jMDP User's Guide

Germán Riaño, Andrés Sarmiento and Daniel F. Silva

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 publicly available language developed 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 `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.

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.

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

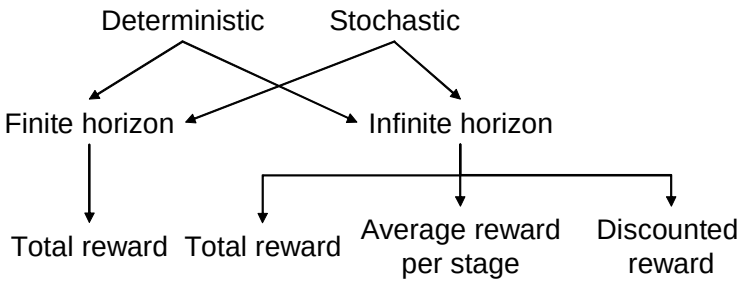

Figure 1: Taxonomy for MDP problems. WARNNING: "rewards" need to be changed to "costs".

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, respectively, 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 $t = 0, 1, \ldots$, and all $i \in \mathcal{S}$ and $a \in \mathcal{A}$,

$$P_\pi\{A_t = a | X_t = i\} = P_\pi\{A_0 = a | X_0 = i\}.$$

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

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

Whenever 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 $v_t^\pi(i)$ obtained from time t to the final stage T following policy π; this is called the *value function*

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

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.

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

Such optimal value function can be shown to satisfy *Bellman's optimality equation*

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

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 $t = 0, 1, \ldots$. 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.

2.2.1 Discounted Cost

In the discounted cost problem the costs in the first stages are more important than the later ones. In particular, a cost 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 cost gives rise to a value function under policy π defined as

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

In this case, the optimal value function is

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

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

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

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 cost 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

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

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 $\max |v_\beta(i) - v^*(i)| < \epsilon$. 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

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

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

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

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.

2.2.2 Total Cost

The value function in the total cost case is given by

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

and the optimal value function is

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

The total 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 (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 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'indexStochastic Shortest Path problem, since since if expected total cost can be thought of as the minimal expected cost accumulated before absorption in a graph with random costs.

2.2.3 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

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

and the optimal value function is the same for every state

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

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

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

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 cost 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 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 $c_t(i,a) = \frac{1}{T} c(i,a)$.*

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

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

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 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 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	$p_{ijt}(a)$
Costs	$c_t(i,a)$

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 defined 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.

```
1  /*
2   * Created on 26/06/2004
3   *
4   */
5  package examples.jmdp;
6
7  import jmarkov.MarkovProcess;
8  import jmarkov.basic.PropertiesState;
9
10 /**
11  * This class allows to represent a State with a songle integer.
12  * It's used in many of the examples.
13  * @author Daniel Silva, German Riano, Andres Sarmiento. Universida de los Andes
14  */
15 public class InvLevel extends PropertiesState {
16     /**
17      * Default constructor.
18      * @param k The level
19      */
20         public InvLevel(int k) {
21                 super(new int[] {k});
22         }
23
24         /**
25          * Return the inventory level
26          *
27          * @return The level.
28          */
29         public int getLevel() {
30                 return prop[0];
31         }
32
33         @Override
34         public String label() {
35                 return "Level " + getLevel();
36         }
37
38     /**
39      * @see jmarkov.basic.State#computeMOPs(MarkovProcess)
40      */
41     @Override
42     public void computeMOPs(MarkovProcess mp) {
43         // TODO Auto-generated method stub
44     }
45
46
47     /**
48      * @see jmarkov.basic.State#isConsistent()
49      */
50     @Override
51     public boolean isConsistent() {
52         // TODO Auto-generated method stub
53         return true;
54     }
55
56 }
```

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.

```java
package examples.jmdp;

import jmarkov.basic.Action;

/**
 * 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 with 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. **Costs**. The final method is the one representing the cost $c_t(i, a)$ received by 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.

```
1 public static void main(String args[]) {
2
3 InventoryProblem prob = new InventoryProblem(maxInventory,
4   maxItemsPerOrder, truckCost, holdingCost, theta);
5
6 prob.solve()
7 prob.printSolution()
8 }
```

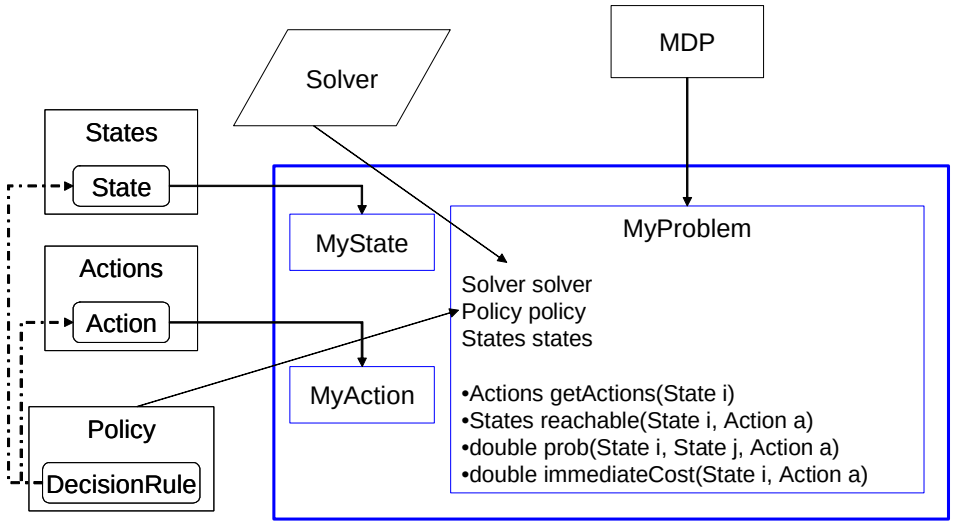

Figure 2: Problem's structure.

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)
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.

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 $\mathcal{S} = \{-B, \ldots, 0, \ldots, M\}$ 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. 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 incured 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 profit 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.

