, probsOfObservation, probsOfRewards, discount);
        // printf("Here!!!\n");
        DSL_doubleArray vec(sizeOfState);
        back(vec, valueOfPol, iaction, iobservation, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState, probsOfObservation, probsOfRewards, discount);
        double val = 0.0;
    }
}
for (int i = 0; i < sizeOfState; i++)
{
    val = val + vec[i] * belief[i];
}

if ((val > bestval) || ((fabs(val - bestval) < epsilon) && superior(vec, bestvec)))
{
    bestpol = vPrevious;
    Getdata();
    bestvec = vec;
    bestval = val;
}

vPrevious.Next();

// printf("Here!!!\n");

setchoice(bestpol, newPol, iobservation, sizeOfObservation, t);

/*
 printf("the best policy of action %i is ", iaction);
 for (int i = 0; i < newPol.GetSize(); i++)
 {
     printf("%i", newPol[i]);
 }
 printf("\n"); */

// give back the pointer of Qta and vPrevious;

vPrevious.m_current = pvPrevious;

// printf("End of besttree\n");

}
DSL_Dmatrix* probsOfRewards, double discount) {

    // printf("Now in back!!!\n");
    for (int istate = 0; istate < sizeOfState; istate++)
        // for each s in S
        { 
            alphaSharp[istate] = 0.0;
            for (int istate1 = 0; istate1 < sizeOfState; istate1++)
            { 
                int logT = istate * sizeOfAction * sizeOfState + istate * sizeOfState + istate1;
                double T = (*probsOfState)[logT];
                int logO = istate1 * sizeOfAction * sizeOfObservation + istate1 * sizeOfObservation + iobservation;
                double O = (*probsOfObservation)[logO];

            }
        }

    // printf("End of back!!!\n");
}

void choice (DSL_intArray& polPre, DSL_intArray& polCur, int iobservation, int sizeOfObservation, int t)
{
    // the P t-1 saved in polPre
    // printf("Now in choice!!!\n");

    int sizeOfpolCur = polCur.GetSize();
    sizeOfpolPre = (sizeOfpolCur - 1)/sizeOfObservation
    int sizeOfpolCurPre = polPre.GetSize();


// printf("size of polPre is \%i\n",sizeofpolCurPre)
if (sizeofpolCurPre == 0)
{
    return;
}
int depthCur = t;
int depthPre = t - 1;
int sizeOfLevel = 1;
int log = 0;
polPre[log] = polCur[iobservation+1];
log++;
int logParent = iobservation+1+1;
for (int i = 1; i < t-1; i++)
{
    sizeOfLevel = sizeOfLevel*
        sizeOfObservation;
    int logChild = logParent *
        sizeOfObservation - sizeOfObservation
            +2 -1;
    logParent = logChild+1;
    for (int j = 0; j < sizeOfLevel; j++)
    {
        polPre[log] = polCur[logChild];
        logChild++;
        log ++;
    }
}

void setchoice (DSL_intArray& polPre, DSL_intArray& polCur,
    int iobservation, int sizeOfObservation, int t)
{
    // insert a t-1 policy tree to a t policy tree,
    // in iobservation branch;
    // printf("Now in setchoice !!!\n");
    // printf("iobservation is \%i ",iobservation);
    // printf("size of observation is \%i ",
        sizeOfObservation);
    // printf("t is \%i\n",t);
    /*printf("the t-1 policy polPre is: ");

for (int i = 0; i < polPre.GetSize(); i++)
{
    printf(" %i ", polPre[i]);
}
printf("\n");
printf("the t policy is polCur: ");
for (int i = 0; i < polCur.GetSize(); i++)
{
    printf(" %i ", polCur[i]);
}
printf("\n");

int sizeOfpolCur = polCur.GetSize();///<
    sizeOfpolPre = (sizeOfpolCur - 1)/sizeOfObservation
int sizeOfpolCurPre = polPre.GetSize();
if (sizeOfpolCurPre == 0)
{
    return;
}
int depthCur = t;
int depthPre = t - 1;
int sizeOfLevel = 1;
int log = 0;
polCur[iobservation+1]= polPre[log];
log++;
int logParent = iobservation+1+1;

for (int i = 1; i < t-1; i++)
{
    sizeOfLevel = sizeOfLevel*
        sizeOfObservation;
    int logChild = logParent * sizeOfObservation - sizeOfObservation +2 -1;
    logParent = logChild+1;
    for (int j = 0; j < sizeOfLevel; j++)
    {
        polCur[logChild] = polPre[log];
        logChild++;
        log ++;
    }
}
void valueP (DSL::doubleArray& valueOfPol, int t,
DSL::intArray& polCur, SetOfTree& vPrevious, int
sizeOfState, int sizeOfAction, int sizeOfObservation,
DSL::Dmatrix* probsOfState, DSL::Dmatrix* probsOfObservation, DSL::Dmatrix* probsOfRewards, double
discount)
{
    // printf("Now in valueP!!!\n");
    // save the pointer of Qta and vPrevious;
    
    treeNode* pvPrevious = vPrevious.m_current;
    
    // int flag = 0;
    // for (int i=0; i < valueOfPol.GetSize(); i++)
    // { 
    //     valueOfPol[i] = 0.0;
    // }
    
    if (polCur.GetSize()==0)
    {
        vPrevious.m_current=pvPrevious;
        return;
    }
    
    *printf("vPrevious is :");
    vPrevious.Print();
    printf("\n");
    
    // use valueOfPol to store value of policy;
    
    // check if polCur is in vPrevious;
    int flagOfIn = vPrevious.Find(polCur);
    if (polCur.GetSize() == 1)
    {
        int i = action = polCur[0];
        for (int istate =0; istate < sizeOfState; istate++)
        {

107
int logR = istate*sizeOfAction+ iaction;
double R =(*probsOfRewards)[logR ];
valueOfPol[istate] = R ;
}
} else if ((flagOfIn ==1)&&vPrevious.GetLength() >0)&&vPrevious.m_current->GetFlag() == true))
{
    DSL::doubleArray temp = vPrevious.GetValue () ;
    for (int istate =0; istate < sizeOfState ; istate++)
    {
        valueOfPol[istate] = temp[istate] ;
    }
}
} else {
    int iaction = polCur[0] ;
    for (int istate =0; istate < sizeOfState ; istate++)
    {
        double local =0.0;
        for (int iobservation = 0 ;
            iobservation< sizeOfObservation ; iobservation++)
        {
            int localSize = 1;
            for (int i =0; i<=t-1; i++)
            {
                localSize =
                localSize* sizeOfObservation ;
            }
            int sizeOfTree =(localSize -1)/( sizeOfObservation -1);
            //printf("t: %i o: %i
            localSize: %i the size
            of tree is %i \n",t , sizeOfObservation ,
        }
localSize, sizeOfTree);
DSL_intArray polPre(
    sizeOfTree);
choice(polPre, polCur, 
    iobervation, 
    sizeOfObservation, t);
DSL_doubleArray
    valueOfPolNext(
        sizeOfState);
for (int i = 0; i <
    sizeOfState; i++)
{
    valueOfPolNext[i] = 0.0;
}
valueP(valueOfPolNext, t
    -1, polPre, vPrevious, 
    sizeOfState, 
    sizeOfAction, 
    sizeOfObservation, 
    probsOfState, 
    probsOfObservation, 
    probsOfRewards, 
    discount);
DSL_doubleArray
    alphaSharp(sizeOfState
    );
for (int i = 0; i <
    sizeOfState; i++)
{
    alphaSharp[i] = 0.0;
}
back(alphaSharp, 
    valueOfPolNext, iaction 
    , iobervation, 
    sizeOfState, 
    sizeOfAction, 
    sizeOfObservation, 
    probsOfState, 
    probsOfObservation, 
    probsOfRewards,
discount;

local = local + alphaSharp[istate];

}

int logR = istate*sizeOfAction+ iaction;
double R =(*probsOfRewards)[logR ];

valueOfPol[istate] = R + discount *local;

} 

/*

printf("valueOfpolicy is:");
for (int i =0;i< valueOfPol.GetSize();i++)
{
    printf(" %f ",valueOfPol[i]);
}
printf("\n");*/

