**jsdp: a Java Stochastic Dynamic Programming Library**

Roberto Rossi  
*Business School*  
*University of Edinburgh*  
Edinburgh, UK  
roberto.rossi@ed.ac.uk

**Abstract**—Stochastic Programming is a framework for modelling and solving problems of decision making under uncertainty. Stochastic Dynamic Programming is a branch of Stochastic Programming that takes a “functional equation” approach to the discovery of optimal policies. By leveraging constructs — lambda expressions, functional interfaces, collections and aggregate operators — implemented in Java to operationalise the MapReduce framework, jsdp provides a general purpose library for modelling and solving Stochastic Dynamic Programs.

**Index Terms**—stochastic dynamic programming; java; jsdp

## I. INTRODUCTION

Stochastic Programming [1] is a framework for modeling and solving problems of decision making under uncertainty. Stochastic Dynamic Programming, originally introduced by Bellman in his seminal book “Dynamic Programming” [2], is a branch of Stochastic Programming that deals with multistage decision processes and takes a “functional equation” approach to the discovery of optimum policies.

In “Reminiscences about the origins of linear programming” [3], Dantzig stressed the pivotal role that declarative modelling — the ability to state objectives and constraints in clear terms, as opposed to developing ad-hoc heuristics for generating solutions - played in the development of (Linear) “Programming;” a tool he developed to “compute more rapidly a time-staged deployment, training and logistical supply program.”

Powell recently remarked that the field of mathematical programming has benefited tremendously from a common canonical framework consisting of decision variables, constraints, and an objective function, while stochastic optimisation has not enjoyed this common declarative framework [4]. In other words, whilst declarative modelling has been a staple of Operations Research since these early days, to the best of my knowledge, no declarative modelling framework has been developed for Stochastic Dynamic Programming.

By leveraging declarative constructs — lambda expressions, functional interfaces, collections and aggregate operators — that have recently been implemented in Java to operationalise the MapReduce [5] framework, jsdp aims to tackle this gap and provide a general purpose Java library [6] for modelling and solving Stochastic Dynamic Programs.

Our library exploits built-in parallelisation available in Java and paves the way towards a high level modelling framework that may seamlessly leverage BigData computing environments such as Hadoop and Apache Spark. In this sense, it contributes to Freuder’s quest in Pursuit of the Holy Grail [6]: “The user states the problem, the computer solves it.”

The rest of this work is structured as follows. In Section II we outline the general structure of a stochastic dynamic program; in Section III we survey relevant language constructs leveraged by the library; in Section IV we illustrate how these language constructs can be used to implement a general purpose approach to modelling and solving stochastic dynamic programs; in Section V we introduce our new library; in Section VI we briefly survey a range of applications; finally, in Section VII we survey related works and draw conclusions.

## II. STRUCTURE OF A STOCHASTIC DYNAMIC PROGRAM

Stochastic dynamic programming [2] is a branch of stochastic programming that deals with multistage decision processes and takes a “functional equation” approach to the discovery of optimal policies. In what follows, we adopt the canonical framework in [4] to capture the dimensions of a stochastic optimisation problem. Without loss of generality, we will consider a cost minimisation setting. To model a problem via stochastic dynamic programming one has to specify:

- a **planning horizon** comprising \( n \) periods;
- the finite set \( S_t \) of possible **states** in which the system may be found in period \( t \), for \( t = 1, ..., n \);
- the finite set \( A_t \) of possible **actions** that may be taken in state \( s \in S_t \);
- the transition probability \( p_{s,j}^{t} \) from state \( s \in S_t \) towards state \( j \in S_{t+1} \), when action \( a \in A_t \) is taken;
- the **expected immediate cost** \( c_{t}(s,a) \) incurred if action \( a \in A_t \) is taken in state \( s \in S_t \) at the onset of period \( t \);
- the **discount factor** \( \alpha \);
- the **functional equation** \( f_{t}(s) \) denoting the minimum expected total cost incurred over periods \( t, t+1, ..., n \); if the system is in state \( s \) at the beginning of period \( t \).