```
1  package examples.jmdp;
2
3  import jmarkov.basic.Actions;
4  import jmarkov.basic.ActionsSet;
5  import jmarkov.basic.StatesSet;
6  import jmarkov.basic.exceptions.SolverException;
7  import jmarkov.jmdp.FiniteDP;
8  import jmarkov.jmdp.solvers.FiniteSolver;
9
10 /**
11  * This example solves a deterministic, dynamic lot-sizing problem, also known
12  * as a Wagner Whitin problem.
13  *
14  * @author Andres Sarmiento, Germán Riaño - Universidad de Los Andes
15  */
16
17 public class WagnerWhitin extends FiniteDP<InvLevel, Order> {
18     int lastStage, maxInventory, maxBackorders, truckSize;
19
20     double K, b, h, price, cost;
21
22     int[] demand;
23
24     // Constructor
25
26     /**
27      * Crates a dynamic economic lot sizing problem to be solved by Wagner
28      * Whitin algorithm.
29      *
30      * @param initialInventory
31      *            Inventory at time t=0.
32      * @param lastStage
33      *            the last stage of the problem
34      * @param maxInventory
35      *            maximum physical capacity in inventory, warehouse size.
36      * @param maxBackorders
37      *            maximum backorders allowed
38      * @param truckSize
39      *            maximum items in each fixed cost order. Orders can be greater
40      *            than this value, but will be charged more than one fixed cost.
41      * @param K
```

```
42          *            fixed cost per order
43          * @param b
44          *            unit cost per backordered item per stage
45          * @param price
46          *            unit price for all stages
47          * @param cost
48          *            unit costo for all stages
49          * @param h
50          *            inventory percentual holding cost as a fraction of cost
51          * @param demand
52          *            demand of items in each stage
53          */
54
55         public WagnerWhitin(int initialInventory, int lastStage, int maxInventory,
56                 int maxBackorders, int truckSize, double K, double b, double price,
57                 double cost, double h, int[] demand) {
58             super(new StatesSet<InvLevel>(new InvLevel(initialInventory)),
59                     lastStage);
60             this.maxInventory = maxInventory;
61             this.maxBackorders = maxBackorders;
62             this.truckSize = truckSize;
63             this.K = K;
64             this.b = b;
65             this.h = h;
66             this.demand = demand;
67             this.price = price;
68             this.cost = cost;
69             init();
70         }
71
72         void init() {// This method builds all the states and the actions.
73             Order acts[] = new Order[this.maxInventory + maxBackorders + 1];
74             InvLevel ssts[] = new InvLevel[maxInventory + maxBackorders + 1];
75             for (int k = 0; k < maxInventory + maxBackorders + 1; k++) {
76                 acts[k] = new Order(k);
77                 ssts[k] = new InvLevel(k - maxBackorders);
78             }
79             // states = new StatesCollection<InvLevel>(ssts);
80         }
81
82
83
84         private double holdingCost(int x) {
85             return (x > 0) ? h * cost * x : 0.0;
86         } // holding cost
87
88         private double orderCost(int x) {
89             return (x > 0) ? Math.ceil((double) x / truckSize) * K : 0.0;
90         } // Order cost
91
92         double backorderCost(int x) {
93             return (x < 0) ? -b * x : 0.0;
94         }
95
96         double lostOrderCost(int x, int t) {
97             return (x + maxBackorders < demand[t]) ? (price - cost)
98                     * (demand[t] - x - maxBackorders) : 0.0;
99         }
100
101        /**
102         * Returns the optimal cost for this level of starting inventory.
103         *
104         * @param inventory
105         * @return The optimal cost for this level of starting inventory.
106         * @throws SolverException
107         */
108        public double getOptimalCost(int inventory) throws SolverException {
109            return getOptimalValueFunction().get(new InvLevel(inventory));
110        }
111
112        @Override
113        public double immediateCost(InvLevel i, Order a, int t) {
114            int s = i.getLevel();
115            int o = a.getSize();
116            return lostOrderCost(o, t) + orderCost(o) + holdingCost(s + o)
117                    + backorderCost(s + o);
118            // return -2000*(Math.max(s+o-demand[t],0))+holding(s + o, t)+
119            // orderCost(o, t);
120        }
121
```

