An Object Oriented Framework for Modeling MDPs with Events

Thesis
presented to
Departamento de Ingeniería Industrial

by

Andrés Sarmiento Plata

Advisor: Germán Riaño, Ph.D.

For title of
Magíster en Ingeniería - Ingeniería Industrial

Ingeniería Industrial
Universidad de Los Andes
December, 2005

An Object Oriented Framework for Modeling MDPs with Events

Approved by:

____

Germán Riaño, Ph.D., Advisor

____

Silvia Takahashi

____

Andrés Medaglia

Date Approved by Chairman ______________

Acknowledgements

The development of this project could not have been possible without the help of many people in very different aspects. Some researchers at the University have given their feedback, others gave key hints in some stages when the project was stuck. Some have been there since the beginning and others have helped in crucial moments, others built the basis for this to be possible.

To start from the beginning, I want to thank Professor Robert Foley from Georgia Tech because he planned the basic structure together with Germán Riaño back in year 2000, when Germán conceived an object oriented tool for this modeling purpose. Professor Foley's support guaranteed Germán's availability to develop the first steps.

In the second phase, I began working in the project thanks to Germán's motivation. He presented me the main idea for my graduation project. At some moment in time Julio Goez and Juan Fernando Pérez were essential because they gave the key ideas to make possible an independent modeling process from the solving process. When this stage was finishing, Professor Fernando Palacios was a juror in the presentation of the project and his feedback was helpful. Professor Andrés Medaglia got involved at that time and also contributed with his feedback on the initial document.

In the third phase, the project was presented at the Institute of Industrial Engineers Conference in Atlanta giving us the opportunity to show the work done and hear opinions from academics and industrials outside the Universidad de Los Andes. Among them I particularly want to thank Professor Paul Griffin from Georgia Tech. At the same time, Professor Hayriye Ayhan and Professor Robert Foley had the chance to see the project and their opinions were considered.

I want to thank INFORMS for giving us the opportunity of presenting this work at the Annual Conference this November, and Germán for presenting it, in spite of my impossibility of assisting by myself. I want to thank the current jurors of the masters thesis for their time. And finally I want to thank those interested in continuing with further work on this topic. Diego Bello has been working on JMDP for his graduation projects.

Contents

Summary

Variability is present in many real life problems and optimization models able to capture randomness can produce significantly better results than deterministic approaches (See Birge and Louveaux[8]. Markov Decision Processes (MDPs) are a specific model for stochastic optimization, and faces the curse of dimensionality, reason for which plenty of work is devoted to developing better algorithms. Some approximations have been developed for state reduction, others study the possibility of skipping some states during iterations, and others intend to achieve reasonable near optimal approximations that run faster.

But the state of the art, in general, is moving ahead from the tools that the industry is using to solve its problems. Modeling tools are often an important mean through which new models can become accessible to the industrial sector. Modeling tools are such that permit a user to focus on the modeling process and not in the solving process, which tends to be more complicated and often understandable just by few people. For example, many mathematical programming languages allow a user to formulate and model a problem, and solves the problem for the user, without needing her to fully understand non linear optimality conditions. Many statistical methods have also become reachable to industrial needs such as time series analysis. Some computational applications allow the user to introduce the data and try different models in order to find a proper one, without getting involved in maximum likelihood estimation for the parameters of each model. Modeling

tools of this type reduce the gap between the border of knowledge and the applicability of such information.

This work presents a generic tool for modeling MDPs. The main purpose is to facilitate the modeling process and to provide a solution to the problem in order to avoid distractions from the modeling process. No proposals are being made in the solution algorithms. The modeling process is aided by some essential features we believe should make it significantly simpler. The first of these is Object Oriented Programming (OOP). Each type of MDP has some mathematical elements that completely characterize the problem. OOP permits that each of these elements is represented computationally in a way that an analogy is kept with the mathematical representation in order to achieve a natural modeling process. Abstract classes and methods allow the framework to require these characteristic elements to be specified, and once fulfilled, the problem is modeled and can be solved.

The second important modeling facilitating feature is the introduction of events in the modeling process. To our knowledge no models have been presented with events, probably because they do not improve the computational results for the solution algorithms but could event worsen it. But, for the purpose of this work, the introduction of events simplifies the modeling process by characterizing simpler transitions from a state to smaller destination sets. During the development of the events model, a queuing control problem was formulated which could not be represented with the previous model. Studying this problem deeper we arrived to a new model where actions depend on events, useful in many applications. The advance in this field consisted in proving that a problem without events can be formulated being equivalent to each of the problems with events. Finally, the last modeling features include a state exploration algorithm and the features that make Java a friendly language.

This work was first started by Germán Riaño in 1997 in GeorgiaTech under direction of Professor Robert Foley. At that time Professor Riaño conceived the object oriented structure of the program. He developed a finite horizon solver using abstract classes and abstract methods to force the user to provide the characteristic elements of the stochastic and deterministic problems and return a solution.

In 2004 I started my work on the subject for my undergraduate senior project. Some months later, when my conception of the mathematical model was clearer, we restructured the framework. The data structures were modified in order to save the value functions and policies in an efficient way without losing flexibility. The sets of states and actions were defined and the deterministic problem was extended from the deterministic one and not viceversa. The solution methodology was changed completely to allow independent modeling from solving and interchangeable solvers, in fact the structure allows new solvers to be coded modularly. The framework was extended to handle infinite horizon discrete time Markov decision processes (DTMDPs) and various solvers were developed to deal with discounted cost infinite horizon problems.

The work developed since then, started with documentation and a first version of the JMDP User's Manual was finished for the existing version. It was also tried as academic material in the Stochastic Processes Seminar given by Dr. Riaño. The research target changed completely and focused on similar tools for modeling MDPs. In terms of the program, the algorithms were slightly improved using iterators, but the new modeling features were included. The state exploration algorithm was developed for each type of problem and average criteria solvers were implemented. The problem structure was reformulated to be able to represent continuous time Markov decision processes, but without requiring new solvers. So cover functions were included that formulate a DTMDP equivalent to the CTMDP in order to solve it with the previous algorithms. Finally, the mathematical model was formulated with events. Similar problems without events were proved equivalent to the problems with events and the problem structure was modified to include the event model for each type of problem. At that point the research process also included searching for models with events in MDPs but these were not found.

Chapter I

Introduction

Markov Decision Processes have seen a lot of development since the pioneering work of Bellman and others [6]. Not only there is a significant wealth of research on the subject but also they have been successfully applied to a wide range of real life systems including inventory management, revenue management[43, 25] systems used nowadays by airlines and other industries[20, 2, 37]. Yet, they remain a rather sophisticated tool mastered by a few researchers and consultants. To our knowledge they are not taught as standard material at the undergraduate level in many IE or Management schools as are other optimization techniques, quite notably linear programming[4].

In our opinion there are two main issues that have prevented the popularization of dynamic programming and MDP. First, the so-called *curse of dimensionality*, by which we mean that the number of states required to describe a system grows exponentially as the size of the system increases. Second, as far as we know, there is not any available software that allows the user to model a system without the need to worry about solving and implementation details. The curse of dimensionality, by its own nature, will always impose a limitation on the problems that can be solved by MDPs, but certainly the advent of faster computers will allow larger systems to be modeled and solved efficiently. Also there are some state reduction techniques that have proved to be quite effective [40, 44, 47, 48]. This paper, however, does not deal with this issue, but rather is a proposal for the second: a generic solver can be build that enables the user to focus on modeling. We present an object oriented framework to attack this issue and describe the implementation we developed.

When an O.R. practitioner today wants to solve a problem, modeling it as an LP or MIP she has at her disposal an excellent array of software packages like AMPL, Mosel, GAMS, Lingo, and OPL-Studio, that allows her to write the model in a very natural way, very similar to its mathematical counterpart, without the need to worry about the solving details. The software then builds the matrices and passes the problem to its solver. Some of them allow (or force) the user to select a third party solver (GAMS[30], AMPL[19]), so there is a complete separation between modeling and solving the problem. The user does not need to know any details about the simplex method to solve an LP or the subtleties of Branch and Bound to solve an MIP. Granted, the solving time and the solution quality can improve a great deal by means of a clever formulation or by tweaking the solver parameters, but in principle she does not have to do the tweaking. In fact, some of the more sophisticated cut generation techniques used by the solver are proprietary and she would not be able to know about them. If every time that a practitioner wanted to solve an MIP model she needs to build the model and code Branch and Bound then LP and MIP would not be as popular as they are.

In this paper we present an MDP tool analogous to the one described for Linear programming or MIP. The analyst can concentrate on modeling the problem and choose from a set of solvers one at her convenience. The design of the system is modular, so third party developers can build their own solvers. The paradigm used is not that of a text file with commands written in a specific language, like those used in linear programming that we mentioned before, but rather it is based on Object Oriented Programming (OOP). OOP is not a new idea, but it has gained a widespread diffusion thanks to programming languages like C++ and Java [11, 46]. OOP enables the encapsulation of mathematical entities end expose its functionality independently from its internal implementation.

When modeling an MDP the analyst has to follow these steps: she must characterize the states of the system, the actions that can be considered in every state, the conditional probability distribution of the destination state once an action has been taken in a given state, and the immediate cost received per transition for every state and action (a brief review on MDP is given below). Our framework establishes a computational element for each of these mathematical elements that the user has to extend or implement in order to represent the model. The elements have been designed in a way that it is natural for the analyst, since there is a one to one correspondence.

The main purpose is to present a tool that facilitates the modeling process, and avoids distortions in this process caused by the solving stage. In order to achieve this, various features have been included in the present framework. The first one is clearly Object Oriented Programming which permits a very close analogy between the mathematical elements and their computational representation. OOP also permits a hierarchical ordering of the problems and the ability to oblige the user to define some characteristic elements of each problem. The second important feature is the ability of modeling with events. To our knowledge, few literature has been written about MDPs with events[5, 28], probably because they do not improve computational performance, but instead they may worsen it. Using events improves the modeling process because the user has to describe simpler transitions. The third feature is the exploration algorithm which finds all the states in the chain, and avoids the possible mistakes the user could make while describing the whole set of states. Finally the use of Java avoids the user to get distracted by technical processes like memory allocation and releasing, and avoids mistakes due to type unsafe declarations.

We know about tools similar to this in certain aspects. For instance, a few queuing software is available ranging from winQSB[13] to PROBMELA[3], and MARCA[41]. WinQSB is a simple tool that calculates steady state probabilities given the matrices. MARCA instead is a more powerful package capable of handling various models. None of these are capable of managing optimization problems. From another perspective, stochastic linear programming languages are being developed like Xpress-SP[16] for stochastic programming, some extensions are being studied for AMPL[1], and SAMPL/SPInE[45]. These are not fully comparable to the framework that is being presented in this paper because they are not intended specifically for MDPs. From the perspective of discrete event systems, Julien[22] and Mohanty[32] have developed frameworks for problem modeling. Specifically on MDPs, one program able to solve them is winQSB that receives the matrices for each action. Professor Riaño started an object oriented framework for finite horizon problems called DPS (Dynamic Programming Solver)[35] on 1997 under the direction of Robert Foley and has been working jointly with Julio Goez, on a continuous time Markov chain modeling framework called JMarkov [36]. Specifically on Markov Decision Processes we have heard of MDPLab[24] as an educational tool developed by Lamond to test different algorithms. MDPLab has its own language and reads the information from files. It is the most powerful program among the ones researched by us, and allows sparse matrix handling for the probability transitions. It permits discrete time Markov chains modeling given the initial amount of states from the beginning and indicating the transition probabilities and rewards.

In our opinion, much of the development is done in the mathematical ground improving the algorithms. In spite of this development, new solutions for companies take a long time to be implemented, until simple tools are at reach for professionals that do not need an advanced mathematical formation. The previous software and packages are an enormous advance in closing the gap between the leading mathematical models and their application in real life problems.

The present document is organized in the following way. The second chapter describes the mathematical MDP models and the notation that will be used. The third chapter describes the mathematical models handling events and develops an example. The fourth chapter describes the computational structure used to build the framework and how the different problems are handled. Fragments from the example introduced earlier are included. Finally, the fifth chapter presents general conclusions.

Chapter II

Mathematical Background - Markov Decision Process

The problems that can be modeled and solved with the present framework can be classified in various categories. They can be divided in finite or infinite horizon problems, or they can also be divided in deterministic and stochastic problems. Problems with events can be considered the general case and those without events would be a special case, or the ones with events could be considered an extension of those without events. A suggested taxonomy is shown in Figure 1 because the framework's structure will be analogous.

For instance, consider an inventory problem. When the future demands are known beforehand the problem is deterministic, while the stochastic case would consider random demand with known distribution. A finite horizon problem would be useful to solve the optimal order policy of the year when seasonality is strong, while an infinite horizon problem would be useful to determine a general order policy if the demand stationary.

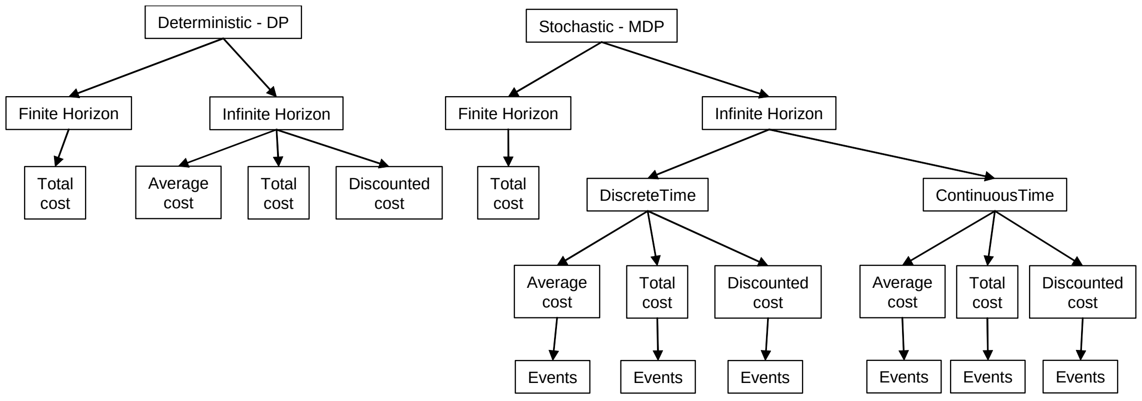


Figure 1: Taxonomy for MDP problems

The deterministic problems are known as Dynamic Programming problems, and the stochastic problems are commonly called Markov Decision Processes (MDPs). The framework's object oriented structure permits a classification of the modeling classes analogous to the mathematical classification presented in this section.