The decision maker’s goal is to minimise the expected (discounted) total cost \( \mathbb{E}[f_{n}(s)] \) over the planning horizon. The functional equation typically takes the following structure

\[
f_{t}(s) = \min_{a \in A_t} c_{t}(s,a) + \alpha \sum_{j \in S_{t+1}} p_{s,j}^{t} f_{t+1}(j),
\]

\[1\] The jsdp library is available at: [http://gwr3n.github.io/jsdp/](http://gwr3n.github.io/jsdp/)

\[2\] or to maximise the expected (discounted) total reward.

---

1 The jsdp library is available at: [http://gwr3n.github.io/jsdp/](http://gwr3n.github.io/jsdp/)

2 or to maximise the expected (discounted) total reward.
where the boundary condition of the system is \( f_{n+1}(s) \equiv 0 \), for all \( s \in S_{n+1} \). The goal is to determine \( f_1(s) \), where \( s \) is the state of the system at the beginning of the first period.

### A motivating example: stochastic inventory control

Consider a 3-period inventory control problem. At the beginning of each period the firm should decide how many units of a product should be produced. If production takes place for \( x \) units, where \( x > 0 \), we incur a production cost \( c(x) \). This cost comprises both fixed (\( K = 3 \)) and variable (\( v = 2 \)) components: \( c(x) = 0 \), if \( x = 0 \); \( c(x) = K + vx \), otherwise. Production in each period cannot exceed 4 units. Demand in each period takes two possible values: 1 or 2 units with equal probability (0.5). Demand is observed in each period only after production has occurred. After meeting current period’s demand holding cost \( h = 1 \) per unit is incurred for any item that is carried over from one period to the next. Because of limited capacity the inventory at the end of each period cannot exceed 3 units. All demand should be met on time (no backorders). If at the end of the planning horizon (i.e. period 3) the firm still has units in stock, these can be salvaged at cost \( b = 2 \) per unit. The initial inventory is 1 unit.

We formulate the stochastic inventory control problem as a stochastic dynamic program.

- there are \( n = 4 \) periods in the planning horizon;
- the state \( s \in S_t \) in period \( t \) represents the initial inventory level at the beginning of period \( t \), where \( S_t = \{1, \ldots, 3\} \), for \( t = 1, \ldots, n \);
- the action \( a \) given state \( s \) in period \( t \) is the order quantity \( Q \), where \( A_s = \{0, \ldots, 4-s\} \);
- the transition probability \( p_{s,j}^a \) from state \( s \in S_t \) towards state \( j \in S_{t+1} \), when action \( a \in A_j \) is taken, immediately follows from the probability distribution of the demand \( d \) in period \( t \);
- the expected immediate cost incurred if action \( a \in A_s \) is taken in state \( s \in S_t \) at the beginning of period \( t = 1, \ldots, 3 \) is
  \[
  c_t(s, a) = \begin{cases} 
  K + va + hE[\max(s + a - d, 0)] & a > 0 \\
  hE[\max(s + a - d, 0)] & \text{otherwise}
  \end{cases}
  \]
  finally, in period \( t = 4 \), \( c_4(s, Q) = bE[\max(s + a - d, 0)] \), where \( E \) denoted the expected value;
- the discount factor \( \alpha \) is 1;
- the functional equation is \( f_t(s) = \min c_t(s, a) + E[f_{t+1}(s + a - d)] \), with boundary condition \( f_{n+1}(s) = \min c_4(s, a) \). The aim is to determine \( f_1(1) \).

Given this functional equation, an optimal policy can be obtained via forward recursion or backward recursion. In Section [IV] we shall present a novel approach for implementing a forward recursion algorithm using modelling constructs originally introduced in Java 8, which are next surveyed in Section [III]. The discussion in these two sections will provide insights on the nature of the jsdp optimisation engine, which leverages the MapReduce framework originally discussed in [5] and seamlessly takes advantage of its parallelism and scalability.

### III. LANGUAGE CONSTRUCTS

In order to implement a compact forward recursion algorithm in Java we will rely on lambda expressions, functional interfaces, collections, and aggregate operations. We next survey each of these constructs in order.