```
122     @Override
123     public double finalCost(InvLevel i) {
124         return 0.0;
125     }
126
127     @Override
128     public Actions<Order> feasibleActions(InvLevel i, int t) {
129         ActionsSet<Order> actionSet = new ActionsSet<Order>();
130         int min_order = Math.max(-maxBackorders - i.getLevel() + demand[t], 0);
131         int max_order = maxInventory - i.getLevel() + demand[t];
132         for (int n = min_order; n <= max_order; n++) {
133             actionSet.add(new Order(n));
134         }
135         return actionSet;
136     }
137
138     @Override
139     public InvLevel destination(InvLevel i, Order a, int t) {
140         int o = a.getSize();
141         int iLevel = i.getLevel();
142         return new InvLevel(Math.max(iLevel + o - demand[t], -maxBackorders));
143     }
144
145     /**
146      * Test Program.
147      *
148      * @param a
149      * @throws Exception
150      */
151     public static void main(String a[]) throws Exception {
152         int lastStage = 12;
153         int maxInventory = 15;
154         int maxBackorders = 5;
155         int truckSize = 6;
156         double K = 500;
157         double b = 2000;
158         double p = 22000;
159         double c = 20000;
160         double h = Math.pow(1.3, 1.0 / 52) - 1.0;
161         int[] demand = new int[] { 10, 4, 3, 6, 3, 2, 0, 1, 7, 3, 4, 5 };
162
163         WagnerWhitin prob = new WagnerWhitin(0, lastStage, maxInventory,
164                 maxBackorders, truckSize, K, b, p, c, h, demand);
165
166
167         FiniteSolver<InvLevel, Order> theSolver = new FiniteSolver<InvLevel, Order>(
168                 prob);
169         prob.setSolver(theSolver);
170         prob.solve();
171         prob.getSolver().setPrintValueFunction(true);
172         // System.out.println(theSolver.bestPolicy(initial));
173         prob.printSolution();
174         prob.getOptimalCost(0);
175     }
176
177 }
```

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

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

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. Costs. The net profit (minus cost) 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.

```
1  package examples.jmdp;
2
3  import jmarkov.basic.Actions;
4  import jmarkov.basic.ActionsSet;
5  import jmarkov.basic.States;
6  import jmarkov.basic.StatesSet;
7  import jmarkov.jmdp.FiniteMDP;
8
9  /**
10  * This class belongs to the examples supplied in the package jmdp. The
11  * objective of this file is to show as clear as possible a simple way to use
12  * the jmdp package as a tool for solving real life problems. The complete
13  * details of the present problems are explained in the documentation.
14  *
15  * @author Andres Sarmiento, German Riano - Universidad de Los Andes
16  */
17 public class StochasticDemand extends FiniteMDP<InvLevel, Order> {
18         //TODO: This example needs more documentation
19     int lastStage, maxInventory, maxBackorders, truckSize;
20
21     double K, b, h, theta, price, cost;
22
23     double[] demandProbability, demandCumulativeProbability;
24
25     // demandProbabililty[i] = P[Demand = i]
26     // demandCDF[i] = P{Demand >= i}
27
28     // Constructor
29
30     /**
31      * @param initSet
32      *             Initial level of inventory of the system
33      * @param lastStage
34      *             the last stage of the problem
35      * @param maxInventory
36      *             maximum physical capacity in inventory, warehouse size.
37      * @param maxBackorders
38      *             maximum backorders allowed
39      * @param truckSize
40      *             maximum items in each fixed cost order. Orders can be greater
```

```
41          *              than this value, but will be charged more than one fixed cost.
42          * @param K
43          *              fixed cost per order
44          * @param b
45          *              unit cost per backordered item per stage
46          * @param price
47          *              unit price for all stages
48          * @param cost
49          *              unit costo for all stages
50          * @param h
51          *              inventory percentual holding cost as a fraction of cost
52          * @param theta
53          *              demand mean
54          */
55
56      public StochasticDemand(States<InvLevel> initSet, int lastStage,
57              int maxInventory, int maxBackorders, int truckSize, double K,
58              double b, double price, double cost, double h, double theta) {
59          super(initSet, lastStage);
60          this.maxInventory = maxInventory;
61          this.maxBackorders = maxBackorders;
62          this.truckSize = truckSize;
63          this.K = K;
64          this.b = b;
65          this.price = price;
66          this.cost = cost;
67          this.h = h;
68          this.theta = theta;
69          // initStates();
70          initializeProbabilities();
71      }
72
73      double holdingCost(int x) {
74          double temp = (x > 0) ? h * cost * x : 0.0;
75          return temp;
76      } // holding cost
77
78      double orderCost(int x) {
79          double temp = (x > 0) ? Math.ceil((new Integer(x)).doubleValue()
80                  / truckSize)
81                  * K /* + x * cost */: 0.0;
82          return temp;
83      } // Order cost
84
85      double backorderCost(double x) {
86          return (x < 0) ? -b * x : 0.0;
87      }
88
89      double lostOrderCost(int x) {
90          int mB = maxBackorders;
91          double expectedBackorders = 0;
92          for (int n = Math.max(x + 1, 0); n <= x + mB; n++)
93              expectedBackorders += (n - x) * demandProbability[n];
94          double expectedLostDemand = demandCumulativeProbability[x + mB]
95                  * (theta - x - mB) + (x + mB) * demandProbability[x + mB];
96          return (price - cost) * expectedLostDemand
97                  + backorderCost(-expectedBackorders);
98      }
99
100     @Override
101     public double finalCost(InvLevel i) {
102         return 0.0;
103     }
104
105     // see documentation for the explanation for this formula
106
107     @Override
108     public double prob(InvLevel i, InvLevel j, Order a, int t) {
109         int iLevel = i.getLevel();
110         int jLevel = j.getLevel();
111         int orderSize = a.getSize();
112
113         // with stock & demand is positive & order is feasable
114         if ((-maxBackorders < jLevel) && (jLevel <= orderSize + iLevel)
115                 && (orderSize + iLevel <= maxInventory))
116             return demandProbability[orderSize + iLevel - jLevel];
117         else if ((orderSize + iLevel <= maxInventory)
118                 && (jLevel == -maxBackorders)) // End up stockless
119             return demandCumulativeProbability[Math.max(orderSize + iLevel, 0)];
120         else
```

