jMDP User's Guide

Germán Riaño and Andrés Sarmiento
Universidad de Los Andes

Contents

Introduction

Java package for Markov Decision Process Package (JMDP) is an object oriented framework designed to model dynamic programming problems (DP) and Markov Decision Processes (MDPs).

1 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 interests us most. Java allows us to define abstract types like Actions, States, etc. We also define abstract functions like immediateReward(). 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.

2 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 1.

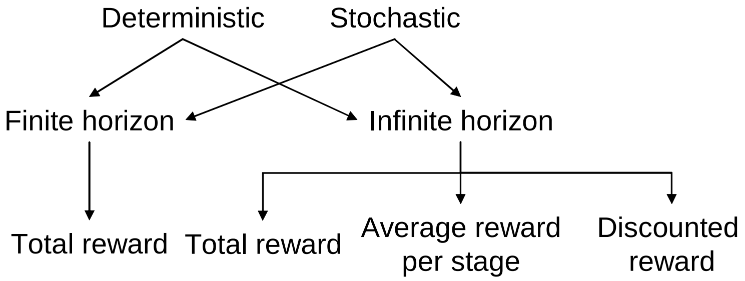

Figure 1: Taxonomy for MDP problems

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

2.1 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

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

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 reward $r_t(i, a)$ is obtained. Consequently it is possible to define a total expected reward $v_t^\pi(i)$ obtained from time t to the final stage T following policy π; this is called the *value function*

$$v_t^\pi(i) = E_\pi\left[\sum_{s=t}^{T} r_s(X_s, A_s) \middle| X_t = i\right], \quad i \in \mathcal{S}_0 \tag{1}$$

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

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

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

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

where $\mathcal{A}_t(i)$ is the set of *feasible actions* that can be taken from state i at stage t and $\mathcal{S}_t(i, a)$ is the set of reachable states from state i taking action a at stage t. Observe that equation (2) 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 maximizes the right hand side (breaking ties arbitrarily).

2.2 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 $t = 0, 1, \ldots$. Rewards are also time homogeneous so $r_t(i, a) = r(i, a)$ stands for the reward obtained when action a is taken from state i. However it is customary to define two objective functions, besides total reward: discounted reward, and average reward. We will explain these three problems in the next subsections.

2.2.1 Discounted reward

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

$$v_\alpha^\pi(i) = E_\pi\left[\sum_{t=0}^{\infty} \alpha^t r(X_t, A_t) \Big| X_0 = i\right], \quad i \in \mathcal{S} \tag{3}$$

In this case, the optimal value function is

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

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

$$v_\alpha^*(i) = \max_{a \in \mathcal{A}(i)} \left\{ r(i, a) + \alpha \sum_{j \in \mathcal{S}(i,a)} p_{ij}(a) v_\alpha^*(j) \right\}, \quad i \in \mathcal{S} \tag{4}$$

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 reward problem. One of them is almost implicit in equation (4). 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

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

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

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

A policy is said to be *ϵ-optimal* if its corresponding value function satisfies $\max |v_\beta(i) - v^*(i)| < \epsilon$. If the previous algorithm stops when $\max |v_\alpha^{(n)}(i) - v_\alpha^{(n-1)}(i)| < \epsilon(1 - \alpha)/(2\alpha))$ 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

$$v^f(i) = r(i, f(i)) + \alpha \sum_{j \in \mathcal{S}(i,f(i))} p_{ij}(f(i)) v^f(j), \quad i \in \mathcal{S}, \tag{5}$$

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

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

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 reward problem can also be solved with a linear program. See [6] for details.

2.2.2 Total Reward

The value function in the total reward 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

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

The total reward problem can be thought of as a discounted reward 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 (5) 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 reward might be infinite. One of the conditions is to assume that there exists an absorbing state with zero-reward 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 rewards are negative then the maximum expected total reward can be thought of as the minimal expected cost accumulated before absorption in a Graph with random costs.