2.1 Finite Horizon Problems

We will show how a finite horizon Markov decision process is built (for a complete reference see Puterman [34] and Bertsekas [7]). Each type of problem is modeled with a class in the presented framework. The finite horizon problems are a branch of classes. Consider a discrete space, discrete time, bivariate random process $\{(X_t, A_t), t = 0, 1, \ldots, T-1\}$. Each of the $X_t \in \mathcal{S}_t$ represents the state of the system at stage t, and each $A_t \in \mathcal{A}_t$ is the action taken at that stage. The quantity $T < \infty$ is called the *horizon* of the problem. The sets $\mathcal{S}_t$ and $\mathcal{A}_t$ are the states and actions available at stage t, and we will assume that both are finite. Let H_t be the *history* of the process up to time t, where $H_t = (X_0, A_0, X_1, A_1, \ldots, X_{t-1}, A_{t-1}, X_t)$. The dynamics of the system is governed by a state in which an action is taken, leading the system to another state according to a probability distribution. In general, the resulting state after a transition, depends on the history and the action taken. The system satisfies the Markov property which implies $P\{X_{t+1} = j | H_t = h, A_t = a\} = P\{X_{t+1} = j | X_t = i, A_t = a\} = p_{ijt}(a)$. A *decision rule* is a function π_t that, given a history realization, assigns a probability distribution over the set $\mathcal{A}_t$. A sequence of decision rules $\pi = (\pi_0, \pi_1, \ldots, \pi_T)$ is called a *policy*. A policy π is called Markov if given X_t all previous history becomes irrelevant, that is

$$P_\pi\{A_t = a | H_t = h\} = P_\pi\{A_t = a | X_t = i\}.$$

where P_π means the distribution following policy π. A Markov policy π is called *deterministic* if there is a function $f_t(i) \in \mathcal{A}$ such that

$$P_\pi\{A_t = a | X_t = i\} = \begin{cases} 1 & \text{if } a = f(i) \\ 0 & \text{otherwise.} \end{cases}$$

For each action a taken from state i at stage t, a finite cost $c_t(i, a)$ is incurred. Consequently it is possible to define a total expected cost $v_t^\pi(i)$ incurred from time t to the final stage T following policy π; this is called the *value function*

$$v_t^\pi(i) = E_\pi\left[\sum_{s=t}^{T} c_s(X_s, A_s) \middle| X_t = i\right], \quad i \in \mathcal{S}_t \tag{1}$$

where E_π is the expectation operator following the probability distribution associated with policy π. The problem is to find the policy $\pi \in \Pi$, where Π is the set of all policies, that minimizes the objective function shown above $v_t^*(i) = \inf_{\pi \in \Pi} v_t^\pi(i)$.

Such optimal value function has to satisfy *Bellman's optimality equation*

$$v_t^*(i) = \min_{a \in \mathcal{A}_t(i)} \left\{ c_t(i,a) + \sum_{j \in \mathcal{S}_t(i,a)} p_{ijt}(a) v_{t+1}^*(j) \right\}, \quad i \in \mathcal{S}_t,\, t = 0, 1, \ldots, T-1, \tag{2}$$

where $\mathcal{A}_t(i)$ is the set of *feasible actions* that can be taken from state i at stage t and $\mathcal{S}_t(i,a) = \{j \in \mathcal{S}_{t+1} | p_{ijt}(a) > 0\}$ is the set of reachable states from state i taking action a at stage t. It can be shown that there exists a deterministic decision rule $\pi_t : \mathcal{S}_t \rightarrow \mathcal{A}_t, \quad t = 0, 1, \ldots, T-1$ that is optimal. So it is only necessary to search among the deterministic decision rules for each stage.

Observe that equation (2) suggests an algorithm to solve the optimal value function, and consequently the optimal policy. It starts from some final values of $v_T(i)$ and solves backward the optimal decisions for the other stages. The optimal decision rule can be obtained taking, in every state, the action that minimized the right hand side (breaking ties arbitrarily). Notice that, in order to solve the problem, $\mathcal{A}_t(i)$ and $\mathcal{S}_t(i,a)$ are essential. The implicit algorithm in (2) also requires $c_t(i,a)$ and $p_{ijt}(i,a)$. Thus these four elements completely characterize a finite horizon MDP.

2.2 Infinite Horizon Problems

The family of infinite horizon problems are represented by another branch of classes, and extend from another class, independently from the finite horizon classes in the presented framework. Consider a discrete space discrete time bivariate random process $\{(X_t, A_t), t \in \mathbb{N}\}$. The discrete time infinite horizon problem will be denoted DTMDP. Notice the time horizon is now infinite. In particular we will assume that the the system is *time homogeneous*, this means that at every stage the state space and action space remain constant and the transition probabilities are independent of the stage $p_{ijt}(a) = p_{ij}(a) = P\{X_{t+1} = j | X_t = i, A_t = a\}$ for all stages. Consequently, a policy $\pi = (\pi_0, \pi_0, \ldots)$ must also be time homogeneous.

Costs are also time homogeneous so $c_t(i,a) = c(i,a)$ stands for the cost incurred when action a is taken from state i for every stage. Besides the total cost objective function presented in the finite horizon problem, it is customary to define two other objective functions: discounted cost and average cost. The respective value functions under a policy π are

$$v_\alpha^\pi(i) = E_\pi \left[\sum_{t=0}^{\infty} \alpha^t c(X_t, A_t) \Big| X_0 = i \right], \quad i \in \mathcal{S},$$

for the discounted cost where $0 < \alpha < 1$,

$$v^\pi(i) = E_\pi\left[\sum_{t=0}^{\infty} c(X_t, A_t)\middle| X_0 = i\right], \quad i \in \mathcal{S},$$

for the total cost, and

$$\overline{v}^\pi(i) = \lim_{T\to\infty} \frac{1}{T} E_\pi\left[\sum_{t=0}^{T} c(X_t, A_t)\middle| X_0 = i\right], \quad i \in \mathcal{S},$$

for the average cost. The respective optimal value functions are $v^*_\alpha(i) = \inf_{\pi\in\Pi} v^\pi_\alpha(i)$, $v^*(i) = \inf_{\pi\in\Pi} v^\pi(i)$, and $g = \overline{v}^*(i) = \inf_{\pi\in\Pi} \overline{v}^\pi(i)$, where g is optimal regardless of the state in an irreducible chain. Optimality conditions can be derived for each case. For instance it can be shown that $v^*_\alpha(i)$ satisfies the following Bellman's optimality equation

$$v^*_\alpha(i) = \min_{a\in\mathcal{A}(i)} \left\{ c(i,a) + \alpha \sum_{j\in\mathcal{S}(i,a)} p_{ij}(a) v^*_\alpha(j) \right\}, \quad i \in \mathcal{S} \tag{3}$$

where $\mathcal{A}(i)$ is the set of feasible actions from state i in any stage and $\mathcal{S}(i,a)$ is the set of reachable states. The costs $c(i,a)$, the transition probabilities $p_{ij}(i,a)$ and the sets of feasible actions $\mathcal{A}(i)$ and reachable states $\mathcal{S}(i,a)$ characterize a problem and are enough to determine a solution to the problem.

There are various algorithms for solving the discounted cost problem. One of them is almost implicit in equation (3). The algorithm, called *Value Iteration*, begins with some initial values for each state for the right side and calculates new values on the left side that will be introduced in the right side in the next iteration, until these values converge within a tolerance of ϵ. It can be shown that for $0 < \alpha < 1$ the algorithm converges regardless of the initial values. Further explanation on this or other algorithms can be found in Bertsekas [7].

2.3 Continuous Time Markov Decision Processes

The continuous time problems are represented in classes independently from the DTMDPs. Continuous time problems are commonly used for queuing problems for example in service centers, production networks and call centers. For this section consider an infinite horizon problem with time-homogeneity, where the set $\{(X(t), A(t)), t \geq 0\}$ is the bivariate process that describes the state of the system and the action taken at time t, on a continuous time space. The continuous time infinite horizon problem will be denoted

CTMDP. Time-homogeneous transitions between states are described by a transition rate $\lambda_{ij}(a) = \lim_{h\to 0} P\{X(t+h) = j | X(t) = i, A(t) = a\}$. The Markov property implies that the time between transitions from one state to another has an exponential distribution with parameter $\lambda_i(a)$ equal to the sum of the exit rates from the former state. This implies that a transition occurs only when the state of the system $X(t)$ changes and no self transitions are allowed. Rate λ is such that $\lambda \geq \lambda(i,a)$ for all states and actions. Costs can be lump costs $\tilde{c}(i,a)$ incurred in the instant when an action is taken and can also be continuously incurred at rate $\gamma(i,a)$ while remaining at state i.

In the discounted case, the discount factor $0 < \alpha < 1$ is still relevant but will alternately be referred to as the interest rate $0 < \beta$ where $\alpha = (1+\beta)^{-1}$. It can be shown that the discrete time process with transitions is equivalent to the continuous time process, uniformizing the chain with jumps from one state to itself if its transitions are

$$p_{ij}(a) = \begin{cases} 1 - \frac{1}{\lambda}\lambda_i(a), & i = j; \\ \frac{1}{\lambda}\lambda_{ij}(a), & i \neq j. \end{cases}$$

The discrete instantaneous costs are represented by

$$c(i,a) = \frac{\beta + \lambda_i(a)}{\beta + \lambda}\left(\tilde{c}(i,a) + \frac{\gamma(i,a)}{\beta + \lambda_i(a)}\right), \quad i \in \mathcal{S}, a \in \mathcal{A}.$$

The CTMDP is treated as a DTMDP and the value functions defined in the preceding sections still hold, the algorithms are also valid, and the convergence is not altered. For further references see Stidham [42] and Serfozo [39].

When the criteria is average cost per stage, the uniformization probabilities are the same, but the equivalent cost for the DTMDP is

$$c(i,a) = \frac{\lambda_i(a)}{\lambda}\left(\tilde{c}(i,a) + \frac{\gamma(i,a)}{\lambda_i(a)}\right), \quad i \in \mathcal{S}, a \in \mathcal{A}.$$

A CTMDP is completely characterized by the following elements: the sets of feasible actions $\mathcal{A}(i)$, the sets of reachable states $\mathcal{S}(i,a)$, transition rates $\lambda_{ij}(a)$, the lump costs $\tilde{c}(i,a)$, and the cost rates $\gamma(i,a)$. Only in the discounted case, an interest rate β is needed.

Chapter III

Event modeling

Modeling with events is a common practice in an area called Discrete Event Systems (DES) related to electric engineering (see Marchand [28] or Cassandras [12] for a complete reference). Particularly relating MDPs with events is the work of Becker [5], Mahadevan [27], and Feinberg [17]. Martinelli [29] has worked on parameter dependant DES and Mohanty [32] has developed a framework for optimal control of DES.

The present section introduces events in order to simplify the modeling process. The model presented is an extension of the models presented before in the sense that the same problems can be represented, conditioning each transition to the event triggering it. The advantage of such a presentation lies on the fact that conditioning reduces the reachable set for each state and permits an easier characterization of the system dynamics. For example, in an inventory problem, the transitions are conditioned to the demand that occurs in each state when the action has already been taken. Each of the problems with events is modeled with one class in the presented framework, and each class with events extends the class representing the problem without events.

3.1 Event conditioning

For the mathematical model extending the previous ones with events, consider the discrete time random process $\big\{(X_t, A_t, E_t),\, t = 0, 1, \ldots, T-1\big\}$, where X_t represents the state of the system, A_t represents the action taken, and E_t is the event that happens at stage t that triggers the transition to X_{t+1} and belongs to a set of events at stage t, $\mathcal{E}_t$. The history of the process up to stage t, is defined $H_t = (X_0, A_0, E_0 \ldots, X_{t-1}, A_{t-1}, E_{t-1}, X_t)$. The markovian behavior of the system implies that $P\{X_{t+1} = j | H_t = h, A_t = a, E_t = e\} = P\{X_{t+1} = j | X_t = i, A_t = a, E_t = e\}$. Consequently the dynamics can be described by transition probabilities defined as $p_{ijt}(a, e) = P\{X_{t+1} = j | X_t = i, A_t = a, E_t = e\}$

that describe the *conditional* probability of reaching state j in the set of reachable states $\mathcal{S}_t(i,a,e)$, given that the current state is i, action a is chosen and *given* that event e occurs. The actions also present a markovian property in the sense that $P\{A_t = a|H_t = h\} = P\{A_t = a|X_t = i\}$. Finally, the $P\{E_t = e|H_t = h, A_t = a\} = P\{E_t = e|X_t = i, A_t = a\}$.

Let $c_t(i,a,e)$ be the cost incurred by taking action a at stage t from state i and $p_t(e|i,a) = P\{E_t = e|X_t = i, A_t = a\}$ be the *conditional* probability of occurrence of event e at stage t.