```
121             return 0.0;
122         }
123
124         @Override
125         public double immediateCost(InvLevel i, Order a, int t) {
126             int iLevel = i.getLevel();
127             int orderSize = a.getSize();
128             double toReturn = orderCost(orderSize)
129                     + holdingCost(iLevel /* + orderSize */)
130                     + lostOrderCost(iLevel + orderSize);
131             return toReturn;
132         }
133
134         void initStates() {
135             InvLevel ssts[] = new InvLevel[maxInventory + maxBackorders + 1];
136             for (int n = 0; n <= maxInventory; n++) {
137                 ssts[n] = new InvLevel(n);
138             }
139             for (int n = maxInventory + 1; n <= maxInventory + maxBackorders; n++) {
140                 ssts[n] = new InvLevel(n - maxInventory - maxBackorders - 1);
141             }
142             // states = new StatesCollection<InvLevel>(ssts);
143         }
144
145         void initializeProbabilities() {
146             demandProbability = new double[maxInventory + maxBackorders + 1];
147             demandCumulativeProbability = new double[maxInventory + maxBackorders
148                     + 1];
149             demandProbability[0] = Math.exp(-theta);
150             demandCumulativeProbability[0] = 1; // P[demand >= 0]
151             double q = 1;
152             for (int i = 1; i <= maxInventory + maxBackorders; i++) {
153                 q = demandCumulativeProbability[i - 1];
154                 // P{demand >= i}
155                 demandCumulativeProbability[i] = q - demandProbability[i - 1];
156                 // P{demand = i}
157                 demandProbability[i] = demandProbability[i - 1] * theta / i;
158             }
159         }
160
161         @Override
162         public Actions<Order> feasibleActions(InvLevel i, int t) {
163             int max = maxInventory - i.getLevel();
164             Order[] vec = new Order[max + 1];
165             for (int n = 0; n <= max; n++) {
166                 vec[n] = new Order(n);
167             }
168             return new ActionsSet<Order>(vec);
169         }
170
171         @Override
172         public States<InvLevel> reachable(InvLevel i, Order a, int t) {
173             StatesSet<InvLevel> statesSet = new StatesSet<InvLevel>();
174             for (int n = -maxBackorders; n <= i.getLevel() + a.getSize(); n++) {
175                 statesSet.add(new InvLevel(n));
176             }
177             return statesSet;
178         }
179
180         /**
181          * @param a Not used
182          * @throws Exception
183          */
184         public static void main(String a[]) throws Exception {
185             int lastStage = 12;
186             int maxInventory = 15;
187             int maxBackorders = 5;
188             int truckSize = 6;
189             int K = 500;
190             double b = 1000;
191             double h = 0.0050582;// Math.pow(1.3, 1 / 52)-1;
192             double theta = 4;
193             double price = 22000;
194             double cost = 20000;
195             InvLevel initial = new InvLevel(0);
196             States<InvLevel> initSet = new StatesSet<InvLevel>(initial);
197
198             StochasticDemand pro = new StochasticDemand(initSet, lastStage,
199                     maxInventory, maxBackorders, truckSize, K, b, price, cost, h,
200                     theta);
```

```
201             pro.solve();
202             pro.getSolver().setPrintValueFunction(true);
203             pro.printSolution();
204         }
205
206     }
```

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. Cost. The net profit 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 $x = i + a + B$):

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

The full implementation is provided in the following.

```
1  package examples.jmdp;
2
3  import jmarkov.basic.Actions;
4  import jmarkov.basic.ActionsSet;
5  import jmarkov.basic.States;
6  import jmarkov.basic.StatesSet;
7  import jmarkov.basic.exceptions.SolverException;
8  import jmarkov.jmdp.DTMDP;
9  import jmarkov.jmdp.solvers.PolicyIterationSolver;
10 import jmarkov.jmdp.solvers.RelativeValueIterationSolver;
11 import jmarkov.jmdp.solvers.ValueIterationSolver;
12 import Jama.Matrix;
13
14 /**
15  * This problem is a single item, periodic review, stochastic demand inventory
```