2.2.3 Average reward

In an ergodic chain that reaches stable state, the steady state probabilities are independent of the initial state of the system. Intuitively, the average reward 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 state

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

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

$$
\begin{aligned}
\mathrm{g} = \max_{x_{ia}} \quad & \sum_{i\in\mathcal{S}} \sum_{a\in\mathcal{A}(i)} r(i,a)x_{ia} && \text{(6a)}\\
\text{s.t.} \quad & \sum_{a\in\mathcal{A}(j)} \sum_{\{i:j\in\mathcal{S}(i,a)\}} p_{ij}(a)x_{ia} = \sum_{a\in\mathcal{A}(i)} x_{ja} \quad j\in\mathcal{S} && \text{(6b)}\\
\text{and} \quad & \sum_{i\in\mathcal{S}} \sum_{a\in\mathcal{A}(i)} x_{ia} = 1, && \text{(6c)}
\end{aligned}
$$

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 (6a) is the average reward per transition in steady state, (6b) are analogous to the balance equations in every markovian system and (6c) 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 reward functions that the finite counterpart. That is not the case. The fact that the reward function depends on t, allows us to define a discounted reward as $r_t(i,a) = \alpha^t r(i,a)$, and an average reward as $r_t(i,a) = \frac{1}{T}r(i,a)$.*

2.3 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

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

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

2.4 Main modeling elements in MDP

Recall the Bellman equation (2). 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 reward 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 1 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	$p_{ijt}(a)$
Rewards	$r_t(i,a)$

Table 1: Main elements

3 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.

File InvLevel.java

```java
/*
 * Created on 26/06/2004
 *
 */
package examples.jmdp;

import jmarkov.MarkovProcess;
import jmarkov.basic.PropertiesState;

/**
 * This class allows to represent a State with a songle integer.
 * It's used in many of the examples.
 * @author Germán Riaño, Andres Sarmiento. Universida de los Andes
 */

public class InvLevel extends PropertiesState {
    /**
     * Default constructor.
     * @param k The level
     */
  public InvLevel(int k) {
    super(new int[] {k});
  }

  /**
   * Return the inventory level
   *
   * @return The level.
   */
  public int getLevel() {
    return prop[0];
  }

  @Override
  public String label() {
    return "Level " + getLevel();
  }

    /**
     * @see jmarkov.basic.State#computeMOPs(MarkovProcess)
     */
    @Override
    public void computeMOPs(MarkovProcess mp) {
        // TODO Auto-generated method stub
    }


    /**
     * @see jmarkov.basic.State#isConsistent()
     */
    @Override
    public boolean isConsistent() {
        // TODO Auto-generated method stub
        return true;
    }

}
```

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.

File Order.java

```
package examples.jmdp;

import jmarkov.basic.Action;
```

```java
/**
 * This class represents an order in an inventory system.
 * It is used in many of the Examples.
 *
 * @author Germán Riano, Andres Sarmiento
 */
public class Order extends Action {
    private int size;

    /**
     * Defualt constructor. Recives the size order.
     * @param k
     */
    Order(int k) {
        size = k;
    }

    @Override
    public String label() {
        return "Order " + size + " Units";
    }

    public int compareTo(Action a) {
        if (a instanceof Order)
            return (size - ((Order) a).size);
        else throw new IllegalArgumentException("Comparing to different type of Action
    }

    /**
     * @return Returns the order size.
     */
    public final int getSize() {
        return size;
    }

}
```

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 (2) 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 2: Types of Problems

```java
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. **Rewards**. The final method is the one representing the reward $r_t(i, a)$ received by taking action a from state i represented by the method **public double** immediateReward(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()
}
```

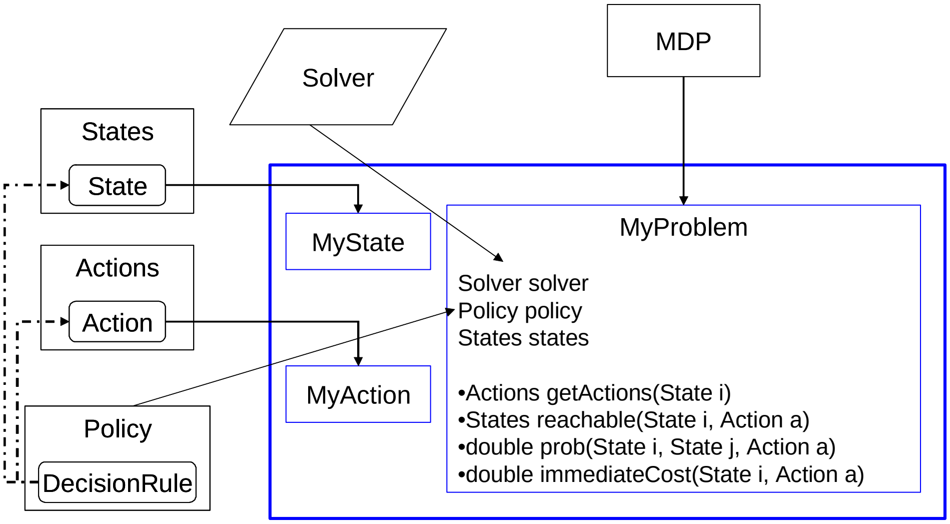
Solver
MDP
States
State
Actions
Action
Policy
DecisionRule
MyState
MyAction
MyProblem
Solver solver
Policy policy
States states
•Actions getActions(State i)
•States reachable(State i, Action a)
•double prob(State i, State j, Action a)
•double immediateCost(State i, Action a)

