Concurrent Factored Planning

Kin-Hon Chan

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Computer Science (Honours) at
The Department of Computer Science
Australian National University

May 2012

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this thesis is my own original work.

Kin-Hon Chan
31 May 2012

To my parents.

Acknowledgements

I would like to thank my family for all their love and support. This would not have been possible without them. I am very fortunate to have Scott Sanner as my Honours supervisor and grateful for his willingness to supervise me even though I started my Honours project three months late due to a change in thesis topic. Scott has been very patient and helpful in providing valuable advices and guidance throughout my Honours year. Many thanks to Jinbo Huang, Stephen Gould, Peter Strazdins and Stephen Roberts for giving me the opportunity to work with them in the past. Finally, a big thank you to all my friends who helped me out one way or the other.

Abstract

For the past decade, Markov decision processes (MDPs) have been widely adopted for solving decision-theoretic planning. However, there has been very little progress made in concurrency with concurrent actions and yet, many real world decision-theoretic planning problems are inherently concurrent. Concurrent problems could be modelled as standard MDPs but concurrency aggravates the curse of dimensionality due to the exponential combination of concurrent actions. In this thesis, we investigate a structured approach based on factored action spaces in an exact dynamic programming solution that obviates the need to enumerate concurrent actions. In addition, we propose an approximation approach that projects value functions into additive structure that could be exploited efficiently by our dynamic programming solution. Empirical results show that our solution outperforms previous methods by orders of magnitude in time and space.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

It ought to be remembered that there is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in the introduction of a new order of things.
Niccoló Machiavelli

Planning is an area of artificial intelligence (AI) that aims to generate plans or action sequences to achieve a series of objectives. Typically, actions are executed by intelligent agents, autonomous robots or unmanned vehicles. In problems where state transitions are non-deterministic and actions outcomes are uncertain, decision-theoretic planning has become the standard framework for finding the plan with the maximum expected utility.

In classical planning, a planner has to find a sequence of actions starting from an initial state to a goal state. Decision-theoretic planning distinguishes itself from classical planning by considering uncertainties in actions outcomes and has a notion of utility in the form of reward. It finds an ordered sequence of actions or policy that maximises the expected reward in a stochastic environment.

Most research on decision-theoretic planning focused on problems where actions were executed sequentially but in reality, real world problems often take place in a concurrent environment. For example, consider an everyday problem faced by a logistic company where it has to plan what is the best way to utilise its resources such that operating costs are minimised. The plan has to involve concurrent tasks for all its delivery trucks, ships and planes. If actions were executed sequentially, most deliveries might not be delivered on time and the company would loss its credibility as well as customers. Another motivating example of concurrency is traffic control where an agent has to control a number of traffic lights concurrently in order to minimise traffic congestion.

In this thesis, we consider *concurrency* in decision-theoretic planning based on concurrent Markov decision processes (MDPs) where concurrent actions are executed simultaneously. While concurrent MDPs are interesting in its own rights, it poses a number of challenges. Concurrency adds an extra level of complexity by considering all possible combination of concurrent actions. Consequently, it aggravates the exponential blow up in dynamic programming (in particular, value iteration) that requires explicit state-action representation and exploration of the entire state space. There-

fore, it is imperative to search for an efficient approach to solve concurrent MDPs; the main theme and question of this thesis. In the following, we give an overview of contributions made in this thesis

- From the idea of factoring state variables in factored MDPs [Boutilier et al. 1995], we investigate a method to factor actions in concurrent MDPs and value iteration. Our implementation extends SPUDD [Hoey et al. 1999] to handle concurrent actions that obviate the need to enumerate concurrent actions. To contain the exponential blow up in algebraic decision diagrams (ADDs) [Bahar et al. 1997], the main data structure in SPUDD, value functions are kept in an additive structure for efficient Bellman backup.
- For the purpose of scalability, we propose a couple of projection algorithms to project an ADD into an additive form of weighted basis functions (ADDs) for additive structure exploitation.
- Moreover, we present an approach to automatically identify decent basis functions from capturing causal relationship between variables.

The remainder of the thesis is organised as follows. In Chapter 2, we provide a cursory review of MDPs and value iteration. Chapter 3 describes a structured approach for solving factored MDPs based on ADDs. Then, we present our first contribution in Chapter 4 by proposing an efficient approach to solve concurrent factored MDPs. Chapter 5 discusses projection algorithms in order to scale beyond the limits of exact inference. Empirical results are presented in Chapter 6 followed by the conclusion of this thesis in Chapter 7.

Chapter 2

Markov Decision Processes

Computer Science is a science of abstraction - creating the right model for a problem and devising the appropriate mechanizable techniques to solve it.
Alfred Aho and Jeffrey Ullman

Markov decision processes (MDPs) [Bellman 1957; Puterman 1994] have been widely adopted as a model of choice for decision-theoretic planning as they provide a convenient and compact mathematical framework for formulating settings where outcomes are uncertain. At each time step t, the process is in some state s, the agent chooses an action a to execute from a set of actions that are available in the current state. The process responds at the next time step $t + 1$ by randomly transitioning into a new state s' based on some probability distribution. In response, the agent receives a corresponding reward as a way to evaluate the value or worth of taking action a at state s. This procedure repeats itself for some finite or infinite horizon h. Thus, the goal of the agent is to choose actions so as to maximise the expected sum of discounted future rewards.

Throughout this thesis, we restrict attention to *discrete* time and *fully* observable MDPs with *finite* set of states and actions. Furthermore, we assume that both the state and action space are *discrete*.

2.1 Overview of MDPs

The description of MDPs follows Puterman [1994]. Formally, a MDP is defined by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ where

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{T}$ is the transition function denoting the conditional probability $P(s'|s, a)$ of reaching state $s' \in \mathcal{S}$ given that action $a \in \mathcal{A}$ was taken at state $s \in \mathcal{S}$
- $\mathcal{R}$ is a real-valued reward function denoted by $\mathcal{R}(s, a)$. It associates a state s with its immediate reward under action a.

An example of MDP is given in Figure 2.1. State transitions of a MDP satisfy the *Markov property* which describe that the next state solely relies on the current state and the agent's action but not on the prior history. In other words, it is conditionally independent of all previous states and actions leading to the current state.

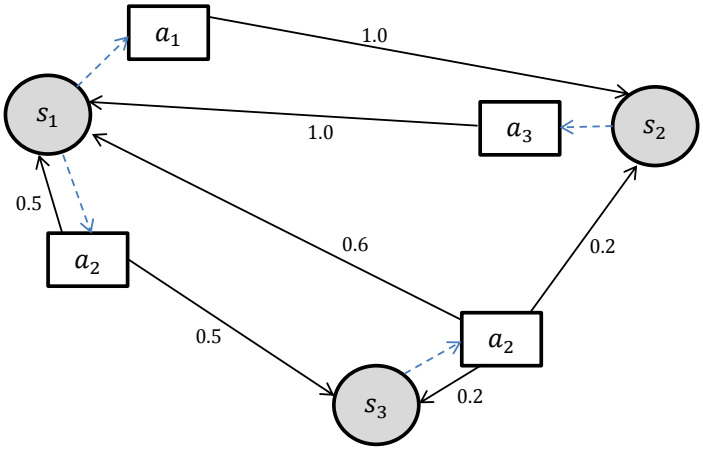


Figure 2.1: An example of MDP with 3 states (s_1, s_2, s_3) and 3 actions (a_1, a_2, a_3). Conditional probability of transitioning to another state is shown next to a solid edge.

A *policy* $\pi : \mathcal{S} \rightarrow \mathcal{A}$ maps a state to an action, with $\pi(s)$ denoting the action to be taken in state s. The core problem of MDPs, roughly speaking, is to find a policy that maximises the expected sum of discounted future rewards over a potentially infinite horizon

$$E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r^t \middle| s_0 \right]$$

where E_π denotes the expected value given that an agent follows policy π, γ is the *discount rate* $0 \leq \gamma \leq 1$, r^t is reward received at time t and s_0 is the initial starting state.

The *value function* under a policy π is the expected sum of discounted reward starting from state s and following π thereafter

$$V_\pi(s) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r^t \middle| s_0 = s \right]$$

A *greedy policy* π_V selects the action which maximises the expected value with respect to some value function V in each state

$$\pi_V(s) = \arg\max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \right\}$$

A policy π in an infinite horizon MDP is known to be *optimal* if $V_\pi \geq V_{\pi'}$ for all $s \in \mathcal{S}$. An optimal policy π^* is a greedy policy with respect to an optimal value

function V^* satisfying the following equation

$$V^*(s) = \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s') \right\}$$

2.2 Value iteration

While there are many methods for solving MDPs, we shall focus on value iteration in this thesis. *Value iteration* [Bellman 1957] is a dynamic programming method for computing an optimal policy and optimal value function by constructing a series of t stages-to-go value functions. Setting $V^0(s) = \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a)\}$, the optimal value function at stage $t+1$ is calculated as follows

$$V^{t+1}(s) = \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^t(s') \right\} \tag{2.1}$$

It could be shown that value iteration converges linearly in the number of iterations [Puterman 1994]. The formula $\sum_{s' \in \mathcal{S}} P(s'|s,a) V^t(s')$ in Equation 2.1 is known as *Bellman regression step*. Each iteration of value iteration is known as *Bellman backup*.

2.3 Summary

In this chapter, we gave a brief review on MDPs and the dynamic programming approach (value iteration) to solve them. However, the MDP representation given in this chapter are rarely utilised in modern implementations due to the lack of structures for exploitation. In the following chapter, we describe how MDPs have evolved into a factored form and elaborate on a structured approach to solve them.

Chapter 3

Factored Planning

In preparing for battle, I have always found that plans are useless but planning is indispensable.
Dwight D. Eisenhower

The state representation of MDP presented in Chapter 2 is neither intuitive nor compact. Thus, modern approach typically represents a state by factoring it into a number of properties for structure exploitation. In this chapter, we *do not* consider concurrency.

3.1 Factored MDPs

In *factored MDPs* [Boutilier et al. 1995], a state s is factored into and defined by a vector of *state variables* $\vec{\mathbf{x}} = (x_1, \ldots, x_n)$. Each state variable x_i may take an assignment from a set of possible values. For instance, suppose that we want to model the state of a room and one of the objects we could use for describing the state is the door. We could have a state variable called *door* and it may take either the value *open* or *closed*. Intuitively, state variables provide a more natural way of portraying a state. For simplicity, we assume state variables are Boolean where x_i could either be *true* or *false* [1].

In this framework, a state transition is triggered by an action that affects a subset of state variables. The value of a state variable changes depending on the executed action and the state variables it is conditioned on. For example, the value of the state variable *door* in the next time step relies on its current value and the action leading to the next state. Hence, *dynamic Bayesian network* (DBN) [Dean and Kanazawa 1989] is an appropriate candidate for formalising state transitions as it provides a factored representation of the transition function and allows us to specify causal relationship between state variables and action's consequences. In formalising state transitions, two set of variables are required. One of them, $\vec{\mathbf{x}} = (x_1, \ldots, x_n)$, referring to the current state of the system. The other, $\vec{\mathbf{x}'} = (x'_1, \ldots, x'_n)$, indicating the state of the system in the next time step (post-action state). We note that directed edges among

[1] For the case of multi-valued variables, we could use multiple Boolean type variables to represent a single multi-valued variable. Refer to [Rossi et al. 1990] and [Stergiou and Walsh 1999] for more details.

variables in $\vec{\mathbf{x}}'$ are prohibited [2].

The reward function is also factored by decomposing it into an additive form $\mathcal{R}(\vec{\mathbf{x}}, a) = \sum_i \mathcal{R}_i(\vec{\mathbf{x}}_i, a)$ where $\mathcal{R}_i(\vec{\mathbf{x}}_i, a)$ is a local reward dependant on action a and $\vec{\mathbf{x}}_i$ which is a subset of state variables. Factor MDPs will be clear to the reader once we provide a concrete example later in the chapter.

The classical method of solving MDPs via value iteration uses a tabular representation for explicit enumerations of the state space for transition function, reward function and value function. However, typical AI planning problems has large state space and fall prey to Bellman's *curse of dimensionality* [Bellman 1957]. For example, suppose that there are n state variables, then there are 2^n total states. Computation time and space would become intractable even for a relatively small n. Therefore, rather than a tabular representation, we employ a structured representation for solving MDPs using ADDs that obviate the need to enumerate the state space.

3.2 Algebraic Decision Diagrams

A *binary decision diagram* (BDD) is a common data structure used in representing a Boolean function $f : \mathcal{B}^n \to \mathcal{B}$. It takes n Boolean variables as arguments and returns a Boolean result. BDD is a binary tree where decision nodes (non-terminal nodes) are labelled with Boolean variables while terminal nodes are labelled with 0 or 1. Each decision node has two child called low child and high child. A dash edge from a decision node to a low child assigns 0 (*false*) to the variable labelled in the decision node. Conversely, a solid edge to a high child implies an assignment of 1 (*true*) to the variable.

A BDD is said to be *reduced* after applying the following two reduction rules [Bryant 1986]: (1) merge duplicate nodes having the same label and children; and (2) if both the outgoing edges of a node is pointing to the same child, eliminate the node and have the parents of the node point directly to the child of the eliminated node. Moreover, a BDD is termed *ordered* provided that it has an ordering for some list of variables i.e. decision nodes have a fixed order from root to leaf. Variable ordering plays a vital role in determining the size of a BDD. A function could be represented by more than one reduced BDDs with different variable orderings and sizes. A good variable ordering helps by reducing the size of the BDD.

Algebraic decision diagrams (ADD) [Bahar et al. 1997] generalise BDDs to represent real-valued functions $f : \mathcal{B}^n \to \mathbb{R}$. Hence, terminal nodes of ADD are labelled with real numbers. Figure 3.1 illustrates an ADD (Figure 3.1b) expressing the tabular representation of Figure 3.1a. To construct an ADD from a table, the decision tree is built while conforming to the desired variable orderings and obeying to the restriction that no variable appears more than once along any path from root to leaf. Then, apply the reduction rules as described previously.

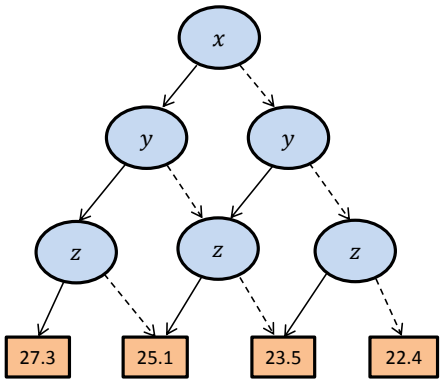
ADDs provide a space-efficient way in representing real-valued functions due to the removal of redundant nodes. Nevertheless, in the worst case of a complete ADD

[2] A remedy for such correlations is detailed in Boutilier [1997].

tree (full binary tree), there are a total of $2^{n+1} - 1$ nodes. Fortunately, this happens very rarely in practice. In comparison, a tabular representation is not space efficient since one would require 2^n lines to represent the same function. Apart from space efficiency, a fully-reduced ordered ADD has an additional benefit of having a *canonical representation*. In other words, for a given variable ordering, each distinct function has a unique reduced ordered ADD representation. From here on, we assume that ADDs are ordered and fully-reduced unless stated otherwise. As ADDs are the main data structure, they play a significant role and form one of the main discussions in this work.

x	y	z	$\mathbf{f(x,y,z)}$
0	0	0	**22.4**
0	0	1	**23.5**
0	1	0	**23.5**
0	1	1	**25.1**
1	0	0	**23.5**
1	0	1	**25.1**
1	1	0	**25.1**
1	1	1	**27.3**

(a)



(b)

Figure 3.1: Representation of function $f(x, y, z)$ in: (a) tabular; and (b) ADD.

3.3 Factored MDPs and ADDs

ADDs provide an extremely compact way of representing value functions, transition function and reward function. States that have identical values are grouped together under the reduction rules. In this section, we provide a concrete example of representing a factored MDP in ADDs.

In the problem of SysAdmin [Guestrin et al. 2003], a system administrator has to maintain a network of n computers $\{c_1, \ldots, c_n\}$ connected as a directed graph. The computers could be connected in any network topology such as those given in Figure 3.2. Each machine c_i is associated with a random variable x_i denoting whether it is *running* or *downed*. Furthermore, each computer has a probability of failing at every time step and this probability is conditioned on its current state and the state of its neighbouring computers connected to it. The system administrator has to decide which computer to *reboot* at each time step or do nothing. One could even choose to reboot a machine that is currently running. If a computer is rebooted, it will be running in the next time step. However, rebooting a computer would incur a penalty in the form of an action cost. The reward at each time step is number of running computers minus action cost. Therefore, the objective of the system administrator is to reboot the computer that has the most influence on the expected future reward.

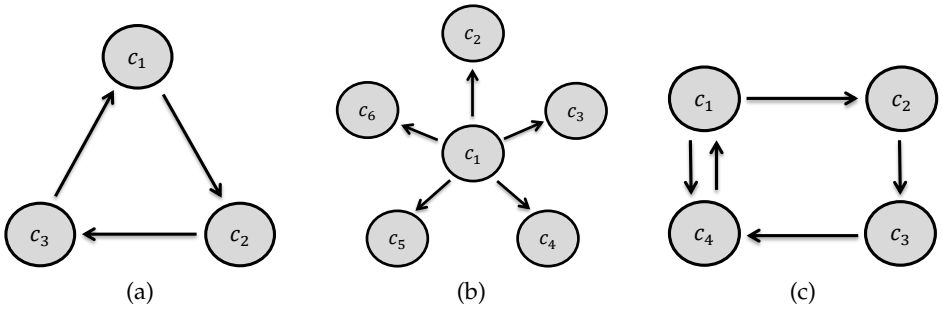


Figure 3.2: SysAdmin network topology: (a) unidirectional ring; (b) star; and (c) random graph.

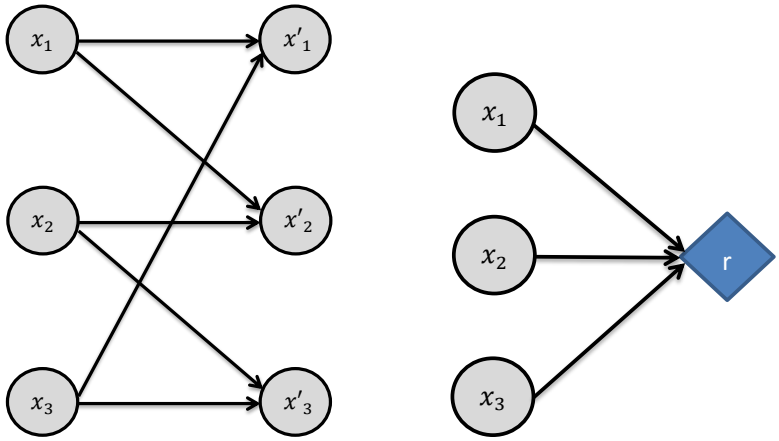
Example 3.1. *Consider an instance of the* SysAdmin *problem with three computers, c_1, c_2 and c_3, connected in a unidirectional ring, shown in Figure 3.2a. The state space is described by $\vec{\mathbf{x}} = (x_1, x_2, x_3)$. If x_i is* true*, then computer c_i is running, otherwise it is downed. The next state of the computer is denoted by the primed state variables. The set of actions is $\mathcal{A} = \{a_1, a_2, a_3, \text{noop}\}$, where a_i means rebooting computer c_i while* noop *corresponds to not rebooting any computer (do nothing). Recall that it is possible to reboot a computer which is currently running. The reward at each time step is just the number of running computers, however, whenever we choose to reboot a computer, an action cost of -0.75 is incurred on the rebooted computer. Let c_j be the computer which has an outgoing edge to c_i, then the transition of state variables is given by the conditional probability below*

$$P(x_i' = true | x_i, x_j, a_i) = \begin{cases} a_i = true & : 1 \\ a_i = false \wedge x_i = false & : 0 \\ a_i = false \wedge x_i = true \wedge x_j = true & : 0.8 \\ a_i = false \wedge x_i = true \wedge x_j = false & : 0.5 \end{cases}$$

Hence, if computer c_i is rebooted ($a_i = true$), c_i will be running in the next time step. Otherwise, the state of c_i depends on x_i and x_j. If computer c_i is downed, then it is impossible for c_i to be running in the next time step. On the other hand, if c_i and c_j are both running, then c_i will be running in the next time step with probability 0.8. For the case where c_i is running but c_j is not, c_i might be running with probability 0.5 *in the next time step.*

The DBN for Example 3.1 is given in Figure 3.3a. From the DBN, we observe that x_i' is conditioned on its parents, denoted as $Parents(x_i')$. Thus, we could rewrite the conditional probability in a more compact form: $P(x_i' | Parents(x_i'), a)$ where $a \in \mathcal{A}$. Now, the transition function for the DBN in Figure 3.3a takes the following factored form

$$\begin{aligned} P(s'|s,a) &= P(\vec{\mathbf{x}}'|\vec{\mathbf{x}}, a) \\ &= P(x_1', x_2', x_3'|x_1, x_2, x_3, a) \\ &= P(x_1'|Parents(x_1'), a) P(x_2'|Parents(x_2'), a) P(x_3'|Parents(x_3'), a) \\ &= \prod_i P(x_i'|Parents(x_i'), a) \end{aligned}$$



(a) (b)

Figure 3.3: Factored MDP illustration: (a) transition function as DBN; and (b) reward function as reward network.