```
16   * problem. It is modeled like a discounted cost, infinite horizon, Markov
17   * Decision Problem. Demand is assumed to be random according to a Poisson
18   * Process distribution with given rate per period. The system is a periodic
19   * review problem in which an entity periodically checks the inventory level and
20   * takes decisions according to the states he finds. There is a price of selling
21   * each item and a cost for buying it. Besides, there is a holding cost incurred
22   * when holding one item in stock from one period to another. There is also a
23   * truckCost ordering cost independent of the size of the order placed. The
24   * objective is to minimize the expected discounted long run cost.
25   *
26   * @author Germán Riaño and Andres Sarmiento - Universidad de Los Andes
27   */
28  public class InfStochasticDemand extends DTMDP<InvLevel, Order> {
29          //TODO This example needs more specific documentation.
30      // Problem parameters:
31      private int maxInv, maxBO, truckSize;
32      // Cost and demand parameters:
33      private double truckCost, backorderCost, holdingCost, intRate, expDemand,
34              price, cost;
35      private double[] demPMF, demCDF, demandLoss1;
36      private boolean isdisc = false;
37
38      // demPMF[i] = P{Demand = i}
39      // demCDF[i] = P{Demand <= i}
40      // demandLoss1[i] = E[(Demand - i)^+ ]
41      // Constructor
42
43      /**
44       * @param maxInv
45       *              maximum physical capacity in inventory, warehouse size.
46       * @param maxBO
47       *              maximum backorders allowed
48       * @param truckSize
49       *              maximum items in each fixed cost order. Orders can be greater
50       *              than this value, but will be charged more than one fixed cost.
51       * @param truckCost
52       *              fixed cost per order
53       * @param backorderCost
54       *              unit cost per backordered item per stage
55       * @param price
56       *              unit price
57       * @param cost
58       *              unit aquistion costo
59       * @param holdingCost
60       *              non-finantial holding cost (it does NOT include finantial
61       *              cost)
62       * @param intRate
63       *              interest per period
64       * @param expDemand
65       *              demand mean
66       * @param discounted
67       *              Whether a discounted model (rather than average) is to be
68       *              used.
69       */
70
71      @SuppressWarnings("unchecked")
72      public InfStochasticDemand(int maxInv, int maxBO, int truckSize,
73              double truckCost, double backorderCost, double price, double cost,
74              double holdingCost, double intRate, double expDemand,
75              boolean discounted) {
76          super(new StatesSet<InvLevel>(new InvLevel(0)));
77          this.maxInv = maxInv;
78          this.maxBO = maxBO;
79          this.truckSize = truckSize;
80          this.truckCost = truckCost;
81          this.backorderCost = backorderCost;
82          this.price = price;
83          this.cost = cost;
84          this.holdingCost = holdingCost;
85          this.expDemand = expDemand;
86          // initStates();
87          initializeProbabilities();
88          this.isdisc = discounted;
89          this.intRate = intRate;
90          if (discounted)
91              setSolver(new ValueIterationSolver(this, intRate));
92          else
93              setSolver(new RelativeValueIterationSolver(this));
94      }
95
```

```
96      private void initializeProbabilities() {
97          demPMF = new double[maxInv + maxBO + 1];
98          demCDF = new double[maxInv + maxBO + 1];
99          demandLoss1 = new double[maxInv + maxBO + 1];
100         double cdf, p = Math.exp(-expDemand);
101         cdf = demCDF[0] = demPMF[0] = p;
102         demandLoss1[0] = expDemand;
103         int maxlevel = maxInv + maxBO;
104         for (int i = 1; i <= maxlevel; i++) {
105             demPMF[i] = (p *= expDemand / i); // P{demand = i}
106             demCDF[i] = (cdf += p); // P{demand <= i}
107             demandLoss1[i] = (expDemand - i) * (1 - cdf) + expDemand * p;
108             // = E[(D-i)^+]
109         }
110     }
111
112     @Override
113     public States<InvLevel> reachable(InvLevel i, Order a) {
114         StatesSet<InvLevel> statesSet = new StatesSet<InvLevel>();
115         // Available inventory upon order receival:
116         int maxLevel = i.getLevel() + a.getSize();
117         for (int n = -maxBO; n <= maxLevel; n++) {
118             statesSet.add(new InvLevel(n));
119         }
120         return statesSet;
121     }
122
123     @Override
124     public double prob(InvLevel i, InvLevel j, Order a) {
125         int iLevel = i.getLevel();
126         int jLevel = j.getLevel();
127         int orderSize = a.getSize();
128         // with stock & demand is positive & order is feasable
129         int demand = orderSize + iLevel - jLevel;
130         assert (demand >= 0);
131         try {
132             if (jLevel == -maxBO)
133                 return 1.0 - ((demand > 0) ? demCDF[demand - 1] : 0.0);
134             else
135                 // End up stockless
136                 return demPMF[demand];
137         } catch (IndexOutOfBoundsException e) {
138             throw new IllegalArgumentException(
139                     "'prob' called on non-reachable state!!. i=" + iLevel
140                             + ", j=" + jLevel + ", a=" + orderSize, e);
141         }
142     }
143
144     @Override
145     public Actions<Order> feasibleActions(InvLevel i) {
146         int max = maxInv - i.getLevel();
147         Order[] vec = new Order[max + 1];
148         for (int n = 0; n <= max; n++) {
149             vec[n] = new Order(n);
150         }
151         return new ActionsSet<Order>(vec);
152     }
153
154     double holdingCost(int x) {
155         double totHoldCost = holdingCost + ((isdisc) ? intRate * cost : 0.0);
156         return (x > 0) ? totHoldCost * x : 0.0;
157     } // holding cost
158
159     double orderCost(int x) {
160         return truckCost * Math.ceil((double) x / truckSize) + x * cost;
161     } // Order cost
162
163     double backorderCost(double x) {
164         return (x < 0) ? -backorderCost * x : 0.0;
165     }
166
167     // see documentation for the explanation for this formula
168     @Override
169     public double immediateCost(InvLevel i, Order a) {
170         int maxSale = i.getLevel() + a.getSize() + maxBO;
171         double expectedSales = expDemand - demandLoss1[maxSale];
172         double netProfit = price * expectedSales - orderCost(a.getSize())
173                 - holdingCost(i.getLevel()) - backorderCost(i.getLevel());
174         return -netProfit;
175     }
```