Figure 2: 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 getActions(S i, **int** t)
Reachable states	$\mathcal{S}_t(i, a)$	**public** States reachable(S i, A a, **int** t)
Transition probabilities	$p_{ijt}(a)$	**public double** prob(S i, S j, A a, **int** t)
Rewards	$r_t(i, a)$	**public double** immediateReward(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.

4 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.

4.1 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 (3).

	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 3: Demand forecast.

Let's first formulate the mathematical model, and then the computational one. The parameters in the word problem are in Table 4.

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 4: 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 $\mathcal{S} = \{-B, \dots, 0, \dots, M\}$ 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. 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. Reward. Finally, the reward obtained 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 reward for selling each car given by the difference between price and cost.

$$OC(a) = \left\lceil \frac{a}{L} \right\rceil$$

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

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

Now, computationally, the file would look like this.

File WagnerWhitin.java

```java
package examples.jmdp;

import jmarkov.basic.Actions;
import jmarkov.basic.ActionsSet;
import jmarkov.basic.StatesSet;
import jmarkov.basic.exceptions.SolverException;
import jmarkov.jmdp.FiniteDP;
import jmarkov.jmdp.solvers.FiniteSolver;

/**
 * This example solves an deterministic, dynamic lot-sizing problem, also known
 * as a Wagner Whitin problem.
 *
 * @author Andres Sarmiento, Germán Riaño - Universidad de Los Andes
 */

public class WagnerWhitin extends FiniteDP<InvLevel, Order> {
    int lastStage, maxInventory, maxBackorders, truckSize;

    double K, b, h, price, cost;

    int[] demand;

    // Constructor
```

```java
/**
 * Crates a dynamic economic lot sizing problem to be solved by Wagner
 * Whitin algorithm.
 *
 * @param initialInventory
 *            Inventory at time t=0.
 * @param lastStage
 *            the last stage of the problem
 * @param maxInventory
 *            maximum physical capacity in inventory, warehouse size.
 * @param maxBackorders
 *            maximum backorders allowed
 * @param truckSize
 *            maximum items in each fixed cost order. Orders can be greater
 *            than this value, but will be charged more than one fixed cost.
 * @param K
 *            fixed cost per order
 * @param b
 *            unit cost per backordered item per stage
 * @param price
 *            unit price for all stages
 * @param cost
 *            unit costo for all stages
 * @param h
 *            inventory percentual holding cost as a fraction of cost
 * @param demand
 *            demand of items in each stage
 */

public WagnerWhitin(int initialInventory, int lastStage, int maxInventory,
        int maxBackorders, int truckSize, double K, double b, double price,
        double cost, double h, int[] demand) {
    super(new StatesSet<InvLevel>(new InvLevel(initialInventory)),
            lastStage);
    this.maxInventory = maxInventory;
    this.maxBackorders = maxBackorders;
    this.truckSize = truckSize;
    this.K = K;
    this.b = b;
    this.h = h;
    this.demand = demand;
    this.price = price;
    this.cost = cost;
    init();
}

void init() {// This method builds all the states and the actions.
    Order acts[] = new Order[this.maxInventory + maxBackorders + 1];
    InvLevel ssts[] = new InvLevel[maxInventory + maxBackorders + 1];
    for (int k = 0; k < maxInventory + maxBackorders + 1; k++) {
        acts[k] = new Order(k);