// give back the pointer of Qta and vPrevious;

vPrevious.m_current=pvPrevious;
//printf("End of valueP!!!\n");

bool superior (DSL_doubleArray& vec,DSL_doubleArray& bestvec){
    //printf("Now in superior!!!\n");
    bool tempflag = false;
    for (int i =0; i<vec.GetSize();i++)
    {
        if (fabs(vec[i])>fabs(bestvec[i]))
        {
            tempflag = true;
            110
return tempflag;

if (fabs(vec[i])<fabs(bestvec[i]))
{
    tempflag = false;
    return tempflag;
}

return tempflag;

}

void findb(DSL_doubleArray& newbelief, int iaction,
SetOfTree& vPrevious, int t, SetOfTree& Qta, int
sizeOfState, int sizeOfAction, int sizeOfObservation,
DSL_Dmatrix* probsOfState, DSL_Dmatrix*
probsOfObservation, DSL_Dmatrix* probsOfRewards, double
discount)
{
    // save the pointer of Qta and vPrevious;
    treeNode* punionQta = Qta.m_current;
    treeNode* pvPrevious = vPrevious.m_current;
    // printf("Now in findb!!!\n");
    double minvalue = -1000000;
    int localSize1 = 1;
    int sizeOfTree1 = 0;

    for (int i =0; i<t-1; i++)
    {
        localSize1 = localSize1*sizeOfObservation;
    }
    if ( t == 1)
    {
        sizeOfTree1 = 0;
    }
    else
    {
        sizeOfTree1 = (localSize1 -1)/(sizeOfObservation -1);
    }

    //compute the size of t-1 policy tree;
    int localSize = 1;

    111
for (int i = 0; i < t; i++)
{
    localSize = localSize * sizeOfObservation;
}

int sizeOfTree = (localSize - 1) / (sizeOfObservation - 1);

// compute the size of t policy tree;
Qta.First();

for (int ipol = 0; ipol < Qta.GetLength(); ipol++)
{
    DSLIntArray polTree(sizeOfTree);
    polTree = Qta.GetData();

    for (int iObservation = 0; iObservation < sizeOfObservation; iObservation++)
    {
        DSLIntArray polPre(sizeOfTree1);
        choice(polPre, polTree, iObservation, sizeOfObservation, t);

        DSLDoubleArray valueOfPolPre(sizeOfState);
        valueP(valueOfPolPre, t - 1, polPre, vPrevious, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState, probsOfObservation, probsOfRewards, discount);

        DSLDoubleArray vecOfPolPre(sizeOfState);
        back(vecOfPolPre, valueOfPolPre, iAction, iObservation, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState, probsOfObservation, probsOfRewards, discount);

        vPrevious.First();
    }
for (int ipolSharp = 0; ipolSharp < vPrevious.GetLength(); ipolSharp++)
{
    DSL_intArray polTreeSharp
        (sizeof(tree1));
    polTreeSharp = vPrevious.GetData();

    DSL_doubleArray
        valueOfPolTreeSharp
        (sizeofState);
    valueP(
        valueOfPolTreeSharp, t
        -1, polTreeSharp, vPrevious, sizeofState, sizeofAction, 
        sizeofObservation, probsOfState, probsOfObservation, 
        probsOfRewards, discount);

    DSL_doubleArray
        vecOfPolTreeSharp
        (sizeofState);
    back( vecOfPolTreeSharp, 
        valueOfPolTreeSharp, iaction, iobservation, 
        sizeofState, sizeofAction, sizeofObservation, probsOfState, 
        probsOfObservation, probsOfRewards, 
        discount);

    DSL_doubleArray vecb(
        sizeofState);

    for (int i = 0; i < 
        sizeofState; i++)
    {
        vecb[i] = 
            vecOfPolTreeSharp
            [i] -

113
vecOfPolPre[i];

DSL_doubleArray
  tempbelief(sizeOfState);
double objective =
  minvalue;
objective = lpfindwitness(tempbelief, vecb, polTree, vPrevious, t, Qta, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState, probsOfObservation, probsOfRewards, discount);

if(objective > epsilon)
{
  for(int i = 0; i < sizeOfState; i++)
  {
    newbelief[i] =
      tempbelief[i];

    // give back the
    // pointer of Qta
    // and vPrevious
    :

    Qta.m_current=
      unionQta;
    vPrevious.
      m_current=
        pvPrevious;
    return;
  }

  vPrevious.Next();
}
`Qta . Next ( ) ;

printf("the lp find belief state is :");
for ( int i =0; i<newbelief.GetSize(); i++)
{
    printf(" %f ", newbelief[i]);
}
printf(" ]\n") ;
// give back the pointer of Qta and vPrevious;
Qta . m.current= punionQta ;
vPrevious . m.current=pvPrevious ;
printf("End of findb\n");

}

double lpfindwitness (DSL_doubleArray& tempbelief ,
DSL_doubleArray& vecb ,DSL_intArray& polTree ,SetOfTree& vPrevious ,int t, SetOfTree& Qta ,int sizeOfState ,int sizeOfAction ,int sizeOfObservation ,DSL_Dmatrix* probsOfState ,DSL_Dmatrix* probsOfObservation ,
DSL_Dmatrix* probsOfRewards ,double discount )
{
    // printf("Now in lpfindwitness !!!\n");
    // save the pointer of Qta and vPrevious;
treeNode* punionQta = Qta . m.current ;
treeNode* pvPrevious = vPrevious . m.current ;
    // Set LP Empty model
    ClpSimplex model;

    int *objIndex = new int [sizeOfState ];
    double * objValue = new double [sizeOfState ];
    for ( int i =0; i<sizeOfState ;i++)
    {
        objIndex[i] = i ;
        objValue[i] = vecb[i] ;
        // printf(" %f", vecb[i]);
    }

    115
// printf("\n");
model.resize(0, sizeOfState);

for (int i = 0; i < sizeOfState; i++)
{
    model.setObjectiveCoefficient(objIndex[i], objValue[i]);
}

for (int i = 0; i < sizeOfState; i++)
{
    model.setColumnLower(i, 0.0);
    model.setColumnUpper(i, 1.0);
}

// add constraint b1+b2+b3...=1
int *rowIndex = new int[sizeOfState];
double *rowValue = new double[sizeOfState];

for (int i = 0; i < sizeOfState; i++)
{
    rowIndex[i] = i;
    rowValue[i] = 1.0;
}

model.addRow(sizeOfState, rowIndex, rowValue, 1.0, 1.0);

// add other constraints
int localSize = 1;

for (int i = 0; i < t; i++)
{
    localSize = localSize * sizeOfObservation;
}

int sizeOftree = (localSize - 1) / (sizeOfObservation - 1);

DSL_doubleArray valueOfPol(sizeOfState);
valueP(valueOfPol, t, polTree, vPrevious, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState,
probsOfObservation, probsOfRewards, discount);

Qta.First();

for (int ipol=0; ipol<Qta.GetLength(); ipol++)
{
    // loop for every pol != p
    DSLIntArray polTree2(sizeOfTree);
    polTree2 = Qta.GetData();
    if (!(polTree==polTree2))
    {
        DSLDoubleArray valueOfPol2(
            sizeOfState);
        valueP(valueOfPol2, t, polTree2,
            vPrevious, sizeOfState, 
            sizeOfAction, sizeOfObservation,
            probsOfState, 
            probsOfObservation,
            probsOfRewards, discount);

        DSLDoubleArray valueOfTemp(
            sizeOfState);
        for (int i=0; i<sizeOfState; i++)
        {
            valueOfTemp[i] =
                valueOfPol[i] -
                valueOfPol2[i];
        }

        for (int i = 0; i<sizeOfState; i++)
        {
            rowIndex[i] = i;
            rowValue[i] = valueOfTemp[i];
        }

        model.addRow(sizeOfState, rowIndex,
            rowValue, 0, COIN_DBL_MAX);
    }
}
model.setOptimizationDirection(-1.0);
model.primal();

double objective_value = model.getObjValue();
double * solution = model.primalColumnSolution();

for (int i = 0; i < sizeOfState; i++)
{
    tempbelief[i] = solution[i];
    // printf("the solution %i is %f\n", i, solution[i]);
}
// printf("the objective is %f\n", objective_value);

delete [] objIndex;
delete [] objValue;

delete [] rowIndex;
delete [] rowValue;
// give back the pointer of Qta and vPrevious;
Qta.m_current= punionQta;
vPrevious.m_current=pvPrevious;
// printf("End of lpfindwitness");
return objective_value;
void purge(SetOfTree& vCurrent, SetOfTree& unionQta,
SetOfTree& vPrevious, int t, int sizeOfState, int
sizeOfAction, int sizeOfObservation, DSL_Dmatrix* probsOfState, DSL_Dmatrix* probsOfObservation,
DSL_Dmatrix* probsOfRewards, double discount)
{