```
176
177     /**
178      * Very stupid method to see what this is doing!!
179      */
180     public void printMatrices() {
181         double[][] cost = new double[maxBO + maxInv + 1][maxBO + maxInv + 1];
182         double[][][] prb = new double[maxBO+maxInv + 1][maxBO+maxInv + 1][maxBO+maxInv + 1];
183         for (InvLevel s: getAllStates()) {
184             int i = s.getLevel();
185             for (Order o: feasibleActions(s)) {
186                 int a = o.getSize();
187                 cost[i+maxBO][a] = immediateCost(new InvLevel(i), new Order(a));
188                 for (InvLevel y:reachable(s,o)) {
189                     int j = y.getLevel();
190                     prb[a][i+maxBO][j+maxBO] = prob(new InvLevel(i), new InvLevel(j),
191                             new Order(a));
192                 }
193             }
194         }
195         (new Matrix(cost)).print(8, 2);
196         for (int a = 0; a < maxInv; a++) {
197             (new Matrix(prb[a])).print(10, 6);
198         }
199         (new Matrix(new double[][] { demPMF })).print(10, 6);
200         (new Matrix(new double[][] { demCDF })).print(10, 6);
201         (new Matrix(new double[][] { demandLoss1 })).print(10, 6);
202     }
203
204     /**
205      * Simple test Program.
206      *
207      * @param a
208      * @throws SolverException
209      */
210     public static void main(String a[]) throws SolverException {
211         int maxInventory = 25;
212         int maxBackorders = 0;
213         int truckSize = 4;
214         int truckCost = 1000;
215         double b = 0;// 1000;
216         double holdCost = 50;
217         double intRate = Math.pow(1.3, 1 / 52);
218         double theta = 20;
219         double price = 1100;// 22000;
220         double cost = 500;// 20000;
221
222         InfStochasticDemand prob = new InfStochasticDemand(maxInventory,
223                 maxBackorders, truckSize, truckCost, b, price, cost, holdCost, intRate, theta,
224                 false);
225
226         RelativeValueIterationSolver<InvLevel, Order> solv = new RelativeValueIterationSolver<InvLevel, Order>(
227                 prob);
228
229         prob.setSolver(solv);
230         prob.getSolver().setPrintValueFunction(true);
231         prob.solve();
232         prob.printSolution();
233
234
235     }
236
237 }
```

4.4 A step-by-step description of the inventory problem

In this section we present an inventory management example to illustrate the process of modeling and solving an MDP with the `jMDP` module of jMarkov. We also show how a user can implement an MDP model using `jMDP` and then, due to jMarkov's flexibility, call this implementation from a different software program and use a different tool to solve it.

4.4.1 An inventory management model

Consider the following problem. A car dealership sells only one type of car and uses a weekly (periodic) inventory review system. Each car is bought at a cost c and sold at a price p. The dealership must pay a fee K per truck for carrying the cars from the distributor to its location, and each truck can carry at most L cars. The dealership has a maximum capacity of M cars, and orders arrive instantly. If a customer places an order and there are no cars available, the sale is lost. We assume a fixed inventory holding cost of h per car and week. The demands for cars each week, D_n, are independent, identically distributed Poisson random variables with a mean of λ cars per week. The objective is to find an optimal ordering policy that maximizes weekly profits.

The problem is an infinite-horizon, discrete-time stochastic decision-making problem, and the objective is to minimize the long run average cost. The time periods are weeks because the inventory review occurs weekly. We model it as DTMDP with events, as this description is more natural than without events (`jMDP` supports both options). In the following we describe the step-by-step mathematical modeling process and the implementation in `jMDP`.

Define the states. Let X_n be the level of physical inventory at the end of week n. The state space is $\mathcal{S} = \{0, 1, ..., M\}$. In the code below we declare the class `InvLevel`, which represents the state. It extends `PropertiesState` which is used to represent states as arrays of integers (in this case the array has only one entry). In line 2 we provide a constructor for the class. In the interest of space, we will only include key portions of the code. Ellipses (...) indicate that further code is used; in this case for example, the class includes methods such as `getLevel` to return the inventory level.

```
1 public class InvLevel extends PropertiesState {
2         public InvLevel(int k) {super(new int[] {k});}
3 (...) }
```

Define the potential actions. Let a_n represent the size of the order placed at the start of week n. In the code below we create the class `Order`, which represents the actions and extends `Action`. We define the field `size` in line 2 to represent the amount ordered, and in line 3 we provide an appropriate constructor.

```
1 public class Order extends Action {
2         private int size;
3         Order(int k) {size = k;}
4 (...) }
```

Define the events. Here the events are the random demands e_n that occur each week. Notice that events occur after action a_n is taken. The event definition below includes two variables. An integer `d`, represents the size of the demand. And a boolean variable `greaterThan`, which takes the value "true" if the demand `d` is greater than or equal to the total inventory $X_{n-1} + a_n$ at the beginning of the period, and "false" otherwise. Here we extend the class `PropertiesEvent`, which represents events as arrays of integers. In lines 3-5 we provide an appropriate constructor.