#### A. Lambda calculus and lambda expressions

In computer programming, an anonymous function (also known as lambda function or lambda expression) is a function definition that is not bound to an identifier. Anonymous functions originate in the work of Church and in his invention of the lambda calculus [7]. Java supports anonymous functions, named Lambda Expressions, starting with JDK 8. In Java, a lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method. The following lambda expression captures the immediate value function for a given state, action and demand value.

```java
(state, action, demand) -> {
  double cost = (action > 0 ? 3 + 2*action : 0);
  double h = state.initialInventory + action - demand;
  cost += h*(state.initialInventory + action - demand);
  return cost;
}
```

#### B. Functional interfaces

In Java, any interface with a single abstract method is a functional interface, and its implementation may be treated as lambda expressions. The most simple and general case of a lambda is a functional interface with a method that receives one value and returns another. This function of a single argument is represented by the `Function` interface, which is parameterized by the types of its argument and a return value. For instance, we may use the following functional interface to capture the behaviour of a function that, given a state, generates an array of feasible actions, each of which is modelled as a double value.

```java
@FunctionalInterface
interface StateTransitionFunction <S, A, R> {
  public R apply (S s, A a, R r);
}
```

More complex functional interfaces, which may receive or return more than one value, may be defined by leveraging the `@FunctionalInterface` annotation. For instance, we may define the following bespoke functional interface to capture a state transition function.

```java
@FunctionalInterface
interface StateTransitionFunction <S, A, R> {
  public S apply (S s, A a, R r);
}
```

#### C. Collections and aggregate operations

The Java collections framework is a set of classes and interfaces that implement commonly reusable collection data structures.
Collections are used to store objects and retrieve them efficiently. Rather than accessing elements in a collection using traditional for loop constructs, one may opt for aggregate operations instead. For instance, given a collection actions, one may print all actions in it as follows.

```
actions.stream().forEach(a -> System.out.println(a));
```

A stream is a sequence of elements. Unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source through a pipeline. A pipeline is a sequence of aggregate operations, which contains the following components: a source collection, zero or more intermediate operations, which produce a new stream, and a terminal operation that produces a non-stream result, such as a primitive value (like a double value), a collection, or in the case of aggregate operation for each, no value at all.

A key aggregate operation is the map function, which takes a lambda expression as argument, and returns a new stream consisting of the results of applying the given lambda expression to the elements of the stream. Java streams and aggregate operations find their origin in the seminal work introducing the MapReduce framework. Aggregate operations sum and average are available for computing the respective functions. Finally, aggregate operator filter is used to return a stream that contains a subset of the elements in the original stream that meet a certain condition expressed as a lambda expression.

**IV. STOCHASTIC DYNAMIC PROGRAMMING IN JAVA**

In this section, we illustrate how to implement a forward recursion algorithm in Java by relying on modelling constructs introduced in Java 8, which we illustrated in the previous section. The reader should note that the discussion in this section does not rely on the jsdp library; conversely, the aim here is to illustrate core modelling constructs and abstractions which jsdp is built upon.

To illustrate the concepts we are about to introduce, we will rely once more upon the motivating example previously introduced: the stochastic inventory control problem. To implement our forward recursion algorithm, we will rely on the following libraries.

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.DoubleStream;
```

In file `InventoryControl.java`, we create the following class, which will contain our code.

```java
public class InventoryControl {
    int planningHorizon;
    double[][] pmf;
    ...
}
```

Member variables `planningHorizon` denotes the number of periods in the planning horizon, while `pmf` is a two-dimensional array that records a given probability mass function describing random demand in each period.

We define the following constructor

```java
public class InventoryControl {
    ...
    public InventoryControl(int planningHorizon, double[][] pmf) {
        this.planningHorizon = planningHorizon;
        this.pmf = pmf;
    }
    ...
}
```

and we introduce a nested class `State` to model the state of the system.

```java
public class InventoryControl {
    ...
    class State{
        int period;
        int initialInventory;
        public State(int period, int initialInventory){
            this.period = period;
            this.initialInventory = initialInventory;
        }
        public double[] getFeasibleActions(){
            return actionGenerator.apply(this);
        }
        @Override
        public int hashCode(){
            String hash = "";
            hash = (hash + period) + "_" + this.initialInventory;
            return hash.hashCode();
        }
        @Override
        public boolean equals(Object o){
            if(o instanceof State) {
                return ((State) o).period == this.period &&
                ((State) o).initialInventory ==
                this.initialInventory;
            } else
                return false;
        }
        @Override
        public String toString(){
            return this.period + " * " + this.initialInventory;
        }
    }
    ...
}
```

Method `hashCode()` is needed because we will store states in hashtables, which require each state to be uniquely identified by a hashcode for direct indexing; method `getFeasibleActions()` relies on `actionGenerator`, a function defined as follows.