    // printf("Now in purge!!!\n");
    // save the pointer of Qta and vPrevious;
    treeNode* punionQta = unionQta.m_current;
    treeNode* pvPrevious = vPrevious.m_current;
    //treeNode* pvCurrent = vCurrent.m_current;

    int localSize = 1;
    for (int i =0; i <t; i++)
    {
        localSize = localSize * sizeOfObservation;
    }
    int sizeOftree = (localSize-1)/(sizeOfObservation
            -1);

    unionQta.First();
    //unionQta.Print();
    for (int ipol = 0; ipol < unionQta.GetLength();
            ipol++)
    {
        DSL_intArray polTree(sizeOftree);
        polTree = unionQta.GetData();
        double delta = 1.0; //set a flag
        delta = lpfindpurge(polTree, unionQta,
                vPrevious, t, sizeOfState, sizeOfAction
                , sizeOfObservation, probsOfState,
                probsOfObservation, probsOfRewards,
                discount);

    }

/printf("delta is : \%f\n", delta);
if (delta <= 0)
{
    unionQta.Remove();
}
else
{
    DSL_doubleArray valueOfPol(
        sizeOfState);
    valueP(valueOfPol, t, polTree, 
        vPrevious, sizeOfState, 
        sizeOfAction, sizeOfObservation 
        , probsOfState 
        , probsOfObservation 
        , probsOfRewards 
        , discount);

    /*
    printf("here!!!\n");
    vCurrent.Print();
    printf("polTree is:");
    for (int i =0;i< polTree.GetSize();i++)
    {
        printf(" %i ",polTree[i])
    }
    printf("\n");
    printf("valueOfpolTree is:");
    for (int i =0;i< valueOfPol. 
        GetSize();i++)
    {
        printf(" %f ",valueOfPol[i]);
    }
    printf("\n");*/

    vCurrent.Insert(polTree, 
        valueOfPol); 
    unionQta.Next();
}

    // give back the pointer of Qta and vPrevious;
double lpfindpurge(DSLIntArray& polTree, SetOfTree& unionQta, SetOfTree& vPrevious, int t, int sizeOfState, int sizeOfAction, int sizeOfObservation, DSL_Dmatrix* probsOfState, DSL_Dmatrix* probsOfObservation, DSL_Dmatrix* probsOfRewards, double discount)
{
    // printf("Now in lptindpurge!!!\n");
    // Set LP Empty model
    // Save the pointer of Qta and vPrevious;
    treeNode* punionQta = unionQta.m_current;
    treeNode* pvPrevious = vPrevious.m_current;
    ClpSimplex model;

    int *objIndex = new int[sizeOfState+1];
    double * objValue = new double [sizeOfState+1];
    for (int i = 0; i<sizeOfState+1; i++)
    {
        if (i == 0)
        {
            objIndex[i] = i;
            objValue[i] = 1.0;
        }
        else
        {
            objIndex[i] = i;
            objValue[i] = 0;
        }
    }

    model.resize(0,suseState+1);
    for (int i = 0; i<sizeOfState+1; i++)
    {
        model.setObjectiveCoefficient(objIndex[i], objValue[i]);
    }
}

for (int i = 0; i<sizeOfState+1; i++)
{
    if (i == 0)
    {
        model.setColumnLower(i,-COIN_DBL_MAX);
        model.setColumnUpper(i,COIN_DBL_MAX);
    }
    else
    {
        model.setColumnLower(i,0.0);
        model.setColumnUpper(i,1.0);
    }
}

//add constraint b1+b2+b3...=1
int *rowIndex = new int[sizeOfState+1];
double * rowValue = new double[sizeOfState+1];
for (int i = 0; i<sizeOfState+1; i++)
{
    if (i==0)
    {
        rowIndex[i] = i;
        rowValue[i] = 0;
    }
    else
    {
        rowIndex[i] = i;
        rowValue[i] = 1.0;
    }
}
model.addRow(sizeOfState+1,rowIndex,rowValue,1.0,1.0);

//add other constraints
int localSize = 1;
for (int i =0; i<t; i++)
{
    localSize = localSize*sizeOfObservation;
}

int sizeOfTree = (localSize -1)/(sizeOfObservation -1);

DSL_doubleArray valueOfPol(sizeOfState);
valueP(valueOfPol ,t,polTree,vPrevious,sizeOfState ,sizeOfAction ,sizeOfObservation ,probsOfState ,
probsOfObservation ,probsOfRewards ,discount);

unionQta.First();

for (int ipol =0; ipol<unionQta.GetLength(); ipol++)
{
    // loop for every pol != p
    DSLIntArray polTree2(sizeOfTree);
polTree2 = unionQta.GetData();

    if (!(polTree==polTree2))
    {
        DSL_doubleArray valueOfPol2(
            sizeOfState);
        valueP(valueOfPol2 ,t,polTree2 ,
            vPrevious,sizeOfState ,
            sizeOfAction ,sizeOfObservation ,
            probsOfState ,
            probsOfObservation ,
            probsOfRewards ,discount);

        DSL_doubleArray valueOfTemp( sizeOfState);        
        for (int i =0; i<sizeOfState;i++)
        {
            valueOfTemp[i] =
            valueOfPol[i]-
            valueOfPol2[i];
        }
    }
}

123
for (int i = 0; i < sizeOfState + 1; i++)
{
    if (i == 0)
    {
        rowIndex[i] = i;
        rowValue[i] = -1.0;
    }
    else
    {
        rowIndex[i] = i;
        rowValue[i] = valueOfTemp[i - 1];
    }
}

model.addRow(sizeOfState + 1, rowIndex, rowValue, 0, COIN_DBL_MAX);

unionQta.Next();

// solve
model.setOptimizationDirection(-1.0);
model.primal();

doctrine objective_value = model.getObjValue();
doctrine * solution = model.primalColumnSolution();

// printf("the objective is \%f\n", objective_value );
double computedifference(SetOfTree& vCurrent, SetOfTree& vPrevious, int t, int sizeOfState, int sizeOfAction, int sizeOfObservation, DSL_Dmatrix* probsOfState, DSL_Dmatrix* probsOfObservation, DSL_Dmatrix* probsOfRewards, double discount) {
    // printf("Now in computedifference !!!\n");
    // save the pointer of Qta and vPrevious;
    treeNode* pvCurrent = vCurrent.m_current;
    treeNode* pvPrevious = vPrevious.m_current;

    double minvalue = -10000000;
    double maxdiff = minvalue;

    vCurrent.First();
    for (int ipol = 0; ipol < vCurrent.GetLength(); ipol++)
    {
        // give back the pointer of Qta and vPrevious;
        unionQta.m_current = punionQta;
        vPrevious.m_current = pvPrevious;

        delete [] objIndex;
        delete [] objValue;

        delete [] rowIndex;
        delete [] rowValue;

        return objective_value;
    }
}

{  
  vPrevious.First();
  for (int ipolSharp = 0; ipolSharp < vPrevious.GetLength(); ipolSharp++)
  {
    double objective1 = minvalue;
    objective1 = lpfinddiff(vCurrent.GetData(), vCurrent, vPrevious.GetData(), vPrevious, vPrevious, t, t - 1, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState, probsOfObservation, probsOfRewards, discount);
    double objective2 = minvalue;
    objective2 = lpfinddiff(vPrevious.GetData(), vPrevious, vCurrent.GetData(), vCurrent, vPrevious, t - 1, t, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState, probsOfObservation, probsOfRewards, discount);
    if (maxdiff < objective1)
    {
      maxdiff = objective1;
    }
    if (maxdiff < objective2)
    {
      maxdiff = objective2;
    }

    vPrevious.Next();
  }
  vCurrent.Next();
}
// give back the pointer of Qta and vPrevious;
vCurrent.m_current= pvCurrent;
vPrevious.m_current= pvPrevious;
// printf("Now End of computedifference!!!\n");
return maxdiff;
}

double lpfinddiff(DSL_intArray& polTreeX, SetOfTree& setX, 
DSL_intArray& polTreeY, SetOfTree& setY, SetOfTree&
vPrevious, int tX, int tY, int sizeOfState, int sizeOfAction, int sizeOfObservation, DSL_Dmatrix* probsOfState, DSL_Dmatrix* probsOfObservation,
DSL_Dmatrix* probsOfRewards, double discount)
{
    // printf("Now in lpfinddiff!!\n");
    // Set LP Empty model
    // save the pointer of Qta and vPrevious;
    treeNode* pX = setX.m_current;
    treeNode* pY = setY.m_current;
    treeNode* pvPrevious = vPrevious.m_current;