```
1 public class DemandEvent extends PropertiesEvent {
2         private boolean greaterThan;
3         public DemandEvent(int d, boolean greater) {
4                 super(new int[] { d });
5                 greaterThan = greater;}
6 (...)}
```

Define the MDP. This is a DTMDP, therefore we extend the class `DTMDPev`. When extending this class, we use the corresponding classes that represent the states, actions and events. In our example, `InvLevel`, `Order` and `DemandEvent` represent the states, actions, and events, respectively. In the following code `CarDealerProblem` is the class representing the problem, and it includes fields, not shown for brevity, for each problem parameter, namely `maxInventory`, `truckSize`, `fixedCost`, `lambda`, `price`, `cost`, `holdCost`, and `truckCost`.

```
1  public class CarDealerProblem extends DTMDPEv<InvLevel, Order, DemandEvent> {(...)}
```

Define the feasible actions. For each state i the feasible actions are those that do not exceed the dealership's capacity. The maximum feasible order in state i is thus $M - i$, an amount that we calculate in line 2 in the code below, and the feasible set of actions is $\mathcal{A}(i) = \{0, \ldots, M - i\}$. The method `feasibleActions` receives the state i as a parameter. In line 3 we create an empty set of actions. In the loop in lines 4-6 we add each of these actions to the set.

```
1  public Actions<Order> feasibleActions(InvLevel i){
2          int max = maxInventory - i.getLevel();
3          ActionsSet<Order> actionSet = new ActionsSet<Order>();
4          for (int n = 0; n <= max; n++){
5                  actionSet.add(new Order(n));
6          }
7          return actionSet;}
```

Define the active events. For each state i, and given that action a is taken, we have to define the events that can occur. For example, the zero-demand event is active in every state, while a demand equal to $i + a$ is indistinguishable from larger demands as it empties the available inventory. We define the method `activeEvents`, which starts by creating an empty set of events `eventSet` in line 2, to which we add the active events. Since each event is defined by both the amount demanded and the boolean `greaterThan` indicating whether the demand empties the inventory or not, we specify each event with these two parameters. In line 3 we add the event where the demand is at least equal to $i + a$ and `greaterThan` is true. The loop in lines 4-6 adds events where the demands is less than $i + a$, and set `greaterThan` to false as the inventory remains positive after the demand event.

```
1  public Events<DemandEvent> activeEvents(InvLevel i, Order a) {
2          EventsSet<DemandEvent> eventSet = new EventsSet<DemandEvent>();
3          eventSet.add(new DemandEvent(i.getLevel() + a.getSize(), true));
4          for (int n = 0; n < i.getLevel() + a.getSize(); n++) {
5                  eventSet.add(new DemandEvent(n, false));
6          }
7    return eventSet;}
```

Define the set of reachable states Here we define the states that the MDP can transition to from state i, given that action a is taken and event e occurs. In the method `reachable`, displayed below, we create a new set of states in line 2. If the demand event is greater than $(i + a)$ the new state is 0, as depicted in line 4. Otherwise, if the demand is d, the only reachable state is $(i + a - d)$, as set in line 6.

```
1  public States<InvLevel> reachable(InvLevel i, Order a, DemandEvent e) {
2          StatesSet<InvLevel> stSet = new StatesSet<InvLevel>();
3          if (e.getGreaterThan())
4                  stSet.add(new InvLevel(0));
5          else
6                  stSet.add(new InvLevel(i.getLevel() + a.getSize() - e.getDemand()));
7          return stSet;}
```

Define the event probabilities. We condition on the event $d < i + a$. The probability of going from state i to a reachable state j when action a is taken is given by

$$p_{ij}(a) = \begin{cases} P\{D_n = d\} & \text{if } j = i + a - d,\ d < i + a, \\ P\{D_n \geq i + a\} & \text{if } j = 0,\ d \geq i + a, \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

In the code below, `demCCDF` denotes the cumulative distribution function of the demand, and `demPMF` its probability mass function. These values have been previously generated and correspond to a Poisson distribution. The condition in line 2 is equivalent to the second case in (7), while the complementary case in line 4 is equivalent to the first case in (7).

```
1 public double prob(InvLevel i, DemandEvent e) {
2         if (e.getGreaterThan())
3                 return demCCDF[e.getDemand()];
4         return demPMF[e.getDemand()];}
```

Define the immediate cost As `jMDP` assumes a minimization objective function, we minimize the negative of the net profit, defined in the method `immediateCost` in the code below. The profit has three major components. First, the revenues, which are calculated as the selling price times the expected sales $p \times (\mathbb{E}[D_n] - L_{D_n}[i+a])$, where L_{D_n} is the first-order loss function of the demand distribution [**?**]. The expected sales are calculated in lines 2 and 3 of the code below. Second, the ordering cost includes a charge per truck and a charge per car, and when the truck is only partially occupied the whole truck is charged, thus it is given by $K\left\lceil \frac{a}{L} \right\rceil + ca$. This cost is computed in lines 7 and 8 below. Third, the holding cost, which depends only on the state and is charged only when the stock is positive. Hence, it can be calculated as $h \times i$. The immediate cost is therefore given by:

$$c(i,a) = -\left(p \times (\mathbb{E}[D_n] - L_{D_n}[i+a]) - K\left\lceil \frac{a}{L} \right\rceil - c \times a - h \times i\right)$$

as calculated in lines 4 and 5 below.