```java
public class InventoryControl {
    ...
    Function<State, double[]> actionGenerator;
    ...
}
```

One should recall that for each state, we must be able to generate all feasible actions. For the moment, we leave `actionGenerator` unimplemented. We will later define an appropriate lambda expression that returns the appropriate set of actions for each relevant state.
In addition to the above functional interface we also define the functional interface `StateTransitionFunction`. We define public class `InventoryControl` to capture the state transition function, a function that, given a state, an action, and a random outcome, returns the associated future state; and the immediate value function, a function that, given a state, an action, and a random outcome, returns the associated immediate cost/profit.

We have now defined all relevant constructs that are necessary to set up our forward recursion procedure, which is presented in Figure 1. The procedure directly implements the functional equation $f_t(s)$. It relies on memoization — method `computeIfAbsent()` — to store the value of the functional equation for states that have been already visited. This ensures that states are not processed more than once.

Finally, we define our main method.

We set up the problem parameters and we call relevant methods to obtain an optimal solution.

After compiling and running the code the output obtained is $f_1(1)=16.25$, and $b_1(1)=3.0$. 

```
public class InventoryControl {
    public static void main(String [] args){
        // set up problem parameters
        int initialPeriod = 1;
        int initialInventory = 1;
        State initialState = inventory.new State(initialPeriod, initialInventory);
        System.out.println("f_1(1)="+inventory.f(initialState));
        System.out.println("b_1(1)="+inventory.cacheActions.get(inventory.new State(initialPeriod, initialInventory)));
    }