    ClpSimplex model;

    int* objIndex = new int[sizeOfState];
    double* objValue = new double[sizeOfState];

    DSL_doubleArray valueOfPolX(sizeOfState);
    valueP(valueOfPolX, tX, polTreeX, vPrevious,
        sizeOfState, sizeOfAction, sizeOfObservation,
        probsOfState, probsOfObservation, probsOfRewards,
        discount);

    DSL_doubleArray valueOfPolY(sizeOfState);
    valueP(valueOfPolY, tY, polTreeY, vPrevious,
        sizeOfState, sizeOfAction, sizeOfObservation,
        probsOfState, probsOfObservation, probsOfRewards,
        discount);

    DSL_doubleArray valueOfTemp(sizeOfState);
    for (int i = 0; i < sizeOfState; i++)
    {
        valueOfTemp[i] = valueOfPolX[i] -
            valueOfPolY[i];
    }

    for (int i = 0; i < sizeOfState; i++)
    {
        objIndex[i] = i;
        objValue[i] = valueOfTemp[i];
    }

    model.resize(0, sizeOfState);

    for (int i = 0; i < sizeOfState; i++)
    { }
{ 
    model.setObjectiveCoefficient(objIndex[i], objValue[i]);
}

for (int i = 0; i < sizeOfState; i++)
{
    model.setColumnLower(i, 0.0);
    model.setColumnUpper(i, 1.0);
}

//add constraint b1+b2+b3+...=1
int *rowIndex = new int[sizeOfState];
double *rowValue = new double[sizeOfState];

for (int i = 0; i < sizeOfState; i++)
{
    rowIndex[i] = i;
    rowValue[i] = 1.0;
}

model.addRow(sizeOfState, rowIndex, rowValue, 1.0, 1.0);

//add other constraints

valueP(valueOfPolX, tX, polTreeX, vPrevious, 
    sizeOfState, sizeOfAction, sizeOfObservation, 
    probsOfState, probsOfObservation, probsOfRewards, 
    discount);

setX.First();

for (int ipol = 0; ipol < setX.GetLength(); ipol++)
{
    //loop for every pol != p

    if (!(polTreeX == setX.GetData()))
    {

    128
DSL doubleArray valueOfPolXSharp(sizeOfState);

valueP(valueOfPolXSharp,tX,setX.Getdata(),vPrevious,
    sizeOfState, sizeOfAction,
    sizeOfObservation, probsOfState,
    probsOfObservation,
    probsOfRewards, discount);

DSL doubleArray localvalueOfTemp(sizeOfState);

for (int i=0; i<sizeOfState; i++)
{
    localvalueOfTemp[i] =
        valueOfPolX[i] -
        valueOfPolXSharp[i];
}

for (int i = 0; i<sizeOfState; i++)
{
    rowIndex[i] = i;
    rowValue[i] =
        localvalueOfTemp[i];
}

model.addRow(sizeOfState, rowIndex, rowValue, 0, COIN, DBL_MAX);

}  //add other constraints

valueP(valueOfPolY,tY,polTreeY,vPrevious,
    sizeOfState, sizeOfAction, sizeOfObservation,
    probsOfState, probsOfObservation, probsOfRewards,
    discount);

129
setY.First();

for (int ipol=0; ipol<setY.GetLength(); ipol++)
{
    // loop for every pol != p

    if (! (polTreeY==setY.GetData()))
    {
        doubleArray valueOfPolYSharp(sizeOfState);
        valueP(valueOfPolYSharp, tY, setY.GetData(), vPrevious, 
        sizeOfState, sizeOfAction, 
        sizeOfObservation, probsOfState, 
        probsOfObservation, 
        probsOfRewards, discount);

        doubleArray localvalueOfTemp(sizeOfState):
        for (int i=0; i<sizeOfState; i++)
        {
            localvalueOfTemp[i] =
            valueOfPolY[i] -
            valueOfPolYSharp[i];
        }

        for (int i = 0; i<sizeOfState; i++)
        {
            rowIndex[i] = i;
            rowValue[i] =
            localvalueOfTemp[i];
        }

        model.addRow(sizeOfState, rowIndex, rowValue, 0, COIN_DBL_MAX);
    }

    setY.Next();
}

130
// solve
model.setOptimizationDirection(-1.0);
model.primal();

double objective_value = model.getObjValue();
double * solution = model.primalColumnSolution();

delete [] objIndex;
delete [] objValue;

delete [] rowIndex;
delete [] rowValue;
// give back the pointer of Qta and vPrevious;
setX.m.current= pX;
setY.m.current= pY;
vPrevious.m.current=pvPrevious;
// printf("END of lpfinddif!!!\n");

return objective_value;

}  
void demoCoin()
{
/*variables: x,y,z (names don’t show up in the
code and aren’t really necessary)
objective: Maximize (1.0*x+4.0*z)
variable bounds (set by setColumnLower,
setColumnUpper)
0.0 <= x <= 2.0
0.0 <= y <= COIN_DBL_MAX (Some maximum value
defined by the solver)
0.0 <= z <= 4.0
*/
constraints:
2.0 <= 1.0*x + 1.0*z <= COIN_MAX
1.0 <= 1.0*x - 5.0*y + 1.0*z <= 1.0 (This
  translates too: 1.0*x-5.0*y+1.0*z == 1.0)
*/

// Empty model
ClpSimplex model;

// Objective - just nonzeros
int objIndex[] = {0,2};
double objValue[] = {1,4};
// Upper bounds - as dense vector
double upper[] = {2.0,COIN_MAX,4.0};

// Create space for 3 columns
model.resize(0,3);
// Fill in
// int i;
// Virtuous way
// First objective
for (int i=0;i<2;i++)
    model.setObjectiveCoefficient(objIndex[i],objValue[i]);
// Now bounds (lower will be zero by default but
do again)
/
const double* objValueCheck;
objValueCheck = model.getObjCoefficients();
for (int i=0;i<3;i++)
{
    printf("coe %i =%f\n",i, objValueCheck[i])
;
}
*/
for (int i=0;i<3;i++)
{
    model.setColumnLower(i,0.0);
}
model.setColumnUpper(i, upper[i]);
*/

const double *collb = model.getColLower();
const double *colub = model.getColUpper();
for (int i=0; i<3;i++)
{
    printf("column %i between %f and %f\n",i,
collb[i],colub[i]);
}
*/

/*
We could also have done in non-virtuous way e.g.
double * objective = model.objective();
and then set directly
*/
// Faster to add rows all at once – but this is
easier to show
// Now add row 1 as >= 2.0
int row1Index[] = {0,2};
double row1Value[]={1.0,1.0};
model.addRow(2,row1Index,row1Value,
2.0,COIN_DBL_MAX);
// Now add row 2 as == 1.0
int row2Index[] = {0,1,2};
double q =1.0;
double row2Value[]={1.0,-5.0,1.0};
model.addRow(3,row2Index,row2Value,q,q);
// solve
model.setOptimizationDirection(-1.0);
model.primal();

double objective_value = model.getObjValue();
//printf("the objective is %f\n", objective_value);

const double * solution = model.getColSolution();
printf("the solution x is %f\n", solution[0]);
printf("the solution y is %f\n", solution[1]);
printf("the solution z is %f\n", solution[2]);
printf("the objective is %f\n", objective_value);
*/

133
int numberRows = model.numberRows();
// double * rowPrimal = model.primalRowSolution();
double * rowDual = model.dualRowSolution();