Since each action $a \in \mathcal{A}$ has different outcomes, each action has its respective *conditional probability table* (CPT). The tabular representation of CPT for executing action a_1 is displayed in Table 3.1. Rather than working with the locally exponential tabular representation of CPTs, ADDs are used to represent CPTs. This is due to the capability of ADDs in capturing regularities in CPTs, therefore, saving memory space. Based on the CPTs, we could easily construct the respective ADDs. The corresponding ADDs are given in Figure 3.4.

x_1	x_3	x'_1	$\mathbf{P}(\mathbf{x'_1}\|\mathbf{Parents}(\mathbf{x'_1}), \mathbf{a_1})$
0	0	0	**0**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**0**
1	1	1	**1**

(a)

x_2	x_1	x'_2	$\mathbf{P}(\mathbf{x'_2}\|\mathbf{Parents}(\mathbf{x'_2}), \mathbf{a_1})$
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**0**
1	0	0	**0.5**
1	0	1	**0.5**
1	1	0	**0.2**
1	1	1	**0.8**

(b)

x_3	x_2	x'_3	$\mathbf{P}(\mathbf{x'_3}\|\mathbf{Parents}(\mathbf{x'_3}), \mathbf{a_1})$
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**0**
1	0	0	**0.5**
1	0	1	**0.5**
1	1	0	**0.2**
1	1	1	**0.8**

(c)

Table 3.1: Tabular representation of CPTs for taking action a_1 (rebooting computer c_1): (a) $P(x'_1|Parents(x'_1), a_1)$; (b) $P(x'_2|Parents(x'_2), a_1)$; and (c) $P(x'_3|Parents(x'_3), a_1)$.

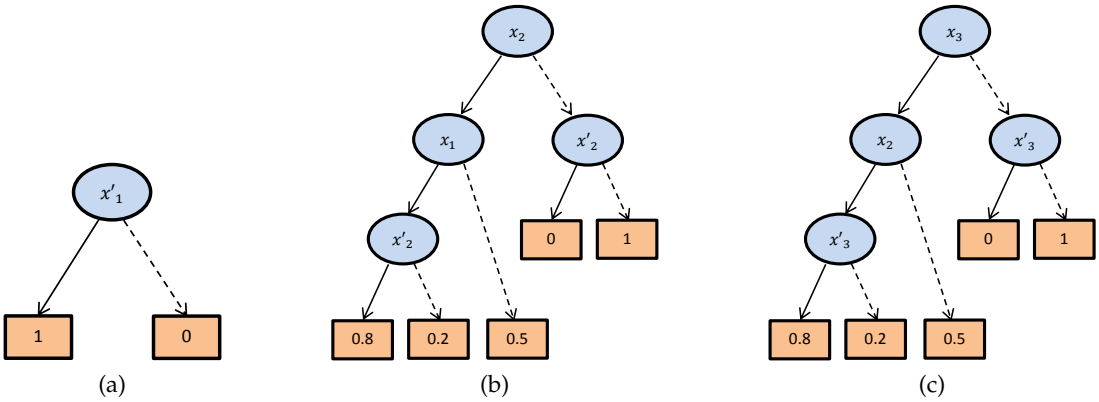


Figure 3.4: ADD representation of CPTs in Table 3.1: (a) $P(x'_1|Parents(x'_1), a_1)$; (b) $P(x'_2|Parents(x'_2), a_1)$; and (c) $P(x'_3|Parents(x'_3), a_1)$.

Finally, reward function is depicted as influence diagram [Howard and Matheson 1984] and is known as reward network (Figure 3.3b). From the reward network, the final reward is actually determined by the values of x_1, x_2 and x_3. Recall that reward function in factored MDPs is factored into an additive form $\mathcal{R}(\vec{\mathbf{x}}, a) = \sum_i \mathcal{R}_i(\vec{\mathbf{x}}_i, a)$ where $\mathcal{R}_i(\vec{\mathbf{x}}_i, a)$ is a local reward. Like CPTs, each local reward is represented by an ADD and is constructed in a similar manner. Figure 3.5 shows the local rewards of rebooting computer c_1.

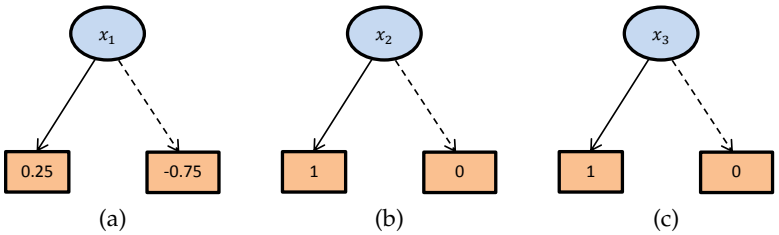


Figure 3.5: Local rewards for performing action a_1: (a) $\mathcal{R}_1(x_1, a_1)$; (b) $\mathcal{R}_2(x_2, a_1)$; and (c) $\mathcal{R}_3(x_3, a_1)$. Note that the terminal values in (a) have been affected by the action cost.

Altogether, there will be a total of 24 ADDs since each action would produce 6 ADDs (3 ADDs for CPTs and 3 ADDs for local rewards) and we have 4 actions including *noop*. Essentially, with m actions and n state variables, there will be $\mathcal{O}(m \cdot n)$ (quadratic in the worst case) ADDs from formalising a planning problem.

The depiction of transition function and reward function as ADDs often saves tremendous amount of space mainly due to the reduction rules in merging identical subtrees. As an example, compare Table 3.1a to Figure 3.4a. Figure 3.4a only has 3 nodes while Table 3.1a requires $2^3 = 8$ rows to represent the same CPT. Let $n = |Parents(x'_i)|$. Then, a standard tabular representation of a CPT under action a for state variable x'_i requires 2^{n+1} rows ($+1$ to include x'_i). This is exponential in terms of n.

There are two primary internal structures we could exploit simultaneously in factored MDPs for efficient decision making: (1) *additive or multiplicative independence*; and (2) *context-specific independence* . Additive independence relates to the observation where typical large-scale systems could often be broken down into a combination of locally interacting components. For example, consider a car assembly line. Assume that the tasks of the assembly line include the installation of the engine, hood and wheels. Then, the *local* quality of the engine installation task relies entirely on the state when the task is taken and the quality of the engine. The sum of all these local qualities from engine installation, setting up the hood and wheels installation determines the final quality of the car. Such additive structure is present in the reward function where we decomposed the function into a sum of local rewards. Additive independence in rewards has been studied extensively in utility theory and related fields [Keeney and Raiffa 1993; Bacchus and Grove 1995]. In addition, multiplicative independence is a generalisation of additive independence and occurs in value functions.

Context-specific independence [Boutilier et al. 1996] structure encodes a different type of locality of influence and is best explained using an example. Consider again the car assembly line example but this time we want to paint the car once all the other tasks have been *completed*. In all context where the installation of the engine or hood or wheels has not been completed, the probability that the car has been painted is 0 *independent* of the taken action. In simple terms, the values of some variables in the CPT are irrelevant under certain settings. Context-specific structure is present in both the transition function (the CPTs) and the reward function.

3.4 Manipulations of ADDs

While we have shown how to represent the CPTs and reward function as ADDs, we have yet to elaborate on the methods for manipulating them. Basically, three operations on ADDs are required throughout this thesis: (1) scalar operations on terminal nodes; (2) binary operations involving two ADDs; and (3) removal of variables in an ADD.

3.4.1 Scalar Operations

Quite often, there is a need to apply scalar operations like addition and multiplication on the terminal nodes. This is easily accomplished by traversing the ADD to search for terminal nodes. At the terminal node, we apply the scalar operation on the value of the terminal node and replace the old value with new old one. Figure 3.6 provides some illustrations.

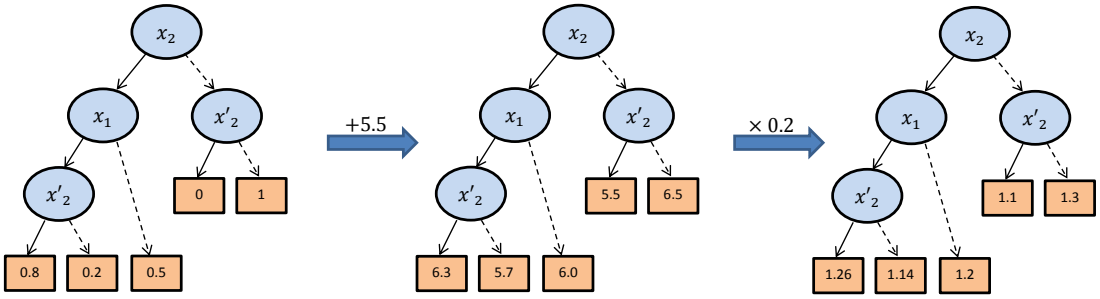


Figure 3.6: Scalar operation on the terminal nodes. The first operation is scalar addition of 5.5 followed by scalar multiplication of 0.2.

3.4.2 Apply Operation

The *Apply* procedure [Bahar et al. 1997] performs a binary operation on two ADDs and returns an ADD as the result. Although numerous binary operations are supported, we only require *addition*, *multiplication* and *max* in this entire thesis. Suppose that A and B are ADDs, then we define $Apply(A, B, op)$ to be the *Apply* procedure for performing binary operation op on A and B. An example of the *Apply* procedure is given in Figure 3.7. Complexity of *Apply* is $\mathcal{O}(n^2)$ where n is number of nodes in the larger ADD.

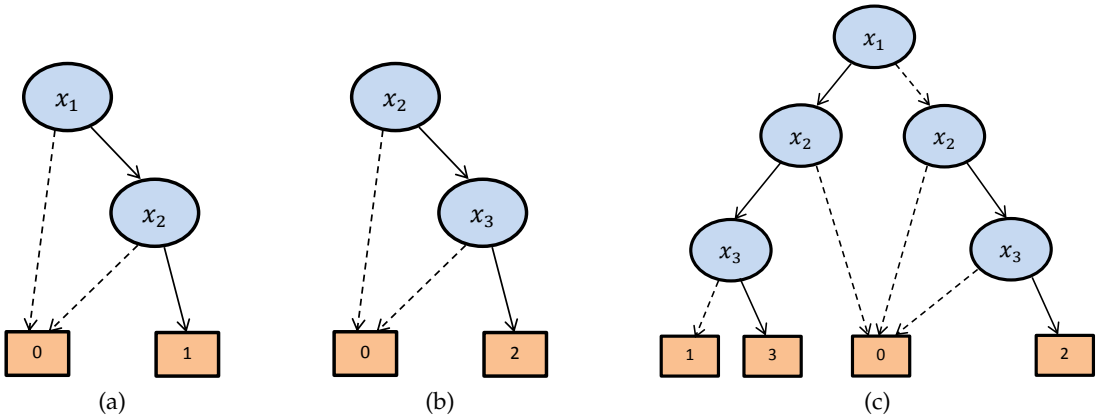


Figure 3.7: *Apply* procedure of $Apply(A, B, +)$ where op is *addition*: (a) ADD denoted by A; (b) ADD denoted by B; and (c) the resulting ADD.

3.4.3 OpOut Operation

OpOut is one of the core ADD manipulation techniques for removing or eliminating a variable from an ADD such as summing out (i.e. $\sum_x f(x, y, z)$) or maxing out variables ($\max_x f(x, y, z)$). It consists of two steps, *Restrict* then *Apply*. *Restrict* [Bryant 1986] works by putting a restriction on a variable, say x_i, in an ADD representing a function

f to either *true* ($f|_{x_i=true}$) or *false* ($f|_{x_i=false}$). Nodes with label x_i are removed from the ADD while the parents of the eliminated nodes point directly to either the low child (if x_i is restricted to *false*) or high child (if x_i is restricted to *true*) of the removed nodes. Figure 3.8 shows an example of the *Restrict* procedure.

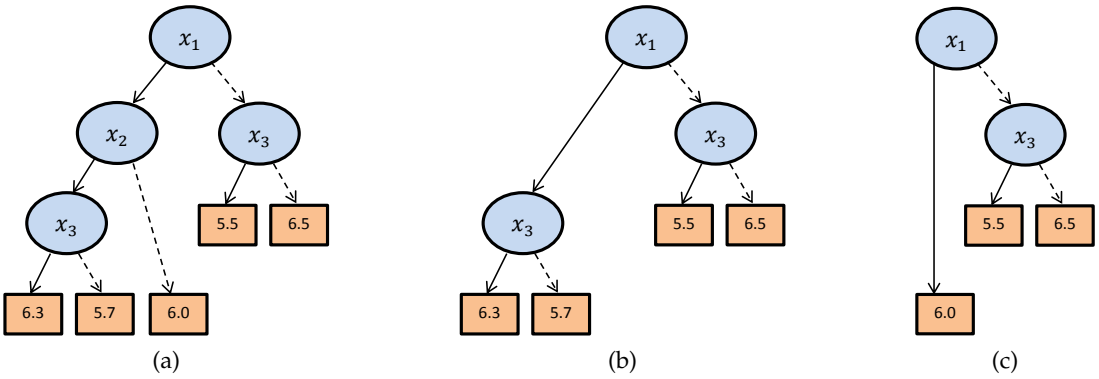


Figure 3.8: *Restrict* procedure: (a) the original ADD; (b) restricting x_2 to *true*; and (c) restricting x_2 to *false*. Note the redirection of the solid edge belonging to the node labelled with x_1 and that the node labelled with x_2 is removed from the ADD.

The *OpOut* procedure is defined as $OpOut(A, x_i, op)$ where A is an ADD, x_i is the variable to be removed and op is the desired operation. Using Figure 3.8a as an example, assume that we want to perform $\sum_{x_2} f(x_1, x_2, x_3)$ which is $OpOut(f, x_2, +)$. First, we restrict x_2 to *true* (Figure 3.8b) and x_2 to *false* (Figure 3.8c). Then, perform the *Apply* routine on the restricted ADDs with addition as the binary operation. For the case of maxing out a variable, $\max_{x_2} f(x_1, x_2, x_3)$ or $OpOut(f, x_2, max)$, use max as the binary operation for *Apply* instead. Refer to Figure 3.9 for the results produced by *OpOut*. Similar to *Apply*, time complexity of *OpOut* is $\mathcal{O}(n^2)$ of the larger ADD.

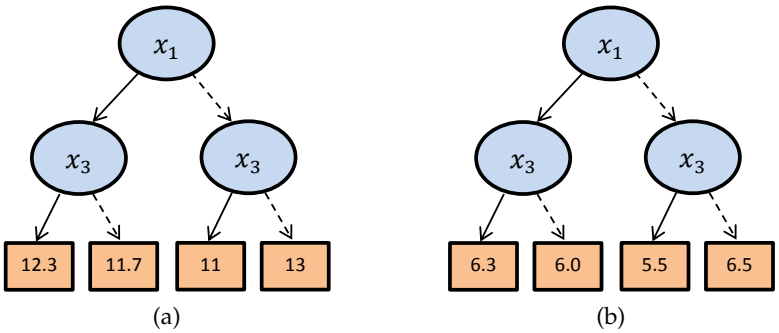


Figure 3.9: *OpOut* procedure for: (a) $\sum_{x_2} f(x_1, x_2, x_3)$ - $OpOut(f, x_2, +)$; and (b) $\max_{x_2} f(x_1, x_2, x_3)$ - $OpOut(f, x_2, max)$.

3.5 Factored Value Iteration

Having described the representation of factored MDPs as ADDs and the operations on ADDs, we are now ready to present the algorithm that implements a form of value iteration for factored MDPs. Recall that the value iteration [Bellman 1957] for MDPs is given as

$$V^{t+1}(s) = \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^t(s') \right\}$$

In factored MDPs, a state s is represented by $\vec{\mathbf{x}} = (x_1, \ldots, x_n)$

$$V^{t+1}(\vec{\mathbf{x}}) = \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(\vec{\mathbf{x}},a) + \gamma \sum_{\vec{\mathbf{x}'}} P(\vec{\mathbf{x}'}|\vec{\mathbf{x}},a) V^t(\vec{\mathbf{x}'}) \right\}$$

In addition to that, the reward function is factored into an additive form while transition function is formalised as a DBN. The transition function in factored notation is $P(\vec{\mathbf{x}'}|\vec{\mathbf{x}},a) = \prod_i^n P(x_i'|Parents(x_i'),a)$. Consequently

$$V^{t+1}(\vec{\mathbf{x}}) = \max_{a \in \mathcal{A}} \left\{ \sum_i \mathcal{R}_i(\vec{\mathbf{x}}_i,a) + \gamma \sum_{\vec{\mathbf{x}'}} \prod_i^n P(x_i'|Parents(x_i'),a) V^t(\vec{\mathbf{x}'}) \right\} \tag{3.1}$$

Also, the expected value calculation $\sum_{\vec{\mathbf{x}'}}$ in Equation 3.1 could instead be written in a factored form $\sum_{x_1'} \cdots \sum_{x_n'}$. Here lies the efficiency of factored MDPs. For the sake of explanation, suppose that one *OpOut* operation of $\sum_{x_i'}$ takes $\mathcal{O}(k)$ time. On account of having n state variables, the factored form would require $\mathcal{O}(n \cdot k)$ time complexity. On the contrary, the summing out of $\sum_{\vec{\mathbf{x}'}}$ would involve the enumeration of 2^n possible combinations of state variables. Rephrasing Equation 3.1, the value iteration for factored MDP is

$$V^{t+1}(\vec{\mathbf{x}}) = \max_{a \in \mathcal{A}} \left\{ \sum_i \mathcal{R}_i(\vec{\mathbf{x}}_i,a) + \gamma \sum_{x_1'} \cdots \sum_{x_n'} \prod_i^n P(x_i'|Parents(x_i'),a) V^t(\vec{\mathbf{x}'}) \right\} \tag{3.2}$$

where $\mathcal{R}_i(\vec{\mathbf{x}}_i,a)$, $P(x_i'|Parents(x_i'),a)$ and $V^t(\vec{\mathbf{x}'})$ are represented in ADDs.

The *SPUDD* algorithm [Hoey et al. 1999] is a value iteration procedures based on ADDs. It computes a series of t stages-to-go value functions until the termination condition is satisfied. These value functions are represented and manipulated as ADDs. The outline of SPUDD is discussed in Algorithm 3.1.

The first two step of the algorithm is the initialisation step where we formalise the reward function and transition function as ADDs. Initial value function V^0 is set to $\max_{a \in \mathcal{A}} \{\mathcal{R}(\vec{\mathbf{x}},a)\}$. There are many termination conditions we could use but just to name a few, finite horizon h or the difference between V^{t+1} and V^t. The value function V^t in Equation 3.2 is regarded as representing values at future states. Therefore, V^t is primed in Step 3(a) causing the state variables to be replaced by their primed counterparts that refer to the next state or post-action state. Hence, this allows us to

distinguish between the state with t stages-to-go and the un-primed variables referring to the state with $i+1$ stages-to-go.

The careful reader may notice that we ignore primed variables which are not in $V^{t'}$ in Step 3(b)ii. Recall that $\sum_{x'_i} P(x'_i|\ldots) = 1$ in conditional probabilities. If the primed variable x'_i is not in $V^{t'}$, then x'_i will only appear in the CPT $P(x'_i|Parents(x'_i), a)$ since we disallow directed edges among post-action variables in the DBN. Thus, we could immediately sum out the CPT to obtain 1. Considering the fact that 1 is the multiplicative identity, we could just ignore variables which are not in $V^{t'}$ and skip the multiplication along with the marginalisation step. Hence, saving substantial amount of time when $V^{t'}$ is a large ADD.