    public static void main(String [] args){
        InventoryControl inventory = new InventoryControl(planningHorizon, pmf);
        // call relevant methods
        System.out.println("f_1(1)="+inventory.f(initialState));
        System.out.println("b_1(1)="+inventory.cacheActions.get(inventory.new State(initialPeriod, initialInventory)));
    }
}
```
public class InventoryControl {
  ...
  Map<State, Double> cacheActions = new HashMap<>();
  Map<State, Double> cacheValueFunction = new HashMap<>();
  double f(State state) {
    return cacheValueFunction.computeIfAbsent(state, s -> {
      double val = Arrays.stream(s.getFeasibleActions())
        .map(orderQty -> Arrays.stream(pmf)
          .mapToDouble(p -> p[1] * immediateValueFunction.apply(s, orderQty, p[0]) +
            (s.period < this.planningHorizon ? p[1] * f(stateTransition.apply(s, orderQty, p[0])) : 0))
          .sum())
        .min().getAsDouble();
      double bestOrderQty = Arrays.stream(s.getFeasibleActions())
        .filter(orderQty -> Arrays.stream(pmf)
          .mapToDouble(p -> p[1] * immediateValueFunction.apply(s, orderQty, p[0]) +
            (s.period < this.planningHorizon ? p[1] * f(stateTransition.apply(s, orderQty, p[0])) : 0))
          .sum() == val)
        .findAny().getAsDouble();
      cacheActions.putIfAbsent(s, bestOrderQty);
      return val;
    });
  }
  ...
}

Fig. 1. Generic forward recursion procedure, which leverages lambda expressions, functional interfaces, collections, and aggregate operations

V. THE JSDP LIBRARY

The example presented in the previous section leveraged high level Java modelling constructs (lambda expressions, functional interfaces, collections and aggregate operators) to implement a forward recursion algorithm and tackle a specific stochastic dynamic programming problem in the realm of inventory control.

JSDP builds upon these modelling constructs and provides an additional layer of abstraction to model problems of decision making under uncertainty via Stochastic Dynamic Programming.

The library features off-the-shelf algorithms — forward recursion and backward recursion — as well as scenario reduction techniques to tackle generic problems of decision making under uncertainty with univariate or multivariate state descriptors.

In what follows, we will survey key modelling constructs offered by JSDP and we will demonstrate its flexibility on a well-known problem from stochastic inventory control [9].

The key abstractions required to model a stochastic dynamic program are provided in package jsdp.sdp. More specifically,

- **State**: this class is used to represent an abstract state;
- **Action**: this class is used to represent an abstract action;
- **TransitionProbability**: this class is used to represent transition probabilities;
- **ImmediateValueFunction**: this functional interface is used to capture an abstract immediate value function;
- **StateTransitionFunction** and **RandomOutcomeFunction**: these functional interfaces are used to capture an abstract state transition function.

The abstraction for the functional equation is provided by class Recursion, which generalises classes BackwardRecursion and ForwardRecursion, which in turn implement general purpose backward and forward recursion algorithms, respectively.

In order to model a problem with JSDP one may define concrete implementations of abstract classes in package jsdp.sdp. An example of this approach is given in package jsdp.app.lotsizing. This is a cumbersome solution that should be adopted only for complex problems.

In most cases, it is sufficient to rely upon general purpose concrete implementations provided in package jsdp.sdp.impl. In what follows, we shall illustrate this latter solution. A generic skeleton for a stochastic dynamic program developed in JSDP is provided in package jsdp.app.skeleton, the following example extends this skeleton.

A. Scarf’s stochastic lot sizing problem

In this section, we provide a JSDP implementation for modelling and solving the well-known stochastic lot sizing problem investigated by Scarf in [9].

We consider an $n$-period inventory control problem. At the beginning of each period the firm should decide how many units of a product should be produced. If production takes place for $x$ units, where $x > 0$, we incur a production cost $c(x)$. This cost comprises both a fix $K$ and a variable $v$ component:

$$c(x) = \begin{cases} K + vx & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

The order is delivered immediately at the beginning of the period. Demand in each period $t$ is Poisson distributed with...
known mean. Demand is observed in each period only after production has occurred. After meeting current period’s demand holding cost $h$ per unit is incurred for any item that is carried over from one period to the next. Unmet demand in any given period is backordered at cost $b$ per unit per period. The initial inventory is known.

We formulate the stochastic inventory control problem as a stochastic dynamic program, and then proceed to illustrate a jdp implementation.

- there are $n$ periods in the planning horizon;
- the state $s \in S_t$ in period $t$ represents the initial inventory level at the beginning of period $t$;
- the action $a$ given state $s$ in period $t$ is the order quantity;
- the transition probability $p_{s|j}^a$ from state $s \in S_t$ towards state $j \in S_{t+1}$, when action $a \in A_t$ is taken, immediately follows from the probability distribution of the demand $d$ in period $t$;
- the immediate cost incurred if action $a \in A_t$ is taken in state $s \in S_t$ at the beginning of period $t$ is $c(s,a) = K + va + bE[\max(s+a-d,0)] - bE[\min(s+a-d,0)]$, if $a \geq 0$, and $c(s,a) = hE[\max(s+a-d,0)] - bE[\min(s+a-d,0)]$, otherwise.
- the discount factor $\alpha$ is 1;
- the functional equation is $f_1(s) = \min c_i(s,a) + E[f_{t+1}(s+a-d)]$, with boundary condition $f_n(s) = \min c_n(s,a)$. Let $i$ be initial inventory, the aim is to determine $f_1(i)$.

We next illustrate how to model this stochastic inventory control problem in jdp.

We create a class StochasticInventoryControl and build a main method as follows.

```java
public class InventoryControl {
    ...
    public static void main(String[] args) {
        ...
    }
}
```

First, we define problem parameters.

```java
// Random variables
Distribution[] distributions = IntStream.iterate(0, i->i+1) .limit(meanDemand.length) .mapToDouble(i -> distributions[i].inverseF(1 - truncationQuantile)) .toArray(Distribution[]::new);
```

We create two arrays storing the lower bound and the upper bound of random variable supports.

```java
double[] supportLB = IntStream.iterate(0, i -> i + 1) .limit(meanDemand.length) .mapToDouble(i -> distributions[i].inverseF(1 - truncationQuantile)) .toArray();
double[] supportUB = IntStream.iterate(0, i -> i + 1) .limit(meanDemand.length) .mapToDouble(i -> distributions[i].inverseF(truncationQuantile)) .toArray();
```

We define a variable representing the initial inventory level.