// int iRow;
for (int iRow=0; iRow<numberRows; iRow++)
    printf("Row %d, dual %f\n", iRow, rowDual[iRow]);

int numberColumns = model.numberColumns();
// double * columnPrimal = model.
    primalColumnSolution();
double * columnDual = model.dualColumnSolution();

// int iColumn;
for (int iColumn=0; iColumn<numberColumns; iColumn++)
    printf("Column %d, dual %f\n", iColumn, columnDual[iColumn]);

/*
int numberRows = model.numberRows();
// double * rowPrimal = model.primalRowSolution();
// double * rowDual = model.dualRowSolution();

// int iRow;
/*for (int iRow=0; iRow<numberRows; iRow++)
    printf("Row %i, primal %f\n", iRow, rowPrimal[iRow]);*/
/*
int numberColumns = model.numberColumns();
double * columnPrimal = model.
    primalColumnSolution();
// double * columnDual = model.dualColumnSolution()
()
int iColumn;
for (int iColumn=0;iColumn<numberColumns;iColumn++)
    printf("Column %d, primal %f\n", iColumn, columnPrimal[iColumn]);*/

void pomdpPBVI()
{
    // need belief_initial, Set_initial, N, and T
    printf("we are going to solve pomdp with witness \n");
    DSL_network theNet;
    theNet.ReadFile("pomdp.xdsl");

    // test information of the Net
    int numberOfNet = theNet.GetNumberOfNodes();
    printf("the node number of net is: %i \n", numberOfNet);
    int numberOfSample = theNet.GetNumberOfSamples();
    printf("the numberofSample is: %i \n", numberOfSample);
    int numberOfTarget = theNet.GetNumberOfTargets();
    printf("the numberOfTarget is: %i \n", numberOfTarget);
    int NumberOfSlices = theNet.GetNumberOfSlices();
    printf("the NumberOfSlices is: %i \n", NumberOfSlices);
    int MaxTemporalOrder = theNet.GetMaxTemporalOrder();
    printf("the MaxTemporalOrder is: %i \n", MaxTemporalOrder);
    // initialize the value array;
    double minValue = -100000.0;

    // get the data from the model
    int state = theNet.FindNode("State");
    int action = theNet.FindNode("Action");
int rewards = theNet.FindNode("Rewards");
int observation = theNet.FindNode("Observation");

int sizeOfState = theNet.GetNode(state)->
    Definition()->GetSize();
printf("the size of State is: %i \n",
    sizeOfState);
int sizeOfAction = theNet.GetNode(action)->
    Definition()->GetSize();
printf("the size of Action is: %i \n",
    sizeOfAction);
int tempsizeOfObservation = theNet.GetNode(
    observation)->Definition()->GetSize();
int sizeOfObservation = tempsizeOfObservation /
    sizeOfState;
printf("the size of Observation is: %i \n",
    sizeOfObservation);

// access transition prob
DSL_nodeDefinition *temporalStateDef = theNet.
    GetDbn()->GetDefinition(state, 1);
DSL_Dmatrix *probsOfState = temporalStateDef->
    GetMatrix();
for (int i=0; i<probsOfState->GetSize(); i++)
{
    printf("probOfState [] is : %i %f\n", i, (*
        probsOfState)[i]);
}

// access observation prob
DSL_nodeDefinition *temporalObservationDef =
    theNet.GetDbn()->GetDefinition(observation, 1);
DSL_Dmatrix *probsOfObservation =
    temporalObservationDef->GetMatrix();
for (int i=0; i<probsOfObservation->GetSize(); i++)
{
    printf("probsOfObservation [] is : %i %f\n",
        i, (*probsOfObservation)[i]);
}

// access rewards
DSL_nodeDefinition *RewardsDef = theNet.GetNode(
    rewards)->Definition();
DSL_Dmatrix *probsOfRewards = RewardsDef->GetMatrix();
for (int i=0;i<probsOfRewards->GetSize();i++)
{
    printf("probOfRewards [] is : %i %f
",i,(*probsOfRewards)[i]);
}

int initialNum = 1;
DSL_intArray initialHead(initialNum);
for (int i=0; i<initialNum; i++)
{
    initialHead[i] = -1;
    printf("head nod %i
",initialHead[i]);
}

DSL_doubleArray initialHeadValue(initialNum);
for (int i=0; i<initialNum; i++)
{
    initialHeadValue[i] = 0.0;
    printf("head nod value %f
", initialHeadValue[i]);
}

double discount = 0.5;

//get Rmin
double Rmin = -minValue;
for (int i=0;i<probsOfRewards->GetSize();i++)
{
    if (((*probsOfRewards)[i]<= Rmin)
        { Rmin = (*probsOfRewards)[i];
        }
}

//get Rmax
double Rmax = minValue;
for (int i=0;i<probsOfRewards->GetSize();i++)
{
    if (((*probsOfRewards)[i] >= Rmax)
    {  
    
137
R\text{max} = (**probsOfRewards**)[i];

```c
}
}
double difMM = R\text{max} - R\text{min};
double logdifMM = \varepsilon / \text{difMM};
// discount \neq 1; note
double T = \log(\log\text{difMM})/\log(\text{discount});
int intT = 0;
while (intT < T)
{
    intT++;
}
// get the integer intT of T
int expaN = 10; // for expansions
// dataNode(2): (action, observation)
DSL::intArray dataNode(2);
dataNode[0] = -1;
dataNode[1] = -1;
```

```c
DSL::doubleArray initialbelief(sizeOfState);
for (int i = 0; i < sizeOfState; i++)
{
    initialbelief[i] = 0;
}
initialbelief[0] = 1.0;
SetOfTree beliefSet(initialHead, initialHeadValue);
beliefSet.Insert(dataNode, initialbelief);
// begin with (1, 0, 0, 0, 0, 0, 0)
```

```c
DSL::doubleArray alpha0(sizeOfState);
for (int i = 0; i < sizeOfState; i++)
{
    alpha0[i] = R\text{min} / (1 - \text{discount});
}
SetOfTree vPrevious(initialHead, initialHeadValue);
```
vPrevious.Insert(dataNode, alpha0);

for (int iExpansion = 0; iExpansion < expanN; iExpansion++)
{
    for (int iT = 0; iT < intT; iT++)
    {
        // printf("now the expand is %i, 
        and the T iteration is %i\n", 
        iExpansion, iT);

        SetOfTree vCurrent(initialHead, 
        initialHeadValue);// to store 
        the return Set of backup(); 
        // vPrevious.Print(); 
        // test 
        /*
        if (iExpansion == 1) {
        printf(" before blief is ");}
        bliefSet.Print();*/

        backup(vCurrent, vPrevious, 
        bliefSet, sizeOfState, 
        sizeOfAction, 
        sizeOfObservation, 
        probsOfState, 
        probsOfObservation, 
        probsOfRewards, discount);

        // now transfer vCurrent to 
        vPrevious 
        // clean vPrevious 
        // test 
        /*
        if (iExpansion == 1) {
        printf(" vPrevious : \n");
        printf(" the num is %i\n", 
        vPrevious.GetLength());
        vPrevious.Print();
        printf(" vCurrent : \n");
    
    139
printf("the num is %i\n",
    vCurrent.GetLength());
vCurrent.Print();
}

vPrevious.First();
int lengthOfSet = vPrevious.GetLength();
for (int ipol = 0; ipol < lengthOfSet; ipol++)
{
    vPrevious.Remove();
    //vPrevious.Next(); do
    //not need next();
    //printf("in clean the
    //previous set is %i :\n    ",ipol);
    //vPrevious.Print();
    //printf("\n");
}

vCurrent.First();
vPrevious.First();
for (int ipol = 0; ipol < vCurrent.GetLength(); ipol++)
{
    vPrevious.Insert(vCurrent.GetData(),vCurrent.GetValue());
    vPrevious.m_current->
        SetFlag();
    vCurrent.Next();
}

//test
/*
if (iExpansion == 1){
    printf("vPrevious : \n");
    vPrevious.Print();
    printf("vCurrent : \n");
    vCurrent.Print();
}
SetOfTree newBelief(initialHead, initialHeadValue); // to store the return Set of backup(); expandbelief( newBelief, beliefSet, 
vPrevious, sizeOfState, sizeOfAction, sizeOfObservation, probsOfState, 
probsOfObservation, probsOfRewards, discount);