```
1 public double immediateCost(InvLevel i, Order a) {
2         int maxSale = i.getLevel() + a.getSize();
3         double expectedSales = expDemand - demandLoss1[maxSale];
4         double netProfit = price * expectedSales - orderCost(a.getSize())- holdCost*i.getLevel();
5         return -netProfit;
6 }
7 double orderCost(int x) {
8         return truckCost * Math.ceil((double) x / truckSize) + x * cost;}
```

Generate and solve the model. In this method we set the values of the parameters to define a specific instance of the problem. As an example, the values of the parameters for this instance are $M = 10$, $L = 4$, $\lambda = 9$, $p = 1100$, $c = 500$, $h = 50$, $K = 1000$, which are set in lines 2 and 3 below. In the next lines we generate an instance of the problem with the given parameters and solve it. This is where the state-space search algorithm is executed to build the whole state space $\mathcal{S}$. Finally, we call the solver, and print the solution. We use the default Relative Value Iteration Algorithm, the solver takes the model object as input.

```
1 public static void main(String a[]) throws SolverException {
2         int maxInventory = 10; int truckSize = 4; double lambda = 9; double price = 1100;
3         double cost = 500; double holdCost = 50; int truckCost = 1000;
4         CarDealerProblem prob = new CarDealerProblem(maxInventory, truckSize, fixedCost,
5                 lambda, price, cost, holdCost, truckCost);
6         prob.solve();
7         prob.printSolution();
8 }
```

The results for this problem are stored as a `Solution` object, and the last line above prints the following optimal policy.

```
1 STATE       ------> ACTION
2 LEVEL 0     ------> ORDER 8 UNITS
3 LEVEL 1     ------> ORDER 8 UNITS
4 LEVEL 2     ------> ORDER 8 UNITS
5 LEVEL 3     ------> ORDER 7 UNITS
6 LEVEL 4     ------> ORDER 4 UNITS
7 (...)
```

This policy contains the optimal action to be taken in each possible state. For instance, if the current inventory level is 4 then it is optimal to order 4 cars. This example can be found in [**?**]. In fact, a finite horizon variation of this example is included as a one of the `jUnit` tests used to test the framework for correctness.

4.4.2 Modeling with jMarkov and solving with another tool

The flexibility of jMarkov, coupled with the fact that it is implemented in Java, provides the user with multiple alternatives for solving larger problems. The user can choose to model and solve the problem using only `jMDP` as in the previous section. Alternatively, the user can build the model with `jMDP`, exploiting the modeling capabilities of the module, and then use a different tool for the solution step. This option can be carried out in three ways: (i) by generating the model in `jMDP`, exporting the parameters and then importing them to another tool; or, (ii) by writing a solver class in Java using the jMarkov framework, which invokes the desired solver tool (this is the way LP solvers work in `jMDP`); or, (iii) by importing a model constructed with `jMDP` into another tool in order to solve it. In Section **??** we followed the third option to use SMCSolver in MATLAB to solve a QBD model built with jMarkov. Here we follow a similar procedure to import the `jMDP` model developed in the previous section into MATLAB and use MDPtoolbox [**?**] to solve it.

The code below shows how to import a `jMDP` model into MATLAB and use the functions provided by MDPtoolbox to find a solution. Line 1 imports the model class `CarDealerProblem`, which we described in detail in Section 4.4.1. Lines 3 and 4 define the model parameters, and Line 6 creates the model object. Lines 7 and 8 generate the model parameters that the MDPtoolbox solver uses as input. Specifically, the method `getTheP` generates a 3-dimensional array of transition probabilities $p_{ij}(a)$, whereas the method `getTheR` generates a matrix of immediate costs $c(i,a)$. Notice that the cost matrix is multiplied by -1 because the default setting of MDPtoolbox is maximization, while the costs are calculated for a minimization problem. Finally, line 10 calls one of the MDPtoolbox solver functions to solve the problem and return the optimal policy, long-run average cost and solution time.

```
1  import examples.jmdp.CarDealerProblem;
2
3  maxInventory = 10; truckSize = 4; lambda = 7.0; truckCost = 800.0;
4  price = 1100.0; cost = 500.0; holdCost = 50.0;
5
6  model=CarDealerProblem(maxInventory, truckSize, truckCost, price, cost, holdCost, lambda);
7  P=model.getTheP();
8  R=-1*model.getTheR();
9
10 [policy, cost, cpu_time] = mdp_relative_value_iteration (P, R);
```

This example illustrates how the modeling capabilities of jMarkov can be exploited to build a complex model using events and to solve it with a tool that does not support MDP models with events. But, because `jMDP` automatically converts the event-dependent model into a DTMDP without events and automatically calculates the non-event-dependent version of the parameters, the process is completely seamless for the user. This could not have been achieved without jMarkov. If the user only had MDPtoolbox available, she would have had to manually generate the parameters for the MDP with events and transform it into a DTMDP without events in order to solve it.

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 `State`s 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 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 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 minimize the total cost 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 cost 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 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 discountFactor) that only receives InfiniteMDP<S,A> objects as an argument. This solver maximizes the discounted cost ${}_{DR}v^\pi$ 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.

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