```java
double initialInventory = 0;
```

We then proceed to the model definition; the first step is to characterize the state space.

```java
// State space
double stepSize = 1; // Step size must be 1 for discrete distributions double minState = -50; // Inventory level lower bound in each period double maxState = 150; // Inventory level upper bound in each period
```

The following instruction bounds the state space.

```java
StateImpl.setStateBoundaries(stepSize, minState, maxState);
```

We introduce a functional interface that dynamically computes the `ArrayList<Action>` feasibleActions storing feasible actions for a given state $s$.

```java
// Actions
Function<State, ArrayList<Action>> buildActionList = s -> {
    StateImpl state = (StateImpl) s;
    ArrayList<Action> feasibleActions = new ArrayList<Action>();
    for(double i = state.getInitialState();
        i <= stateImpl.getMaxState();
        i += stateImpl.getStepSize()) {
        feasibleActions.add(new ActionImpl(state, 0 - state.getInitialState()));
    }
    return feasibleActions;
};
```

The next functional interface defines the idempotent action: an action that does not affect the state variable.

```java
Function<State, Action> idempotentAction = s -> new ActionImpl(s, 0);
```
The following two functional interfaces define the immediate value function

```java
// Immediate Value Function
ImmediateValueFunction<State, Action, Double>
ImmediateValueFunction = (initialState, action, finalState) -> {
    ActionImpl a = (ActionImpl)action;
    StateImpl fs = (StateImpl)finalState;
    double orderingCost =
        a.getAction() > 0 ? (fixedOrderingCost +
            a.getAction() * proportionalOrderingCost) : 0;
    double holdingAndPenaltyCost =
        holdingCost * Math.max(fs.getInitialState(), 0) +
        penaltyCost * Math.max(-fs.getInitialState(), 0);
    return orderingCost + holdingAndPenaltyCost;
};
```

and the random outcome function,

```java
// Random Outcome Function
RandomOutcomeFunction<State, Action, Double>
randomOutcomeFunction = (initialState, action, finalState) -> {
    double realizedDemand =
        ((StateImpl)initialState).getInitialState() +
        ((ActionImpl)action).getAction() -
        ((StateImpl)finalState).getInitialState();
    return realizedDemand;
};
```

which — given present state, future state, and action chosen — returns the associated random outcome. This is essentially equivalent to defining the dynamics — i.e. state transition function — of the system; adopting a random outcome function in lieu of a state transition function is useful if one aims to rely on sampling for speeding up the computation and obtain an approximate policy.

To summarize, in order to model a problem the key steps are the following: bound the state space, define the functional interfaces to compute the action list, the immediate value function, and the random outcome function for the system.

Once these modeling components are defined, we can move to the solution process. The first step is to determine if we want to sample the state space or to adopt an exhaustive state space enumeration. SamplingScheme.NONE carries out an exhaustive state space enumeration; in the following example we will adopt simple random sampling.

The maximum sample size (maxSampleSize) refers to the sample size adopted for random variables at stage 1; if the reduction factor per stage (reductionFactorPerStage) is 1, this sample size will be adopted also in subsequent stages, otherwise the sample size in subsequent stages will shrink exponentially fast according to the rule

where period denotes the stage for which we are computing the the sample size. We name this approach sample waning. To the best of our knowledge, a similar approach has not yet been discussed in the literature. It should be noted that the appeal of sample waning in the context of a forward or backward recursion algorithm is related to the advantage brought by memoization. A combination of sample waning and forward recursion in jsdp has been successfully adopted to solve large instances of a routing problem discussed in [10].

Finally, we proceed and apply backward recursion to solve our problem. The discount factor (discountFactor) captures value discounting from one period to the next. stateSpaceLowerBound and loadFactor are parameters utilised to initialise hashtables. In this specific instance, we use HashType.THASHMAP as hash table to store the state space. Enum HashType contains other possible choices, including a hash table (MapDB) that provide disk storage in place of RAM for large state spaces.

Backward recursion is invoked as follows.