// transfer newBelief to belief

// clean belief
beliefSet.First();
int lengthOfSet = beliefSet.GetLength();
for (int ib = 0; ib < lengthOfSet; ib++)
{
    beliefSet.Remove();
    // beliefSet.Next();
}
// printf(" here\n");

newBelief.First();
beliefSet.First();
printf(" here\n");

int iblengthofNewBelief = newBelief.Getlength();
printf(" here legth = %i\n", iblengthofNewBelief);
newBelief.Print();

for (int ib = 0; ib < iblengthofNewBelief; ib++)
{


bliefSet.Insert(newBelief.GetData(), newBelief.GetValue());
// bliefSet.m_current->SetFlag();
newBelief.Next();

// printf("expand n = %i\n", iExpansion);

printf("vPreviousTreeset is :\n");
vPrevious.Print();

void backup(SetFont& vCurrent, SetOfTree& vPrevious, 
SetOfTree& bliefSet, int sizeOfState, int sizeOfAction, 
int sizeOfObservation, DSL_Dmatrix* probsOfState, 
DSL_Dmatrix* probsOfObservation, DSL_Dmatrix* 
probsOfRewards, double discount)
{
   // printf("backup\n");
   // vCurrent.Print();
   // save the pointer of Qta and vPrevious;
   treeNode* pliefSet = bliefSet.m_current;
   treeNode* pvPrevious = vPrevious.m_current;
   //treeNode* pvCurrent = vCurrent.m_current;

   double minvalue = -100000.0;
   int initialNum = 1;
   DSL_intArray initialHead(initialNum);
   for (int i = 0; i < initialNum; i++)
   {
      initialHead[i] = -1;
      // printf("head nod %i\n", initialHead[i]);
   }

   DSL_doubleArray initialHeadValue(initialNum);
   for (int i = 0; i < initialNum; i++)
initialHeadValue[i] = 0.0;
// printf("head nod value %f\n",
   initialHeadValue[i]);

SetOfTree vTemp(initialHead, initialHeadValue);//
all the alpha with iaction and observation

for (int iaction = 0; iaction < sizeOfAction;
   iaction++)
{
   for (int iobservation = 0; iobservation <
      sizeOfObservation; iobservation++)
   {
      vPrevious.First();
      int lengthVP1 = vPrevious.
         Getlength();
      for (int ialpha = 0; ialpha <
         lengthVP1; ialpha++)
      {
         DSL_intArray dataNode(2);
         dataNode[0] = iaction;
         dataNode[1] =
            iobservation;

         DSL_doubleArray
            alphaSharp(sizeOfState
   );
         for (int istate = 0;
            istate < sizeOfState;
            istate++)
         {
            double local = 0;
            for (int istate1
               = 0; istate1 <
               sizeOfState;
               istate1++)
            {
               int logT
               =
               istate
               *
               sizeOfAction
               *
               sizeOfState
double T = (probsOfState) \[ \text{logT} \];

int logO = istate1 * sizeOfAction * sizeOfObservation + iaction * sizeOfObservation + iobservation;

double O = (probsOfObservation) \[ \text{logO} \];

local = local + T * O * vPrevious . Getvalue () [ istate1 ];
}
alphaSharp[istate] = discount * local;

vTemp.Insert(dataNode, alphaSharp);

vPrevious.Next();

vPrevious.Last();

//vTemp.Print();

// vCurrent = null;

SetOfTree vTempAction(initialHead, initialHeadValue);  // all the alpha with iaction
for (int iaction = 0; iaction < sizeOfAction; iaction++)
{
    briefSet.First();
    int lengthBriefSet = briefSet.GetLength();
    for (int i = 0; i < lengthBriefSet; i++)
    {
        DSL.doubleArray alphaba(sizeOfState);
        for (int i = 0; i < sizeOfState; i++)
        {
            alphaba[i] = 0;
        }
        for (int iobservation = 0; iobservation < sizeOfObservation; iobservation++)
        {
            vTemp.First();
            double localMax = minvalue;
        }
    }
}

145
DSL_doubleArray alphaBAO(sizeOfState);

int lengthVTemp = vTemp.GetLength();
for (int j = 0; j < lengthVTemp; j++)
{
    if (vTemp.GetData()[1] == iobservation)
    {
        double localab = 0;
        for (int istate = 0; istate < sizeOfState; istate++)
        {
            localab =
                localab +
                vTemp.GetData()[istate]*bliefSet.GetData()[istate];
        }
        if (localab >= localMax)
        {
        }
    }
}
localMax =

localab;

for (int i = 0; i < sizeofState;
    i +=)
{
    alphaBAO[
        i
    ] = vTemp . Getvalue () [
        i
    ];
for (int i = 0; i < sizeof(state); i++)
{
    alphaba[i] = alphaba[i] + alphaBAO[i];
}

int logR = i * sizeOfAction + iAction;
double R = (*probsOfRewards)[logR];
alphaba[i] = alphaba[i] + R;
}

DSLIntArray dataNode(2);
dataNode[0] = iAction;
dataNode[1] = -1;

vTempAction.Insert(dataNode, alphaba);
bliefSet.Next();
}
bliefSet.Last();

//now we build vCurrent
//printf("now we build vCurrent\n");
vCurrent.First();
//clean vcurrent
int lengthVC1 = vCurrent.GetLength();
for (int i = 0; i < lengthVC1; i++)
{
    vCurrent.Remove();
    //vCurrent.Next(); donot need
}
//vCurrent.Print();

bliefSet.First();
int lengthBS2 = bliefSet.GetLength();
for (int i = 0; i < lengthBS2; i++)
{
    DSLIntArray dataNode(2);
    }
dataNode[0] = -1;
dataNode[1] = -1;

DSL_doubleArray alphabrief(sizeOfState);
int flagofaction = -1;

//we don't need iaction now
double localMax = minvalue;

int lengthvTA1 = vTempAction.GetLength();
vTempAction.First();
for (int j = 0; j < lengthvTA1; j++)
{

double localb = 0;

for (int istate = 0; istate <
    sizeOfState; istate++)
{
    localb = localb +
        vTempAction.GetValue()
        [istate] *-bliefSet.
        GetValue()[istate];
}
if (localb >= localMax)
{
    localMax = localb;
    for (int i = 0; i <
        sizeOfState; i++)
    {
        alphabrief[i] =
            vTempAction.
            GetValue()[i];
        //change frome
        alphabrief[i] = vTempAction.
            GetValue()[istate];
    }
    flagofaction =
        vTempAction.GetData()
        [0];
}
vTempAction.Next();

vTempAction.Last();

vCurrent.First();
bool insert = true;

int lengthVC2 = vCurrent.GetLength();
for (int i = 0; i < lengthVC2; i++)
{
    if (vCurrent.GetData()[0] == flagofaction)
    {
        double localZero = 0;
        for (int j = 0; j < sizeofState; j++)
        {
            double localdiff = vCurrent.GetValue()[j] - alphablename[j];
            localZero = localZero + fabs(localdiff);
        }
        if (localZero * 100 < epsilon)
        {
            insert = false;
        }
    }
    vCurrent.Next();
}

vCurrent.Last();

if (insert == true)
{
    dataNode[0] = flagofaction;
}

151
void expandBelief(SetOfTree& newBelief, SetOfTree& beliefSet, SetOfTree& vPrevious, int sizeOfState, int sizeOfAction, int sizeOfObservation, DSL_Dmatrix* probsOfState, DSL_Dmatrix* probsOfObservation, DSL_Dmatrix* probsOfRewards, double discount)
{
    // printf("in EXPand \n");
    // save the pointer of Qta and vPrevious;
    treeNode* pbeliefSet = beliefSet.m_current;
    treeNode* pvPrevious = vPrevious.m_current;
    //treeNode* pvCurrent = vCurrent.m_current;
    //test
    /*
    printf("belief tree\n");
    beliefSet.Print();
    printf("new belief tree\n");
    newBelief.Print();*/