        ssts[k] = new InvLevel(k - maxBackorders);
    }
    // states = new StatesCollection<InvLevel>(ssts);
}
```

```java
private double holdingCost(int x) {
    return (x > 0) ? h * cost * x : 0.0;
} // holding cost

private double orderCost(int x) {
    return (x > 0) ? Math.ceil((double) x / truckSize) * K : 0.0;
} // Order cost

double backorderCost(int x) {
    return (x < 0) ? -b * x : 0.0;
}

double lostOrderCost(int x, int t) {
    return (x + maxBackorders < demand[t]) ? (price - cost)
            * (demand[t] - x - maxBackorders) : 0.0;
}

/**
 * Returns the optimal cost for this level of starting inventory.
 *
 * @param inventory
 * @return The optimal cost for this level of starting inventory.
 * @throws SolverException
 */
public double getOptimalCost(int inventory) throws SolverException {
    return getOptimalValueFunction().get(new InvLevel(inventory));
}

@Override
public double immediateCost(InvLevel i, Order a, int t) {
    int s = i.getLevel();
    int o = a.getSize();
    return lostOrderCost(o, t) + orderCost(o) + holdingCost(s + o)
            + backorderCost(s + o);
    // return -2000*(Math.max(s+o-demand[t],0))+holding(s + o, t)+
    // orderCost(o, t);
}

@Override
public double finalCost(InvLevel i) {
    return 0.0;
}

@Override
public Actions<Order> feasibleActions(InvLevel i, int t) {
    ActionsSet<Order> actionSet = new ActionsSet<Order>();
    int min_order = Math.max(-maxBackorders - i.getLevel() + demand[t], 0);
    int max_order = maxInventory - i.getLevel() + demand[t];
    for (int n = min_order; n <= max_order; n++) {
        actionSet.add(new Order(n));
    }
    return actionSet;
}

@Override
public InvLevel destination(InvLevel i, Order a, int t) {
        int o = a.getSize();
        int iLevel = i.getLevel();
        return new InvLevel(Math.max(iLevel + o - demand[t], -maxBackorders));
    }

    /**
     * Test Program.
     *
     * @param a
     * @throws Exception
     */
    public static void main(String a[]) throws Exception {
        int lastStage = 12;
        int maxInventory = 15;
        int maxBackorders = 5;
        int truckSize = 6;
        double K = 500;
        double b = 2000;
        double p = 22000;
        double c = 20000;
        double h = Math.pow(1.3, 1.0 / 52) - 1.0;
        int[] demand = new int[] { 10, 4, 3, 6, 3, 2, 0, 1, 7, 3, 4, 5 };

        WagnerWhitin prob = new WagnerWhitin(0, lastStage, maxInventory,
                maxBackorders, truckSize, K, b, p, c, h, demand);


        FiniteSolver<InvLevel, Order> theSolver = new FiniteSolver<InvLevel, Order>(
                prob);
        prob.setSolver(theSolver);
        prob.solve();
        prob.getSolver().setPrintValueFunction(true);
        // System.out.println(theSolver.bestPolicy(initial));
        prob.printSolution();
        prob.getOptimalCost(0);
    }

}
```

4.2 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 5

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.

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 5: Parameters

When there are backorders, they will be denoted as a negative inventory level. The set of states $\mathcal{S} = \{-B, \dots, 0, \dots, M\}$ 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. $\mathcal{A} = \{0, \dots, B + M\}$.

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 $\mathcal{A}_t(i) = \{0, \dots, M - i\}$.

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: $\mathcal{S}_t(i, a) = \{-B, \dots, i\}$.

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

$$OC(a) = \left\lceil \frac{a}{L} \right\rceil$$

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 $x = i + a + B$):

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

Now, computationally, the file would look like this.