In the finite horizon problem, the value function is defined as

$$v_t^\pi(i) = E_\pi\left[\sum_{s=t}^{T} c_s(X_s, A_s, E_s)\Big|X_t = i\right], \quad i \in \mathcal{S}_0,\, t = 0,1,\ldots,T-1.$$

The purpose of this section is to show that the process with events is equivalent to another process without events such as the ones described earlier in section 2.1. Consider $\mathcal{E}_t(i,a)$ to be the set of active events from state i and action a is chosen at stage t, then

$$\begin{aligned}
\mathcal{S}_t(i,a) &= \bigcup_{e\in\mathcal{E}_t(i,a)} \mathcal{S}_t(i,a,e), \quad i \in \mathcal{S}_t,\, a \in \mathcal{A}_t,\, t = 0,1,\ldots,T-1,\\
p_{ijt}(a) &= \sum_{e\in\mathcal{E}_t(i,a)} p_t(e|i,a)p_{ijt}(a,e), \quad i,j \in \mathcal{S}_t,\, a \in \mathcal{A}_t,\, t = 0,1,\ldots,T-1,\\
c_t(i,a) &= \sum_{e\in\mathcal{E}_t(i,a)} p_t(e|i,a)c_t(i,a,e), \quad i \in \mathcal{S}_t,\, a \in \mathcal{A}_t,\, t = 0,1,\ldots,T-1.
\end{aligned}$$

It is then possible to build an equivalent finite horizon MDP, such that an optimal solution for it will be optimal for the initial problem. The characteristic elements are $\mathcal{A}_t(i)$, $\mathcal{E}_t(i,a)$, $S_t(i,a,e)$, $p_{ijt}(a,e)$, $p_t(e|i,a)$, and $c_t(i,a,e)$. In the infinite horizon problem, a similar equivalence is valid but the time homogeneity assumption is necessary. Actions are chosen from a set $\mathcal{A}(i)$ and the set of active events is $\mathcal{E}(i,a)$. Notice that for the problems treated in this section, the set of active events can depend on the action. The transition probabilities are defined as $p_{ij}(a,e)$ and determine the probability of reaching state j in the set of reachable states $\mathcal{S}(i,a,e)$, given action a is taken from state i, and given event e triggers the transition. Let $c(i,a,e)$ be the cost incurred by taking action a from state i when event e occurs, and let $p(e|i,a) = P\{E_t = e|X_t = i, A_t = a\}$ be the *conditional* probability of occurrence of event e at every stage. The equivalence is as follows:

$$\begin{aligned}
\mathcal{S}(i,a) &= \bigcup_{e\in\mathcal{E}(i,a)} \mathcal{S}(i,a,e), \quad i\in\mathcal{S},\ a\in\mathcal{A},\\
p_{ij}(a) &= \sum_{e\in\mathcal{E}(i,a)} p(e|i,a)p_{ij}(a,e), \quad i,j\in\mathcal{S},\ a\in\mathcal{A},\\
c(i,a) &= \sum_{e\in\mathcal{E}(i,a)} p(e|i,a)c(i,a,e), \quad i\in\mathcal{S},\ a\in\mathcal{A}.
\end{aligned}$$

The three optimization criteria described in section 2.2 are also valid as before,

$$\begin{aligned}
v_\alpha^\pi(i) &= E_\pi\left[\sum_{t=0}^{\infty}\alpha^t c(X_t,A_t,E_t)\Big|X_0=i\right], \quad i\in\mathcal{S}\\
v^\pi(i) &= E_\pi\left[\sum_{t=0}^{\infty} c(X_t,A_t,E_t)\Big|X_0=i\right], \quad i\in\mathcal{S}\\
\overline{v}^\pi(i) &= \lim_{T\to\infty}\frac{1}{T}E_\pi\left[\sum_{t=0}^{T} c(X_t,A_t,E_t)\Big|X_0=i\right], \quad i\in\mathcal{S}.
\end{aligned}$$

The characteristic elements are $\mathcal{A}(i)$, $\mathcal{E}(i,a)$, $S(i,a,e)$, $p_{ij}(a,e)$, $p(e|i,a)$, and $c(i,a,e)$. In the continuous time problem there are no explicit probabilities. Consequently, the equivalent CTMDP problem without events is built with the following transformations.

$$\begin{aligned}
\mathcal{S}(i,a) &= \bigcup_{e\in\mathcal{E}(i,a)} \mathcal{S}(i,a,e), \quad i\in\mathcal{S},\ a\in\mathcal{A},\\
\lambda_{ij}(a) &= \sum_{e\in\mathcal{E}(i,a)} \lambda_{ij}(a,e), \quad i,j\in\mathcal{S},\ a\in\mathcal{A},\\
\tilde{c}(i,a) &= \sum_{e\in\mathcal{E}(i,a)} p(e|i,a)\tilde{c}(i,a,e), \quad i\in\mathcal{S},\ a\in\mathcal{A},\\
\gamma(i,a) &= \sum_{e\in\mathcal{E}(i,a)} p(e|i,a)\gamma(i,a,e), \quad i\in\mathcal{S},\ a\in\mathcal{A},
\end{aligned}$$

where,

$$p(e|i,a) = \frac{\sum_{k\in\mathcal{S}(i,a,e)}\lambda_{ik}(a,e)}{\lambda_i(a)} \qquad e\in\mathcal{E}, i\in\mathcal{S}, a\in\mathcal{A}. \tag{4}$$

The characteristic elements are $\mathcal{A}(i)$, $\mathcal{E}(i,a)$, $S(i,a,e)$, $\lambda_{ij}(a,e)$, $\tilde{c}(i,a,e)$, and $\gamma(i,a,e)$. Once an equivalent CTMDP without events is formulated, it is solved formulating an equivalent DTMDP as described in section 2.3.

3.2 Modeling procedure

In order to illustrate the mathematical modeling procedure for the DTMDP with events, we will develop an inventory example. The modeling process could be generally described as

1. Identify stages, states, actions, events
2. Identify a type of problem
3. Identify sets of active events, feasible actions, reachable states.
4. Identify transitions and costs.

Consider a car dealer selling identical cars an handling a weekly (periodic) review inventory system. Each car is bought at \$$c$ and sold at \$$p$. A transporter charges a fixed fee of \$$K$ per truck for carrying the cars from the distributor to the car dealer, and each truck can carry L cars, and the order arrives the same day it is ordered. The exhibit hall has space for M cars. If a customer orders a car and there are no cars available, the car dealer gives him the car as soon as it arrives with a \$$b$ discount. The car dealer does not allow more than B backorders of this type. Holding inventory implies a cost of capital of β annually (interest rate). The marketing department has informed that the demand follows a Poisson process with mean θ cars per week. Which is the optimal order policy that minimizes costs? The rest of this section describes the detailed modeling process.

Let D_t represent the random weekly demand, $p_n = P\{D_t = n\}$ and $q_n = P\{D_t \geq n\}$. The problem is an infinite horizon stochastic problem. It is possible to model the problem with or without events.

1. States. Each state X_t is the inventory level at the end of week t, where the stages are weeks.
2. Actions. Each action A_t is the order placed at week t.
3. Events. Each event E_t represents the realization of the random demand that occurs in each week t.
4. Type of problem. The problem on-hand is an infinite horizon problem, it is stochastic and the time is discrete. We will use events.

5. Feasible Actions. For each state i the feasible actions that can be taken are those that will not exceed the exhibit hall's capacity. The maximum order feasible is $M-i$, so the feasible set of actions for each state i is $\mathcal{A}(i) = \{0, \ldots, M-i\}$.

6. Active Events. For each state i and given action a is taken, zero-demand is active in every state. Demand equal to $i+a+B$ is indistinguishable from larger demands. So the sets of active events are $\mathcal{E}(i,a) = \{0, \ldots, i+a+B\}$.

7. Reachable States. The only reachable state when action a is taken from state i and event e occurs is $(i+a-e)$. The set of reachable states is $\mathcal{S}(i,a,e) = \{i+a-e\}$.

8. Transition Probabilities. When demand is greater than or equal to $i+a+B$ the destinations state is (0), and all indistinguishable.

$$p_{ij}(a,e) = \begin{cases} p_{i+a+B} & \text{if } e < i+a+B \\ q_{i+a+B} & \text{if } e \geq i+a+B \end{cases}$$

9. Costs. The cost incurred depends on various factors. The ordering cost only depends on the order and is only charged when it is positive, and charged per truck, and when only part of the truck is occupied, the whole truck is charged $\lceil \frac{a}{L} \rceil$. The holding cost depends only on the state and is charged only when there is positive stock, and the pending orders cost charged only when there is negative stock. $(ih)^+$ and $(ib)^-$. Finally, there is an expected lost sales cost(See Zipkin [49]) $L(i,a,e) = E[D_t - (i+a+B)]^+ = q_{i+a+B}\big(\theta - (i+a+B)\big) + (i+a+B)p_{i+a+B}$. The total cost is $c(i,a,e) = \lceil \frac{a}{L} \rceil + (ih)^+ + (ib)^- + q_{i+a+B}\big(\theta - (i+a+B)\big) + (i+a+B)p_{i+a+B}$.

Chapter IV

Framework Architecture

This section explains the main core of the object oriented framework that we introduce. The main idea is to have objects representing the basic structures in an MDP, like states, actions and events. A problem is also represented as an object. The idea used is to include abstract classes for each problem type. An abstract class cannot be instantiated, it can only be subclassed [11]. The abstract methods in each class oblige the user to define the characteristic elements for each type of problem as explained in the mathematical description of the various types of problems. Finally, some classes represent solvers for different types of problems and optimization criteria, and receive a specific type of problem. All the architecture is intended to facilitate the modeling process and allow the user to focus on problem formulation and modeling without distracting with solving algorithms.

This means that the procedure to model a problem starts by defining the basic elements (states, actions, and events when present). Then the user chooses an abstract problem type and extends such class, implementing the abstract methods that characterize his specific problem. The problem is at this point modeled. In order to solve it, a default solving method is configured, but the user can prove different solvers, methods or even plug in his own solver.

The framework was built in Java because of its flexibility and because it has become common for these type of applications like [31, 23, 18, 26, 33, 21] or see[9] for an up to date information. The speed problems faced at the beginning are being overcame with the recent versions of the hot spot compiler. Besides, Java 1.5 presents a new feature that facilitates the use of abstract classes called generics [10]. This one of the modeling features included.

The solvers included are designed to solve only DTMDPs and finite horizon problems. The strategy to solve any infinite horizon problem, is to formulate an equivalent DTMDP when it is not one, and then solve it. Analogies were described above in order to formulate problems without events that are equivalent to the problems with events, and DTMDPs equivalent to the CTMDP without events. The state expansion for the event dependent actions problem is also done in a user transparent way.

The framework has three packages which group objects of different types as described above. The first package is called `jmdp.basic` and contains the basic elements that make up an MDP model and the objects interacting with them, that are not problems themselves. The `jmdp` package has the classes that represent the problems to be modeled. Finally, the `jmdp.solvers` package contains the solving methods coded in this framework. The solution is completely independent from the modeling, which is an interesting feature present in advanced modeling languages like Mosel[15]. Independent modeling is another modeling facilitating feature. A brief description of these packages follows.

4.1 Basic elements package

This package is a collection of classes designed to represent the most basic elements that interact in a MDP. Abstract classes `State`, `Action`, and `Event` represent those objects. The user must extend each class in order to have her own states, actions, and events when needed. The abstract classes allow as general an implementation as possible for the user to use his/her most suitable structure. For ease of use, the abstract classes `StateArray<S>`, `ActionArray<A>`, and `EventArray<E>` are extensions of the previous ones, and represent each structure with an internal array. Getters and setters are encouraged to encapsulate the objects' attributes.

This package also includes classes that represent sets of the previous objects which are `States<S>`, `Actions<A>`, and `Events<E>`. Notice the use of generics in the sets, where for instance, `States<S>` means that the class represents a set of objects of type `S`. The abstract classes `StateSet<S>`, `ActionSet<A>` and `EventSet<E>` are extensions of the set classes and internally implement a `java.util.TreeSet` that stores the objects in an ordered way avoiding duplicates.

Finally this package includes the objects that represent some type of interaction between basic elements. The class `ValueFunction<S>`, assigns a value to each state. Class `Policy<S,A>` stores objects `DecisionRule<S,A>` for each stage of a finite horizon problem, or a single decision rule for an infinite horizon problem. Each `DecisionRule<S,A>` represents a deterministic decision rule and stores the action for each state. Class `Solution<S,A>`

includes a `Policy<S,A>` and a `ValueFunction<S,A>` and is the object returned by every `solve()` method in every solver. Some `Exceptions` are also included in this package.

4.2 Modeling package

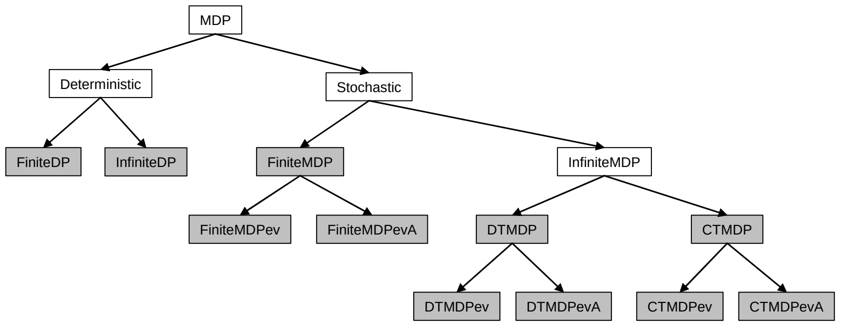


Figure 2: Classes

The modeling package contains the classes that represent all the different types of problems described in section 2 and represented in figure 1. The class structure is depicted in figure 2. Class `MDP<S,A>` has the principal attributes and procedures an MDP must have, like a default solver, a `Solution<S,A>`, and the `getSolution()` method. The higlighted classes in fugure 2 represent problems and they have or inherit abstract methods that the user must implement in order to correctly model the intended problem. Only class `CTMDP<S,A>` will be described in this section and the inventory example presented in section 3.2 will be shown at the end illustrating class `DTMDPev<S,A>`. The other classes are described in the user's manual[38]. Notice that all the classes handle generics and the specific state, action, and event types used must be indicated.

The classes that are not highlighted are structural classes. This permits that some attributes that are common to various classes are only included in the superclass. Another procedure common to various classes is the state exploration procedure. Once the set of reachable states is implemented for a certain type of problem, and given a set of initial states, this algorithm explores the states in a Breadth First Search mode (see Ciardo[14]). Given a finite state structure, it is possible to explore all the states in the chain that can be reached from these initial states after finite transitions. The method iterates until it finds no new states. It was observed that the user definition of the whole set of states in the modeling process was a source for errors. When the chain structure becomes more complex and the set of actions is significant, the beforehand determination of such a set is complicated. This is another one of the important modeling features included.