    // transfer belief set to new belief set
    newBelief.First();
    beliefSet.First();
    DSL_intArray dataNode(2);
    dataNode[0] = -1;
    dataNode[1] = -1;
    int iblength = beliefSet.GetLength();
for (int ib = 0; ib < ibLength; ib++)
{
    newBelief.Insert(blfSet.GetData(),
    blfSet.GetValue());
    // newBelief.m_current->SetFlag();
    blfSet.Next();
}

// test
/*
 printf("belief tree\n");
 blfSet.Print();
 printf("new belief tree\n");
 newBelief.Print();*/

DSL_doubleArray bTemp(sizeOfState+1);
DSL_doubleArray bNew(sizeOfState);

for (int ib = 0; ib < ibLength; ib++)
{
    for (int i = 0; i < sizeOfState+1; i++)
    {
        int randN;
        randN = rand() % 10001;
        // get from 0 10000
        double M = 10000.0;
        bTemp[i] = randN/M;
        // from 0 1
    }

    // sort bTemp[] in ascending order
    for (int j = 0; j < sizeOfState+1; j++)
    {
        for (int k = sizeOfState; k > j; k--)
        {
            if (bTemp[k] < bTemp[k-1])
            {
                double local =
                bTemp[k];
                bTemp[k] = bTemp[k-1];
            }
        }
    }
}
bTemp[k-1] =
    local;

    }
}

for (int i = 1; i<sizeOfState;i++)
{
    bNew[i] = bTemp[i+1] - bTemp[i];
}

double localb0 = 1.0;
for (int i=1; i<sizeOfState;i++)
{
    localb0 = localb0 - bNew[i];
}

bNew[0] = localb0;

bool insert = true;

newBelief.First();
int ilengthofNewBelief = newBelief.GetLength();

for (int i = 0; i < ilengthofNewBelief;i++)
{

double localZero = 0;
for (int j = 0; j<sizeOfState; j++)
{
    double localdiff
    = newBelief.GetValue()[j]-
    bNew[j];
    localZero =
    localZero + fabs(localdiff);
}

if (localZero*100<epsilon)
{
    insert = false;
}
if (insert == true)
{
    newBelief.Insert(dataNode, bNew);
}

// bliefSet.Next();

// test
/*
bliefSet.Print();
printf("new\n");
newBelief.Print();
printf("new end\n");
*/

bliefSet.m_current = pbliefSet;
vPrevious.m_current = pvPrevious;

// printf("End of expand\n");

} // forwardhmm

void forwardhmm()
{
    DSL_network theNet;
    theNet.ReadFile("hmm.xdsl");

    // Vector<DSL_rNetwork*> *net_list = Relevant(theNet);

    155
// test information of the Net
int numberOfNet = theNet.GetNumberOfNodes();
// cout << "the node number of net is" << numberOfNet << endl;
printf("the node number is: %i\n", numberOfNet);
int numberOfSample = theNet.GetNumberOfSamples();
printf("the number of Sample is: %i\n", numberOfSample);
int numberOfTarget = theNet.GetNumberOfTargets();
printf("the number of Target is: %i\n", numberOfTarget);
int NumberOfSlices = theNet.GetNumberOfSlices();
printf("the number of Slices is: %i\n", NumberOfSlices);
int MaxTemporalOrder = theNet.GetMaxTemporalOrder();
printf("the MaxTemporalOrder is: %i\n", MaxTemporalOrder);

int state = theNet.FindNode("state");
int observation = theNet.FindNode("observation");

int sizeOfState = theNet.GetNode(state)->Definition()->GetSize();
printf("the size of State is: %i\n", sizeOfState);
int tempSizeOfObservation = theNet.GetNode(observation)->Definition()->GetSize();
int sizeOfObservation = tempSizeOfObservation / sizeOfState;
printf("the size of Observation is: %i\n", sizeOfObservation);

int t = 0; // t step
DSL_nodeDefinition *temporalPiDef = theNet.GetNode(state)->Definition();
DSL_Dmatrix *probsOfPi = temporalPiDef->GetMatrix();
for (int i = 0; i < probsOfPi->GetSize(); i++)
{
    printf("probsOfPi[] is: %i %f\n", i, (*probsOfPi)[i]);
}
DSL_nodeDefinition *temporalStateDef = theNet.GetDbn().GetDefinition(state, 1);
DSL_Dmatrix *probsOfState = temporalStateDef->GetMatrix();
for (int i=0;i<probsOfState->GetSize();i++)
{
    printf(" probOfState [] is : %i %f\n", i, (*probsOfState)[i]);
}
DSL_nodeDefinition *temporalObservationDef = 
    theNet.GetNode(observation)->Definition();
DSL_Dmatrix *probsOfObservation = 
    temporalObservationDef->GetMatrix();
for (int i=0;i<probsOfObservation->GetSize();i++)
{
    printf(" probsOfObservation [] is : %i %f\n", i, (*probsOfObservation)[i]);
}

// let t be 3
int t = 3;
// give the evidence
DSL_intArray evidenceArray(t);
evidenceArray[0]=0;
evidenceArray[1]=2;
evidenceArray[2]=3;

// define variable alpha
DSL_doubleArray alpha(sizeOfState);
DSL_doubleArray Prealpha(sizeOfState);

// initialization
for (int i =0; i<sizeOfState; i++)
{
    int evidenceO = evidenceArray[0];
    // get pi[i]
    int logPi = i;
    double valueOfPi = (*probsOfPi)[logPi];
    // get bi(Oi)
    int iobservation = evidenceO;
    
    // put evidence
    Prealpha[iobservation] = Prealpha[iobservation] + valueOfPi;
}
int logO = i*sizeOfObservation +
iobservation;
double O = (*probsOfObservation)[logO];

// printf("i is : %i pi is: %f, b is : %f\n",i,valueOfPi, O);

// think about log method
alpha[i] = valueOfPi * O;
printf("0 i is : %i alpha is : %f\n", i,
alpha[i]);
}

// transfer alpha to prealpha
for (int i = 0; i < sizeOfState; i++)
{
    Prealpha[i] = alpha[i];
}

// induction
for (int itime = 1; itime < t; itime++)
{
    int evidenceO = evidenceArray[itime];
    for (int j = 0; j < sizeOfState; j++)
    {
        // get bj(Oitime)
        int iobservation = evidenceO;
        int logO = j*sizeOfObservation +
iobservation;
double O = (*probsOfObservation)[logO];

double localsum = 0;
    for (int i = 0; i < sizeOfState; i++)
    {
        int logS = i*sizeOfState
            + j;
        double S = (*probsOfState)
            [logS];
localsum = localsum + S * Prealpha[i];
}

// printf("i is : %i pi is: %f, b is : %f\n",i,valueOfPi, O);

// think about log method
alpha[j] = localsum * O;
// printf("i is : %i alpha is : %f
",i,alpha[i]);
}

// transfer alpha to Prealpha
for (int i = 0; i < sizeOfState; i++)
{
    Prealpha[i] = alpha[i];
    // for debug
    // printf("%i i is : %i alpha is : %f\n",itime, i, alpha[i]);
}

double PrOfsequence = 0;
for (int i=0; i < sizeOfState; i++)
{
    PrOfsequence = PrOfsequence + Prealpha[i];
}

printf("the probability of the observation sequence is %f\n", PrOfsequence);
void viterbihmm()
{
    DSL_network theNet;
    theNet.ReadFile("hmm.xdsl");

    // Vector<DSL_network> *net_list = Relevant(theNet);

    // test information of the Net
    int numberOfNet = theNet.GetNumberOfNodes();
    // cout<<"the node number of net is"<<numberOfNet
    //<<endl;
    printf("the node number is: %i \n", numberOfNet);
    int numberOfSample = theNet.GetNumberOfSamples();
    printf("the number of Sample is: %i \n", numberOfSample);
    int numberOfTarget = theNet.GetNumberOfTargets();
    printf("the number of Target is: %i \n", numberOfTarget);
    int NumberOfSlices = theNet.GetNumberOfSlices();
    printf("the Number of Slices is: %i \n", NumberOfSlices);
    int MaxTemporalOrder = theNet.GetMaxTemporalOrder();
    printf("the MaxTemporalOrder is: %i \n", MaxTemporalOrder);

    int state = theNet.FindNode("state");
    int observation = theNet.FindNode("observation");

    int sizeOfState = theNet.GetNode(state)->
        Definition()->GetSize();
    printf("the size of State is: %i \n", sizeOfState);
    int tempsizeOfObservation = theNet.GetNode(
        observation)->Definition()->GetSize();
    int sizeOfObservation = tempsizeOfObservation/
        sizeOfState;
    printf("the size of Observation is: %i \n", sizeOfObservation);

    int t = 0; // t step
DSL_nodeDefinition *temporalPiDef = theNet.
    GetNode(state)->Definition();
DSL_Dmatrix *probsOfPi = temporalPiDef->GetMatrix();
for (int i=0; i<probsOfPi->GetSize(); i++)
{
    printf("probsOfPi [] is : %i %f\n", i, (*probsOfPi)[i]);
}