File StochasticDemand.java

```java
package examples.jmdp;

import jmarkov.basic.Actions;
import jmarkov.basic.ActionsSet;
import jmarkov.basic.States;
import jmarkov.basic.StatesSet;
import jmarkov.jmdp.FiniteMDP;

//TODO explain!!
/**
 * This class belongs to the examples supplied in the package jmdp. The
 * objective of this file is to show as clear as possible a simple way to use
 * the jmdp package as a tool for solving real life problems. The complete
 * details of the present problems are explained in the documentation.
 *
 * @author Andres Sarmiento, Germán Riaño - Universidad de Los Andes
 */
public class StochasticDemand extends FiniteMDP<InvLevel, Order> {
    int lastStage, maxInventory, maxBackorders, truckSize;

    double K, b, h, theta, price, cost;

    double[] demandProbability, demandCumulativeProbability;

    // demandProbabililty[i] = P[Demand = i]
    // demandCDF[i] = P{Demand >= i}

    // Constructor
```

```
/**
 * @param initSet
 *            Initial level of inventory of the system
 * @param lastStage
 *            the last stage of the problem
 * @param maxInventory
 *            maximum physical capacity in inventory, warehouse size.
 * @param maxBackorders
 *            maximum backorders allowed
 * @param truckSize
 *            maximum items in each fixed cost order. Orders can be greater
 *            than this value, but will be charged more than one fixed cost.
 * @param K
 *            fixed cost per order
 * @param b
 *            unit cost per backordered item per stage
 * @param price
 *            unit price for all stages
 * @param cost
 *            unit costo for all stages
 * @param h
 *            inventory percentual holding cost as a fraction of cost
 * @param theta
 *            demand mean
 */

public StochasticDemand(States<InvLevel> initSet, int lastStage,
        int maxInventory, int maxBackorders, int truckSize, double K,
        double b, double price, double cost, double h, double theta) {
    super(initSet, lastStage);
    this.maxInventory = maxInventory;
    this.maxBackorders = maxBackorders;
    this.truckSize = truckSize;
    this.K = K;
    this.b = b;
    this.price = price;
    this.cost = cost;
    this.h = h;
    this.theta = theta;
    // initStates();
    initializeProbabilities();
}

double holdingCost(int x) {
    double temp = (x > 0) ? h * cost * x : 0.0;
    return temp;
} // holding cost

double orderCost(int x) {
    double temp = (x > 0) ? Math.ceil((new Integer(x)).doubleValue()
            / truckSize)
            * K /* + x * cost */: 0.0;
    return temp;
} // Order cost

double backorderCost(double x) {
        return (x < 0) ? -b * x : 0.0;
    }

    double lostOrderCost(int x) {
        int mB = maxBackorders;
        double expectedBackorders = 0;
        for (int n = Math.max(x + 1, 0); n <= x + mB; n++)
            expectedBackorders += (n - x) * demandProbability[n];
        double expectedLostDemand = demandCumulativeProbability[x + mB]
                * (theta - x - mB) + (x + mB) * demandProbability[x + mB];
        return (price - cost) * expectedLostDemand
                + backorderCost(-expectedBackorders);
    }

    @Override
    public double finalCost(InvLevel i) {
        return 0.0;
    }

    // see documentation for the explanation for this formula

    @Override
    public double prob(InvLevel i, InvLevel j, Order a, int t) {
        int iLevel = i.getLevel();
        int jLevel = j.getLevel();
        int orderSize = a.getSize();

        // with stock & demand is positive & order is feasable
        if ((-maxBackorders < jLevel) && (jLevel <= orderSize + iLevel)
                && (orderSize + iLevel <= maxInventory))
            return demandProbability[orderSize + iLevel - jLevel];
        else if ((orderSize + iLevel <= maxInventory)
                && (jLevel == -maxBackorders)) // End up stockless
            return demandCumulativeProbability[Math.max(orderSize + iLevel, 0)];
        else
            return 0.0;
    }

    @Override
    public double immediateCost(InvLevel i, Order a, int t) {
        int iLevel = i.getLevel();
        int orderSize = a.getSize();
        double toReturn = orderCost(orderSize)
                + holdingCost(iLevel /* + orderSize */)
                + lostOrderCost(iLevel + orderSize);
        return toReturn;
    }

    void initStates() {
        InvLevel ssts[] = new InvLevel[maxInventory + maxBackorders + 1];
        for (int n = 0; n <= maxInventory; n++) {
            ssts[n] = new InvLevel(n);
        }
        for (int n = maxInventory + 1; n <= maxInventory + maxBackorders; n++) {
            ssts[n] = new InvLevel(n - maxInventory - maxBackorders - 1);
        }
        // states = new StatesCollection<InvLevel>(ssts);
    }

    void initializeProbabilities() {
        demandProbability = new double[maxInventory + maxBackorders + 1];
        demandCumulativeProbability = new double[maxInventory + maxBackorders
                + 1];
        demandProbability[0] = Math.exp(-theta);
        demandCumulativeProbability[0] = 1; // P[demand >= 0]
        double q = 1;
        for (int i = 1; i <= maxInventory + maxBackorders; i++) {
            q = demandCumulativeProbability[i - 1];
            // P{demand >= i}
            demandCumulativeProbability[i] = q - demandProbability[i - 1];
            // P{demand = i}
            demandProbability[i] = demandProbability[i - 1] * theta / i;
        }
    }

    @Override
    public Actions<Order> feasibleActions(InvLevel i, int t) {
        int max = maxInventory - i.getLevel();
        Order[] vec = new Order[max + 1];
        for (int n = 0; n <= max; n++) {
            vec[n] = new Order(n);
        }
        return new ActionsSet<Order>(vec);
    }

    @Override
    public States<InvLevel> reachable(InvLevel i, Order a, int t) {
        StatesSet<InvLevel> statesSet = new StatesSet<InvLevel>();
        for (int n = -maxBackorders; n <= i.getLevel() + a.getSize(); n++) {
            statesSet.add(new InvLevel(n));
        }
        return statesSet;
    }

    /**
     * @param a Not used
     * @throws Exception
     */
    public static void main(String a[]) throws Exception {
        int lastStage = 12;
        int maxInventory = 15;
        int maxBackorders = 5;
        int truckSize = 6;
        int K = 500;
        double b = 1000;
        double h = 0.0050582;// Math.pow(1.3, 1 / 52)-1;
        double theta = 4;
        double price = 22000;
        double cost = 20000;
        InvLevel initial = new InvLevel(0);
        States<InvLevel> initSet = new StatesSet<InvLevel>(initial);

        StochasticDemand pro = new StochasticDemand(initSet, lastStage,
                maxInventory, maxBackorders, truckSize, K, b, price, cost, h,
                    theta);
          pro.solve();
          pro.getSolver().setPrintValueFunction(true);
          pro.printSolution();
     }

}
```