The elements that characterize a continuous time infinite horizon Markov decision problem (CTMDP) were commented in section 2.3. Given the states and actions that are independent of the type of problem, the procedure to model the problem needs the identification of the type of problem and the class that represents such problem. The abstract methods in class `CTMDP<S,A>` are those listed in table 1. This information is enough to completely characterize a problem of this type. The mentioned procedures are abstract methods in the problem class and the user must implement them in the extending class. The framework uses this information to formulate the characteristic elements of `DTMDP<S,A>`, and the solution algorithms then solve the problem as a DTMDP.

Mathematical representation	Object or Method
$X(t) \in \mathcal{S}$	`public class MyState extends State`
$A(t) \in \mathcal{A}$	`public class MyAction extends Action`
$\big\{\big(X(t), A(t)\big), t > 0\big\}$	`public class MyProblem extends CTMDP<MyState,MyAction>`
$\mathcal{A}(i)$	`public Actions<MyAction> feasibleActions(MyState i)`
$\mathcal{S}(i,a)$	`public States<MyState> reachable(MyState i, MyAction a)`
$\lambda_{ij}(a)$	`public double rate(MyState i, MyState j, MyAction a)`
$\tilde{c}(i,a)$	`public double lumpCost(MyState i, MyAction a)`
$\gamma(i,a)$	`public double continuousCost(MyState i, MyAction a)`

Table 1: `CTMDP` abstract methods and related objects

One of the most important modeling features in the framework is the ability of modeling with events. In section 3 the models handling events were presented. The modeling package includes these as can be seen in figure 2 where classes ending with `ev` represent that type of problem with events. These classes extend from their equivalent classes without events but now include abstract methods receiving events as parameters. For example, the `CTMDPev<S,A,E>` class $\big(\big\{\big(X(t), A(t), E(t)\big)\big\}\big)$ has the following abstract requirements $\mathcal{A}(i)$, $\mathcal{E}(i,a)$, $\mathcal{S}(i,a,e)$, $\lambda_{ij}(a,e)$, $\tilde{c}(i,a,e)$, and $\gamma(i,a,e)$.

4.3 Solving package

The solving mechanisms are not the main topic of this article; no particular novelty is presented in them. Nevertheless the flexibility of the framework allows different solvers to be easily coded and introduced in the package. All solver classes inherit from the `Solver<S,A>` abstract class. The constructor receives a problem to be solved and all the implementing classes must implement a `solve()` method that returns a `Solution<S,A>` object with the optimal policy and the optimal value function. Currently, solvers are implemented to solve the finite horizon problem (deterministic or stochastic). The value iteration algorithm is implemented for the infinite horizon discounted problem and allows activating or deactivating the Gauss-Seidel modification and using error bounds [7]. Policy iteration algorithm has been implemented using JMP [21] and modified policy iteration has also been implemented for discounted problems. In the case of average cost optimization criteria, only relative value iteration is currently implemented and interaction with a linear programming solver is under development. For the continuous time case and the problems with events, all the algorithms presented for the discrete time case still work with the same convergence properties. The problems where actions depend on events experiment a significant state multiplication that will affect the computational time. Each problem class has a default solver assigned for that type of problem. Users are able to focus on modeling and then simply ask for the optimal solution.

Other solvers will be shortly included in the package that are not intended to reach a solution to the optimization problem, but, for a given policy, calculate measures of performance. Estimating a steady state probability when possible, the solver calculates the probability of each state and calculates the expected value of some measure of performance. These measures could be effective rate, queue length, server utilization, etc. Having the effective rate, the time dependent measures can also be calculated like the waiting time.

4.4 Application and Results

Recall the problem mentioned in section 3.2. Each state is the level of physical inventory. Class `InvLevel` represents each state.

Consider :

```
public class InvLevel extends StateArray {
  public InvLevel(int k) {
    super(new int[] {k});
  }

  public int getLevel() {
    return status[0];
  }
  @Override
  public String label() {
        return "Level␣" + getLevel();
    }
}
```

The actions that can be taken in each state are the orders placed. Class `Order` represents each of these objects.

```
public class Order extends Action {
    private int size;
    Order(int k) {
        size = k;
    }
    public final int getSize() {
        return size;
    }
    public int compareTo(Action a) {
        return (size - ((Order) a).size);
    }
    public String label() {
        return "Order␣" + size + "␣Units";
    }
}
```

The events that can happen in each state are demands. Class `Demand` represents each of these objects, and its structure is analogous to the one presented in `Order` and `InvLevel`. This problem corresponds to an infinite horizon, discrete time Markov decision process (DTMDP) with events, and the class able to model these type of problem is `DTMDPev<S extends State,A extends Action,E extends Event>`. It is first necessary to implement an object that extends `DTMDPev` and indicate the states, actions, and events that will be used.

```
public class InventoryProblem extends DTMDPev<InvLevel, Order, Demand> {
```

Then the methods declared as abstract in the superclass need to be implemented, but their parameters are the states, actions, and events indicated.

```
public abstract Actions<Order> feasibleActions(InvLevel i);
public abstract Events<Demand> activeEvents(InvLevel i, Order a);
public abstract States<InvLevel> reachable(InvLevel i, Order a, Demand e);
public abstract double prob(InvLevel i, InvLevel j, Order a, Demand e);
public abstract double prob(InvLevel i,Demand e);
public abstract double immediateCost(InvLevel i, Order a, Demand e);
```

These are the characteristic elements of this type of problem: $\mathcal{A}(i)$, $\mathcal{E}(i, a)$, $S(i, a, e)$, $p_{ij}(a, e)$, $p(e|i, a)$, and $c(i, a, e)$. For example, the set of feasible actions $\mathcal{A}(i) = \{0, \ldots, M - i\}$ is implemented as follows.

```
  @Override
  public Actions<Order> feasibleActions(InvLevel i){
    ActionsSet<Order> actionSet = new ActionsSet<Order>(); //empty set
    int max = maxInventory - i.getLevel();
    for (int n = 0; n <= max; n++){
      actionSet.add(new Order(n)); //add each feasible action
    }
    return actionSet; //return the set
  }
}
```

The result for this problem, with parameters $K = 6$, $B = 0$, $\theta = 4$, $M = 15$, $h = 10$, $A = 100$, $b = 600$, is stored as a `Solution<InvLevel,Order,Demand>` and the policy can be out printed as follows.

```
Value Iteration Solver for Average reward problem
Using Gauss-Seidel modification
********* Best Policy *********

In every stage do:
STATE      ------> ACTION
LEVEL 0    ------> ORDER 15 UNITS
LEVEL 1    ------> ORDER 14 UNITS
LEVEL 2    ------> ORDER 12 UNITS
LEVEL 3    ------> ORDER 12 UNITS
LEVEL 4    ------> ORDER 6 UNITS
LEVEL 5    ------> ORDER 6 UNITS
LEVEL 6    ------> ORDER 0 UNITS
LEVEL 7    ------> ORDER 0 UNITS
...
LEVEL 14   ------> ORDER 0 UNITS
LEVEL 15   ------> ORDER 0 UNITS
```

Chapter V

Conclusions

Tools for modeling and optimizing stochastic environments are scarce. The generic framework presented is able to model a general type of MDP. It permits a structured modeling process by including abstract methods that oblige to implement the key elements that characterize the problem. The ease of modeling is the main concern and is supported on independent element identification to characterize a problem using OOP. The introduction of events permits an easier and structured procedure for modeling. The framework also includes a state exploration algorithm in order to facilitate the modeling process.

The framework allows modeling finite and infinite horizon problems independently with various optimization criteria. Deterministic dynamic programming problems are solved as DTMDPs with trivial probability transitions. Continuous time problems are transformed into DTMDPs for their solution. The inclusion of events is handled by transforming the problem into the same type of problem but without events, and then applying the solving procedure for such.

The framework is flexible enough to handle new data structures and new solution algorithms can be included. Some users can focus on modeling without distracting on solving procedures, but other advanced users can tune up any of the included algorithms, or build their own. The contribution of the present work is the introduction of an object oriented framework, that is generic, and is focused on modeling and uses events for modeling.

There are still many aspects to develop and others to research. The development of an interface that interacts with linear programming solvers and MPS formats is under development. There are pleanty state reduction techniques [40, 44, 47, 48] that should be implemented in order to solve complex systems. The development of efficient linear systems solvers is important in order to be able to compute the steady state probabilities of the system under some specific policy. In that case, a solution would be characterized by a policy, a value function and some measures of performance like the effective rate of the system, a throughput, queue lengths, average waiting time, etc. Professor Takahashi, juror of this work, suggested the `State`, `Action`, and `Event` classes to be changed to interfaces.

As mentioned before, the curse of dimensionality causes the state space to grow very fast. In that case it would be useful to use the implementation of `Serializable` in `TreeSet` that is inside each object `States`, in order to use hard disk memory to save the set of part of the set of states.

In terms of research, the problems of dynamic queuing control were studied and were handled with a model where actions depend on the event. This model should be studied closer and compared with the classical models used in queuing control. Further research should be done around the use of events in MDPs.

Appendix A

User's Manual

A.1 Introduction

The present document is the user's manual of Java Markov Decision Process Package (JMDP). It is an object oriented framework designed to model dynamic programming problems (DP) and Markov Decision Processes (MDPs).

A.2 Java and Object Oriented Programming

Java is a relatively new publicly available language c by Sun Microsystems. The main characteristics that Sun intended to have in Java are:

- Object-Oriented.
- Robust.
- Secure.
- Architecture Neutral
- Portable
- High Performance
- Interpreted
- Threaded
- Dynamic

Object Oriented Programming (OOP) is not a new idea. However it has not have an increased development until recently. OOP is based on four key principles:

- abstraction.

- encapsulation

- inheritance

- polymorphism

An excellent explanation of OOP and the Java programming language can be found in [7].

The abstraction capability is the one that interest us most. Java allows us to define abstract types like Actions, States, etc. We also define abstract functions like immediateCost(). We can program the algorithm in terms of this abstract objects and functions, creating a flexible tool. This tool can be used to define and solve DP problems. All the user has to do is to *implement* the abstract functions. What it is particularly nice is that if a function is declared as abstract, then the compiler itself will require the user to implement it before attempting to run the model.

A.3 Markov Decision Process - The Mathematical Model

The general problems that can be modeled and solved with the present framework can be classified in finite or infinite horizon problems. In any of these cases, the problem can be deterministic or stochastic. See Figure 3.

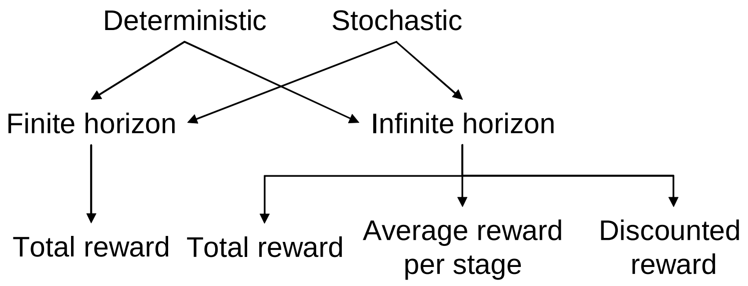


Figure 3: Taxonomy for MDP problems

The deterministic problems are known as Dynamic Programming problems, and the stochastic problems are commonly called MDPs.

A.3.1 Finite Horizon Problems

We will show how a Markov Decision Process is built. Consider a discrete space, discrete time, bivariate random process $\{(X_t, A_t), t = 0, 1, \ldots, T\}$. Each of the $X_t \in \mathcal{S}_t$ represents the state of the system at stage t, and each $A_t \in \mathcal{A}_t$ is the action taken at that stage. The quantity $T < \infty$ is called the *horizon* of the problem. The sets $\mathcal{S}_t$ and $\mathcal{A}_t$ are called the space state and the action space, and represent the states and actions available at stage t; we will assume that both are finite. The dynamics of the system are defined by two elements. First, we assume the system has the following Markov property

$$\begin{aligned} &P\{X_{t+1} = j | X_t = i, A_t = a\} \\ &\qquad = P\{X_{t+1} = j | X_t = i, A_t = a, X_{t-1} = i_{t-1}, A_{t-1} = a_{t-1}, \ldots, X_0 = i_0\}. \end{aligned}$$

We call $p_{ijt}(a) = P\{X_{t+1} = j | X_t = i, A_t = a\}$ the *transition probabilities*. Next, actions are taken when a state is realized. In general the action taken depends on the *history* of the process up to time t, i.e. $H_t = (X_0, A_0, X_1, A_1, \ldots, X_{t-1}, A_{t-1}, X_t)$. A *decision rule* is a function π_t that given a history realization assign a probability distributions over the set $\mathcal{A}$. A sequence of decision rules $\pi = (\pi_0, \pi_1, \ldots, \pi_T)$ is called a *policy*. We call Π is the set of all policies. A policy is called Markov if given X_t all previous history becomes irrelevant, that is

$$P_\pi\{A_t = a | X_t = i, A_{t-1} = a_{t-1}, X_{t-1} = i_{t-1}, \ldots\} = P_\pi\{A_t = a | X_t = i\},$$

where we use $P_\pi\{\cdot\}$ to denote the probability measure (on events defined by (X_t, A_t)) induced by π. A Markov policy is called *stationary* if for all $t = 0, 1, \ldots, i \in \mathcal{S}, a \in \mathcal{A}$,

$$P_\pi(A_t = a | X_t = i) = P_\pi(A_0 = a | X_0 = i).$$

Notice that a stationary policy is completely determined by a single decision rule, and we have $\pi = (\pi_0, \pi_0, \pi_0, \ldots)$. A Markov policy is called *deterministic* if there is a function $f_t(i) \in \mathcal{A}$ such that

$$P\{A_t = a | X_t = i\} = \begin{cases} 1 & \text{if } a = f_t(i) \\ 0 & \text{otherwise.} \end{cases}$$

For each action a taken from state i at stage t, a finite cost $c_t(i, a)$ is obtained. Consequently it is possible to define a total expected cost $v_t^\pi(i)$ obtained from time t to the final stage

T following policy π; this is called the *value function*

$$v_t^{\pi}(i) = E_{\pi}\left[\sum_{s=t}^{T} r_s(X_s, A_s)\Big| X_t = i\right], \quad i \in \mathcal{S}_0 \tag{5}$$

where E_{π} is the expectation operator following the probability distribution associated with policy π. The problem is to find the policy $\pi \in \Pi$, that minimizes the objective function shown above.

$$v_t^*(i) = \inf_{\pi \in \Pi} v_t^{\pi}(i).$$

Such optimal value function has to satisfy *Bellman's optimality equation*

$$v_t^*(i) = \min_{a \in \mathcal{A}_t(i)} \left\{ c_t(i,a) + \sum_{j \in \mathcal{S}_t(i,a)} p_{ijt}(a) v_{t+1}^*(j) \right\}, \quad i \in \mathcal{S},\ t = 0, 1, \ldots, T-1. \tag{6}$$

where $\mathcal{A}_t(i)$ is the set of *feasible actions* that can be taken from state i at stage t and $\mathcal{S}_t(i,a)$ is the set of reachable states from state i taking action a at stage t. Observe that equation (6) implies an algorithm to solve the optimal value function, and consequently the optimal policy. It starts from some final values of $v_T(i)$ and solves backward the optimal decisions for the other stages. Since the action space $\mathcal{A}_t$ is finite, the Bellman equation shows that it is possible to find a deterministic decision rule $f_t(i)$ (and hence a deterministic policy) that is optimal, by choosing in every stage in every state the action that minimizes the right hand side (breaking ties arbitrarily).