DSL_nodeDefinition *temporalStateDef = theNet.
    GetDbn()->GetDefinition(state, 1);
DSL_Dmatrix *probsOfState = temporalStateDef->
    GetMatrix();
for (int i=0; i<probsOfState->GetSize(); i++)
{
    printf("probOfState [] is : %i %f\n", i, (*probsOfState)[i]);
}

DSL_nodeDefinition *temporalObservationDef =
    theNet.GetNode(observations)->Definition();
DSL_Dmatrix *probsOfObservation =
    temporalObservationDef->GetMatrix();
for (int i=0; i<probsOfObservation->GetSize(); i++)
{
    printf("probsOfObservation [] is : %i %f\n","i, (*probsOfObservation)[i]);
}

// let t be 3
int t = 3;
// give the evidence
DSL_intArray evidenceArray(t);
evidenceArray[0]=0;
evidenceArray[1]=2;
evidenceArray[2]=3;

// define variable delta
DSL_doubleArray delta(sizeOfState);
DSL_doubleArray Predelta(sizeOfState);
DSL_intArray pointer(sizeOfState*t);

// initialization
for (int i = 0; i < sizeOfState; i++)
{
    int evidenceO = evidenceArray[0];

    // get pi[i]
    int logPi = i;
    double valueOfPi = (*probsOfPi)[logPi];

    // get bi(Oi)
    int iObservation = evidenceO;
    int logO = i * sizeOfObservation + iObservation;
    double O = (*probsOfObservation)[logO];

    printf("i is : %i pi is : %f, bi is : %f
           %n", i, valueOfPi, O);

    // think about log method
    delta[i] = valueOfPi * O;
    printf("0 i is : %i alpha is : %f\n", i, delta[i]);
    pointer[i] = 0;
}

// transfer alpha to prealpha
for (int i = 0; i < sizeOfState; i++)
{
    Predelta[i] = delta[i];
}

// recursion
for (int itime = 1; itime < t; itime++)
{
    int evidenceO = evidenceArray[itime];

    for (int j = 0; j < sizeOfState; j++)
    {
        // get bj(Oitime)
        int iObservation = evidenceO;
        //...
int logO = j * sizeofObservation + 
    iobservation;
double O = (*probsOfObservation)[logO];

double localmax = -1;
double localmaxI = -1;
int flagI = -1;
for (int i = 0; i < sizeofState; i ++)
{
    int logS = i * sizeofState + j;
double S = (*probsOfState)[logS];
double localV = Predelta[i] * S * O;
double localI = Predelta[i] * S;
    if (localV > localmax)
    {
        localmax = localV;
    }
    if (localI > localmaxI)
    {
        flagI = i;
    }
}

// printf(" i is : %i pi is : %f, b is : %f\n", i, valueOfPi, O);

// think about log method
delta[j] = localmax;
// printf(" i is : %i alpha is : %f\n", i, alpha[i]);
pointer[itime * sizeofState + j] = flagI;

// transfer alpha to Prealpha

163
for (int i = 0; i < sizeOfState; i++) {
    Predelta[i] = delta[i];
    // for debug
    // printf("%i i is : %i alpha is : \\
    %f\n", itime, i, alpha[i]);
}

} // Termination

int finalI = -1;
double finalMax = -1;
for (int i = 0; i < sizeOfState; i++) {
    if (Predelta[i] > finalMax)
    {
        finalI = i;
        finalMax = Predelta[i];
    }
}

// find the path
DSL_intArray pathViterbi(t);
for (int i = 0; i < sizeOfState; i++) {
    pathViterbi[i] = -1;
}

pathViterbi[t - 1] = finalI;
int localIdPath = finalI;
for (int i = t - 2; i >= 0; i --) {
    pathViterbi[i] = pointer[i + sizeOfState +
                          localIdPath];
    localIdPath = pathViterbi[i];
}
printf("The Path is:");

for (int i = 0; i < sizeOfState; i++)
{
    printf(" %i\n", pathViterbi[i]);
}

void testSetOfTree()
{
    int initialNum = 1;
    DSL_intArray initialHead(initialNum);
    for (int i = 0; i < initialNum; i++)
    {
        initialHead[i] = -1;
        printf("head nod %i\n", initialHead[i]);
    }
    DSL_doubleArray initialHeadValue(initialNum);
    for (int i = 0; i < initialNum; i++)
    {
        initialHeadValue[i] = 0.0;
        printf("head nod value %f\n", initialHeadValue[i]);
    }

    SetOfTree vPrevious(initialHead, initialHeadValue);
    if (vPrevious.IsEmpty())
    {
        printf("empty\n");
    }

    DSL_intArray a1(3);
    for (int i = 0; i < 3; i++)
    {
        a1[i] = 1;
    }
    DSL_doubleArray val(2);
    for (int i = 0; i < 2; i++)
    {
        val[i] = 1.1;
    }
}
DSLIntArray a2(3);
for (int i = 0; i < 3; i++)
{
    a2[i] = 2;
}
DSLdoubleArray va2(2);
for (int i = 0; i < 2; i++)
{
    va2[i] = 2.2;
}
DSLIntArray a3(3);
for (int i = 0; i < 3; i++)
{
    a3[i] = 3;
}
DSLdoubleArray va3(2);
for (int i = 0; i < 2; i++)
{
    va3[i] = 3.3;
}
DSLIntArray a4(3);
DSLIntArray a5(3);
for (int i = 0; i < 3; i++)
{
    a4[i] = 4;
}

vPrevious.First();

vPrevious.Insert(a1, va1);
vPrevious.Insert(a2, va2);
vPrevious.Insert(a3, va3);
vPrevious.Print();
if (vPreviousisempty())
{
    printf("empty
");
}
vPrevious.Firster();
printf("the node is %i", vPrevious.GetData()[0]);
printf("the value is %f\n", vPrevious.GetValue()[0]);

vPrevious.First();
for (int i = 0; i < vPrevious.GetLength(); i++)
{   printf("the node is %i", vPrevious.
    GetData() [0]);
   printf(" the value is %f\n", vPrevious.
    GetValue() [0]);
   vPrevious.Next();
}
changeSetOfTree(vPrevious, a4);
  // vPrevious.Remove();
  // vPrevious.Print();
  printf("after arr out is:");
  for (int i =0; i<a4.GetSize(); i++)
  {
    printf(" %i", a4[i]);
  }
  printf("\n");
  printf("after Tree out is:\n");
  vPrevious.Print();
  printf("\n");
  a5 = a4;
  printf("after a5=a4:");
  for (int i =0; i<a5.GetSize(); i++)
  {
    printf(" %i", a5[i]);
  }
  printf("\n");

} void changeSetOfTree(SetOfTree& vPrevious, DSL_intArray&
a4){   printf("before arr is:");
  for (int i =0; i<a4.GetSize(); i++)
  {
    printf(" %i", a4[i]);
    a4[i] = 7;
  }
  printf("\n");

167
```c
printf("after arr in is:");
for (int i =0; i<a4.GetSize(); i++)
{
    printf(" %i",a4[i]);
}
printf("\n");
printf("before tree in is:\n");
vPrevious.Print();
printf("\n");
vPrevious.First();
for (int i =0; i<vPrevious.GetData().GetSize(); i++)
{
    printf("%i ",vPrevious.GetData()[i]);
    vPrevious.GetData()[i] = 9;
    printf("%i ",vPrevious.GetData()[i]);
}
printf("after tree in is:\n");
vPrevious.Print();
printf("\n");
vPrevious.First();
vPrevious.Remove();
printf("after tree in remove is:\n");
vPrevious.Print();
printf("\n");
```