Step 3(b)ii is the Bellman regression step. Note that Step 3(b)ii.A corresponds to $Apply(tmp, P(x'_i|Parents(x'_i), a), *)$ while the sum out in Step 3(b)ii.B is simply $OpOut(tmp, x'_i, +)$. Operations on large ADD are expensive, SPUDD tries to minimise the size of the tmp ADD by summing state variables as soon as possible (after the CPT multiplication). Each time we sum out a variable, we remove the variable from the ADD, resulting in a smaller ADD in general. In Step 3(b)iii, tmp is discounted by γ (ADD scalar operation) and reward is added via $Apply$. The expected future value of executing action a is assigned to v_a. Finally, Step 3(c) maximise over all v_a to produce V^{t+1}. This is implemented through *Apply* with *max* as the binary operation. Step 3(d) increment t to imply the next time step of value iteration.

Algorithm 3.1 Factored Value Iteration

1. formalise the reward function $\mathcal{R}(\vec{\mathbf{x}}, a)$ and CPTs, i.e. $P(x'_i|Parents(x'_i), a)$, as ADDs.
2. $V^0 \leftarrow \max_{a\in\mathcal{A}} \{\mathcal{R}(\vec{\mathbf{x}}, a)\}$.
3. **while** stopping criteria has not been met
   (a) prime the value function, replacing x_i with x'_i. Let $V^{t'}$ denote the primed value function.
   (b) **for** all $a \in \mathcal{A}$
      i. $tmp \leftarrow V^{t'}$
      ii. *// Bellman regression step*
      for all *primed variables* x'_i in $V^{t'}$
         A. $tmp \leftarrow tmp * P(x'_i|Parents(x'_i), a)$
         B. $tmp \leftarrow \sum_{x'_i} tmp$
      iii. *// discount by* γ *and add the reward*
      $v_a \leftarrow \mathcal{R}(\vec{\mathbf{x}}, a) + \gamma * tmp$
   (c) $V^{t+1} \leftarrow$ maximise over all v_a
   (d) increment t

3.6 Summary

We began this chapter by introducing factored MDPs and ADDs. Then, we illustrated the ADDs representation of factored MDPs using SYSADMIN as an example. In addition, we showed how to perform value iteration with ADDs via the SPUDD algorithm. In the next chapter, we propose an efficient method for solving concurrent factored MDPs by factoring actions.

Chapter 4

Concurrent Factored Planning

Thus, the task is, not so much to see what no one has yet seen;
but to think what nobody has yet thought, about that which everybody sees.
Erwin Schrödinger

The focus so far in this thesis has been on *non-concurrent* factored planning where only *one* action may be executed at any time step. Such restriction that only allows one action per time step is unrealistic in solving real-world applications which are inherently concurrent. The term *concurrent* refers to the situation where multiple actions are executed simultaneously at every time step. For instance, in concurrent SysAdmin, the system administrator is able to operate on all the machines concurrently in each time step.

One could always take the naïve approach of introducing concurrency into factored planning by enumeration of all possible combination of actions. To illustrate, if there are two *single* actions a_1 and a_2, then the set of possible combination of actions is $\{noop, a_1, a_2, a_1a_2\}$ where a_1a_2 implies taking both the action a_1 and a_2 concurrently. It is rather easy to tell that this is exponential in the number of actions. With m single actions, the size of the action space $\mathcal{A}$ would turn out to be 2^m. This would in turn, leads us to two non-trivial complications. First, performing 2^m *Bellman regression* step. Second, exponential space to formalise a MDP problem in ADD. Recall that if there are n state variables, there will be a total of $\mathcal{O}(2^m \cdot n)$ ADDs. Subsequently, both time and space grow to be intractable relatively fast.

In this chapter, as our first major contribution, we propose an efficient approach to use dynamic programming in concurrent factored MDPs without enumerations of concurrent actions.

4.1 Concurrent Factored MDPs

In factored MDPs, we observed how a state is factored into state variables, how a transition function is factored by DBN and how the reward function is factored into an additive form. *Concurrent factored MDPs* extend factored MDPs by factoring actions into *action variables*. Like state variables, action variables are denoted by an action vector $\vec{\mathbf{a}} = (a_1, \ldots, a_m)$. Action variables have the same properties as state variables in

factored MDPs. On top of that, they are applied in the same manner. One could regard action variables as state variables while keeping in mind that action variables relate to actions. At each time step, m concurrent actions are taken where each action variables a_i could either be *true* or *false* (not taking action a_i is considered as an action itself). The most distinctive feature of concurrent factored MDPs is that actions are now variables in the DBN, reward network and ADD formalisation. Such representation provides an even more intuitive way of understanding actions' consequences from multiple concurrent actions as we will see in the example below.

Example 4.1. *Consider a* concurrent *instance of the* SysAdmin *problem with three computers* $\{c_1, c_2, c_3\}$ *connected in a unidirectional ring. Using the same notation as Example 3.1, states are denoted by* $\vec{\mathbf{x}} = (x_1, x_2, x_3)$ *with concurrent actions* $\vec{\mathbf{a}} = (a_1, a_2, a_3)$*. Setting* a_i *to* true *implies rebooting computer* c_i*.* Each *action would incur an action cost of* -0.75*. Reward at any time step is total number of running computers subtracted by total action cost. If* c_j *is the machine that has an outgoing edge to* c_i*, the conditional probability below details the transition of state variables*

$$P(x_i' = true | x_i, x_j, a_i) = \begin{cases} a_i = true & : 1 \\ a_i = false \wedge x_i = false & : 0 \\ a_i = false \wedge x_i = true \wedge x_j = true & : 0.8 \\ a_i = false \wedge x_i = true \wedge x_j = false & : 0.5 \end{cases}$$

Referring to Figure 4.1a, concurrent factored MDPs have a slightly different DBN. Action variables currently play an important role in indicating direct casual effect on next state variables. Due to the participation of action variables in the DBN, conditional probability is now rephrased to $P(x_i'|Parents(x_i'))$. The transition function is defined as $P(\vec{\mathbf{x}'}|\vec{\mathbf{x}}, \vec{\mathbf{a}})$ where the factored form is $\prod_i P(x_i'|Parents(x_i'))$. In general, a next state variables x_i' may be affected by multiple action variables. An example of a concurrent CPT represented by an ADD is given in Figure 4.2. We note that the concurrent CPT is constructed in a similar manner as before.

Like the DBN, the reward network shown in Figure 4.1b is now influenced by action variables. Reward function is factored into $\mathcal{R}(\vec{\mathbf{x}}, \vec{\mathbf{a}}) = \sum_i \mathcal{R}_i(\vec{\mathbf{x}}_i, \vec{\mathbf{a}}_i)$ where $\mathcal{R}_i(\vec{\mathbf{x}}_i, \vec{\mathbf{a}}_i)$ is a local reward dependant on state variable subset $\vec{\mathbf{x}}_i$ and action variable subset $\vec{\mathbf{a}}_i$.

Reward in Example 4.1 is number of running computers subtracted by total action costs or $\sum_i \mathcal{R}_i(\vec{\mathbf{x}}_i, \vec{\mathbf{a}}_i) = (x_1 + x_2 + x_3) - 0.75(a_1 + a_2 + a_3)$. Again, observe the additive structure in the reward function (note that subtraction is just a special case of addition). ADDs for local reward x_1 and action cost a_1 are illustrated in Figure 4.3. In total, there will be 3 ADDs for CPTs and 6 ADDs from actions costs and local rewards. Concurrent factored MDPs have an additional advantage over factored MDPs when it comes to formalising CPTs and reward function in ADDs. Factored MDPs require different sets of CPTs and reward for *each* action. On the other hand, concurrent factored MDPs only have one set of CPTs and reward. With m action variables and n state variables, there will be $\mathcal{O}(m + n)$ ADDs in contrast to $\mathcal{O}(m \cdot n)$ in factored MDPs.

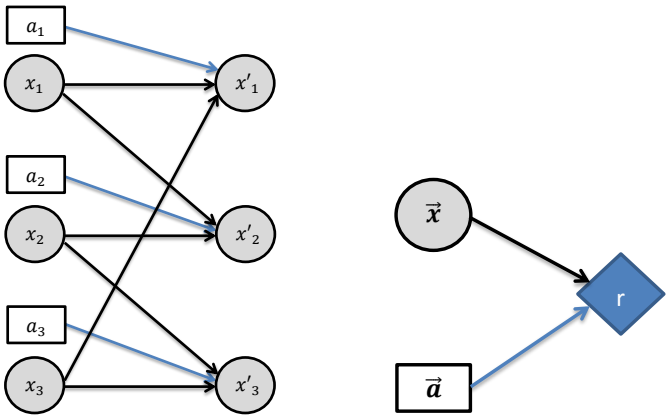


Figure 4.1: Concurrent factored MDP illustration: (a) DBN; and (b) the reward network. Observe the participation of action variables.

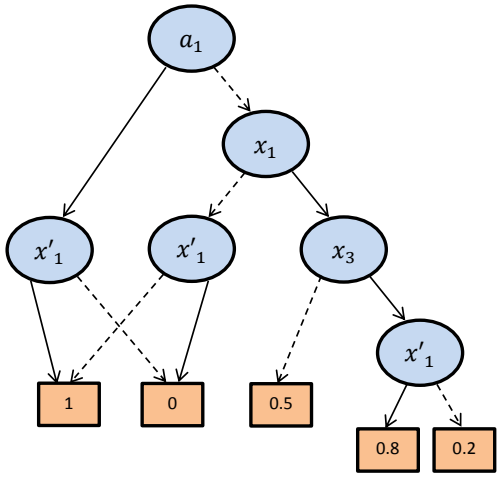


Figure 4.2: ADD representation of concurrent CPT. Note that action variables are now part of the ADD.

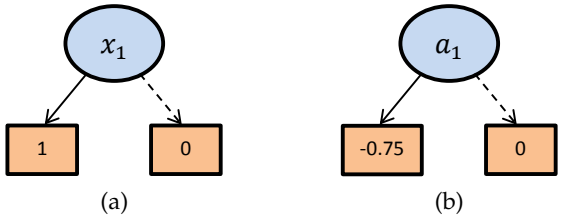


Figure 4.3: Concurrent local reward x_1 and action cost a_1: (a) local reward x_1; and (b) action cost a_1.

4.2 Concurrent Factored Value Iteration

In this section, we present the algorithm for solving concurrent factored MDPs based on factored value iteration. Recall that factored value iteration (Equation 4.1) only allows one action to be taken at each time step by determining which action a is the most desirable from the max out operation

$$V^{t+1}(\vec{\mathbf{x}}) = \max_{a \in \mathcal{A}} \left\{ \sum_i \mathcal{R}_i\left(\vec{\mathbf{x}}_i, a\right) + \gamma \sum_{x'_1} \cdots \sum_{x'_n} \prod_i^n P(x'_i | Parents(x'_i), a) V^t(\vec{\mathbf{x}'}) \right\} \tag{4.1}$$

In concurrent factored MDPs, concurrent actions are denoted by $\vec{\mathbf{a}} = (a_1, \ldots, a_m)$. From the DBN, conditional probabilities are now defined as $P(x'_i | Parents(x'_i))$ while the reward function is simply $\mathcal{R}\left(\vec{\mathbf{x}}, \vec{\mathbf{a}}\right) = \sum_i \mathcal{R}_i\left(\vec{\mathbf{x}}_i, \vec{\mathbf{a}}_i\right)$. Instead of determining the best action, the vector $\vec{\mathbf{a}}$ is maxed out in the concurrent case

$$V^{t+1}(\vec{\mathbf{x}}) = \max_{\vec{\mathbf{a}}} \left\{ \sum_i \mathcal{R}_i\left(\vec{\mathbf{x}}_i, \vec{\mathbf{a}}_i\right) + \gamma \sum_{x'_1} \cdots \sum_{x'_n} \prod_i^n P(x'_i | Parents(x'_i)) V^t(\vec{\mathbf{x}'}) \right\} \tag{4.2}$$

Now, we are able to solve concurrent problems with Equation 4.2. However, the complication associated with Equation 4.2 is that, with m action variables, we need to max over all 2^m possible combination of actions. A better way to perform the max out procedure is to factor *max* into the factored form $\max_{a_1} \cdots \max_{a_m}$ as shown in *concurrent factored value iteration* (Equation 4.3)

$$V^{t+1}(\vec{\mathbf{x}}) = \max_{a_1} \cdots \max_{a_m} \left\{ \sum_i \mathcal{R}_i\left(\vec{\mathbf{x}}_i, \vec{\mathbf{a}}_i\right) + \gamma \sum_{x'_1} \cdots \sum_{x'_n} \prod_i^n P(x'_i | Parents(x'_i)) V^t(\vec{\mathbf{x}'}) \right\} \tag{4.3}$$

With that, we only have to max out every action variable a_i individually. If a max out operation takes $\mathcal{O}(k)$ time, then with m action variables, it would require a total of $\mathcal{O}(m \cdot k)$ in time complexity which is an exponential reduction compared to the previous case. One could easily infer that the factoring of the max operation borrows the same idea from factored MDPs where the expected value calculation is factored to $\sum_{x'_1} \cdots \sum_{x'_n}$. For simplicity, the equation $\gamma \sum_{x'_1} \cdots \sum_{x'_n} \prod_i^n P(x'_i | Parents(x'_i)) V^t(\vec{\mathbf{x}'})$ would be referred as *Regress*.

In order to exploit the factored max efficiently, it is necessary to have some additive structure. Since ADDs are functions and $\mathcal{R}(\vec{\mathbf{x}}, \vec{\mathbf{a}})$ has an additive form, setting $V^0 = \max_{a_1} \cdots \max_{a_m} \{\mathcal{R}(\vec{\mathbf{x}}, \vec{\mathbf{a}})\}$ would lead to an additive form $V^0 = f_1 + \cdots + f_k$ where f_i is an ADD. The trick is *not* to sum functions together but to keep them in an additive form. Consequently, each function f_i contains a small subset of variables and the corresponding ADD will be relatively small.

Due to the fact that $Regress\left(\sum_i f_i\right) = \sum_i Regress(f_i)$, we are able to apply *Regress* on each f_i *individually* in the next time step of value iteration as demonstrated in Equation 4.4. Considering that f_i is represented by a small ADD, the *Regress* procedure would

be very *efficient*

$$
\begin{aligned}
V^{t+1}(\vec{\mathbf{x}}) &= \max_{a_1} \cdots \max_{a_m} \left\{ \sum_i \mathcal{R}_i(\vec{\mathbf{x}}_i, \vec{\mathbf{a}}_i) + \gamma \sum_{x'_1} \cdots \sum_{x'_n} \prod_i^n P(x'_i | Parents(x'_i)) V^t(\vec{\mathbf{x}'}) \right\} \\
&= \max_{a_1} \cdots \max_{a_m} \left\{ \sum_i \mathcal{R}_i(\vec{\mathbf{x}}_i, \vec{\mathbf{a}}_i) + \gamma \sum_{x'_1} \cdots \sum_{x'_n} \prod_i^n P(x'_i | Parents(x'_i)) (f_1 + \ldots + f_k) \right\} \\
&= \max_{a_1} \cdots \max_{a_m} \left\{ \sum_i \mathcal{R}_i(\vec{\mathbf{x}}_i, \vec{\mathbf{a}}_i) + Regress(f_1 + \ldots + f_k) \right\} \\
&= \max_{a_1} \cdots \max_{a_m} \left\{ \sum_i \mathcal{R}_i(\vec{\mathbf{x}}_i, \vec{\mathbf{a}}_i) + Regress(f_1) + \ldots + Regress(f_k) \right\} \qquad (4.4)
\end{aligned}
$$

In contrast to the previous chapter where the additive structure in reward function did not play any important role, here, we exploit the additive structure. By exploiting the additive structure in Equation 4.4 and *not* summing functions together, we end up with Equation 4.5

$$V^{t+1}(\vec{\mathbf{x}}) = \max_{a_1} \cdots \max_{a_m} \{g_1 + \ldots + g_l\} \qquad (4.5)$$

For maxing out action variable a_i, functions containing a_i are summed together, producing an ADD encompassing all variables in those functions. Then, action variable a_i is maxed out via *OpOut*. For example,

$$
\begin{aligned}
&max_{a_1} max_{a_2} \{g_1(x_1, a_1) + g_2(x_2, a_1) + g_3(x_1, a_2)\} \\
&= max_{a_1} max_{a_2} \{g_4(x_1, x_2, a_1) + g_3(x_1, a_2)\} \\
&= max_{a_2} \{g_5(x_1, x_2) + g_3(x_1, a_2)\} \\
&= g_5(x_1, x_2) + max_{a_2} \{g_3(x_1, a_2)\} \\
&= g_5(x_1, x_2) + g_6(x_1)
\end{aligned}
$$

Utilising the same trick and *not* summing functions as before, V^{t+1} will also have a *factored additive structure* after maxing out all action variables in Equation 4.5. The function h_i in Equation 4.6 will be known as *additive value function*

$$V^{t+1}(\vec{\mathbf{x}}) = h_1 + \ldots + h_o \qquad (4.6)$$

One might be concerned that retaining all functions in an additive structure like Equation 4.6 would increase the number of functions h_i in V^{t+1} as time progresses. We emphasize that this is not the case because in the next iteration when h_i is regressed, $Regress(h_i)$ would contain action variables. More specifically, the multiplication of h_i with CPTs $P(x'_i | Parents(x'_i))$ produces an ADD containing action variables. During the max out procedure, these regressed functions would gradually merge into larger ADDs.

Algorithm 4.1 is an extension to SPUDD (Algorithm 3.1) for implementing concurrent factored value iteration while exploiting the additive structure mentioned above. Step 1 to 3(a) are very much alike to the ones in Algorithm 3.1, the only difference is that V^t is in an additive form. Additive structure of the algorithm is enforced and implemented by *tmp* whereas the *Regress* step is described in Step 3(c). Reward is added into *tmp* in Step 3(d). Action variables are maxed out one at a time in Step 3(e). For efficiency, they are maxed out in some order ensuring that the resulting ADDs from the max out operation are as small as possible.

Algorithm 4.1 Concurrent Factored Value Iteration

1. formalise the reward function $\mathcal{R}(\vec{\mathbf{x}}, \vec{\mathbf{a}})$ and CPTs, i.e. $P(x_i'|Parents(x_i'))$, as ADDs.
2. $V^0 \leftarrow \max_{a_1} \cdots \max_{a_m} \{\mathcal{R}(\vec{\mathbf{x}}, \vec{\mathbf{a}})\}$.
3. **while** stopping criteria has not been met
   (a) prime all *state variables* in all functions in V^t. Let $V^{t'}$ denote the primed value function.
   (b) $tmp \leftarrow \{\}$
   (c) *// regress each f_i individually and add the result into tmp*
   for all functions $f_i \in V^{t'}$
      i. $ret \leftarrow Regress(f_i)$
      ii. $tmp \leftarrow tmp \cup ret$
   (d) *// add the reward*
   $tmp \leftarrow tmp \cup \mathcal{R}(\vec{\mathbf{x}}, \vec{\mathbf{a}})$
   (e) *// max out all action variables in tmp*
   $V^{t+1} \leftarrow \max_{a_1} \cdots \max_{a_m} \{tmp\}$
   (f) increment t

4.3 Related Work

In the past, various researches were conducted on factored action MDPs but most of these works do not evaluate an optimal policy and value function over all states. Earlier works on goal-based oriented problem [Younes and Simmons 2004; Little and Thiebaux 2006] require specification of initial states but do not evaluate an optimal policy over all states nor do they guarantee optimality. Furthermore, they are not able to handle arbitrary rewards. Although Mausam and Weld [2004] considered general reward, their work requires initial state. On top of that, it does not guarantee optimality nor compute an optimal solution over all states. Guestrin, Koller, and Parr [2001] proposed a method for solving general MDPs that computes policies over all state but requires additional knowledge without optimality guarantee.

Furthermore, there are some works on weakly coupled MDPs [Singh et al. 1998; Meuleau et al. 1998] where the state space of the MDP is divided to form subMDPs. Concurrency is implemented by treating each subMDP as a concurrent process.

4.4 Summary

In this chapter, we proposed a method to solve concurrent problems without enumeration of concurrent actions. Actions are factored into action variables and play an important role in DBN as well as the reward network. From the fact that $Regress(\sum_i f_i) = \sum_i Regress(f_i)$, value functions are kept in an additive form so that we could perform the *Regress* step efficiently. Consequently, the regressed value functions would be in a factored additive form that is later exploited by factored max. Perhaps, the most distinctive feature of our work is that no one has ever considered factored MDPs as expressively as our research. In the following chapter, we shall talk about the projection of ADDs into an additive form for scalability.