A.3.2 Infinite Horizon Problems

Consider a discrete space discrete time bivariate random process $\{(X_t, A_t), t \in \mathbb{N}\}$. Notice the time horizon is now infinite. Solving a general problem like this is difficult unless we make some assumptions about the regularity of the system. In particular we will assume that the the system is *time homogeneous*, this means that at every stage the space state and action space remain constant and the transition probabilities are independent of time $p_{ijt}(a) = p_{ij}(a) = P\{X_{t+1} = j | X_t = i, A_t = a\}$ for all $t = 0, 1, \ldots$. Costs are also time homogeneous so $c_t(i,a) = c(i,a)$ stands for the cost incurred when action a is taken from state i. However it is customary to define two objective functions, besides total cost: discounted cost, and average cost. We will explain these three problems in the next subsections

A.3.2.1 Discounted cost

In the discounted cost problem the costs in the first stages are more important than the further ones. In particular, a cost incurred at time t is assumed to have a present value $\alpha^t c(i,a)$, where $0 < \alpha < 1$ is a discount factor. If the interest per period is r then $\alpha = 1/(1+r)$. The total expected discounted cost gives rise to a value function under policy π defined as

$$v_\alpha^\pi(i) = E_\pi\left[\sum_{t=0}^{\infty} \alpha^t r(X_t, A_t)\Big| X_0 = i\right], \quad i \in \mathcal{S} \tag{7}$$

In this case, the optimal value function is

$$v_\alpha^*(i) = \inf_{\pi\in\Pi} v_\alpha^\pi(i),$$

and it can be shown that it satisfies the following Bellman's optimality equation

$$v_\alpha^*(i) = \min_{a\in\mathcal{A}(i)}\left\{c(i,a) + \alpha \sum_{j\in\mathcal{S}(i,a)} p_{ij}(a) v_\alpha^*(j)\right\}, \quad i \in \mathcal{S} \tag{8}$$

where $\mathcal{A}(i)$ is the set of feasible actions from state i in any stage and $\mathcal{S}(i,a)$ is the set of reachable states. Notice that since t does not appear in the equation it is possible to find an stationary policy that is optimal.

There are various algorithms for solving the discounted cost problem. One of them is almost implicit in equation (8). The algorithm is called *Value Iteration* and begins with some initial values $v_\alpha^{(0)}(i)$ and iteratively defines the n-th iteration value function $v_\alpha^{(n)}(i)$ in terms of $v_\alpha^{(n-1)}(i)$ according to

$$v_\alpha^{(n)}(i) = \min_{a\in\mathcal{A}(i)}\left\{c(i,a) + \alpha \sum_{j\in\mathcal{S}(i,a)} p_{ij}(a) v_\alpha^{(n-1)}(j)\right\}, \quad i \in \mathcal{S}.$$

It can be shown that for $0 < \alpha < 1$ the algorithm converges regardless of the initial function. For further details see Bertsekas[2] or Stidham[6]. If the algorithm has stopped after N iterations , then the recommended policy will be

$$f(i) = \operatorname*{argmin}_{a\in\mathcal{A}(i)}\left\{c(i,a) + \alpha \sum_{j\in\mathcal{S}(i,a)} p_{ij}(a) v_\alpha^{(N)}(j)\right\}, \quad i \in \mathcal{S}.$$

A policy is said to be *ϵ-optimal* if its corresponding value function satisfies $\min|v_\beta(i) - v^*(i)| < \epsilon$. If the previous algorithm stops when $\min|v_\alpha^{(n)}(i) - v_\alpha^{(n-1)}(i)| < \epsilon(1-\alpha)/(2\alpha))$ then it can be shown that the stationary policy $\pi = (f, f, \ldots)$ is ϵ-optimal.

The *Policy Iteration algorithm* starts with a deterministic policy $f(i)$ and through a series of iterations find improving policies. In every iteration for a given policy $f(i)$ its corresponding value function is computed solving the following linear system

$$v^f(i) = r(i, f(i)) + \alpha \sum_{j \in \mathcal{S}(i,f(i))} p_{ij}(f(i)) v^f(j), \quad i \in \mathcal{S}, \tag{9}$$

where $v^f(i)$ if the total expected discounted cost under the deterministic stationary policy $\pi = \{f, f, f, \ldots\}$. A new policy f' is found through the following policy-improvement step

$$f'(i) = \operatorname*{argmin}_{a \in \mathcal{A}(i)} \left\{ r(i, f(i)) + \alpha \sum_{j \in \mathcal{S}(i,a)} p_{ij}(f(i)) v^f(j) \right\}, \quad i \in \mathcal{S}.$$

After a succession of value computation and policy improvement steps the algorithm stops when no further improvement can be obtained. This guarantees an optimal solution instead of an ϵ-optimal one, but can be very time consuming to solve the systems. The discounted cost problem can also be solved with a linear program. See [6] for details.

A.3.2.2 Total Cost

The value function in the total cost case is given by

$$v^\pi(i) = E_\pi \left[\sum_{t=0}^{\infty} r(X_t, A_t) \Big| X_0 = i \right], \quad i \in \mathcal{S}$$

and the optimal value function is

$$v^*(i) = \inf_{\pi \in \Pi} v^\pi(i)$$

The total cost problem can be thought of as a discounted cost with $\alpha = 1$. However the algorithms presented do not work in this case. The policy evaluation in the policy iteration algorithm fails since the linear system (9) is always singular; and there is no guarantee that the value iteration algorithm converges unless we impose some additional condition. This is due to the fact that the total cost might be infinite. One of the conditions is to assume that there exists an absorbing state with zero-cost and that every policy eventually reaches it. (Weaker conditions can also be used, see [2]). This problem is also called the Stochastic Shortest Path problem, since since if the costs are negative then the minimum expected total cost can be thought of as the minimal expected cost accumulated before absorption in a Graph with random costs.

A.3.2.3 Average cost

In an ergodic chain that reaches stable state, the steady state probabilities are independent of the initial state of the system. Intuitively, the average cost per stage should be a constant regardless of the initial state. So the value function is

$$\overline{v}^{\pi}(i) = \lim_{T\to\infty} \frac{1}{T} E_{\pi}\left[\sum_{t=0}^{T} r(X_t, A_t)\Big| X_0 = i\right], \quad i \in \mathcal{S}$$

and the optimal value function is the same for every step

$$g = \overline{v}^{*}(i) = \inf_{\pi\in\Pi} \overline{v}^{\pi}(i)$$

The average cost per stage problem can be obtained by solving the following linear program

$$\begin{aligned} \text{g} = \min_{x_{ia}} \quad & \sum_{i\in\mathcal{S}} \sum_{a\in\mathcal{A}(i)} c(i,a)x_{ia} && \text{(10a)} \\ \text{s.t.} \quad & \sum_{a\in\mathcal{A}(j)} \sum_{i\in\mathcal{S}(j,a)} p_{ij}(a)x_{ia} = \sum_{a\in\mathcal{A}(i)} x_{ja} \quad j \in \mathcal{S} && \text{(10b)} \\ \text{and} \quad & \sum_{i\in\mathcal{S}} \sum_{a\in\mathcal{A}(i)} x_{ia} = 1 && \text{(10c)} \end{aligned}$$

where the solution is interpreted as

$$x_{ia} = \lim_{t\to 0} P\{X_t = i, A_t = a\} \qquad i \in \mathcal{S}, a \in \mathcal{A}(i).$$

The equation (10a) is the average cost per transition in steady state, (10b) are analogous to the balance equations in every markovian system and (10c) is the normalization condition. The optimal policy can be obtained after the LP has been solved as

$$\pi_i(a) = P\{A_t = a | X_t = i\} = \frac{x_{ia}}{\sum_{b\in\mathcal{A}(i)} x_{ib}}. \qquad i \in \mathcal{S}, a \in \mathcal{A}(i)$$

It can be shown that for every $i \in \mathcal{S}$ the is only one $a \in \mathcal{A}(i)$ that is positive, so the optimal policy is always deterministic. There is also an iterative solution based on a modification of the value iteration algorithm. See [5] for details.

Remark 1 *It may seem to the reader that the infinite horizon admits more type of cost functions that the finite counterpart. That is not the case. The fact that the cost function depends on* t*, allows us to define a discounted cost as* $c_t(i,a) = \alpha^t c(i,a)$*, and an average cost as* $c_t(i,a) = \frac{1}{T}c(i,a)$*.*

A.3.3 Deterministic Dynamic Programming

This is a particular case of the finite horizon problem defined earlier. When the set of reachable states $\mathcal{S}_t(i,a)$ has only one state for all $t \in \mathbb{N}$, $i \in \mathcal{S}$, $a \in \mathcal{A}$, then it is clear that all the probability of reaching this state has to be 1.0, and 0 for every other state. This would be a deterministic transition. So it is possible to define a transition function $h : \mathcal{S} \times \mathcal{A} \times \mathbb{N} \to \mathcal{S}$, that assigns to each state and action to be taken at the given stage, a unique destination state. Under this conditions, the Bellman equation would look like

$$v_t(i) = \min_{a \in \mathcal{A}_t(i)} \left\{ c_t(i,a) + v_{t+1}\big(h(i,a,t)\big) \right\}, \quad i \in \mathcal{S}, t \in \mathbb{N}.$$

Naturally, there are also infinite horizon counterparts as in the probabilistic case.

A.3.4 Main modeling elements in MDP

Recall the Bellman equation (6). As explained before, X_t and A_t are the state and the action taken at stage t respectively. The set $\mathcal{A}_t(i)$ is the set of actions that can be taken from state i at stage t. So the optimal action is selected only from this feasible action set, for the statement to make sense. In the equation, the first cost is taken, and then it is added to the expected future value function.

The expected future value function is a sum over the states in $\mathcal{S}_t(i,a)$. This is the set of reachable states from state i given that action a is taken at stage t. If this set was not defined, then the sum would be over all the possible states $\mathcal{S}$, and its value would be the same, only that there would be many probabilities equal to zero.

As a summary, if the elements in Table 2 are clearly identified, then it is possible to say that the Markov Decision Process has been defined.

Element	Mathematical representation
States	$X_t \in \mathcal{S}$
Actions	$A_t \in \mathcal{A}$
Feasible actions	$\mathcal{A}_t(i)$
Reachable states	$\mathcal{S}_t(i,a)$
Transition probabilities	$p_{ijt}(a)$
Costs	$c_t(i,a)$

Table 2: Main elements

A.4 Framework Design

As stated before, the intention is to make this framework as easy to use as possible. An analogy is stated between the mathematical elements presented above and the computational elements that will be explained. There is first a general overview of the framework, and specific details of each structure will be presented afterwards. This first part should be enough to understand the examples.

The framework is divided in two packages. The modeling package is called jmdp, and the solving package is jmpd.solvers. The user does not need to interact with this second one, because a standard solver is defines for every type of problem. However, as the user gains experience he mights want to fine-tune the solvers or even define his/her own solver by using the package jmdp.solvers.

The following steps will show how to model a problem. An inventory problem will be used.

1. **Defining the states**. The first thing to do when modeling a problem, is to define which will be the states. Each state X_t is represented by an object or class, and the user must modify the attributes to satisfy the needs of each problem. The class State is declared abstract and can not be used explicitly; the user must extend class State and define his own state for each type of problem. Once each state has been defined, a set of states $\mathcal{S}$ can be defined with the class States. For example, in an inventory problem, the states are inventory levels. The following file defines such a class. It has a constructor, and, very important implemente compareTo() to establish a total ordering among the states. If no comparator is provided, then the sorting will be made according to the name, which might be very inefficient in real problems.

2. **Defining the actions**.The next step is to define the actions of the problem. Again, each action A_t is represented by an object called Action, and this is an abstract class that must be extended in order to use it. In an inventory problem, the actions that can be taken from each state are orders placed.

3. **Defining the problem**. In some way, the states and actions are independent of the problem itself. The rest of the modeling corresponds to the problem's structure that is also represented by an object. In this case, the object is more complex than the ones defined earlier, but it combines the important aspects of the problem. The

classes that represent the problem are also abstract classes and must be extended in order to be used. See table (3) for reference on which class to extend for each type of problem.