4.3 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 (6).

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	$P\{D_t \geq n\}$

Table 6: Parameters

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.

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. Reward. The reward obtained depends on various factors. The ordering cost is only charged when the order is positive, and charged per truck.

$$OC(a) = \left\lceil \frac{a}{L} \right\rceil$$

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.

File InfStochasticDemand.java

```
package examples.jmdp;

import jmarkov.basic.Actions;
import jmarkov.basic.ActionsSet;
import jmarkov.basic.States;
import jmarkov.basic.StatesSet;
import jmarkov.basic.exceptions.SolverException;
import jmarkov.jmdp.DTMDP;
import jmarkov.jmdp.solvers.PolicyIterationSolver;
import jmarkov.jmdp.solvers.RelativeValueIterationSolver;
import jmarkov.jmdp.solvers.ValueIterationSolver;
```

```java
import Jama.Matrix;

//TODO documentar bien
/**
 * This problem is a single item, periodic review, stochastic demand inventory
 * problem. It is modeled like a discounted cost, infinite horizon, Markov
 * Decision Problem. Demand is assumed to be random according to a Poisson
 * Process distribution with given rate per period. The system is a periodic
 * review problem in which an entity periodically checks the inventory level and
 * takes decisions according to the states he finds. There is a price of selling
 * each item and a cost for buying it. Besides, there is a holding cost incurred
 * when holding one item in stock from one period to another. There is also a
 * truckCost ordering cost independent of the size of the order placed. The
 * objective is to minimize the expected discounted long run cost.
 *
 * @author Germán Riaño and Andres Sarmiento - Universidad de Los Andes
 */
public class InfStochasticDemand extends DTMDP<InvLevel, Order> {
    // Problem parameters:
    private int maxInv, maxBO, truckSize;
    // Cost and demand parameters:
    private double truckCost, backorderCost, holdingCost, intRate, expDemand,
            price, cost;
    private double[] demPMF, demCDF, demandLoss1;
    private boolean isdisc = false;

    // demPMF[i] = P{Demand = i}
    // demCDF[i] = P{Demand <= i}
    // demandLoss1[i] = E[(Demand - i)^+ ]
    // Constructor

    /**
     * @param maxInv
     *            maximum physical capacity in inventory, warehouse size.
     * @param maxBO
     *            maximum backorders allowed
     * @param truckSize
     *            maximum items in each fixed cost order. Orders can be greater
     *            than this value, but will be charged more than one fixed cost.
     * @param truckCost
     *            fixed cost per order
     * @param backorderCost
     *            unit cost per backordered item per stage
     * @param price
     *            unit price
     * @param cost
     *            unit aquistion costo
     * @param holdingCost
     *            non-finantial holding cost (it does NOT include finantial
     *            cost)
     * @param intRate
     *            interest per period
     * @param expDemand
     *            demand mean
     * @param discounted
     *            Whether a discounted model (rather than average) is to be
     *            used.
     */

    @SuppressWarnings("unchecked")
    public InfStochasticDemand(int maxInv, int maxBO, int truckSize,
            double truckCost, double backorderCost, double price, double cost,
            double holdingCost, double intRate, double expDemand,
            boolean discounted) {
        super(new StatesSet<InvLevel>(new InvLevel(0)));
        this.maxInv = maxInv;
        this.maxBO = maxBO;
        this.truckSize = truckSize;
        this.truckCost = truckCost;
        this.backorderCost = backorderCost;
        this.price = price;
        this.cost = cost;
        this.holdingCost = holdingCost;
        this.expDemand = expDemand;
        // initStates();
        initializeProbabilities();
        this.isdisc = discounted;
        this.intRate = intRate;
        if (discounted)
            setSolver(new ValueIterationSolver(this, intRate));
        else
            setSolver(new RelativeValueIterationSolver(this));
    }

    private void initializeProbabilities() {
        demPMF = new double[maxInv + maxBO + 1];
        demCDF = new double[maxInv + maxBO + 1];
        demandLoss1 = new double[maxInv + maxBO + 1];
        double cdf, p = Math.exp(-expDemand);
        cdf = demCDF[0] = demPMF[0] = p;
        demandLoss1[0] = expDemand;
        int maxlevel = maxInv + maxBO;
        for (int i = 1; i <= maxlevel; i++) {
            demPMF[i] = (p *= expDemand / i); // P{demand = i}
            demCDF[i] = (cdf += p); // P{demand <= i}