Chapter 5

Projection

... all models are approximations. Essentially, all models are wrong, but some are useful. However, the approximate nature of the model must always be borne in mind ...
George E. P. Box

Even with all those careful manipulations of ADDs to keep them small in size by exploitation of additive structure, the size of ADDs would still blow up due to dependencies between variables and the max out procedure. For example, consider the max out operation $\max_{a_1} \{f_1(x, z, a_1) + f_2(y, a_1, a_2)\}$. Note that both f_1 and f_2 contain a_1. Before maxing out a_1, these two functions are summed together to obtain a *larger* ADD $f_3(x, y, z, a_1, a_2)$ consisting of even more variables. In other words, there will eventually be one additive value function in V^{t+1} which *converged* into a large ADD encompassing *all* state variables (action variables are all eliminated after the max out procedure).

Ultimately, the converged ADD dominates computation time as size increases. In addition, since most ADD operations are of the order $\mathcal{O}(n^2)$, it does not require a large problem size for time to become intractable. Even for a trivial problem like SysAdmin, scalability issues start to emerge with 12 computers.

In order to scale to a larger problem size, we have to opt for an approximate solution. We showed in the previous chapter that concurrent factored value iteration is very efficient in exploiting additive structure in reward and value functions. Therefore, our approximation method projects converged ADDs into an *additive* list of smaller ADDs for additive structure exploitation. Additive projection of ADDs is our second major contribution in this thesis.

5.1 Additive Projection of Value Functions

Efficiency in concurrent factored value iteration comes from applying *Regress* on each additive value function or $\sum_i Regress(f_i)$ where f_i is assumed to be a small ADD. However, dependencies between variables often caused additive value functions to merge into larger ADDs. Eventually, there exists an additive value function f encompassing

all state variables that blows up the ADD size. As a result, we lost the efficiency of concurrent factored value iteration.

Projection allows us to regain the efficiency by projecting f into an *additive* list of *weighted* basis functions where each basis function is essentially a *small* ADD consisting of a subset of variables. Figure 5.1 highlights the process. Formally, projection projects an ADD denoted as $y(\vec{\mathbf{x}})$ to $\hat{y}(\vec{\mathbf{x}}) = \sum_{i=1}^{k} w_i \phi_i$ where w_i is the weight of the basis function ϕ_i. Note that $w_i \phi_i$ is basically a scalar multiplication operation of constant w_i on basis function ϕ_i. Adhering to common practice, we set $\phi_k = 1$ so $w_k \phi_k$ is simply a constant. Moreover, we stress that $\hat{y}(\vec{\mathbf{x}})$ retained its additive form for additive structure exploitation.

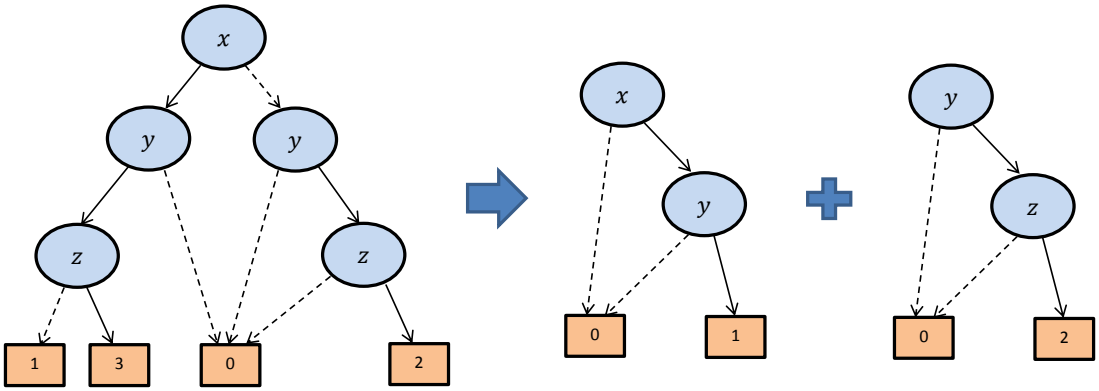


Figure 5.1: Projection of the large ADD on the left to the two *weighted* basis functions on the right.

Converged ADDs are projected after all action variables have been maxed out and before proceeding to the next time step of value iteration. For instance, assume that $V^{t+1} = f_1 + f_2 + f_3$ before projection and f_3 is the converged ADD. If f_3 is projected into $g_1 + g_2 + g_3$, then $V^{t+1} = f_1 + f_2 + g_1 + g_2 + g_3$.

5.2 Basis Functions

Given $y(\vec{\mathbf{x}})$ and a list of basis functions, a projection algorithm has to determine the weight for each basis function. Calculating weights are pretty straightforward but deciding which basis functions to use is more like an art than a science. There are many good heuristics for selecting basis functions; our method uses the dependency information in DBN. Notice that the next state variable x_i' in Figure 5.2 is conditioned on $Parents(x_i')$. Constructing basis functions that contain x_i' and $Parents(x_i')$ would allow us to capture these conditional relationships, thus, producing decent basis functions.

From analysing dependencies between variables, we could create different kinds of basis functions. For example, *singleton* basis functions consisting of only one variable or *pairwise* basis functions that consist of two variables in each basis function or higher order basis functions. Depending on implementation and the point where ADDs are projected, one could even include action variables in basis functions. Since

we project ADDs after maxing out all action variables, our basis functions would only contain *current state variables*.

In this thesis, we use pairwise basis functions of the form $\alpha\beta$ where α is the unprimed form of x_i', $\beta \in Parents(x_i')$ and $\alpha \neq \beta$. Figure 5.3 shows an example of the pairwise basis functions. Note that each basis function only has 4 nodes. Two projection algorithms were implemented; greedy linear regression and linear programming.

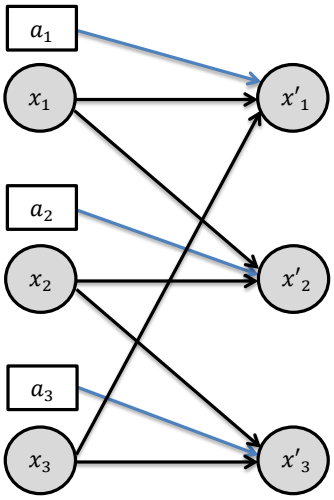


Figure 5.2: DBN from the previous chapter. Heuristic for basis functions by capturing conditional relationships between x_i' and $Parents(x_i')$.

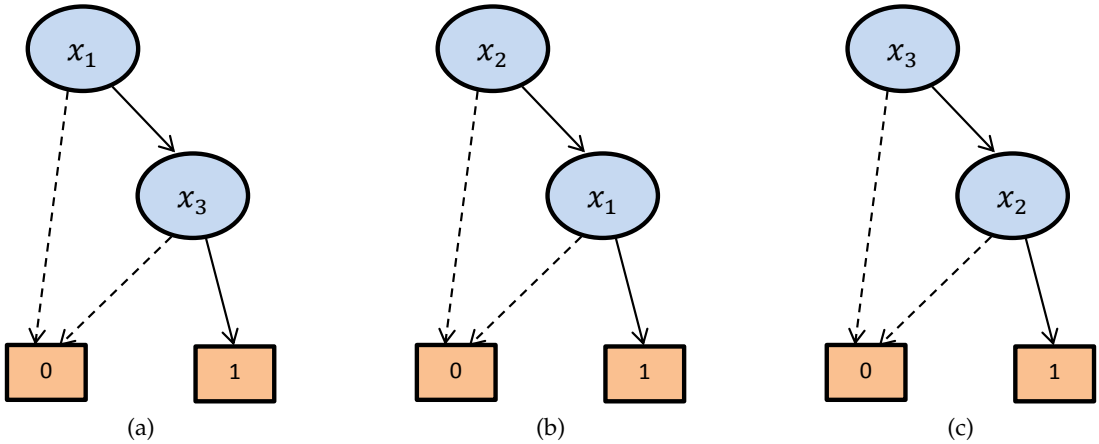


Figure 5.3: Pairwise basis functions: (a) x_1x_3; (b) x_2x_1; and (c) x_3x_2.

5.3 Greedy Linear Regression

Greedy Linear Regression is a *greedy* method based on *simple linear regression*. Simple linear regression computes the parameter α and β such that the following straight line equation

$$\hat{y} = \alpha + \beta x$$

would provide a *best fit* for some given data points $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ via the least squares approach. This method finds the line that minimizes the sum of squared residuals of the linear regression model [Kenney and Keeping 1962]

$$\begin{aligned} E &= \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \\ &= \sum_{i=1}^{n} [y_i - (\alpha + \beta x_i)]^2 \end{aligned} \tag{5.1}$$

To minimize Equation 5.1, take the partial derivatives of E and set them to 0

$$\frac{\partial E}{\partial \alpha} = -2 \sum_{i=1}^{n} [y_i - (\alpha + \beta x_i)] = 0 \tag{5.2}$$

$$\frac{\partial E}{\partial \beta} = -2 \sum_{i=1}^{n} [y_i - (\alpha + \beta x_i)] \, x_i = 0 \tag{5.3}$$

From Equation 5.2, the value for α is

$$\begin{aligned} -2 \sum_{i=1}^{n} [y_i - (\alpha + \beta x_i)] &= 0 \\ \sum_{i=1}^{n} (\alpha + \beta x_i) &= \sum_{i=1}^{n} y_i \\ n\alpha + \beta \sum_{i=1}^{n} x_i &= \sum_{i=1}^{n} y_i \\ \alpha &= \frac{1}{n} \sum_{i=1}^{n} y_i - \frac{\beta}{n} \sum_{i=1}^{n} x_i \\ &= \bar{y} - \beta \bar{x} \end{aligned} \tag{5.4}$$

where a horizontal bar over a variable means the sample average for that variable e.g. $\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$.

From Equation 5.3 and substitution of α according to Equation 5.4, β is calculated by

$$
\begin{aligned}
-2\sum_{i=1}^{n}\left[y_i-(\alpha+\beta x_i)\right]x_i &= 0\\
\sum_{i=1}^{n}x_iy_i-\sum_{i=1}^{n}(\alpha+\beta x_i)\,x_i &= 0\\
\alpha\sum_{i=1}^{n}x_i+\beta\sum_{i=1}^{n}x_i^2 &= \sum_{i=1}^{n}x_iy_i\\
\left(\frac{1}{n}\sum_{i=1}^{n}y_i-\frac{\beta}{n}\sum_{i=1}^{n}x_i\right)\sum_{i=1}^{n}x_i+\beta\sum_{i=1}^{n}x_i^2 &= \sum_{i=1}^{n}x_iy_i\\
\frac{1}{n}\sum_{i=1}^{n}x_i\sum_{i=1}^{n}y_i-\frac{\beta}{n}\left(\sum_{i=1}^{n}x_i\right)^2+\beta\sum_{i=1}^{n}x_i^2 &= \sum_{i=1}^{n}x_iy_i\\
\beta\sum_{i=1}^{n}x_i^2-\frac{\beta}{n}\left(\sum_{i=1}^{n}x_i\right)^2 &= \sum_{i=1}^{n}x_iy_i-\frac{1}{n}\sum_{i=1}^{n}x_i\sum_{i=1}^{n}y_i\\
\beta &= \frac{\sum_{i=1}^{n}x_iy_i-\frac{1}{n}\sum_{i=1}^{n}x_i\sum_{i=1}^{n}y_i}{\sum_{i=1}^{n}x_i^2-\frac{1}{n}\left(\sum_{i=1}^{n}x_i\right)^2}\\
&= \frac{\frac{1}{n}\sum_{i=1}^{n}x_iy_i-\frac{1}{n^2}\sum_{i=1}^{n}x_i\sum_{i=1}^{n}y_i}{\frac{1}{n}\sum_{i=1}^{n}x_i^2-\frac{1}{n^2}\left(\sum_{i=1}^{n}x_i\right)^2}\\
&= \frac{\frac{1}{n}\sum_{i=1}^{n}x_iy_i-\left(\frac{1}{n}\sum_{i=1}^{n}x_i\right)\left(\frac{1}{n}\sum_{i=1}^{n}y_i\right)}{\frac{1}{n}\sum_{i=1}^{n}x_i^2-\left(\frac{1}{n}\sum_{i=1}^{n}x_i\right)^2}\\
&= \frac{\overline{xy}-\bar{x}\bar{y}}{\overline{x^2}-\bar{x}^2}\\
&= \frac{cov\,(x,y)}{\sigma_x^2}
\end{aligned}
\tag{5.5}
$$

where $cov(x,y)$ is the covariance between x and y data points while σ_x^2 is the variance of x data points.

Back to the main discussion of calculating weights for projection, greedy linear regression evaluates the weight for *every* basis function by solving Equation 5.5 in a slightly different manner. Being greedy, it computes weight of each basis function one at a time based on the current residual $y'(\vec{\mathbf{x}})$ and basis function ϕ_i. Data points for y as well as x are respectively terminal nodes of $y'(\vec{\mathbf{x}})$ and ϕ_i. Algorithm 5.1 summarizes the underlying implementation.

Step 2 of the algorithm determine the number of data points in $y(\vec{\mathbf{x}})$ which is later required in Step 3(d) - (e). Let l = number of state variables in $y(\vec{\mathbf{x}})$, then total data point is 2^l. It might not be intuitive to the reader that total data points depends on number of state variables in $y(\vec{\mathbf{x}})$. Perhaps, it would be better to think of it as a tabular representation like CPT or an *unreduced* ADD. In tabular representation, we require 2^l rows to list down *values* for all combination of state variables. Similarly in

an *unreduced* ADD, there will be 2^l leaves. That explains the fact that total data points is 2^l. The same argument applies to Step 3(a) but this time it depends on number of state variables in ϕ_i.

In Step 3(b) - (e), all data points are summed ($\sum \phi_i$ or $\sum y'(\vec{\mathbf{x}})$). This is analogous to performing *OpOut* on all state variables in ϕ_i or $y'(\vec{\mathbf{x}})$. Once all state variables have been summed out, the resulting ADD will only contain a *single* terminal node labelled with the summed value of all data points. To clarify, ϕ_i^2 is obtained by calling the *Apply* procedure to multiply ϕ_i by itself while $\phi_i \cdot y'(\vec{\mathbf{x}})$ is obtained by multiplying ϕ_i with $y'(\vec{\mathbf{x}})$. Since $\phi_i \cdot y'(\vec{\mathbf{x}})$ would contain all state variables in $y'(\vec{\mathbf{x}})$, Step 3(e) divides the summed value by n. Weight for ϕ_i is determined in Step 3(f) while the residual $y'(\vec{\mathbf{x}})$ is updated in Step 3(g).

The loop in Step 3 skipped ϕ_k, the basis function where $\phi_k = 1$, and compute w_k at Step 4 instead. Step 4 is comparable to Equation 5.4, computing α (which is essentially w_k) from the final residual $y'(\vec{\mathbf{x}})$. Finally, the algorithm returns with a list of *weighted* basis functions.

Algorithm 5.1 Greedy Linear Regression

GreedyLinearRegression $(y(\vec{\mathbf{x}}), \{\phi_1, \ldots, \phi_k\})$

1. $y'(\vec{\mathbf{x}}) \leftarrow y(\vec{\mathbf{x}})$
2. $n \leftarrow$ number of data points in $y(\vec{\mathbf{x}})$
3. **for** $i \leftarrow 1$ to $k-1$
   (a) $m \leftarrow$ number of data points in ϕ_i
   (b) $\bar{x} \leftarrow \frac{1}{m} \sum \phi_i$
   (c) $\overline{x^2} \leftarrow \frac{1}{m} \sum \phi_i^2$
   (d) $\bar{x}\bar{y} \leftarrow \frac{1}{m} \sum \phi_i \cdot \frac{1}{n} \sum y'(\vec{\mathbf{x}})$
   (e) $\overline{xy} \leftarrow \frac{1}{n} \sum (\phi_i \cdot y'(\vec{\mathbf{x}}))$
   (f) *// weight for basis function ϕ_i (Equation 5.5)*
   $w_i = (\overline{xy} - \bar{x}\bar{y})/(\overline{x^2} - \bar{x}^2)$
   (g) *// update residual*
   $y'(\vec{\mathbf{x}}) \leftarrow y'(\vec{\mathbf{x}}) - w_i \cdot \phi_i$
4. *// calculate the weight for constant basis function $\phi_k = 1$ (Equation 5.4)*
   $w_k = \frac{1}{n} \sum y'(\vec{\mathbf{x}})$
5. *// returns weighted basis functions*
   return $\{w_1\phi_1, \ldots, w_k\phi_k\}$

5.4 Linear Programming

Linear programming (LP) is a problem of determining an optimal solution for a mathematical model specified by some linear constraints. Formally, linear programming is an optimisation technique for maximising or minimising a linear objective function subject to linear and non-negativity constraints over a convex polyhedron. LP algorithms like *simplex method* [Wood and Dantzig 1949; Dantzig 1949] finds a point in the polyhedron which produces the smallest (minimisation) or largest (maximisation) value for the objective function if such a point exists.

Our objective is to *minimise* projection error where error is defined to be max norm error $e = max_{\vec{\mathbf{x}}} |y(\vec{\mathbf{x}}) - \hat{y}(\vec{\mathbf{x}})|$. In simple terms, e is the largest absolute value among terminal nodes evaluated from the difference between $y(\vec{\mathbf{x}})$ and $\hat{y}(\vec{\mathbf{x}}) = \sum_{i=1}^{k} w_i \phi_i$. Formally, the objective is

$$min_{\vec{\mathbf{w}}} max_{\vec{\mathbf{x}}} \left| y(\vec{\mathbf{x}}) - \sum_{i=1}^{k} w_i \phi_i \right| \tag{5.6}$$

where $\vec{\mathbf{w}}$ is defined to be the weight vector for basis functions. Let ϵ be the LP variable expressing max norm error, then our objective is the same as

$$min_{\vec{\mathbf{w}}} \epsilon \qquad \textit{subject to} \qquad \epsilon \geq max_{\vec{\mathbf{x}}} \left| y(\vec{\mathbf{x}}) - \sum_{i=1}^{k} w_i \phi_i \right|$$

Nonetheless, LP does not allow non-linear constraints such as *max* and *absolute*. Recall that $max_{\vec{\mathbf{x}}}$ examines *all* assignments of state variables to find the largest value. Hence, we could replace $max_{\vec{\mathbf{x}}}$ by $\forall x_i$

$$min_{\vec{\mathbf{w}}} \epsilon \qquad \textit{subject to} \qquad \epsilon \geq \left| y(\vec{\mathbf{x}}) - \sum_{i=1}^{k} w_i \phi_i \right| ; \forall x_i$$

Treatment for *absolute* is simple, replace *absolute* with two *equivalent* inequalities as below

$$\epsilon \geq y(\vec{\mathbf{x}}) - \sum_{i=1}^{k} w_i \phi_i ; \forall x_i \tag{5.7}$$

$$\epsilon \geq \sum_{i=1}^{k} w_i \phi_i - y(\vec{\mathbf{x}}) ; \forall x_i \tag{5.8}$$

The linear programming problem could now be defined in its *standard form* given below

$$\begin{aligned} \textit{Variables}: &\quad \epsilon, w_1, \ldots, w_k \\ \textit{Minimise}: &\quad \epsilon = 1\epsilon + 0w_1 + \ldots + 0w_k = \vec{\mathbf{c}}^T\vec{\mathbf{v}} \\ \textit{Subject to}: &\quad \epsilon \geq y(\vec{\mathbf{x}}) - \sum_{i=1}^{k} w_i\phi_i; \forall x_i \\ &\quad \epsilon \geq \sum_{i=1}^{k} w_i\phi_i - y(\vec{\mathbf{x}}); \forall x_i \end{aligned}$$

Now, we could pass the standard form above to a LP solver and the solver would return values of w_i which minimises ϵ. Unfortunately, the conventional method for solving linear programming would require us to enumerate all state assignments for each inequalities in Equation 5.7 and 5.8 due to $\forall x_i$. With n state variables, we would end up with 2^n constraints for each inequality, thereby, totalling to $2 \cdot 2^n$ exponential number of constraints. Certainly, this is not computationally practical.

In fact, within these exponentially many constraints, most constraints are found to be redundant and do not play any significant role in computing the optimal solution. Only a small subset of carefully chosen constraints is sufficient in determining the final solution.