```java
double discountFactor = 1.0;
int stateSpaceLowerBound = 100000000;
float loadFactor = 0.8f;
BackwardRecursionImpl recursion = new
    BackwardRecursionImpl(OptimisationDirection.MIN,
        distributions,
        supportLB,
        supportUB,
        immediateValueFunction,
        randomOutcomeFunction,
        buildActionList,
        idempotentAction,
        discountFactor,
        samplingScheme,
        maxSampleSize,
        reductionFactorPerStage,
        stateSpaceLowerBound,
        loadFactor,
        HashType.THASHMAP);
```

The maximum sample size (maxSampleSize) refers to the sample size adopted for random variables at stage 1; if the reduction factor per stage (reductionFactorPerStage) is 1, this sample size will be adopted also in subsequent stages, otherwise the sample size in subsequent stages will shrink exponentially fast according to the rule

[https://mapdb.org/]
After running the code the output obtained is the following.

---Backward recursion---
Expected total cost (assuming an initial inventory level
0.0) : 567.753717866613
Optimal initial action: 91.0
Time elapsed: 10
CPU usage: 168% (4 cores)

jsdp exploits parallelisation made available off-the-shelf by Java streams, this is evidenced in the CPU usage statistics.

VI. APPLICATIONS

Besides Scarf’s stochastic lot sizing model [9] presented in the previous section, which is available in package jsdp.app.lotsizing, the library features further sample applications in stochastic inventory control (package jsdp.app.inventory), an application in maintenance scheduling (package jsdp.app.maintenance), an application in stochastic optimal control (Constrained Linear Quadratic Gaussian Problem [11], package jsdp.app.clgq), and an application in stochastic vehicle routing [10] (package jsdp.app.routing).

VII. RELATED WORKS AND CONCLUSIONS

There exist several commercial algebraic modelling languages (AML) available in the market (e.g. AMPL, OPL). These languages, also known as “Mathematical Programming” (MP) languages, let a user express a mathematical model by using conventional mathematical notation. A compiler translates these high level models to a form understandable by solvers for mathematical programs, such as CPLEX [12], which then find an optimal solution. The counterpart of MP for modelling problems of decision making under uncertainty is “Stochastic Programming” (SP) [11]. In addition to problem parameters, decision variables, constraints and an objective function, SP features a number of high level concepts for modelling uncertainty: e.g. random variables, stochastic constraints, and decision stages. Over fifteen years ago, [13] discussed the fact that most of the difficulties to model uncertainty through SP originate from the lack of an agreed standard of representation and of a widely accepted syntax for a description of stochastic programs. This message has recently been reinforced in [4].

In the past decade significant research and commercial efforts have been directed towards the development of effective modelling languages for stochastic programming, i.e. Stochastic AML (SAML). A number of stochastic extensions has been proposed for several well known existing AML: SMPS [14–16], SAMLP [17–20], XPRESS-SP [21], Stochastic GAMS and AIMMS, Pyomo AML [22, 23] and Stochastic OPL [24]. Libraries such as ApproxRL and OpenAI have been proposed for reinforcement learning [25], in which however the environment is modelled as a Markov Decision Process.

To the best of our knowledge, no attempt has been made to date to develop a high level declarative modelling and solution framework for modelling and solving problems of decision making under uncertainty formulated as stochastic dynamic programs. This work introduced a new Java modelling and solution framework, the jsdp library, to address this gap.

---

REFERENCES

[1] J. R. Birge and F. Louveaux, Introduction to stochastic programming, 2nd ed., ser. Springer Series in Operations Research and Financial Engineering. New York, NY: Springer, Jun. 2011.