Type of Problem	Class to be extended
Finite Horizon Dynamic Programming Problem	FiniteDP<S,A>
Infinite Horizon Dynamic Programming Problem	InfiniteDP<S,A>[1]
Finite Horizon MDP	FiniteMDP<S,A>
Infinite Horizon MDP	InifiniteMDP<S,A>

Table 3: Types of Problems

```
public class InventoryProblem extends FiniteMDP<InvLevel,Order>{
...
}
```

Once one of these classes is extended in a blank editor file, compilation errors will prompt up. This doesn't mean the user has done anything wrong, it is just a way to make sure all the requisites are fulfilled before solving the problem. Java has a feature called generics that allows safe type transitions. In the examples, whenever S is used, it stands for S **extends** State that is the class being used to represent a state. In the same way A is the representation of A **extends** Action. In the inventory example, class FiniteMDP<S,A> will be extended and the editor will indicate the user that there are compilation errors because some methods have not yet been implemented. This means the user must implement this methods in order to model the problem, and also for the program to compile. It is necessary that the state and the action that were defined earlier are indicated in the field <S,A> as state and action as shown in the example. This will allow the methods to know that this class is using these two as states and actions respectively.

4. **Feasible actions**. The first of these methods is **public** Actions getActions(S i, **int** t). For a given state i this method must return the set of feasible actions $\mathcal{A}(i)$ that can be taken at stage t. Notice that the declaration of the method takes element i as of type S but in the concrete example, the compiler knows the states that are being used are called InvLevel and so changes the type.

```
public Actions getActions(InvLevel i, int t){
  Actions<Order> actionSet = new ActionsCollection<Order>();
  for(int n=0; n<=K-i.level; n++){
    actionSet.add(new Order(n));
  }
  return actionSet;
}
```

The example procedure returns the actions corresponding to the set $\{0, 1, \ldots, K-i\}$, the user can declare an empty set called actionSet of type ActionsCollection<Order>, which is an easy-to-use extension of Actions<A>. The generics use is indicating that the set will store objects of type Order. Then for each iteration of the for cycle, create a new order and this new action is added to the set. After adding all the actions needed, the method returns the set of actions.

5. **Reachable states**. The second method in the class FiniteMDP<S,A> that must be implemented **public** States reachable(S i, A a, **int** t) indicates the set of reachable states $\mathcal{S}_t(i, a)$ from state i and given that action a is taken at stage t . The example shows how to define the set of states $\{0, 1, \ldots, a + i\}$. First declare an empty set called statesSet of type StatesCollection<InvLevel> which is an easy-to-use extension of States<S>, that indicates this set will store objects of type InvLevel. Then a for cycle adds a state for each value between 0 and $a + i$.

```
public States reachable(InvLevel i, Order a) {
  States<InvLevel> statesSet = new StatesCollection<InvLevel>();
  for(int n=0;n<=a.size+i.level;n++)
    statesSet.add(new InvLevel(n));
  return statesSet;
}
```

6. **Transition Probabilities**. The method **public double** prob(S i, S j, A a) is still pending to be implemented and represents the transition probabilities $p_{ijt}(a)$.

7. **Costs**. The last method is the one representing costs $c_t(i, a)$ incurred when taking action a from state i represented by the method **public double** immediateCost(S i, A a). Once these methods are implemented the class should compile.

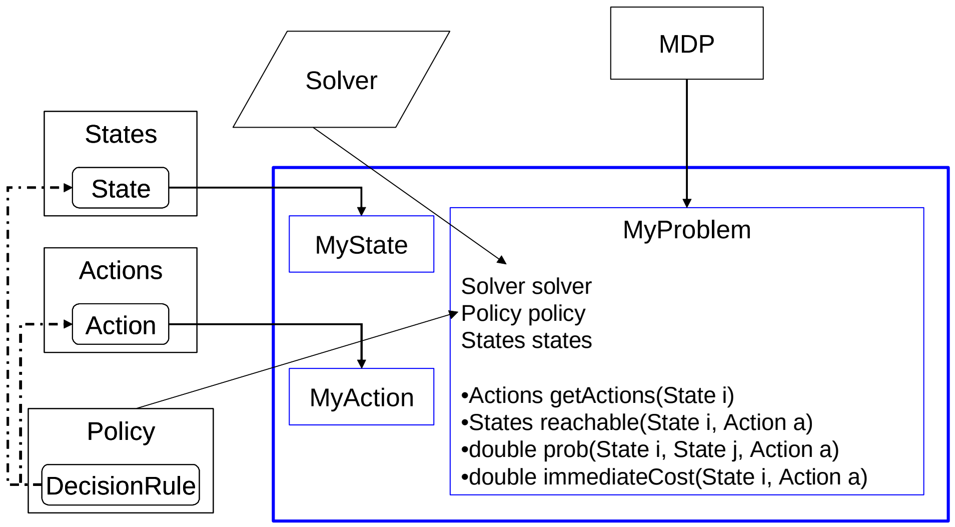
8. **The main method**. In order to test the model and solve it, the class may also have a main method. This is of course not necessary, since the class can be called from

other classes or programs provided you have been careful to declare it constructor public. The following example shows that the name of the class extending FiniteMDP is InventoryProblem so the main method must first declare an object of that type, with the necessary parameters determined in the constructor method. Then the solve() method must be called from such and the problem will call a default solver, solve the problem, store the optimal solution internally. You can obtain information about the optimal policy and value functions by calling the getOptimalPolicy() and getoptimalValue() methods. There is also a convenience method called printSolution() which prints the solution in standard output.

```
public static void main(String args[]) {

InventoryProblem prob = new InventoryProblem(maxInventory,
  maxItemsPerOrder, truckCost, holdingCost, theta);

prob.solve()
prob.printSolution()
}
```



MDP
Solver
States
State
Actions
Action
Policy
DecisionRule
MyState
MyAction
MyProblem
Solver solver
Policy policy
States states
•Actions getActions(State i)
•States reachable(State i, Action a)
•double prob(State i, State j, Action a)
•double immediateCost(State i, Action a)

Figure 4: Problem's structure.

Element	Mathematical representation	Computational representation
States	$X_t \in \mathcal{S}$	**public class** MyState **extends** State
Actions	$A_t \in \mathcal{A}$	**public class** MyAction **extends** Action
Process	$\{X_t, A_t\}$	**public class** MyProblem **extends** FiniteMDP<S,A>
Feasible actions	$\mathcal{A}_t(i)$	**public** Actions feasibleActions(S i, **int** t)
Reachable states	$\mathcal{S}_t(i,a)$	**public** States reachable(S i, A a, **int** t)
Transition probs	$p_{ijt}(a)$	**public double** prob(S i, S j, A a, **int** t)
Costs	$c_t(i,a)$	**public double** immediateCost(S i, A a, **int** t)

For details on the construction of specifc sets, modifying the solver or solver options, see the Java documentation and the Advanced Features section.

A.5 Examples

This sections shows some problems and their solution with JMDP in order to illustrate its use. The examples cover the usage of the DP, FiniteMDP, and InfiniteMDP classes.

A.5.1 Deterministic inventory problem

Consider a car dealer selling identical cars. All the orders to the distributor have to be placed on Friday eve and arrive on Monday morning before opening. The car dealer is open Monday to Friday. Each car is bought at USD \$20.000 and sold at USD\$22.000. A transporter charges a fixed fee of USD\$500 per truck for carrying the cars from the distributor to the car dealer, and each truck can carry 6 cars. The exhibit hall has space for 15 cars. If a customer orders a car and there are not cars available, the car dealer gives him the car a soon as it gets with a USD\$1000 discount. The car dealer does not allow more than 5 pending orders of this type. Holding inventory implies a cost of capital of 30% annually. The marketing department has handed in the following demand forecasts, for the next 12 weeks, shown in table (4).

	Weeks											
t	1	2	3	4	5	6	7	8	9	10	11	12
D_t	10	4	3	6	3	2	0	1	7	3	4	5

Table 4: Demand forecast.

Let's first formulate the mathematical model, and then the computational one. The parameters in the word problem are in Table 5.

K	Fixed cost per truck.
c	Unit cost.
p	Unit price.
D_t	Demand at week t.
h	Holding cost per unit per week.
b	Backorder cost.
M	Maximum exhibit hall capacity.
B	Maximum backorders allowed.
L	Truck's capacity.
T	Maximum weeks to model.

Table 5: Parameters

The problem will be solved using dynamic programming to determine the appropriate amount to order in each week in order to minimize the costs. The problem has a finite horizon and is deterministic.

1. States. Each state X_t is the inventory level at each stage t, where the stages are the weeks. When there are backorders, they will be denoted as a negative inventory level. The set of states $\mathcal{S} = \{-B, \ldots, 0, \ldots, M\}$ are all the levels between the negative maximum backorders and the maximum inventory level.

2. Actions. Each action A_t is the order placed in each stage t. The complete set of actions are the orders from 0 to the addition of the maximum exhibit hall's capacity and the maximum backorders allowed. $\mathcal{A} = \{0, \ldots, B + M\}$.

3. Feasible Actions. For each state i the feasible actions that can be taken are those that will not exceed the exhibit hall's capacity. Ordering 0 is the minimum order and is feasible in every state. The maximum order feasible is $M - i$, so the feasible set of actions for each state i is $\mathcal{A}_t(i) = \{0, \ldots, M - i\}$.

4. Destination. The destination state when action a is taken from state i is the sum of the cars in that state and the cars that are ordered, minus the cars that are sold. $h(i, a, t) = i + a - D_t$.

5. Costs. Finally, the cost incurred depends on various factors. The ordering cost is only charged when the order is positive, and charged per truck. The holding cost

is charged only when there is positive stock, and the backorder cost charged only when there is negative stock. There is finally a cost for selling each car given by the difference between price and cost.

$$OC(a) = \begin{cases} 0 & \text{if } a = 0 \\ \lceil \frac{a}{L} \rceil & \text{if } a > 0 \end{cases}$$

$$HC(i) + BC(i) = \begin{cases} ib & \text{if } i \leq 0 \\ ih & \text{if } i > 0 \end{cases}$$

$$c_t(i, a) = OC(a) + HC(i) + BC(i) + (p - c)D_t$$

Now, computationally, the file would look like this.

A.5.2 Finite horizon stochastic inventory problem

Consider the car dealer in the past example. The car dealer selling identical cars. All the orders placed to the distributor arrive on Monday morning. The car dealer is open Monday to Friday. Each car is bought at USD \$20.000 and sold at USD\$22.000. A transporter charges a fixed fee of USD\$500 per truck for carrying the cars from the distributor to the car dealer, and each truck can carry 6 cars. The exhibit hall has space for 15 cars. If a customer orders a car and there are not cars available, the car dealer gives him the car a soon as it gets with a USD\$1000 discount. The car dealer does not allow more than 5 pending orders of this type. Holding inventory implies a cost of capital of 30% annually. Now instead of receiving demand forecasts, marketing department has informed that the demand follows a Poisson process.

The parameters of the problem are shown in table 6

The problem is a finite horizon stochastic problem. Markov Decision Processes can be used in order to minimize the costs.

1. States. Each state X_t is the inventory level at each stage t, where the stages are the weeks. When there are backorders, they will be denoted as a negative inventory level. The set of states $\mathcal{S} = \{-B, \ldots, 0, \ldots, M\}$ are all the levels between the negative maximum backorders adn the maximum inventory level.

K	Fixed cost per truck.
c	Unit cost.
p	Unit price.
h	Holding cost per unit per week.
b	Backorder cost.
M	Maximum exhibit hall capacity.
B	Maximum backorders allowed.
L	Truck's capacity.
T	maximum weeks to model.
D_t	Random variable that represents the weekly demand.
θ	Demand's mean per week t.
p_n	$P\{D_t = n\}$
q_n	$P\{D_t >= n\}$

Table 6: Parameters

2. Actions. Each action A_t is the order placed in each stage t. The complete set of actions are the orders from 0 to the addition of the maximum exhibit hall's capacity and the maximum backorders allowed. $\mathcal{A} = \{0, \ldots, B + M\}$.

3. Feasible Actions. For each state i the feasible actions that can be taken are those that will not exceed the exhibit hall's capacity. Ordering 0 is the minimum order and is feasible in every state. The maximum order feasible is $M - i$, so the feasible set of actions for each state i is $\mathcal{A}_t(i) = \{0, \ldots, M - i\}$.

4. Reachable States. The minimum reachable state when action a is taken from state i would be $-B$, when the demand is maximum $(b + i)$. The maximum reachable state when action a is taken from state i is i when the demand is minimum (0). So the set of reachable states are all the states ranging between these two: $\mathcal{S}_t(i, a) = \{-B, \ldots, i\}$.

5. Cost. The costs incurred depends on various factors. The ordering cost is only charged when the order is positive, and charged per truck.

$$OC(a) = \begin{cases} 0 & \text{if } a = 0 \\ \lceil \frac{a}{L} \rceil & \text{if } a > 0 \end{cases}$$

The holding cost is charged only when there is positive stock, and the backorder cost charged only when there is negative stock.

$$HC(i) + BC(i) = \begin{cases} ib & \text{if } i \leq 0 \\ ih & \text{if } i > 0 \end{cases}$$

Finally, there is an expected lost sales cost (Using $x = i + a + B$):

$$
\begin{aligned}
E[D_t - x]^+ &= \sum_{d=x+1}^{\infty} \big(d - x\big) p_d \\
&= \sum_{d=x+1}^{\infty} d p_d - \sum_{d=x+1}^{\infty} x p_d \\
&= \sum_{d=x+1}^{\infty} d \frac{\theta^d e^{-\theta}}{d!} - x \sum_{d=x+1}^{\infty} p_d \\
&= \theta \sum_{d=x+1}^{\infty} \frac{\theta^{d-1} e^{-\theta}}{(d-1)!} - x q_{x+1} \\
&= \theta \sum_{d=x+1}^{\infty} p_{d-1} - x q_{x+1} \\
&= \theta \sum_{d=x}^{\infty} p_d - x q_{x+1} \\
&= \theta(q_x) - x(q_x - p_x) \\
&= \theta(q_x - p_x) - x q_x
\end{aligned}
$$

Now, computationally, the file would look like this.