            demandLoss1[i] = (expDemand - i) * (1 - cdf) + expDemand * p;
            // = E[(D-i)^+]
        }
    }

    @Override
    public States<InvLevel> reachable(InvLevel i, Order a) {
        StatesSet<InvLevel> statesSet = new StatesSet<InvLevel>();
        // Available inventory upon order receival:
        int maxLevel = i.getLevel() + a.getSize();
        for (int n = -maxBO; n <= maxLevel; n++) {
            statesSet.add(new InvLevel(n));
        }
        return statesSet;
    }

    @Override
    public double prob(InvLevel i, InvLevel j, Order a) {
        int iLevel = i.getLevel();
        int jLevel = j.getLevel();
        int orderSize = a.getSize();
        // with stock & demand is positive & order is feasable
        int demand = orderSize + iLevel - jLevel;
        assert (demand >= 0);
        try {
            if (jLevel == -maxBO)
                return 1.0 - ((demand > 0) ? demCDF[demand - 1] : 0.0);
            else
                // End up stockless
                return demPMF[demand];
        } catch (IndexOutOfBoundsException e) {
            throw new IllegalArgumentException(
                    "'prob' called on non-reachable state!!. i=" + iLevel
                            + ", j=" + jLevel + ", a =" + orderSize, e);
        }
    }

    @Override
    public Actions<Order> feasibleActions(InvLevel i) {
        int max = maxInv - i.getLevel();
        Order[] vec = new Order[max + 1];
        for (int n = 0; n <= max; n++) {
            vec[n] = new Order(n);
        }
        return new ActionsSet<Order>(vec);
    }

    double holdingCost(int x) {
        double totHoldCost = holdingCost + ((isdisc) ? intRate * cost : 0.0);
        return (x > 0) ? totHoldCost * x : 0.0;
    } // holding cost

    double orderCost(int x) {
        return cost = truckCost * Math.ceil((double) x / truckSize) + x * cost;
    } // Order cost

    double backorderCost(double x) {
        return (x < 0) ? -backorderCost * x : 0.0;
    }

    // see documentation for the explanation for this formula
    @Override
    public double immediateCost(InvLevel i, Order a) {
        int maxSale = i.getLevel() + a.getSize() + maxBO;
        double expectedSales = expDemand - demandLoss1[maxSale];
        double netProfit = price * expectedSales - orderCost(a.getSize())
                - holdingCost(i.getLevel()) - backorderCost(i.getLevel());
        return -netProfit;
    }

    /**
     * Very stupid method to see what this is doing!!
     */
    public void printMatrices() {
        double[][] cost = new double[maxBO + maxInv + 1][maxBO + maxInv + 1];
        double[][][] prb = new double[maxBO+maxInv + 1][maxBO+maxInv + 1][maxBO+maxInv + 1
        for (InvLevel s: getAllStates()) {
            int i = s.getLevel();
            for (Order o: feasibleActions(s)) {
                int a = o.getSize();
                cost[i+maxBO][a] = immediateCost(new InvLevel(i), new Order(a));
                for (InvLevel y:reachable(s,o)) {
                    int j = y.getLevel();
                    prb[a][i+maxBO][j+maxBO] = prob(new InvLevel(i), new InvLevel(j),
                            new Order(a));
                }
            }
        }
        (new Matrix(cost)).print(8, 2);
        for (int a = 0; a < maxInv; a++) {
            (new Matrix(prb[a])).print(10, 6);
        }
        (new Matrix(new double[][] { demPMF })).print(10, 6);
        (new Matrix(new double[][] { demCDF })).print(10, 6);
        (new Matrix(new double[][] { demandLoss1 })).print(10, 6);
    }

    /**
     * Simple test Program.
     *
     * @param a
     * @throws SolverException
     */
    public static void main(String a[]) throws SolverException {
        int maxInventory = 15;
        int maxBackorders = 5;
        int truckSize = 100;// 6;
        int K = 50;// 500;
        double b = 600;// 1000;
        double h = 80;//
        double intRate = Math.pow(1.3, 1 / 52);
        double theta = 4;
        double price = 1000;// 22000;
        double cost = 400;// 20000;

        InfStochasticDemand prob = new InfStochasticDemand(maxInventory,
                maxBackorders, truckSize, K, b, price, cost, h, intRate, theta,
                true);

        PolicyIterationSolver<InvLevel, Order> solv = new PolicyIterationSolver<InvLevel, O
                prob, 0.9);

        prob.setSolver(solv);
        prob.getSolver().setPrintValueFunction(true);
        prob.solve();
        prob.printSolution();

        // ------------------------------

        RelativeValueIterationSolver<InvLevel, Order> solver = new RelativeValueIterationSo
                prob);
        solver.setPrintValueFunction(true);
        prob.setSolver(solver);
        prob.solve();
        prob.printSolution();
    }

}
```