[2] R. Bellman, Dynamic Programming. Princeton, NJ, USA: Princeton University Press, 1957.
[3] G. B. Dantzig, “Reminiscences about the origins of linear programming,” Operations Research Letters, vol. 1, no. 2, pp. 43–48, apr 1982.
[4] W. B. Powell, “A unified framework for optimization under uncertainty,” in Optimization Challenges in Complex, Networked and Risky Systems. INFORMS, oct 2016, pp. 45–83.
[5] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” Commum. ACM, vol. 51, no. 1, p. 107–113, jan 2008.
[6] E. Freuder, “In pursuit of the holy grail,” ACM Comput. Surv., vol. 28, no. 4es, p. 63–es, dec 1996.
[7] A. Church, “A formula of the simple theory of types,” Journal of Symbolic Logic, vol. 5, no. 2, pp. 56–68, jun 1940.
[8] D. Michie,”“Memo’ functions and machine learning,” Nature, vol. 218, no. 5136, pp. 19–22, apr 1968.
[9] H. E. Scarf, “Optimality of (s, S) policies in the dynamic inventory problem,” in Mathematical Methods in the Social Sciences, K. J. Arrow, S. Karlin, and P. Suppes, Eds. Stanford, CA: Stanford University Press, 1960, pp. 196–202.
[10] R. Rossi, M. Tomasella, B. Martin-Barragan, T. Embley, C. Walsh, and M. Langston, “The dynamic bowser routing problem,” European Journal of Operational Research, vol. 275, no. 1, pp. 108–126, may 2019.
[11] B. sen Chen and C. tsung Kuo, “An LQG optimal design in the stochastic system with saturated driving,” in 1986 25th IEEE Conference on Decision and Control. IEEE, dec 1986.
[12] IBM ILOG CPLEX, “V12. 1: User’s Manual for CPLEX,” International Business Machines Corporation, vol. 46, no. 53, p. 157, 2009.
[13] E. Fragnirte and J. Gondzio, “7, stochastic programming from modelling languages,” in Applications of Stochastic Programming. Society for Industrial and Applied Mathematics, jan 2005, pp. 95–113.
[14] J. Birge, M. Dempster, H. Gassmann, E. Gunn, A. King, and S. Wallace, “A standard input format for multiperiod stochastic linear programs,” COAL Newsletter, vol. 17, pp. 1–19, 1987.
[15] H. Gassmann and A. Ireland, “On the automatic formulation of stochastic linear programs,” Annals of Operations Research, vol. 64, no. 1, pp. 83–112, 1996.
[16] H. I. Gassmann and E. Schweitzer, “A comprehensive input format for stochastic linear programs,” Annals of Operations Research, vol. 104, no. 1, pp. 89–125, Apr. 2001.
[17] J. Thiébé, C. Delft, and Vial, “Automatic formulation of stochastic programs via an algebraic modeling language,” Computational Management Science, vol. 4, no. 1, pp. 17–40, Jan. 2007.
[18] P. Valente, G. Mitra, M. Sadki, and R. Fourer, “Extending algebraic modelling languages for stochastic programming,” The Centre for the Analysis of Risk and Optimisation Modelling Applications (CARISA), Brunel University, Tech. Rep. CTR/09/03, Nov 2003.
[19] M. Colombo, A. Grothey, J. Hogg, K. Woodsend, and J. Gondzio, “A structure-conveying modelling language for mathematical and stochastic programming,” Mathematical Programming Computation, vol. 1, no. 4, pp. 223–247, Dec. 2009.
[20] R. Fourer and L. Lopes, “Stampi: A filtration-oriented modeling tool for multistage stochastic recourse problems,” INFORMS J. on Computing, vol. 21, no. 2, pp. 242–256, Apr. 2009.
[21] A. Dormer, A. Vazacopoulos, N. Verma, and H. Tipi, “Modeling & solving stochastic programming problems in supply chain management using xpress-sp,” in Supply Chain Optimization, ser. Applied Optimization. J. Geunes and P. M. Pardalos, Eds. Springer US, 2005, vol. 1, no. 4, pp. 307–354.
[22] J.-P. Watson, D. L. Woodruff, and W. E. Hart, “Pysp: modeling and solving stochastic programs in python,” Mathematical Programming Computation, vol. 4, pp. 109–149, 2012.
[23] B. Knuven, D. Mildebrah, C. Muir, J. Sirola, D. Woodruff, and J.-P. Watson, "mpi-sppy: Optimization under uncertainty for pyomo,” 11 2020, [Online]. Available: https://wwwosti.gov/biblio/1721756
[24] S. Tarim, S. Manandhar, and I. Walsh, “Stochastic constraint programming: A Scenario-Based approach,” Constraints, vol. 11, no. 1, pp. 53–80–80, Jan. 2006.
[25] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. The MIT Press, 2018.