A.5.3 Infinite horizon stochastic inventory problem

Consider the car dealer in the past example. The car dealer selling identical cars. All the orders placed to the distributor arrive on Monday morning. The car dealer is open Monday to Friday. Each car is bought at USD \$20.000 and sold at USD\$22.000. A transporter charges a fixed fee of USD\$500 per truck for carrying the cars from the distributor to the car dealer, and each truck can carry 6 cars. The exhibit hall has space for 15 cars. If a customer orders a car and there are not cars available, the car dealer gives him the car a soon as it gets with a USD\$1000 discount. The car dealer does not allow more than 5 pending orders of this type. Holding inventory implies a cost of capital of 30% annually. Now instead of receiving demand forecasts, marketing department has informed that the demand follows a Poisson process.

The parameters of the problem are shown in table (7).

The problem is a finite horizon stochastic problem. Markov Decision Processes can be used in order to minimize the costs.

K	Fixed cost per truck.
c	Unit cost .
p	Unit price.
h	Holding cost per unit per week.
b	Backorder cost.
M	Maximum exhibit hall capacity.
B	Maximum backorders allowed.
L	Truck's capacity.
D_t	Random variable that represents the weekly demand.
θ	Demand's mean per week t.
p_n	$P\{D_t = n\}$
q_n	$P\{D_t >= n\}$

Table 7: Parameters

1. States. Each state X_t is the inventory level at each stage t, where the stages are the weeks. When there are backorders, they will be denoted as a negative inventory level. The set of states $\mathcal{S} = \{-B, \ldots, 0, \ldots, M\}$ are all the levels between the negative maximum backorders adn the maximum inventory level.

2. Actions. Each action A_t is the order placed in each stage t. The complete set of actions are the orders from 0 to the addition of the maximum exhibit hall's capacity and the maximum backorders allowed. $\mathcal{A} = \{0, \ldots, B + M\}$.

3. Feasible Actions. For each state i the feasible actions that can be taken are those that will not exceed the exhibit hall's capacity. Ordering 0 is the minimum order and is feasible in every state. The maximum order feasible is $M - i$, so the feasible set of actions for each state i is $\mathcal{A}_t(i) = \{0, \ldots, M - i\}$.

4. Reachable States. The minimum reachable state when action a is taken from state i would be $-B$, when the demand is maximum $(b + i)$. The maximum reachable state when action a is taken from state i is i when the demand is minimum (0). So the set of reachable states are all the states ranging between these two: $\mathcal{S}_t(i, a) = \{-B, \ldots, i\}$.

5. Costs. The cost incurred depends on various factors. The ordering cost is only charged when the order is positive, and charged per truck.

$$OC(a) = \begin{cases} 0 & \text{if } a = 0 \\ \lceil \frac{a}{L} \rceil & \text{if } a > 0 \end{cases}$$

The holding cost is charged only when there is positive stock, and the backorder cost charged only when there is negative stock.

$$HC(i) + BC(i) = \begin{cases} ib & \text{if } i \leq 0 \\ ih & \text{if } i > 0 \end{cases}$$

Finally, there is an expected lost sales cost (Using $x = i + a + B$):

$$\begin{aligned} E[D_t - x]^+ &= \sum_{d=x+1}^{\infty} (d-x)p_d \\ &= \sum_{d=x+1}^{\infty} dp_d - \sum_{d=x+1}^{\infty} xp_d \\ &= \sum_{d=x+1}^{\infty} d\frac{\theta^d e^{-\theta}}{d!} - x\sum_{d=x+1}^{\infty} p_d \\ &= \theta \sum_{d=x+1}^{\infty} \frac{\theta^{d-1}e^{-\theta}}{(d-1)!} - xq_{x+1} \\ &= \theta \sum_{d=x+1}^{\infty} p_{d-1} - xq_{x+1} \\ &= \theta \sum_{d=x}^{\infty} p_d - xq_{x+1} \\ &= \theta(q_x) - x(q_x - p_x) \\ &= q_x(\theta - x) + xp_x \end{aligned}$$

Now, computationally, the file would look like this.

A.6 Advanced Features

The sections above were intended to show an easy way to use JMDP. The package has some more features that make it more flexible and powerful than what was shown above. This section is intended for users that are already familiar with the previous sections and want to customize the framework according to their specific needs.

A.6.1 States and Actions

The **public abstract class** State **implements** Comparable<State> is declared as an abstract class. As an abstract class it may not be used directly but must be extended. Abstract classes can't be used directly and must be extended.

This class implements Comparable, which implies that objects of type State have some criterion of order. By default the order mechanism is to order the States according to the String name property. This is the most general case because allows states such as "Active" or "Busy" that don't have any numerical properties. It is not efficent to organize states in such a way because comparing Strings is very slow; but this is flexible. In many cases it will be easier to represent the system state by a vector $(i_1, i_2, \ldots, i_K)$ of integers. In this case, it is more efficient to compare states according to this vector. The class StateArray is an extension of State that has a field called **int**[] status. This class changes the Comparable implementation to order the states accorging to status. This is also an abstract class and must also be extended to be used.

When State objects have to be grouped, for example when the reachable method must return a set of reachable states, the States<S> structure is the one that handles this operation. This class is also an abstract class and implements Iterable<S>. There is no restriction on how the user can store the State objects as long as Iterable<S> is implemented and an **public void** add(S s) method is implemented. This means the user can use an array, a list, a set or any other structures. For beginner users, the class StatesCollection<S> was built to make a faster and easier way to store the State objects. The StatesCollection<S> class extends States and organizes the objects in a Set from the java.util.Collections.

It is important to use the generics in a safe mode in the States object and its extensions. Its declaration is **abstract public class** States<S **extends** State> **implements** Iterable<S>. This means that Every time a States object is declared, it must specify the type of objects stored in it. For example: States<MyState> theSet = **new** StatesCollection<MyState>(); is the right way to ensure that only objects of type MyState are stored in the object theSet. This also makes the iterator that the class returns, to iterate over MyState objects.

The behavior of class Action is completely analogous to that of class State. The class is abstract and must be extended to be used. The default criterion of ordering is alphabetical order of the name attribute. But there is an ActionArray that can have an integer array stored as properties representing the action. This objects compare themselves according to the array instead of the name allowing faster comparisons. The set of actions is called Actions<A **extends** Action> **implements** Iterable<A>. This class does not need to have the add method implemented, but works analogously to class States<S>. For simplicity, class ActionsCollection<A> stores the objects in a Set from java.util.Collections.

A.6.2 Decision Rules and Policies

The deterministic decision rules π_t as referred in the MDP mathematical model, are functions that assign a single action to each state. The computational object representing a decision rule is **public final class** DecisionRule<S **extends** State, A **extends** Action>. Probably the most common method used by a final user will be **public** A getAction(S i) which returns the Action assigned to a State. Remember the generics structure where State and Action are only abstract classes. Consider the following example where the method's result is stored in a: MyAction a = myDecisionR.getAction(**new** MyState(s));, where only extensions of State and Action are being used.

Non stationary problems that handle various stages use a policy $\pi = (\pi_1, \pi_2, \ldots, \pi_T)$ that is represented by **public final class** Policy<S **extends** State, A **extends** Action>. A Policy stores a DecisionRule for each stage. It may be useful to get the action assigned to a state in a particular stage using the method **public** A getAction(S i, **int** t) that used with generics could look like this: MyAction a = pol.getAction(**new** Mystate(s), 0); where again State and Action are only abstract classes that are not used explicitly.

A.6.3 MDP class

The MDP class is the essence of the problem modeling. This class is extended in order to represent a Markov decision process or a dynamic programming problem. For each type of problem, a different extension of class MDP must be extended (See table 3). Remember always to indicate the name of the objects that represent the states and the actions extending State and Action respectively; these are indicated as <S> and <A> in the class declaration.

After declaring a new **public class** MyProblem **extends** FiniteMDP<MyState,MyAction>, various compilation errors pop up. This doesn't mean that something was done wrong, it is just to remember the user that some methods must be implemented for the problem to be completely modeled. A summary of the methods is shown on table (8).

A.6.4 Solver classes

The Solver class is a very general abstract class. It requires the implementing class to have a **public void** solve() method that reaches a policy that is optimal for the desired problem, and stores this policy in the Policy <S,A> policy field inside the problem. The current package has a dynamic programming solver called FiniteSolver, a value iteration solver and

Class	Abstract Methods
FiniteDP<S,A>	**public abstract** Actions<A> getActions(S i, **int** t) **public abstract** S destination(S i, A a, **int** t) **public abstract double** immediateCost(S i, A a, **int** t)
FiniteMDP<S,A>	**public abstract** Actions<A> getActions(S i, **int** t) **public abstract** States<S> reachable(S i, A a, **int** t) **public abstract double** prob(S i, S j, A a, **int** t) **public abstract double** immediateCost(S i, A a, **int** t)
InfiniteMDP<S,A>	**public abstract** Actions<A> getActions(S i) **public abstract** States<S> reachable(S i, A a) **public abstract double** prob(S i, S j, A a) **public abstract double** immediateCost(S i, A a)

Table 8: Abstract methods.

a policy iteration solver. The three of them have convenience methods printSolution() that allow the user to print the solution in standard output or to a given PrintWriter. For larger models the user might not want to see the solution in the screen, but rather extract all the information through getOptimalPolicy(), and getOptimalValueFunction() methods.

A.6.4.1 FiniteSolver

public class FiniteSolver<S **extends** State, A **extends** Action> **extends** AbstractFiniteSolver

This solver is intended to solve only finite horizon problems. The constructors only receive problems modeled with FiniteMDP (or FiniteDP) classes, implying that only finite horizon problems can be solved. The objective function is to minimize the total cost presented in equation (5), in the mathematical model.

A.6.4.2 ValueIterationSolver

public class ValueIterationSolver<S **extends** State, A **extends** Action> **implements** Solver.

This solver minimizes the discounted cost v_α^π presented in equation (7) on the mathematical model. The constructor only receives problems of type InfiniteMDP<S,A>, which implies the class in only intended to solve infinite horizon, discounted problems.

The algorithm used to solve the problem is the value iteration algorithm that consists on applying the transformation described on equation (8) repeatedly until the results are ϵ apart. It can be proved (see Stidham[6]) that the result will be ϵ-optimal. The value functions start in 0.0 by default, but it can be changed using **public void** setInitVal(**double** val), and this may speed up the convergence of the algorithm. The ϵ is also an important criterion for the speed convergence and may be changed from its default value in 0.0001, using **public void** setEpsilon(**double** epsilon); a bigger ϵ will speed up convergence but will make the approximation less accurate.

The Gauss-Seidel modification presented by Bertsekas[2] is used by default and may be deactivated using **public void** setGaussSeidel(**boolean** val). This modification will cause the algorithm to make less iterations because the value function $v(i)$ is changing faster than without the modification. It is also possible to activate the Error Bounds modification presented by Bertsekas[2], that is deactivated by default. This modification changes the stopping criterion and makes each iteration faster.

Finally, it is possible to print the final value function for each state on screen using the **public void** setPrintValueFunction(**boolean** val) method. In some cases, for comparison purposes, it may be useful to be able to see the time it took the algorithm to solve the problem by activating **public void** setPrintProcessTime(**boolean** val). The two last options are deactivated by default.

A.6.4.3 PolicyIterationSolver

The **public class** PolicyIterationSolver is also designed to solve only infinite horizon problems and this is restricted in its constructor that only receives InfiniteMDP<S,A> objects as problem parameter. This solver minimizes the discounted cost ${}_{DR}v^{\pi}$ presented in equation (7) on the mathematical model. The solver uses the policy iteration algorithm. This algorithm has a step in which a linear system of equation needs to be solved, so the JMP[3] package is used. This class also allows to print the final value function for each state on screen using the **public void** setPrintValueFunction(**boolean** val) method. The solving time can be shown by activating **public void** setPrintProcessTime(**boolean** val). These two last options are deactivated by default.

A.7 Further Development

This project is currently under development, and therefore we appreciate all the feedback we can receive. We plan to extend this package so that it is able to solve infinite-horizon models. Also, we will provide more internal routines to store the data, so that decision can be taken on the fly whether to use main or secondary memory.

References

[1] Bellman, Richard. *Dynamic Programming.* Princeton, New Jersey: Princeton University Press, 1957.

[2] Bertsekas, Dimitri. *Dynamic Programming and Optimal Control* Belmont, Massachusetts: Athena Scientific, 1995.

[3] Bjorn-Ove, Heinsund. *JMP-Sparce Matrix Library in Java*, Department of Mathematics, University of Bergen, Norway, September 2003.

[4] Ciardo, Gianfranco. *Tools for Formulating Markov Models* in "Computational Probability" edited by Winfried Grassman. Kluwer Academic Publishers, USA, 2000.

[5] Puterman, Martin. *Markov Decision Processes.* John Wiley & Sons Inc.

[6] Stidham, J. *Optimal Control of Markov Chains* in "Computational Probability" edited by Winfried Grassman. Kluwer Academic Publishers, USA, 2000.

[7] Van der Linden, Peter. *Just Java.* The sunsoft Press. 1996

Appendix B

Unhanded work

This appendix shows some of the work that was not included explicitly in the thesis work. The first section shows another model with events where the events depend on actions. This model was programmed and the classes are available. The second section presents a summary of the technical features included in the program.