5 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.

5.1 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. The class is declared as **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. 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.

5.2 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. An example would be: 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 the object **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.

5.3 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 2). 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.

When declaring a new class **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 (7).

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** immediateReward(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** immediateReward(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** immediateReward(S i, A a)

Table 7: Abstract methods.

5.4 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 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.

5.4.1 FiniteSolver

The **public class** FiniteSolver<S **extends** State, A **extends** Action> **extends** AbstractFiniteSolver is intended to solve only finite horizon problems. The constructors **public** FiniteSolver(FiniteMDP<S,A> problem) only receive problems modeled with FiniteMDP (or FiniteDP) classes, implying that only finite horizon problems can be solved. The objective function is to maximize the total reward presented in equation (1), in the mathematical model.

5.4.2 ValueIterationSolver

The **public class** ValueIterationSolver<S **extends** State, A **extends** Action> **implements** Solver is the solver class that maximizes the discounted reward v_α^π presented in equation (3) on the mathematical model. The constructor only receives InfiniteMDP<S,A> objects as a problem parameter as shown in **public** ValueIterationSolver(InfiniteMDP<S,A> problem, **double** discountFactor). This shows 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 (4) 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 this default 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.

5.4.3 PolicyIterationSolver

The **public class** PolicyIterationSolver is also designed to solve only infinite horizon problems and this is restricted in the arguments of its constructor **public** PolicyIterationSolver(InfiniteMDP<S,A> problem, **double** disco that only receives InfiniteMDP<S,A> objects as an argument. This solver maximizes the discounted reward $_{DR}v^\pi$ presented in equation (3) 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.

6 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

Index