Constraint generation [Schuurmans and Patrascu 2001; Trick and Zin 1997] is a technique for reducing the exponential constraints to a reasonable number. Initialised with a small set of constraints, constraint generation generates new constraints only if the current solution from LP violates some constraints. Among the violated constraints, we find the *most violated constraint* (MVC) which subsumes all other violated constraints. The MVC is then added into the constraints set for LP to solve again. This process repeats itself until the LP solution satisfies all constraints.

Since we have two constraints in this problem, we will have two MVCs, Formally, let $\vec{\mathbf{x_1}}^* = \arg\max_{\vec{\mathbf{x}}} |y(\vec{\mathbf{x}}) - \hat{y}(\vec{\mathbf{x}})|$, $\vec{\mathbf{x_2}}^* = \arg\max_{\vec{\mathbf{x}}} |\hat{y}(\vec{\mathbf{x}}) - y(\vec{\mathbf{x}})|$ and if error $e > 0$, then the MVCs are defined as

$$\begin{aligned} \epsilon &\geq y(\vec{\mathbf{x_1}}^*) - \hat{y}(\vec{\mathbf{x_1}}^*, \vec{\mathbf{w}}) \\ \epsilon &\geq \hat{y}(\vec{\mathbf{x_2}}^*, \vec{\mathbf{w}}) - y(\vec{\mathbf{x_2}}^*) \end{aligned}$$

Figure 5.4 demonstrates the mechanism of constraint generation. Initially, there are five constraints in the constraints set (Figure 5.4a). The solution lies within the constraints set and is depicted by the direction of blue arrows. Since our aim is to minimise ϵ, solution from LP would be the lowest intersection point in the constraints set. This is shown by the blue dot in Figure 5.4b.

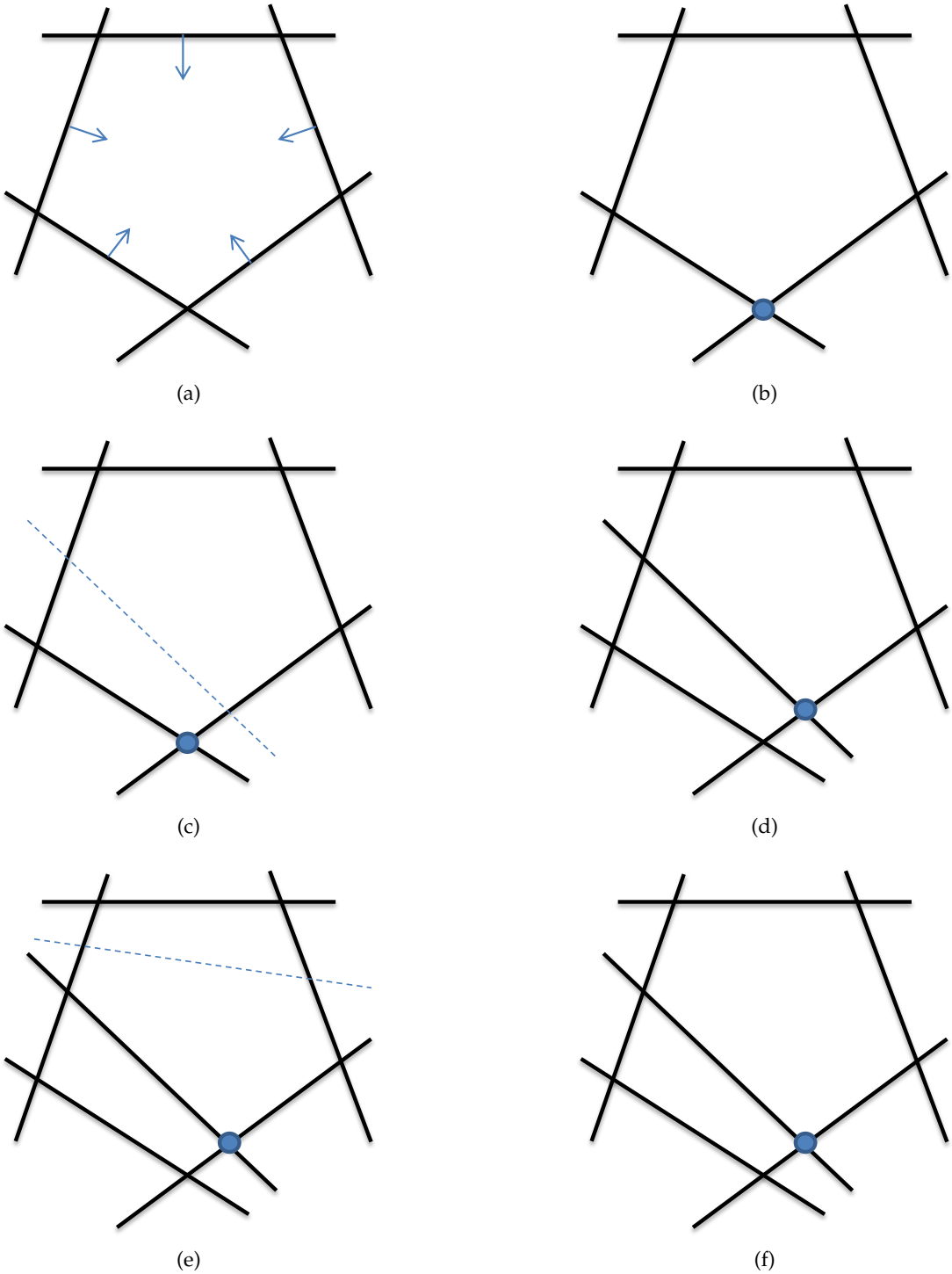


Figure 5.4: Constraint generation. Full details in text.

In each iteration of constraint generation, several constraints are generated as potential candidates to be added into the constraints set. From these candidates, the *best* constraint which subsumes every other generated constraint is selected. Note that this particular constraint (blue constraint) given in Figure 5.4c is *violated* by our current solution. Hence, this constraint is the MVC. On adding the MVC into the set and running LP again, the new solution is shown by the blue dot in Figure 5.4d.

In the next iteration of constraint generation, the *best* constraint is again identified. Observe from Figure 5.4e that the current solution satisfies the blue constraint. Therefore, the constraint is not added into the set. It is said that our current solution satisfies all constraints and constraint generation terminates with the current solution in Figure 5.4f.

Some minor alterations to the ADD data structure are necessary to facilitate the process of identifying MVC for constraint generation. It concerns decision nodes (non-terminal nodes) and records variables assignment leading to the maximum absolute value among terminal nodes. Two additional fields are added to the decision node data structure, namely $absMaxValue$ which documents the larger absolute value of the two children nodes, and $branch$ that records the decision node's variable assignment leading to $absMaxValue$.

ADD trees are usually constructed inductively starting from leaf nodes (bottom-up). Such bottom-up approach is extremely useful for bookkeeping purposes like recording $absMaxValue$. To demonstrate the process, study the ADD in Figure 5.5. Examine that the absolute value in the low child of node x_2 is larger than the high child. Accordingly, $absMaxValue$ and $branch$ are set to 7 and $false$. Likewise for node x_3, $absMaxValue = 9$ whereas $branch = true$ because the larger absolute value comes from the high child. On assigning values to node x_1, we compare values of $absMaxValue$ in nodes x_2 and x_3. Since node x_3 has a larger $absMaxValue$, node x_1's $absMaxValue$ is set to the value of node x_3 and $branch = false$. From the root node x_1, we just have to follow values in $branch$ to get the path leading to the largest absolute value as shown by blue edges in the figure. The tree path documents the assignment of variables.

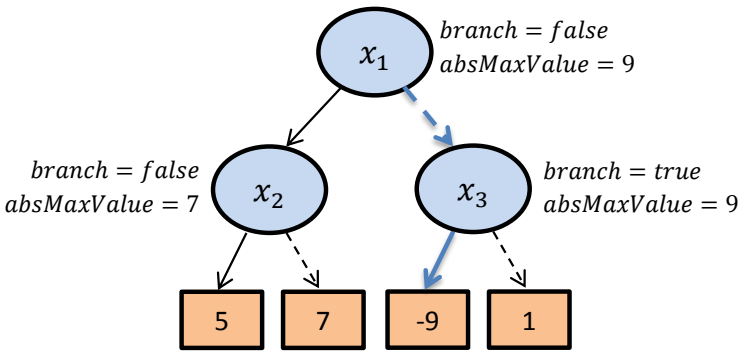


Figure 5.5: ADD with two extra fields, $absMaxValue$ and $branch$. Blue edges indicate the path from the root node to the largest absolute value.

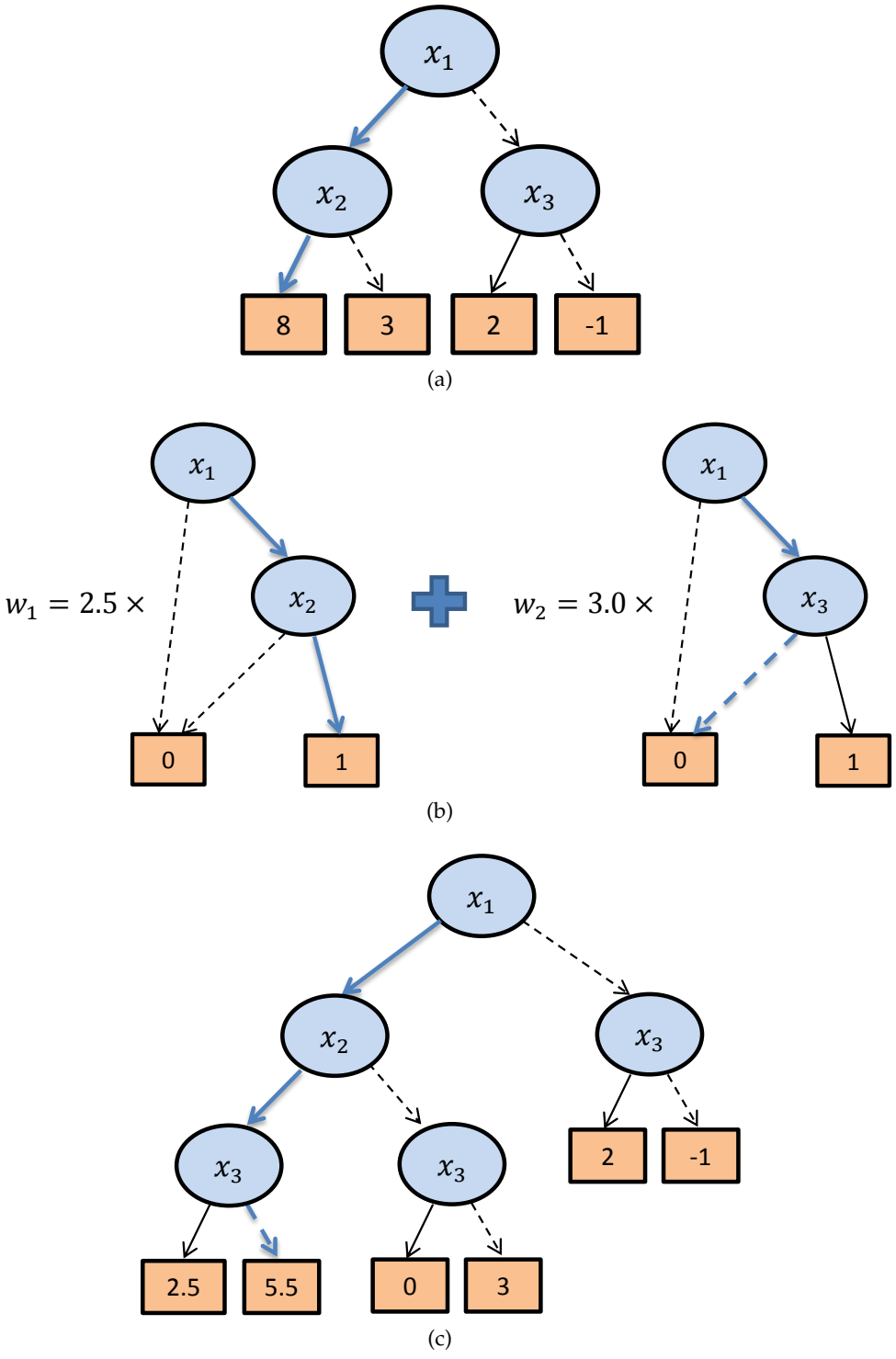


Figure 5.6: Identifying MVC: (a) ADD to be projected $y(\vec{\mathbf{x}})$; (b) weighted basis functions $\sum_{i=1}^{k} w_i \phi_i$ with current solution $w_1 = 2.5$ and $w_2 = 3.0$; and (c) difference of $y(\vec{\mathbf{x}}) - \sum_{i=1}^{k} w_i \phi_i$. Like before, the blue edges show the path of variables assignments leading to e. Refer to text for full details.

Having presented all the fine details, we could elaborate on an example of LP together with constraint generation. Suppose that Figure 5.6a is the $y(\vec{\mathbf{x}})$ ADD to be projected and the weighted basis functions $\sum_{i=1}^{k} w_i \phi_i$ are given in Figure 5.6b. Furthermore, assume that the current LP solution has computed $\epsilon = 2.2, w_1 = 2.5$ and $w_2 = 3.0$. Constraint generation would proceed to identify the *best* constraint based on the current computed weights.

First, we find the max norm error e according to the current LP solution by evaluating the difference between $y(\vec{\mathbf{x}})$ and $\sum_{i=1}^{k} w_i \phi_i$ given in Figure 5.6c. Max norm error e is simply the value of the root node's $absMaxValue = 5.5$ with variables assignments $x_1 = true, x_2 = true$ and $x_3 = false$. We note that variables assignments are vital in evaluating the coefficients of weights for MVCs.

If $\epsilon = e$, then our current solution has satisfy all constraint and we could terminate constraint generation. However, since $\epsilon = 2.2$, $e = 5.5$ and $\epsilon < e$, our current solution has violated some constraints; and constraint generation would proceed to generate the relevant MVCs. Recall that our LP problem is subjected to the following inequalities

$$\epsilon \geq y(\vec{\mathbf{x}}) - \sum_{i=1}^{k} w_i \phi_i; \forall x_i$$
$$\epsilon \geq \sum_{i=1}^{k} w_i \phi_i - y(\vec{\mathbf{x}}); \forall x_i$$

Under the variables assignments of $x_1 = true, x_2 = true$ and $x_3 = false$, $y(\vec{\mathbf{x}})$ evaluates to 8. Likewise, ϕ_1 and ϕ_2 evaluate to 1 and 0 respectively. One could infer that e is calculated from these values i.e. $y(\vec{\mathbf{x}}) - \sum_{i=1}^{k} w_i \phi_i = 8 - (2.5 \cdot 1 + 3.0 \cdot 0) = 5.5$ under the same variables assignments. Moreover, e is the same for both inequalities above since $absMaxValue$ records the *absolute* value. Substituting the coefficients of weights and $y(\vec{\mathbf{x}})$, the MVCs are

$$\epsilon \geq 8 - (w_1 \cdot 1 + w_2 \cdot 0)$$
$$\epsilon \geq (w_1 \cdot 1 + w_2 \cdot 0) - 8$$

with some simplification and reordering

$$\epsilon + w_1 \geq 8$$
$$\epsilon - w_1 \geq -8$$

The simplified MVCs are then added into the constraints set for LP to resolve. Every iteration of constraint generation adds two MVCs into the set, one from each inequality.

Algorithm 5.2 outlines the entire process. Initialisation of LP solver [1] occurs at Step 1 by defining the LP standard form while Step 2 initialises all weights w_i to 1.0. Con-

[1] LP was implemented with the help of the *lpsolve* library.
Available from http://lpsolve.sourceforge.net/5.5/distribution.htm.

straint generation starts from Step 3 where max norm error e is computed in Step 3(a), MVCs are added into the set in Step 3(b) with the solution for the current constraint set is determined and returned by the LP solver in Step 3(c).

There are *three* termination conditions in our implementation, fulfilment of any of these three conditions would terminate constraint generation.

- First condition is the typical stopping criteria for constraint generation in regards to the current solution satisfying all constraints in the set.
- Second condition puts a limit on the number of constraints in the set, directly affecting number of iterations to prevent constraint generation from running for a long period of time.
- Lastly, our third condition accepts the current solution once the max norm error falls within the error threshold of $\delta \cdot t \cdot (MAX\_REW - MIN\_REW)$ where δ is between 0 to 1, t is the current time step, MAX_REW and MIN_REW is maximum or minimum reward an agent could obtain from the reward function. Maximum or minimum reward is extracted by maxing out or minimising out all variables in the reward function. Parameter δ is the control factor for final projection error and processing time. Reducing error comes at the cost of time and vice versa for the reason that smaller δ would require more constraints as well as iterations.

We note that the computational cost of Step 3(a) - (c) is not cheap and it is desirable to minimise number of iterations. In particular, we shall see that Step 3(a) is the main bottleneck for scalability.

Algorithm 5.2 Linear Programming

LinearProgramming $(y(\vec{\mathbf{x}}), \{\phi_1, \ldots, \phi_k\})$

1. Initialise LP.
2. **for** $i \leftarrow 1$ to k
   (a) $w_i \leftarrow 1.0$
3. *// constraint generation*
   while termination condition has not been met
   (a) *// evaluate max norm error e*
   $e \leftarrow max_{\vec{\mathbf{x}}} \left| y(\vec{\mathbf{x}}) - \sum_{i=1}^{k} w_i \phi_i \right|$
   (b) Determine MVCs and add them into the constraints set.
   (c) $(\epsilon, w_1, \ldots, w_k) \leftarrow$ solve LP
4. *// returns weighted basis functions*
   return $\{w_1\phi_1, \ldots, w_k\phi_k\}$

5.5 Summary

At the beginning of this chapter, we stressed the importance of additive projection for scalability. Additive projection was chosen because concurrent factored value iteration is very efficient in exploiting additive structure. Then, we provided a method to identify decent basis functions by analysing dependencies or the causal relationships between variables in the DBN. After that, we introduced the two projection algorithms implemented in this thesis: greedy linear regression and linear programing. In the following chapter, we present the empirical results.

Chapter 6

Empirical Results

There is no result in nature without a cause;
understand the cause and you will have no need of the experiment.
Leonardo da Vinci

In this chapter, we present the empirical results to investigate the following questions

1. How factored max in concurrent factored value iteration helps in solving concurrent problems? How does problem size affect processing time and space? Why is there a need for additive projection?
2. Between greedy linear regression and LP, which projection algorithm is better?
3. How much performance improvement we could gain from using projection on some highly concurrent problems?

Problem domains and instances are written in *RDDL* [Sanner 2010]. Experimental results were obtained from a machine running on Ubuntu 11.10 with Intel ®Core™i7-930 2.80GHz and 6GB of main memory. For some results, both tables as well as graphs are provided for illustrative purposes, discussion of results and ease of identifying values.

6.1 Preliminary Investigations

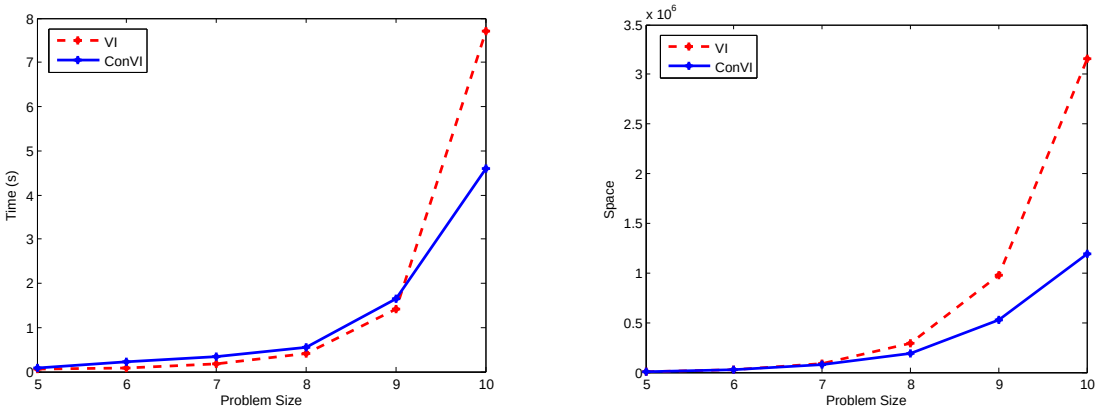
We begin the chapter by investigating how factored max in concurrent factored value iteration helps in solving concurrent problems. For simplicity, we will refer concurrent factored value iteration as *ConVI* and factored value iteration as *VI*. Experimental results were obtained by performing *one* time step of value iteration on a complete ADD tree in a unidirectional ring SysAdmin. *ConVI* is compared against the naïve approach of adding concurrency into *VI* via complete enumeration of all combination of actions. Recall that in SysAdmin, number of computers is equal to number of state variables. For a problem size of n, there are n computers, n state variables, n next state variables and n action variables. Results are shown in Table 6.1 and Figure 6.1. Note that ADDs are *not* projected in this section.