B.1 Event Dependent Actions

It is relevant to discuss the second model, in which the actions taken can depend on the event that occurs. This is a common case that arises, for instance when no preemption is desired in a queuing system: actions can be taken about the servers only these are free but it is not possible to interrupt service, so actions can be taken upon departure and not when an arrival occurs. Originally, in the finite horizon problem $\{(X_t, A_t, E_t),\, t = 0, 1, \ldots, T-1\}$ with event dependent actions, a deterministic decision rule $\pi_t : \mathcal{S}_t \to \mathcal{A}_t$ was defined as a function assigning an action to a state. If actions also depend on events, it would be necessary to redefine a function such that $\pi_t : (\mathcal{S}_t \times \mathcal{E}_t) \to \mathcal{A}_t$. Consider the following alternate approach: define a new state $i' \in (\mathcal{S}_t \times \mathcal{E}_t) = \mathcal{S}'_t$, and include the information about the occurring event inside the state, where the deterministic decision rule $\pi_t : \mathcal{S}'_t \to \mathcal{A}_t$. Consequently $\big|\big|\mathcal{E}_t(i')\big|\big| = 1$ if $i' \in \mathcal{S}'_t$. Consider $i' = (i, e_1)$, $j' = (j, e_2)$, and $\mathcal{E}_t(i)$ is the set of active events from state i at stage t (does not depend on the action) then

$$\begin{aligned}
\mathcal{S}'_t(i', a) &= \bigcup_{s \in \mathcal{S}_t(i,a,e_1)} \{s\} \times \mathcal{E}_{t+1}(s), \quad i' \in \mathcal{S}'_t,\, a \in \mathcal{A},\, t = 0, 1, \ldots, T-1, \\
p_{i'j't}(a) &= p_{t+1}(e_2|j, a) p_{ijt}(a, e_1), \quad i, j \in \mathcal{S},\, a \in \mathcal{A},\, t = 0, 1, \ldots, T-1, \\
c_t(i', a) &= p_t(e_1|i, a) c_t(i, a, e_1), \quad i' \in \mathcal{S}'_t,\, a \in \mathcal{A},\, t = 0, 1, \ldots, T-1.
\end{aligned}$$

The characteristic elements are $\mathcal{E}_t(i)$, $\mathcal{A}_t(i, e)$, $\mathcal{S}_t(i, a, e)$, $p_{ijt}(a, e)$, $p_t(e|i)$, and $c_t(i, a, e)$.

The discrete time finite horizon problem with event dependent actions is equivalent to a discrete time finite problem without events, for which the optimization criteria and optimality conditions formulated in section 2.1 still apply. Mathematically, this is a simple solution to handle events, but the state expansion is computationally inefficient.

An analogous transformation is valid in the infinite horizon case. Consider $i' = (i, e_1)$, $j' = (j, e_2)$, and $\mathcal{E}(i)$ is the set of active events from state i at every stage then

$$\begin{aligned}
\mathcal{S}'(i', a) &= \bigcup_{s \in \mathcal{S}(i,a,e_1)} \{s\} \times \mathcal{E}(s), \quad i' \in \mathcal{S}',\, a \in \mathcal{A}, \\
p_{i'j'}(a) &= p(e_2|j, a) p_{ij}(a, e_1), \quad i', j' \in \mathcal{S}',\, a \in \mathcal{A}, \\
c(i', a) &= p(e_1|i, a) c(i, a, e_1), \quad i' \in \mathcal{S}',\, a \in \mathcal{A}.
\end{aligned}$$

The elements that completely characterize a problem of this type are $\mathcal{E}(i)$, $\mathcal{A}(i, e)$, $\mathcal{S}(i, a, e)$, $p_{ij}(a, e)$, $p(e|i, a)$, and $c(i, a, e)$. In the continuous time problem,

$$\begin{aligned}
\mathcal{S}'(i', a) &= \bigcup_{s \in \mathcal{S}(i,a,e_1)} \{s\} \times \mathcal{E}(s), \quad i' \in \mathcal{S}',\, a \in \mathcal{A}, \\
\lambda_{i'j'}(a) &= p(e_2|j, a) \lambda_{ij}(a, e_1), \quad i', j' \in \mathcal{S}',\, a \in \mathcal{A}, \\
\tilde{c}(i', a) &= p(e_1|i, a) \tilde{c}(i, a, e_1), \quad i' \in \mathcal{S}',\, a \in \mathcal{A}, \\
\gamma(i', a) &= p(e_1|i, a) \gamma(i, a, e_1), \quad i' \in \mathcal{S}',\, a \in \mathcal{A},
\end{aligned}$$

where $p(e|i, a)$ has to be calculated as in (4).

The elements that completely characterize a problem of this type are $\mathcal{E}(i)$, $\mathcal{A}(i, e)$, $\mathcal{S}(i, a, e)$, $\lambda_{ij}(a, e)$, $\tilde{c}(i, a, e)$, and $\gamma(i, a, e)$. These elements are enough to formulate a CTMDP with elements $\mathcal{A}(i')$,$\lambda_{i'j'}(a)$, $\mathcal{S}(i', a)$, $\tilde{c}(i', a)$, and $\gamma(i', a)$. Then, an equivalent DTMDP is formulated. Consider a queuing system with k servers. The only possible events are arrivals A and departures D. When a transition is triggered by a departure, the server that was freed can be disabled. In the case of an arrival, the only possible action is to do nothing. The system is illustrated in figure (5) where states i' are denoted by (i, k, e) where i are the entities in the system, k are the servers, and e is the only active event for that state. Notice the enormous state expansion but it is possible disable servers only upon departures.

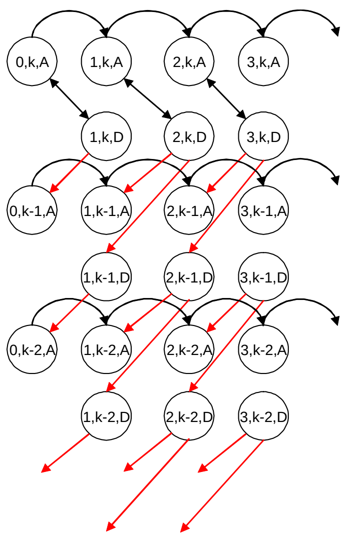


Figure 5: Queue example

B.2 Technical Features

This section describes some technical features of the present software development project.

Ubication	CVS repository of the University
Java features	JDK 1.5 Abstract classes with Generics JUnits for every example Iterators in the solvers Javadoc
Problems modeled	Finite and infinite horizon Stochastic and deterministic Discrete and continuous time
Optimization objectives	Total cost Discounted cost Average cost
Solving algorithms	Backwards finite dynamic programming Value Iteration Gauss-Seidel modification Error Bounds modification ϵ modification for convergence Policy Iteration Modified policy iteration ϵ modification for convergence Gauss-Seidel modification Relative Value Iteration Gauss-Seidel modification Error Bounds modification ϵ modification for convergence
Solution	Process time calculation Iterations report

References

[1] AMPL. Stochastic programming extensions - proposed new AMPL features. Technical report, AMPL - http://www.ampl.com/NEW/FUTURE/stoch.html, August 1996.

[2] Sigrún Andradóttir, Hayriye Ayhan, and Douglas G. Down. Server assignment policies for maximizing the steady-state throughput of finite queueing systems. *Management Science*, 47(10):1422–1439, October 2001.

[3] Christel Baier, Frank Ciesinski, and Marcus Größer. PROBMELA: a modeling language for communicating probabilistic processes. Universität Bonn, Institut fÃ$\frac{1}{4}$r Informatik I, Germany,.

[4] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear programming and network flows.* John Wiley & Sons, Inc., New York, NY, USA, 2 edition, 1990.

[5] Raphen Becker, Shlomo Zilberstein, and Victor Lesser. Decentralized markov decision processes with event-driven interactions. In *Autonomous Agents & multi Agent Systems*, New your, New Your, USA, July 2004.

[6] Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, New Jersey, 1957.

[7] Dimitri Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, Belmont, Massachusetts, 1995.

[8] John R. Birge and Franćois Louveaux. *Introduction to Stochastic Programming.* Springer, New York, 1997.

[9] Ron Boisvert and Roldan Pozo. Java numerics. Technical report, NIST, http://math.nist.gov/javanumerics/, 2003.

[10] Gilad Bracha. Generics in the java programming language. Technical report, Sun Mircrosystems, July 2004.

[11] Mary Campione, Kathy Walrath, and Alison Huml. *The Java Tutorial.* JavaSoft - Sun Microsystems, http://java.sun.com/tutorial, 2004.

[12] Christos G. Cassandras and Stéphane Lafortune. *Introduction to discrete event systems.* Kluwer Academic, Boston, 1999.

[13] Yih-Long Chang and Kiran Desai. *WinQSB, Version 2.0.* John Wiley and Sons, 2003.

[14] Gianfranco Ciardo. Tools for formulating Markov models. In Winfried K. Grassman, editor, *Computational Probability.* Kluwer's International Series in Operations Research and Management Science, Massachusetts, USA, 2000.

[15] Y. Colombani and S. Heipcke. Mosel: An overview. Dash Optimization, September 2005.

[16] Alan Dormer, Alkis Vazacopoulos, Nitin Verma, and Horia Tipi. Modeling & solving stochastic programming problems in supply chain management using Xpress-SP. Dash Optimization, Inc.

[17] Eugene A. Feinberg. Continuous time discounted jump markov decision processes: A discrete-event approach. *Mathematics of Operations Research*, 29(3):492–524, August 2004.

[18] Mark Fong. JiST user guide. Technical report, Cornell Research Foundation, September 2003.

[19] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press / Brooks/Cole Publishing Company, 2 edition, 2002.

[20] Jason H. Goto, Mark E. Lewis, and Martin L. Puterman. Coffee, tea, or...?: A markov decision process model for airline meal provisioning. *Transportation Science*, 38(1):107–118, February 2004.

[21] Bjϕrn-Ove Heimsund. Jmp-sparce matrix library in java. Technical report, Univesity of Bergen, Bergen, Norway, September 2003.

[22] Christine Julien, Joachim Hammer, and William J. O'Brien. A dynamic programming framework for pervasive computing environments. The University of Texas at Austin - Department of Electrical and Computer Engineering.

[23] Ng Yew Kwong and Teo Yong Meng. Spades/java simulation library - user manual. Technical report, National University of Singapore, http://www.comp.nus.edu.sg/ pasta/spades-java/spadesJava.html, 2001.

[24] Bernard Lamond. An interactive computer program for markov dynamic programming. Technical report, Université Laval - Départment d'operations et systémes de décision, October 1984.

[25] Conrad J. Lautenbacher and Shaler Stidham Jr. The underlying markov decision process in the single-leg airline yield-management problem. *Transportation Science*, 33(2):136–146, May 1999.

[26] Pierre L'Ecuyer. SSJ user's guide. Technical report, Université de Montréal,, http://www.iro.umontreal.ca/ simardr/ssj/, 2006.

[27] Sridhar Mahadevan, Nikfar Khaleeli, and Nicholas Marchalleck. Designing agent controllers using discrete-event markov models. Department of Computer Science, Michigan State University.

[28] Hervé Marchand, Olivier Boivineau, and Stéphane Lafortune. Optimal control of discrete event systems under partial observation. In *Proceedings of the 40th IEEE Conference on Decision and Control, CDC*, December 2001.

[29] Francesco Martinelli, Salvatore Nicosia, and Valigi Paolo. A state reconstruction algorithm for parameter dependent discrete event dynamic systems. In *5th IEEE Mediterranean Conference on Control and Systems*, Cyprus, July 1997.

[30] Bruce A. McCarl. Gams user guide: 2004. Technical report, GAMS Development Corporation, February 2004.

[31] Ross McNab. A guide to the simjava package. Technical report, University of Edinburgh, http://www.dcs.ed.ac.uk/home/hase/simjava/.

[32] S. R. Mohanty, V. Chandra, and R. Kumar. A framework for optimal control of discrete event systems. University of Kentucky - Department of Electrical and Computer Engineering.

[33] F. Murtagh. Principal components analysis on correlations. Technical report, The Queen's University of Belfast, http://astro.u-strasbg.fr/ fmurtagh/mda-sw/java/, 2002.

[34] Martin Puterman. *Markov decision processes: discrete stochastic dynamic programming.* John Wiley and Sons, New York, 1994.

[35] Germán Riaño. Dynamic programming solver dps. Technical report, Gerogia Institute of Technology, Atlanta, GA, USA, 1997.

[36] Germán Riaño. Jmarkov userÂ´s manual. Technical report, Universidad de Los Andes, Bogotá, Colombia, 2005.

[37] Paul A. Samuelson. Lifetime portafolio selection by dynamic stochastic programming. *The Review of Economics and Statistics*, 51(3):539–246, August 1969.

[38] Andrés Sarmiento and Germán Riaño. Jmdp user's manual. Technical report, Universidad de Los Andes, Bogotá, Colombia, 2005.

[39] Richard Serfozo. An equivalence between continuous and discrete time markov decision processes. *Operations Research*, 27:616–620, 1979.

[40] Theodore J. Sheskin. State reduction in markov decision process. *int. j. math. educ. sci. technol.*, 30:167–185, 1999.

[41] William J. Stewart. MARCA: Markov chain analyzer, a software package for markov modelling. Technical report, North Carolina State University - Department of Computer Science, 1996.

[42] Shaler Stidham Jr. 9: Optimal control of markov chains. In Winfried K. Grassman, editor, *Computational Probability*. Kluwer's International Series in Operations Research and Management Science, Massachusetts, USA, 2000.

[43] Janakiram Subramanian, Shaler Stidham Jr., and Conrad J. Lautenbacher. Airline yield management with overbooking, cancellations, and no-shows. *Transportation Science*, 33(2):147–167, May 1999.

[44] John N. Tsitsiklis and Benjamin Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 1996.

[45] Patrick Valente. SAMPL/SPInE user's manual. Technical report, OptiRisk Systems, Middlesex, United Kingdom, 2002.

[46] Peter van der Linden. *Just Java*. Java Series. SunSoft Press, Prentice Hall, Mountain View, California, USA, 1996.

[47] Ward Whitt. Approximations of dynamic programs, I. *Mathematics of Operations Research*, 3(3):231–243, August 1978.

[48] Ward Whitt. Approximations of dynamic programs, II. *Mathematics of Operations Research*, 4(2):179–185, May 1979.

[49] Paul Herbert Zipkin. *Foundations of Inventory Management*. Management and Organization Series. McGraw-Hill International Editions, 2000.