Initially, *VI* is relatively faster than *ConVI* because *ConVI* has to manipulate a larger ADD with more variables due to involvement of action variables. However, when the number of computers increased to 10, *ConVI* starts to outperform *VI*. It took *ConVI* 4.6*s* while *VI* took 7.7*s*. The cause for this is *VI* starts to pay a large computational cost for executing 2^n regressions from 2^n enumerated actions. Undoubtedly, *VI* with 2^n regressions is impractical for large n.

Nodes are saved in *hashtables* for constant time retrieval and simple referencing. Measurement for space is actually the size or number of entries in the *hashtable* used to facilitate *Apply* and *OpOut* operations. Such metric for space may not be exact but it serves as a decent estimation. Normally, this *hashtable* is cleared after each *Apply* or *OpOut* operation to free up memory space but in this experiment, it was not cleared to obtain the final space usage. With 2^n regressions and 2^n value functions from enumerated actions, *VI* requires significantly more space than *ConVI*. Moreover, space usage for *VI* increased in an exponential manner. Again, this is not feasible for large problem size. To conclude, *ConVI* is more efficient in both time and space. Hereafter, all discussions will be centred upon *ConVI*.

Problem Size	**5**	**6**	**7**	**8**	**9**	**10**
Nodes	63	127	255	511	1023	2047
Time (s)	0.05	0.09	0.17	0.42	1.42	7.70
	0.08	**0.23**	**0.34**	**0.56**	**1.64**	**4.60**
Space	8124	27187	88566	296674	973861	3153602
	7459	**24926**	**73097**	**188673**	**527683**	**1189164**

Table 6.1: *VI* and *ConVI*. Time and space performance for *VI* and *ConVI* (in **bold**). Problem size refers to number of state variables. Results were taken after *one* time step of value iteration on a complete ADD tree in unidirectional ring SysAdmin.



(a) (b)

Figure 6.1: *VI* and *ConVI*: (a) time; and (b) space. The results show that *ConVI* is more efficient in time and space compared to *VI*. This is attributed to factored max and exploitation of additive structure.

Before proceeding, let us examine the effect of incrementing problem size on time and space. First, we need ADD trees of different sizes but with equal amount of variables. Examine the two ADDs in Figure 6.2, they are constructed on top of a complete ADD tree represented by a triangle. Suppose that there are n variables and $k = 1$ (Figure 6.2a), then the triangle represents a complete tree with $n - 1$ variables. The solid edge of the n-th variable will point to the root of the complete tree while the dash edge points towards the terminal node labelled by 0 which is part of the complete tree. Likewise, $k = 2$ in Figure 6.2b is constructed in a similar manner except that there are $n - 2$ variables in the complete tree and two outer nodes.

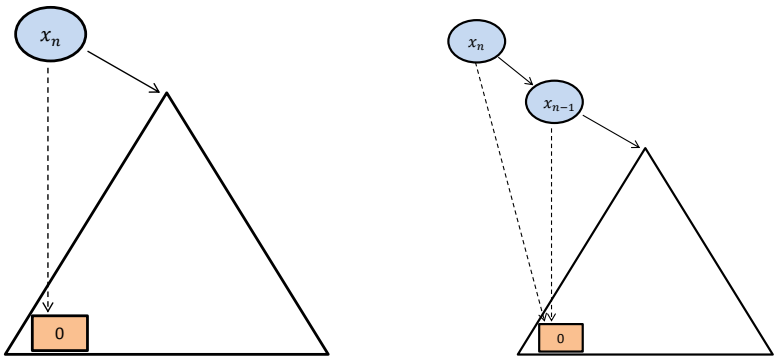


Figure 6.2: ADDs from Table 6.2: (a) k = 1; and (b) k = 2. Triangle represents a complete tree.

Problem Size	**8**		**9**		**10**	
	Time(s)	**Space**	**Time(s)**	**Space**	**Time(s)**	**Space**
k = 3	(66) 0.58	152095	(130) 1.77	470703	(258) 4.62	1259735
k = 2	(129) 0.64	203352	(257) 2.12	606219	(513) 7.14	1522795
k = 1	(256) 0.81	212399	(512) 2.02	608912	(1024) 5.34	1447129
Complete	(511) 0.56	188673	(1023) 1.64	527683	(2047) 4.60	1189164

Table 6.2: Effects of problem size on time and space. The top row refers to problems size while leftmost column refers to ADD type. Given a problem size and type of ADD tree, number of nodes in the ADD is written within the brackets so that all values fit into a single table. ADD type *Complete* refers to a complete ADD without any outer nodes.

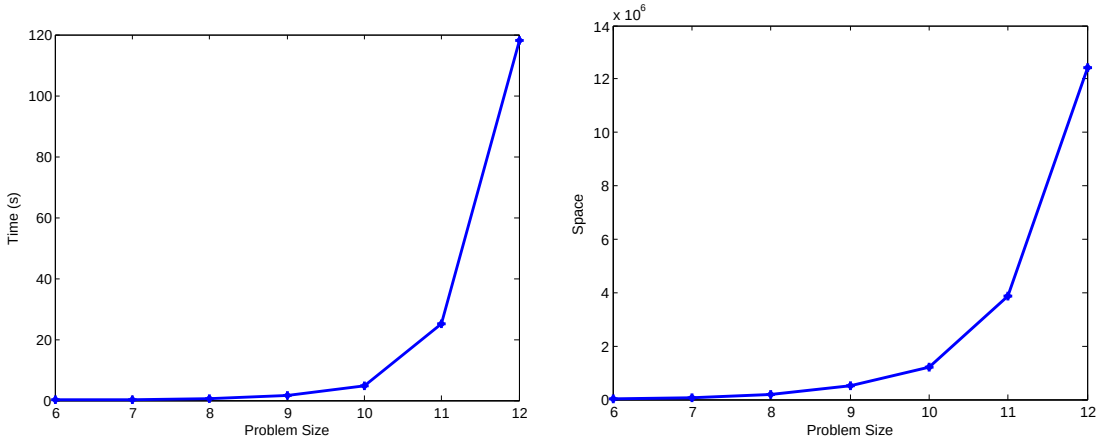
Table 6.2 tabulates the results for performing one iteration of *ConVI* in unidirectional ring SYSADMIN on different ADD types. Reading Table 6.2 from top to bottom, we observe that *ConVI* pays very little space/time for larger diagram with same number of variables but pays a large cost for introducing new variables when the problem size increases (reading from left to right). Consider the case where there are 500+ nodes, problem size 8 took 0.56s but when we increment the problem size to 9 and 10, time increased to 2.02s and 7.14s respectively despite the fact that number of nodes remained almost constant. The explanation is quite simple. Apart from introducing

more variables, a problem becomes more complex as a whole when we increase the problem size. There are more reward functions and CPTs to manipulate, not to mention that number of *OpOut* operations increases as well.

In Chapter 5, we mentioned about the importance and need to project a large ADD containing all state variables into an additive list of smaller ADDs. In the following experiment, we present results supporting this claim and stress again the significance of projection for scalability. Table 6.3 tabulates the results for performing one iteration of *ConVI* on a complete ADD tree in a unidirectional ring SYSADMIN. Figure 6.3 shows the plotted graphs.

Problem Size	Nodes	Time(s)	Space
6	127	0.23	24926
7	255	0.34	73097
8	511	0.56	188673
9	1023	1.64	527683
10	2047	4.60	1189164
11	4095	25.18	3872783
12	8192	118.00	12410384

Table 6.3: The need for projection. Time and space performance as problem size increases in SYSADMIN.



(a) (b)

Figure 6.3: Corresponding graphs illustrating results for: (a) time against problem size; and (b) space against problem size. Both time and space increase exponentially as problem size increases. Thus, it is essential to project ADDs to scale beyond the limits of exact inference.

According to the graphs, both time and space increase exponentially as problem size increases. Results are actually not that surprising taking into account that number of nodes doubles each time and complexity of *OpOut* and *Apply* are both $\mathcal{O}(n^2)$. Also, with n computers, we have to max out n action variables and sum out n next state

variables from all additive value functions. Even though these operations require quadratic time, the impact would eventually become severe and intractable.

For instance, with 12 machines, processing time increased from 25*s* to 118*s* while space experienced similar exponential growth. It is rather disappointing that such a *small* problem already starts to display signs of scalability issues. The reason is due to the fact that *ConVI* actually has to manipulate $12 \cdot 3 = 36$ variables (current state, next state and action variables) and in the worst case, it may have to manipulate an ADD with $2^{36+1} - 1$ nodes. Therefore, it is essential to project ADDs with the intention of scaling beyond larger problems while maintaining feasible processing time and space.

6.2 Additive Projection

Next, we investigate the performance of the two projection algorithms implemented in this thesis, greedy linear regression (Greedy) and LP. Both algorithms are fundamentally different from each other with the former being simple to implement and understand while the latter requires a bit more thought as well as effort to formulate the LP standard form.

The experiment was carried out by assigning random weights to the pairwise basis functions, summing them together to get a large ADD and passing the ADD together with the un-weighted basis functions as arguments to the projection algorithm. The intention was to observe to what extent the projection algorithm was able to recover the weights. Suppose that there are 5 variables, basis functions in this experiment would be $x_1x_2, x_2x_3, x_3x_4, x_4x_5, x_5x_1$ (forming a ring) and constant basis function 1. The error metric is max norm error, a metric that is often used in the literature. For *LP*, $\delta = 0.1$ for error threshold in constraint generation. Table 6.4 summarises the result with the plotted graphs given in Figure 6.4.

Variables	Greedy	LP
12	0.22	0.14
13	0.27	0.20
14	0.54	0.33
15	0.70	0.63
16	1.59	1.05
17	4.10	2.03
18	11.15	3.11
19	20.75	4.94
20	52.08	11.48

(a)

Variables	Max Norm Error (%)
12	15.33 (16.85)
13	17.86 (17.00)
14	20.57 (17.14)
15	23.47 (17.26)
16	26.56 (17.36)
17	29.83 (17.45)
18	33.30 (17.52)
19	36.95 (17.59)
20	40.79 (17.66)

(b)

Variables	Constraints
12	26
13	28
14	30
15	32
16	34
17	36
18	38
19	40
20	42

(c)

Table 6.4: Greedy and LP: (a) time(s) for projection; (b) Greedy max norm error, % was calculated by dividing max norm error by the maximum absolute value in the original ADD. LP was able to recover close to exact weights resulting in *no error* (error was reported to be exceedingly small, less than 1×10^{-6}); and (c) LP constraints generated by constraint generation.

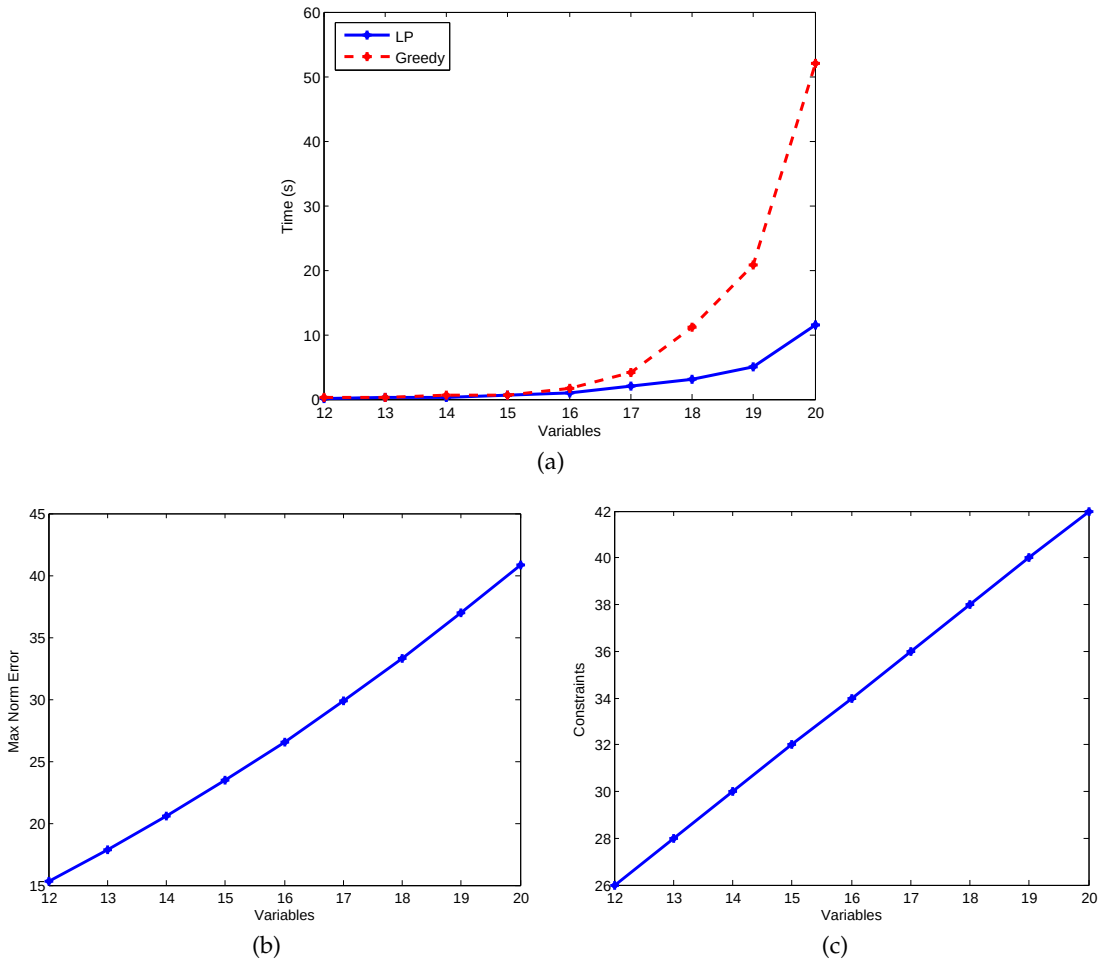


Figure 6.4: Greedy and LP: (a) time in seconds; (b) Greedy max norm error; and (c) generated LP constraints. From the results, the LP projection is optimal and faster compared to suboptimal Greedy. Moreover, LP only requires a few constraints to recover the weights.

With respect to time, LP clearly outperforms Greedy from the very beginning because of the large amount of *OpOut* operations in Greedy to determine a *single* weight. On top of that, LP scales better than Greedy and above all, LP was able to recover close to exact weights with error reported to be less than 1×10^{-6}. It even managed to recover *exact* weights when there were less than 15 variables. Greedy on the other hand, did not fare too well in error and has a linear relationship between error and number of variables. The same occurs to the error percentage.

Finally, the most astonishing results came from total number of generated constraints. Comparing this to the case without constraint generation where we require $2 \cdot 2^n$ constraints, only a few constraints are needed to recover the weights. Furthermore, an addition of 1 variable only necessitate 2 extra constraints. To sum up, the LP projection is optimal and faster compared to suboptimal Greedy. Therefore, we will be using LP for projection.

6.3 Concurrent Problems

This section presents experimental results of applying *ConVI* with LP projection on three concurrent problems. They are SYSADMIN, GAME OF LIFE and LOGISTICS. Domains and instances for SYSADMIN as well as GAME OF LIFE are from ICAPS 2011 IPPC [1]. LOGISTICS is a problem written by us and is included in the Appendix section. Since concurrency is one of the main discussions in this thesis, we have chosen problems which are *highly* concurrent. Similar to SYSADMIN, both GAME OF LIFE and LOGISTICS have number of state variables that is equal to number of action variables.

For all problems, horizon $h = 10$ and $\gamma = 1$. Maximum number of constraints was limited to 1000 while $\delta = 0.1$ *unless* stated otherwise. In discussing results, we use the term *Exact* to refer to *ConVI without* projection whereas *Projection* is *ConVI with* LP projection.

6.3.1 SYSADMIN

At present, unidirectional ring SYSADMIN is well understood by the reader and does not require any further explanations. Table 6.5 and Figure 6.5 illustrate the results.

Problem Size	Exact	Projection
6	0.52	0.23
7	1.30	0.25
8	4.17	0.35
9	18.05	0.58
10	69.67	0.92
11	308.60	1.66
12	10989.72	2.95

(a)

Problem Size	Exact	Projection
6	71	44
7	132	51
8	275	58
9	546	65
10	1103	72
11	2183	79
12	4378	86

(b)

Problem Size	Max Norm Error (%)
6	0.27 (0.50%)
7	0.24 (0.38%)
8	0.40 (0.54%)
9	0.38 (0.46%)
10	0.45 (0.50%)
11	0.44 (0.44%)
12	0.54 (0.50%)

(c)

Problem Size	Estimated Error
6	2.50
7	2.50
8	3.94
9	3.63
10	4.46
11	4.26
12	5.29

(d)

Table 6.5: SYSADMIN - Problem Size. Results taken from unidirectional ring SYSADMIN: (a) time(s) for *Exact* and *Projection*; (b) total sum of nodes in final additive value function (we remind the reader that the converged ADD containing *all* state variables is projected at the end of each time step, thus, the reported values for *Projection* is after projection); (c) max norm error (actual error) from projection; and (d) estimated error from projection (upper bound error).

[1] http://users.cecs.anu.edu.au/∼ssanner/IPPC_2011/index.html.

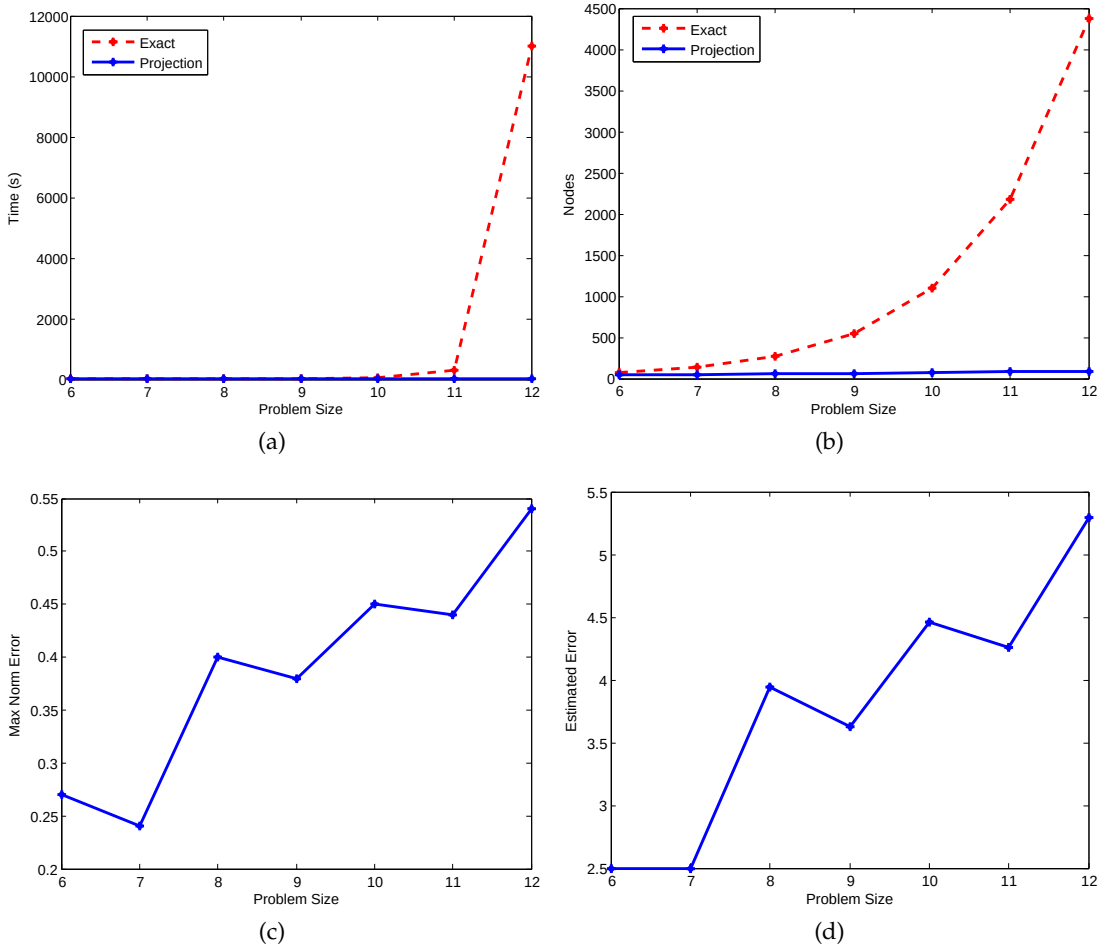


Figure 6.5: SYSADMIN - Problem Size. This shows that *Projection* significantly outperforms *Exact* in time and total nodes while incurring exceedingly small error.

First thing to notice is that *Projection* significantly outperforms *Exact* in both time and total sum of nodes in the final additive value function. At 12 machines, *Exact* took 10989 seconds or 3 hours in comparison to 2.95 seconds required by *Projection*. Such impressive results are attributed to LP for projecting ADDs into small basis functions consisting of 4 nodes each. Regressing on an ADD so small would need very little time. Total nodes in value function determines memory consumption and with so few nodes, *Projection* definitely consumes less memory.

The amazing results also apply to error. Error reported to be remarkably small with the highest error percentage of 0.54%. Part of the reason is because SYSADMIN, in particular, unidirectional ring has a lot of structures and pairwise basis functions were able to capture them. By trading off accuracy for time, we acquired enormous speedup in exchange for a small loss in accuracy.

On termination of each constraint generation, max norm error evaluated from the current solution for weights are summed together to obtain estimated error. Estimated error could be interpreted as an upper bound error from projection. There exists an

interesting interaction between actual error and estimated error from similarities in the two plotted graphs.

δ	Time(s)	Constraints	Estimated Error
0.01	52.92	48	5.86
0.05	52.58	34	6.88
0.10	53.88	34	6.88
0.15	51.39	34	6.88
0.20	53.25	34	6.88
0.25	52.21	34	6.88
0.30	52.79	34	6.88

Table 6.6: SysAdmin - δ. Results were taken from unidirectional ring SysAdmin with 16 computers. *Constraints* is the average amount of LP constraints generated from *each* projection. Note that every iteration of constraint generation adds two constraints. Hence, we are able to determine total iterations from number of constraints.

In Table 6.6, we experimented with different values of δ and found that results remained the same except for $\delta = 0.01$. During the final iteration of constraint generation, previous error was reported to be more than 1000 but on adding two final MVCs, error reduces to less than 1. Such a small error would fulfils any error threshold with $0.05 \leq \delta \leq 0.3$.

Problem Size	Time(s)
14	10.69
16	52.58
18	473.81
20	67863.22

Table 6.7: SysAdmin - Scalability.

Table 6.7 tests scalability of SysAdmin up to 20 computers. Time rose substantially from 18 computers (473 seconds) to 20 computers (67863 seconds or roughly 19 hours) with a couple of explanations. Firstly, problem size 20 has 2^{20} states which is about 1 million states. If one take into account that there are actually a total of 60 variables to manipulate, the problem is in fact quite large with huge ADDs. Recall that most ADD operations are $\mathcal{O}(n^2)$, the effect of manipulating large ADDs start to be a limiting factor for scalability. The second reason is due to memory wall and *JAVA garbage collector*. Since we are dealing with large ADDs, memory fills up pretty quickly and starts swapping to disk, an undesirable predicament known as memory thrashing that greatly hurts performance. Besides that, each time after an ADD operation, *hashtables* are cleared to free up memory space. *Garbage collectors* are invoked to reclaim these memory spaces while worsening performance. The bad news is this happens quite often and together with the first reason, producing the results as we see in Figure 6.7.

6.3.2 GAME OF LIFE

Our next domain is the well-known John H. Conway's GAME OF LIFE [Gardner 1970], a cellular automaton with a few simple rules governing the next state properties of a cell based on the state of its neighbouring cells. The rules are

1. *Survival*: Every living cell with 2 or 3 living neighbours survives for the next generation (next time step).
2. *Deaths*: A living cell with less than 2 living neighbours dies from isolation. Every cell with more than 3 living neighbours dies from overpopulation.
3. *Births*: Each non-living cell adjacent to exactly 3 living neighbours becomes alive in the next time step.

A cell is determined by its x and y coordinates. An agent could concurrently set a number of cells to alive while incurring action costs. Action outcomes and the rules above hold with some probability for stochastic properties. Reward is sum of living cells minus action costs. Like SYSADMIN, n cells would have n state variables and n action variables. Results shown in Table 6.8 and Figure 6.6.

Problem Size	Exact	Projection
6	0.38	0.45
8	2.13	1.52
10	9.88	2.23
12	331.19	26.14
14	3467.71	33.37

(a)

Problem Size	Exact	Projection
6	56	64
8	167	90
10	323	116
12	2826	154
14	2950	168

(b)

Problem Size	Max Norm Error (%)
6	10.51 (27.08)
8	13.48 (26.11)
10	16.15 (23.09)
12	10.35 (14.53)
14	17.94 (18.67)

(c)

Problem Size	Estimated Error
6	30.0
8	44.18
10	55.83
12	51.23
14	66.51

(d)

Table 6.8: GAME OF LIFE - Problem Size: (a) time (s); (b) sum of nodes in additive value function; (c) actual error from projection; and (d) estimated error from projection.

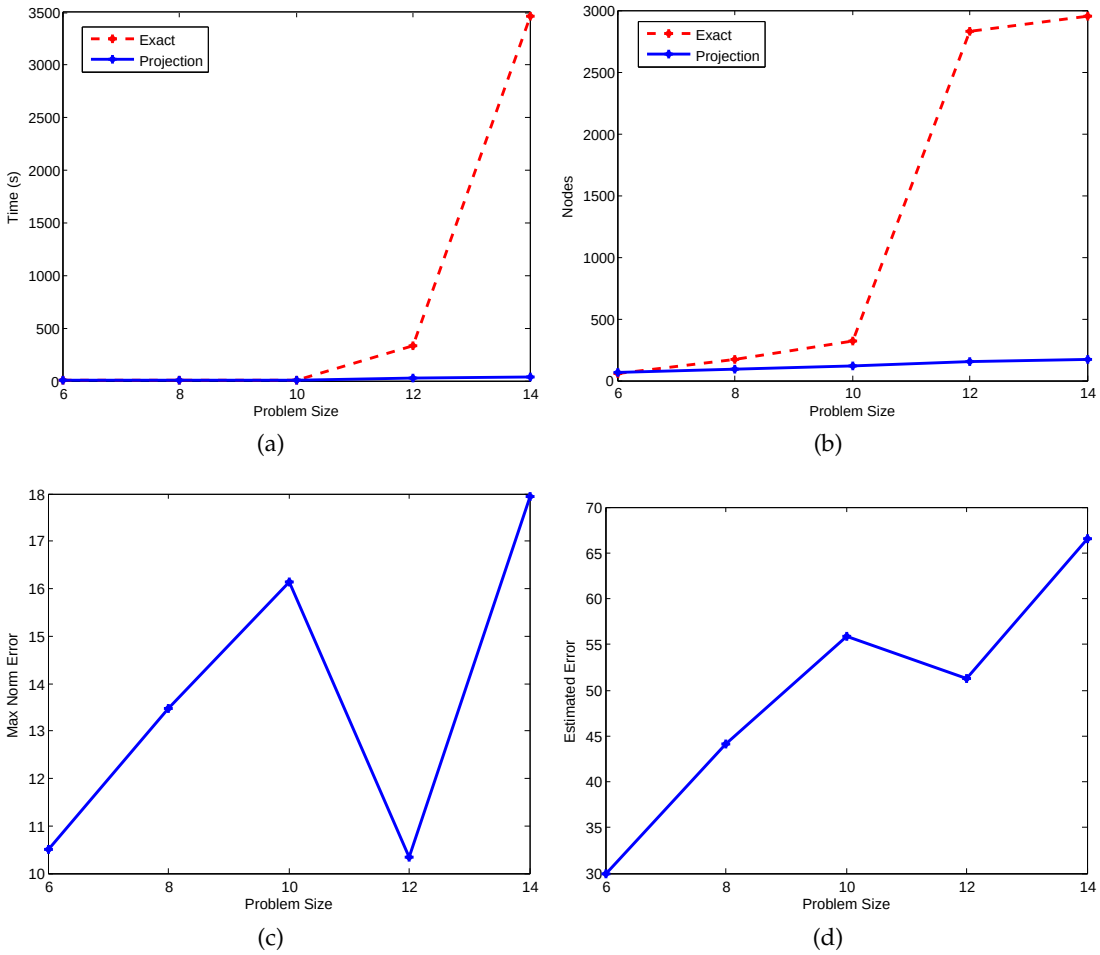


Figure 6.6: GAME OF LIFE - Problem Size. Again, *Projection* significantly outperforms *Exact* but error was reported to be quite large. This is because we need higher order basis functions to capture the more complex causal relationships between variables.

Judging from the fact that *Exact* was able to process up to 14 variables, GAME OF LIFE seems to be easier than SYSADMIN. This claim is further supported by the small number of nodes in GAME OF LIFE. Perhaps, problems size is a bit too small for GAME OF LIFE to be non-trivial. Although *Projection* outperforms *Exact*, actual error percentage (%) is quite large, suggesting that we might need higher order basis functions with more state variables. Alternatively, we could decrease our error threshold by reducing δ at the cost of time.

δ	Time(s)	Constraints	Estimated Error
0.05	6821.06	193	60.38
0.10	1035.41	107	78.48
0.15	438.43	90	79.23
0.20	729.48	86	90.19
0.25	956.28	86	90.19
0.30	379.49	86	90.19

Table 6.9: GAME OF LIFE - δ. Results were taken from GAME OF LIFE with 16 cells.

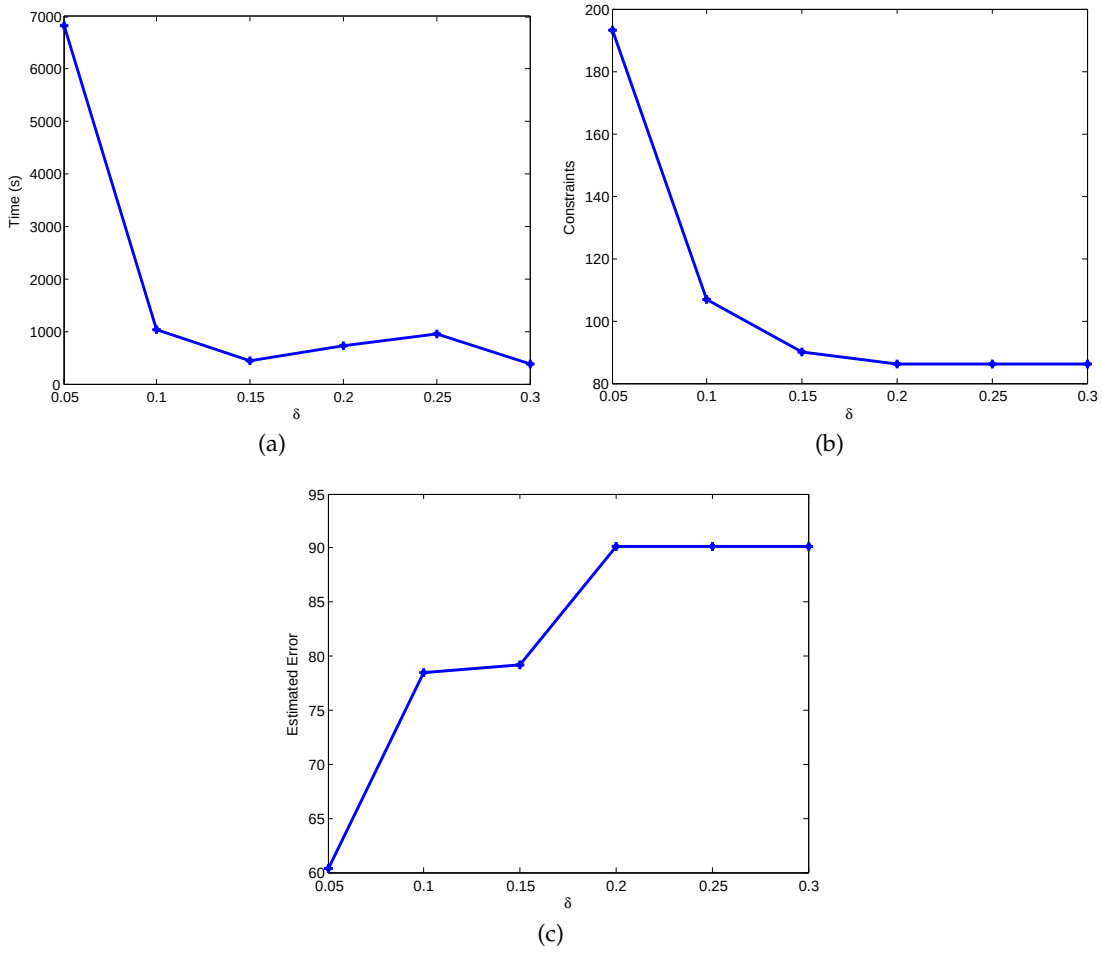


Figure 6.7: GAME OF LIFE - δ: (a) time (s); (b) average constraints; and (c) estimated error. As δ decreases, estimated error decreases as well.

Table 6.9 and Figure 6.7 experiment with different values of δ. As one would expect, reducing δ decreases estimated error at the cost of time due to the increased in the number of LP constraints to satisfy the smaller error threshold.

Variables	Time(s)
16	5556.63
18	60007.59

Table 6.10: GAME OF LIFE - Scalability.

Table 6.10 shows scalability of GAME OF LIFE. On average, GAME OF LIFE has at least 2 to 3 times more constraints than SYSADMIN with the same amount of variables. At 16 variables, an average of 107 constraints are needed for each projection while 18 variables requires about 119 constraints. An average constraints of 119 translates to roughly 60 iterations of constraint generation for each projection (recall that each iteration of constraint generation adds 2 constraints). In each iteration, we compute the difference between $y(\vec{\mathbf{x}})$ and $\hat{y}(\vec{\mathbf{x}})$ to identify the MVCs. Clearly, if each projection requires 60 iterations and if each time step of value iteration requires projection, memory space would fill up very quickly and we will be hitting the memory wall. Then, *garbage collectors* are invoked to free up memory space. Needless to say, scalability of GAME OF LIFE is limited by number of constraints.

6.3.3 LOGISTICS

The final concurrent domain we introduce is motivated by real world logistics planning problem. In LOGISTICS, there are several cities connected in a directed graph. Each city would independently generate a *job* based on some probability distribution at every time step. A job could only be serviced if there is a truck in the city and each job which is not serviced would incur a cost. Moreover, a truck could only move to the next city only if the current city has a directed edge to the next city. At any point in time, there could be zero or more trucks on *field* circling around the cities. The state of the problem is described by the location of all jobs and the location of all trucks that are currently on field. A pictorial depiction of a domain instance is given in Figure 6.8.

Similar to how trucks would incur fuel costs in the real problem, a truck would incur a *field-penalty* cost each time it stays on field. Hence, the *direction* action would decide whether is it better to call a truck back or to have the truck continue its journey. There is also a *dispatch* action which dispatch a truck to a specified city at a cost. The reward at each time step is total cost, hence, the goal of an agent is to minimise total cost.

In the previous two domains, number of state variables is the same as number of computers or number of cells. LOGISTICS is quite different from the other two domains in this aspect. For LOGISTICS, when total cities is incremented by 1, both state and action variables increased by 2. This is because we have two types of state variables (*job* and *field*) and two types of action variables i.e. *dispatch* and *direction*. Like before, we first show the results for both *Exact* and *Projection* in Table 6.11 and Figure 6.9.

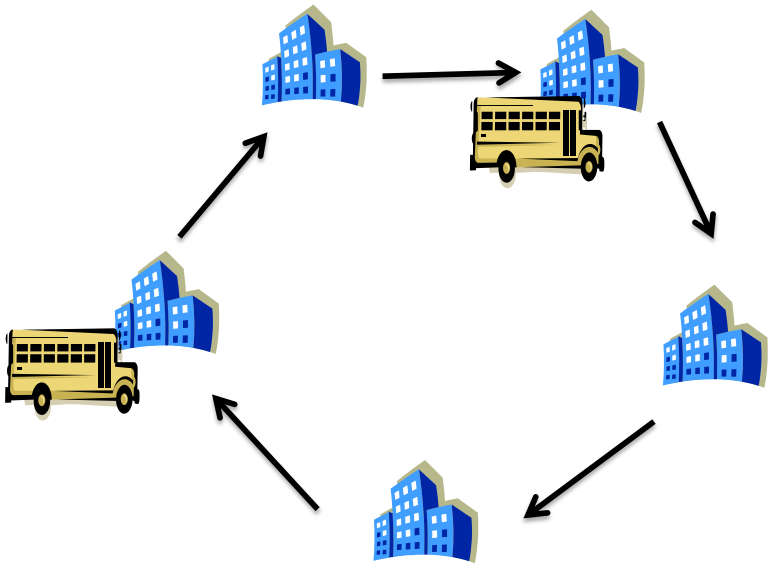


Figure 6.8: An example of LOGISTICS. Cities are connected in a unidirectional ring and trucks may drive along directed edges to move to the next connected city. Each city would generate a job based on some probability distribution.

Problem Size	Exact	Projection
6	0.27	0.17
8	0.73	0.30
10	3.37	0.62
12	22.14	1.41
14	263.54	4.15

(a)

Problem Size	Exact	Projection
6	95	53
8	261	70
10	798	87
12	2728	104
14	10482	121

(b)

Problem Size	Max Norm Error (%)
6	0.68 (35.09)
8	0.80 (31.01)
10	0.77 (23.31)
12	1.04 (26.90)
14	1.18 (26.11)

(c)

Problem Size	Estimated Error
6	0.85
8	1.24
10	1.35
12	1.55
14	1.72

(d)

Table 6.11: LOGISTICS - Problem Size: (a) time (s); (b) sum of nodes in additive value function; (c) actual error from projection; and (d) estimated error from projection.

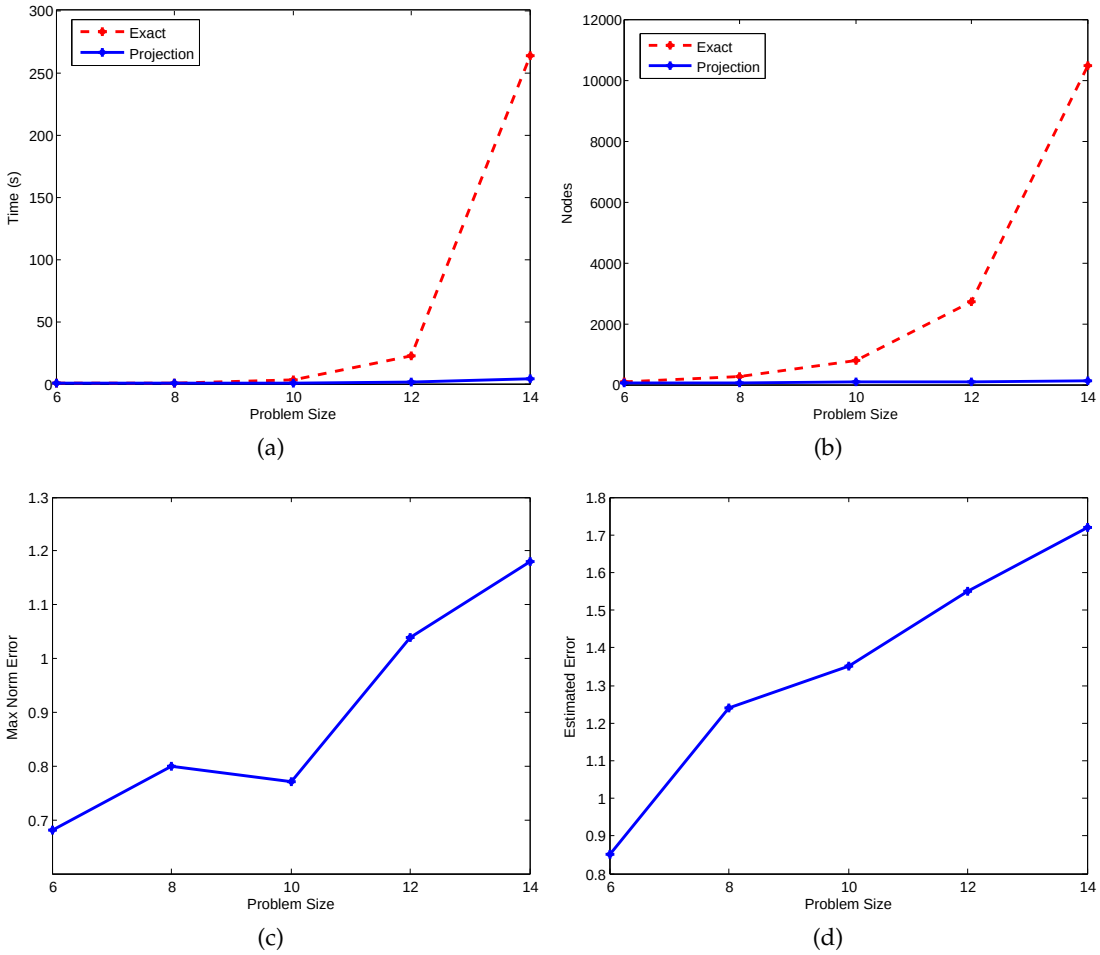


Figure 6.9: LOGISTICS - Problem Size. Like before, *Projection* outperforms *Exact* considerably.

Again, *Projection* significantly outperforms *Exact* in processing time and nodes for the same reasons stated before: regressing on a smaller ADD. Besides that, actual error percentage (%) is quite large probably due to our choice of pairwise basis functions.

δ	Time(s)	Constraints	Estimated Error
0.01	18.26	51	1.69
0.05	12.07	34	1.99
0.10	12.87	34	1.99
0.15	12.80	34	1.99
0.20	12.71	34	1.99
0.25	11.57	34	1.99
0.30	11.59	34	1.99

Table 6.12: LOGISTICS - δ. Results were taken from LOGISTICS with 16 state variables.

Referring to Table 6.12, experiments terminated under the same setting for $\delta = 0.05, \cdots, 0.30$. The cause for such an outcome is identical to SYSADMIN where error decreased from a very large value to a value less than 1. However, if δ is reduced to 0.01, we observed a decrease in estimated error in exchange for more time and constraints.

Problem Size	Time(s)
16	12.87
18	59.01
20	317.30
22	36218.09

Table 6.13: LOGISTICS - Scalability.

Table 6.13 shows scalability of LOGISTICS. Among the three domains, LOGISTICS has the most scalability probably due to the fact that it requires very few constraints and ADD size is relatively small. Nevertheless, as we scale to 22 state variables, memory wall issue, garbage collectors, and ADD operations of $\mathcal{O}(n^2)$ begins to limit scalability.

6.4 Discussion on Scalability

Towards the end of Chapter 5, we mentioned that the identification of MVCs is the *main* bottleneck for scalability. The careful reader might have already gained some insights from some of the discussions above. In identifying MVCs, $y(\vec{\mathbf{x}})$ is subtracted by $\hat{y}(\vec{\mathbf{x}}) = \sum_i w_i \phi_i$ to compute the difference. There are two ways to achieve this

1. Summing all basis functions in $\hat{y}(\vec{\mathbf{x}})$ to get a large ADD, then take the difference between $y(\vec{\mathbf{x}})$ and the large ADD.

2. Subtract $y(\vec{\mathbf{x}})$ with each basis function $w_i \phi_i$ in $\hat{y}(\vec{\mathbf{x}})$ *individually*.

Both ways have their own advantages and disadvantages. For method (1), the advantage is speed. Since basis functions are small, summing them will be very fast. The downside of this approach is high memory consumption because we would end up with 3 large ADDs ($y(\vec{\mathbf{x}})$, $\hat{y}(\vec{\mathbf{x}})$ and the difference ADD). Method (2) is the polar opposite of method (1), it consumes less space at the cost of time. Recall that time complexity of *Apply* is $\mathcal{O}(n^2)$ of the larger ADD. Suppose that $y(\vec{\mathbf{x}})$ is a huge ADD and there are many basis functions, then trying to subtract each basis function individually would be impractical. In our implementation, we have selected the first approach for speed but in return, running into memory wall issues. We note that the second approach also have the same complications but not as severe as the first method.

If constraint generation requires only 1 or 2 iterations, then we do not have to worry too much about memory wall issues but this is generally not the case. Even for SYSADMIN which has very few constraints, it requires at least 17 iterations for

problem size 16 (refer to Table 6.6) and this number increases as problem size gets larger. With larger problem size and more constraints, memory space fills up even faster causing considerable amount of memory thrashing and invoking of garbage collectors.

Although the complexity of *Apply* and *OpOut* is $\mathcal{O}(n^2)$, ADD size blows up very quickly as problem size increases. The impact on performance starts to be evident when there are more than 100000 nodes, an ADD size which is easily achievable by problem size 20 that has a total of 60 variables (20 state variables, 20 next state variables and 20 action variables). In the worst case, we could end up with an ADD encompassing all 60 variables.

While ADDs are meant to provide a space-efficient way to represent real-valued functions, this is only applicable if there are *few* distinct values or leaf nodes in the ADDs. Even if the problem itself has many structures to be exploited by ADDs, the influence of variables and projections tend to *eliminate* these structures over time. As a result, ADD size gets larger from having too many distinct leaf nodes and many variables become relevant in determining real values in leaf nodes. For example, consider two terminal nodes labelled by real values 25.6178 and 25.6177. These values are very close to each other but at the same time distinct from each other.

In effect, all factors above that deter scalability are somehow related to ADD size. By reducing the number of nodes in an ADD, we expect to see an improvement in running time. St-Aubin, Hoey, and Boutilier [2001] presented a way to merge leaf nodes which are *similar* in values like 25.6177 and 25.6178 while maintaining a predefined approximation error. Replacing similar values with a single approximate value leads to size reduction. To prevent confusion between this technique and projection, we refer this approximation technique as *pruning*. One could think of it as pruning unnecessary nodes to reduce the size of ADD.

	Without Prune	With Prune
Time (s)	67863.22	1742.19
Estimated Error	8.32	12.31
Total Pruning	-	13

(a) SYSADMIN

	Without Prune	With Prune
Time (s)	60007.58	38284.86
Estimated Error	84.19	88.26
Total Pruning	-	19

(b) GAME OF LIFE

	Without Prune	With Prune
Time (s)	36218.08	2554.64
Estimated Error	2.81	3.24
Total Pruning	-	9

(c) LOGISTICS

Table 6.14: *Projection* with pruning: (a) SYSADMIN problem size 20; (b) GAME OF LIFE problem size 18; and (c) LOGISTICS problem size 22. Results include time, estimated error from projection as well as total number of pruning.

Our final experiment in this thesis integrate pruning into *Projection*. Average approximation error from pruning is set to $0.01 * (MAX\_REW - MIN\_REW)$. During value iteration where action variables are maxed out from ADDs, ADDs are pruned before the max out operation. Recall that prior to maxing out an action variable, all ADDs containing the action variable are summed together, resulting in a large ADD. Thus, we prune an ADD before maxing out an action variable. Moreover, we prune an ADD before projection since MVCs identification is the main bottleneck. Only ADDs with more than 100000 nodes are pruned.

Results from Table 6.14 were obtained from three problems where *Projection* finds it hard to solve, one from each domain. The purpose of the experiment was to show that ADD size limits scalability and by reducing the size with pruning, we experience a significant performance improvement.

6.5 Summary

We began the chapter with some preliminary investigations on *VI* and *ConVI*. It was observed that *ConVI* is more efficient in terms of time and space. After that, we examined the performance of the two projection algorithms. LP projection with constraint generation is optimal and faster than Greedy. Then, we presented results for three concurrent problems: SysAdmin, Game of Life and Logistics where *Projection* significantly outperforms *Exact* by orders of magnitude in time and space. This was followed by a discussion on scalability.

Chapter 7

Conclusion

A conclusion is the place where you got tired thinking.
Martin H. Fischer

The main theme of this thesis is *concurrency* in highly concurrent decision-theoretic planning, a problem that has not been well-addressed in the literature. We have come a long way since we first review factored MDPs. Since then, we elaborated on the concept of action variables and the role they play in concurrent factored planning. Moreover, we presented concurrent factored value iteration as an efficient approach to maxing out action variables. For scalability, we emphasised the significance of projecting value functions into an additive form for additive structure exploitation. Then, we provided an automatic identification of basis functions from DBN. Finally, we showed some encouraging results for *ConVI* and *Projection*. For the rest of this chapter, we review the major contributions made this thesis, outline some interesting directions for future work and conclude with some final remarks.

7.1 Summary of Contributions

We have come back to how we started at the beginning of this thesis, let us review the major contributions made in this thesis.

- **Factored Action Value Iteration**
  Having identified that complete enumeration of actions in *VI* is infeasible for *highly* concurrent problems, we introduce *ConVI*. Borrowing the idea of factoring states into state variables in *VI*, actions are *factored* into action variables in *ConVI*. Efficiency in *ConVI* comes from applying *Regress* on each *additive* value function containing a subset of variables built on the intuition that $Regress\,(\sum_i f_i) = \sum_i Regress(f_i)$. If the value functions are in small additive form, then each *Regress* could be processed efficiently. The regressed value functions would be in a factored additive form that is later exploited by *factored max*. Empirical results suggest that *ConVI* outperforms *VI* in time and space when there are many concurrent actions. This is due to the (possibly exponential) reduction in complexity of the Bellman backup.

- **Additive Projection of Value Functions**
  Due to the undesirable consequence of factored max and dependency between variables in an underlying problem, compact structure is not preserved during value iteration. As a result, we lost the efficiency of *ConVI*. Thus, there arise a need to *project* ADDs into an additive form in order to regain the efficiency of *ConVI*. Two projection algorithm were presented and discussed, namely *greedy linear regression* and *linear programming*. LP was shown to be optimal and faster than Greedy.

- **Automatic Identification of Basis Functions**
  By analysing the structure of a DBN, we are able to identify casual relationships between variables. This information or knowledge of the underlying problem's structure allows us to automatically identify and construct decent basis functions.

7.2 Future Work

Here, we point out some of the interesting directions to take from here onwards.

- **Better Projections**
  With respect to projection, LP and Greedy are just some of the examples we could employ to project value functions into an additive form. An interesting research direction would be to explore novel approaches to projecting ADDs that is optimal and even more scalable.

- **More Compact Extensions of ADD**
  In this thesis, the main factor that limits scalability is ADD size. Due to the nature of ADDs, it is inevitable for the size to blow up as problem size increases. Sanner and McAllester [2005] presented an affine extension to ADDs known as *affine algebraic decision diagrams* (AADD) to address some of the shortcomings of the ADD representation. AADD is capable of exploiting additive, multiplicative and context-specific independence simultaneously in (concurrent) factored MDP. In their paper, they showed that AADD often yield an exponential to linear reduction in time and space over ADD. It would be interesting to observe how well AADD performs in *ConVI*.

- **Exploiting Initial State Knowledge and Reachability**
  Value iteration experiences the same drawback like many other conventional dynamic programming algorithms. It fully explores all possible states including states which are not reachable from the current state. Consequently, it has very limited utility for problems with large state spaces. That explains some of the scalability issues we encountered earlier. A better and more scalable approach is to implement online algorithm such as *real-time dynamic programming* (RTDP) [Barto et al. 1995] that only explores states which are reachable from the current state.

- **Hybrid Discrete and Continuous State**
  Throughout this thesis, we restrict attention to discrete state MDPs. In reality, most if not all real world applications are both *concurrent* and *continuous*. Recently, a paper by Sanner, Delgado, and de Barros [2011] proposed a method to compute optimal solutions for discrete and continuous state MDPs. A follow up to the paper extends the algorithm to continuous actions [Zamani et al. 2012]. By integrating with our work on concurrency, we should be able to solve problems that are highly concurrent and continuous.

7.3 Concluding remarks

In conclusion, we have presented a very efficient structured approach that computes exact dynamic programming solution for highly concurrent decision-theoretic planning with large action spaces. Our quest for speed and scalability has led us from exploitation of additive structure in value functions to projection and pruning. Given the performance improvements from projection and pruning, approximation approaches are essential to scale beyond the limits of exact inference. Finally, we believe that there are many rooms for improvement in concurrent factored planning and we hope that our work in this thesis would encourage further research in this area.

Appendix A

LOGISTICS

A.1 Domain

```
domain logistic_mdp {

 requirements = { reward-deterministic };

 types {

  // representing cities
  location : object;
 };

 pvariables {

  JOB-PROB         : { non-fluent, real, default = 0.4};
  JOB-PENALTY      : { non-fluent, real, default = -0.1};
  FIELD-PENALTY    : { non-fluent, real, default = -0.1};
  DISPATCH-PENALTY : { non-fluent, real, default = -0.5};

  // use to describe connected cities
  CONNECTED(location, location) : { non-fluent, bool, default = false};

  // states
  job(location)   : {state-fluent, bool, default = false};
  field(location) : {state-fluent, bool, default = false};


  // actions
  direction(location): {action-fluent, bool, default = false};
  dispatch(location) : {action-fluent, bool, default = false};

 };


 cpfs {

  // A job is serviced if there is a truck at the city
  // if there is no job, randomly generate a job
  job'(?l) = if(job(?l)) then KronDelta(~field(?l))
             else Bernoulli(JOB-PROB);
```

```
  // a city will have a truck if the truck was dispatched to the city
  // or the truck moved from its previous city to the current city
  field'(?l) = KronDelta(dispatch(?l) | exists_{?y: location}
                       [CONNECTED(?y,?l) ^ field(?y) ^ direction(?y)]);
 };

 // reward is total cost
 reward =  sum_{?l : location} [DISPATCH-PENALTY * dispatch(?l) +
             FIELD-PENALTY * direction(?l) + JOB-PENALTY * job(?l)];
}
```

A.2 Instance

```
non-fluents nf_logistic_inst_mdp__6 {
 domain = logistic_mdp;
 objects {
  location: {l1,l2,l3};
 };
 non-fluents {
  CONNECTED(l1,l2);
  CONNECTED(l2,l3);
  CONNECTED(l3,l1);
 };
}

instance logistic_6 {
 domain = logistic_mdp;
 non-fluents = nf_logistic_inst_mdp__6;
 init-state {
  job(l1);
 };

 max-nondef-actions = 1;
 horizon  = 10;
 discount = 1.0;
}
```

Bibliography

BACCHUS, F. AND GROVE, A. 1995. Graphical models for preference and utility. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence* (1995), pp. 3–10. Morgan Kaufmann Publishers Inc.

BAHAR, R., FROHM, E., GAONA, C., HACHTEL, G., MACII, E., PARDO, A., AND SOMENZI, F. 1997. Algebric decision diagrams and their applications. *Formal methods in system design 10*, 2, 171–206. (pp. 2, 8, 14)

BARTO, A., BRADTKE, S., AND SINGH, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence 72*, 1-2, 81–138. (p. 62)

BELLMAN, R. E. 1957. *Dynamic Programming*. Princeton University Press, Princeton. (pp. 3, 5, 8, 16)

BOUTILIER, C. 1997. Correlated action effects in decision theoretic regression. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence* (1997), pp. 30–37. Morgan Kaufmann Publishers Inc. (p. 8)

BOUTILIER, C., DEARDEN, R., AND GOLDSZMIDT, M. 1995. Exploiting structure in policy construction. In *International Joint Conference on Artificial Intelligence*, Volume 14 (1995), pp. 1104–1113. LAWRENCE ERLBAUM ASSOCIATES LTD. (pp. 2, 7)

BOUTILIER, C., FRIEDMAN, N., GOLDSZMIDT, M., AND KOLLER, D. 1996. Context-specific independence in bayesian networks. In *Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence* (1996), pp. 115–123. Morgan Kaufmann Publishers Inc. (p. 13)

BRYANT, R. 1986. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on 100*, 8, 677–691. (pp. 8, 14)

DANTZIG, G. 1949. Programming of interdependent activities: II mathematical model. *Econometrica, Journal of the Econometric Society*, 200–211.

DEAN, T. AND KANAZAWA, K. 1989. A model for reasoning about persistence and causation. *Computational intelligence 5*, 2, 142–150. (p. 7)

GARDNER, M. 1970. Mathematical games: The fantastic combinations of john conways new solitaire game life. *Scientific American 223*, 4, 120–123. (p. 51)

GUESTRIN, C., KOLLER, D., AND PARR, R. 2001. Multiagent planning with factored mdps. *Advances in neural information processing systems 14*, 1523–1530. (p. 24)

GUESTRIN, C., KOLLER, D., PARR, R., AND VENKATARAMAN, S. 2003. Efficient solution algorithms for factored mdps. *J. Artif. Intell. Res. (JAIR) 19*, 399–468. (p. 9)

HOEY, J., ST-AUBIN, R., HU, A., AND BOUTILIER, C. 1999. Spudd: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence* (1999), pp. 279–288. Morgan Kaufmann Publishers Inc. (pp. 2, 16)

HOWARD, R. AND MATHESON, J. 1984. Influence diagrams. *Readings on the Principles and Applications of Decision Analysis: Professional collection 2*, 719. (p. 12)

KEENEY, R. AND RAIFFA, H. 1993. *Decisions with multiple objectives: Preferences and value tradeoffs*. Cambridge Univ Pr. (p. 13)

KENNEY, J. F. AND KEEPING, E. S. 1962. Linear regression and correlation. In *Mathematics of statistics, Volume 2* (3rd ed.)., Chapter 15, pp. 252–285. Princeton, NJ: Van Nostrand. (p. 30)

LITTLE, I. AND THIEBAUX, S. 2006. Concurrent probabilistic planning in the graphplan framework. In *Proc. ICAPS*, Volume 6 (2006), pp. 263–272.

MAUSAM, M. AND WELD, D. 2004. Solving concurrent markov decision processes. In *Proceedings of the 19th national conference on Artifical intelligence* (2004), pp. 716–722. AAAI Press. (p. 24)

MEULEAU, N., HAUSKRECHT, M., KIM, K., PESHKIN, L., PACK KAELBLING, L., DEAN, T., AND BOUTILIER, C. 1998. Solving very large weakly coupled markov decision processes. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE* (1998), pp. 165–172. JOHN WILEY & SONS LTD.

PUTERMAN, M. 1994. *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons, Inc. (pp. 3, 5)

ROSSI, F., PETRIE, C., AND DHAR, V. 1990. On the equivalence of constraint satisfaction problems. In *Proceedings of the 9th European Conference on Artificial Intelligence* (1990), pp. 550–556. Citeseer. (p. 7)

SANNER, S. 2010. Relational dynamic influence diagram language (rddl): Language description. http://users.cecs.anu.edu.au/ ssanner/IPPC_2011/RDDL.pdf. (p. 41)

SANNER, S., DELGADO, K., AND DE BARROS, L. 2011. Symbolic dynamic programming for discrete and continuous state mdps. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI-11)* (2011). (p. 63)

SANNER, S. AND MCALLESTER, D. 2005. Affine algebraic decision diagrams (aadds) and their application to structured probabilistic inference. In *International Joint Conference on Artificial Intelligence*, Volume 19 (2005), pp. 1384. (p. 62)

SCHUURMANS, D. AND PATRASCU, R. 2001. Direct value-approximation for factored mdps. In *Advances in Neural Information Processing Systems (NIPS-14)* (2001), pp. 1579–1586. (p. 34)

SINGH, S., COHN, D., ET AL. 1998. How to dynamically merge markov decision processes. *Advances in neural information processing systems* 10, 1057–1063. (p. 25)

ST-AUBIN, R., HOEY, J., AND BOUTILIER, C. 2001. Apricodd: Approximate policy construction using decision diagrams. *Advances in Neural Information Processing Systems*, 1089–1096. (p. 58)

STERGIOU, K. AND WALSH, T. 1999. Encodings of non-binary constraint satisfaction problems. In *Proceedings of AAAI*, Volume 99 (1999), pp. 163–168. (p. 7)

TRICK, M. AND ZIN, S. 1997. Spline approximations to value functions. *Macroeconomic Dynamics 1*, 1, 255–277.

WOOD, M. AND DANTZIG, G. 1949. Programming of interdependent activities: I general discussion. *Econometrica, Journal of the Econometric Society*, 193–199. (p. 33)

YOUNES, H. AND SIMMONS, R. 2004. Policy generation for continuous-time stochastic domains with concurrency. In *ICAPS04*, Volume 325 (2004). (p. 24)

ZAMANI, Z., SANNER, S., AND FANG, C. 2012. Symbolic dynamic programming for continuous state and action mdps. In *Proceedings of the 26th Conference on Artificial Intelligence (AAAI-12), to appear* (2012). (p. 63)