jMarkov User's Guide

Germán Riaño and Julio Góez
Universidad de Los Andes

Contents

1 Introduction

The jMarkov project has been in development since 2002 by the research grup COPA at Universidad de los Andes.

The main purpose of jMarkov is facilitating the development and application of large sacale Markovian models, so that they can be used by engineers with basic programming and stochastic skills.

The project is composed by four modules

- jMarkov
- jQBD
- jPhase
- jMDP

In this manual we explain jMarkov and jQBD, which are used to build Markov Chains and Quasi-Birth and death processes (QBD). The other two modules have their own manulas.

With jPhase a user can easily manipulate Phase-Type distributions (PH). These distibutions are quite flexible and powerful, and a model that is limited to PH in practical terms can model many situations. For details see [8] and [7]

jMDP is used to build and solve Markov Decision Process (MDP). MDP, or, as is often called, Probabilistic Dynamic Programming allows the analyst to design optimal control rules for a Markov Chain. jMDP works for discrete and continous time MDPs. For details see [11] and [10]

For up-to date information, downloads and examples check COPA's web page at `copa.uniandes.edu.co`.

2 Building Large - Scale Markov Chains

In this section, we will describe the basic algorithms used by jMarkov to build Markov Chains. Although we limit our description to Continuous Time Markov Chain (CTMC), jMarkov can handle also Discrete Time Markov Chains (DTMC).

Let $\{X(t), t \geq 0\}$ be a CTMC, with finite space state $\mathcal{S}$ and generator matrix $\mathbf{Q}$, with components

$$q_{ij} = \lim_{t \downarrow 0} P\{X(t) = j | X(0) = i\} \quad i, j \in \mathcal{S}.$$

It is well known that this generator matrix, along with the initial conditions, completely determines the transient and stationary behavior of the Markov Chain (see, e.g, [4]). The diagonal components q_{ii} are non-positive and represent the exponential holding rate for state i, whereas the off diagonal elements q_{ij} represent the transition rate from state i to state j.

The transient behavior of the system is described by the matrix $\mathbf{P}(t)$ with components

$$p_{ij}(t) = P\{X(t+s) = j | X(s) = i\} \quad i, j \in \mathcal{S}.$$

This matrix can be computed as

$$\mathbf{P}(t) = e^{\mathbf{Q}t} \quad t > 0.$$

For an irreducible chain, the stationary distribution $\boldsymbol{\pi} = [\pi_1, \pi_2, \ldots,]$ is determined as the solution to the following system of equations

$$\begin{aligned} \boldsymbol{\pi}\mathbf{Q} &= \mathbf{0} \\ \boldsymbol{\pi}\mathbf{1} &= 1, \end{aligned}$$

where $\mathbf{1}$ is a column vector of ones.

2.1 Space state building algorithm

Transitions in a CTMC are triggered by the occurrence of events such as arrivals and departures. The matrix $\mathbf{Q}$ can be decomposed as $\mathbf{Q} = \sum_{e \in \mathcal{E}} \mathbf{Q}^{(e)}$, where $\mathbf{Q}^{(e)}$ contains the transition rates associated with event e, and $\mathcal{E}$ is the set of all possible events that may occur. In large systems, it is not easy to know in advance how many states there are in the model. However, it is possible to determine what events occur in every state, and the destination states produced by each transition when it occurs. jMarkov works based on this observation, using an algorithm similar to the algorithm buildRS presented by Ciardo [1]; see Figure 1. The algorithm builds the space state and the transition rate by a deep exploration of the graph. It starts with an initial state i_0 and searches for all other states. At every instant, it keeps a set of "unchecked" states $\mathcal{U}$ and the set of states $\mathcal{S}$ that have been already checked. For every unchecked state the algorithm finds the possible destinations and, if they had not been previously found, they are added to the $\mathcal{U}$ set. To do this, it first calls the function `active` that determines if an event can occur. If it does, then the possible destination states are found by calling the function `dests` . The transition rate is determined by calling the function `rate` . From this algorithm, we can see that a system is fully described once the states and events are defined and the functions `active`, `dests`, and `rate` have been specified. As we will see, modeling a problem with jMarkov entails coding these three functions.

```
$\mathcal{S} = \emptyset, \mathcal{U} = \{i_0\}$, $\mathcal{E}$ given.
while $\mathcal{U} \neq \phi$ do
  for all $e \in \mathcal{E}$ do
    if active$(i,e)$ then
      $\mathcal{D} :=$ dests$(i,e)$
      for all $j \in \mathcal{D}$ do
        if $j \notin \mathcal{S} \cup \mathcal{U}$ then
          $\mathcal{U} := \mathcal{U} \cup \{j\}$
        end if
        $R_{ij} := R_{ij} +$ rate$(i,j,e)$
      end for
    end if
  end for
end while
```

Figure 1: BuildRS algorithm

2.2 Measures of Performance

When studying Markovian systems, the analyst is usually interested in the transient and steady state behavior of measures of performance (MOPs). This is accomplished by attaching rewards to the model. Let $\mathbf{r}$ be a column vector such that $r(i)$ represents the expected rate at which the system receives rewards whenever it is in state $i \in \mathcal{S}$. Here the term *reward* is used for any measure of performance that might be of interest, not necessarily monetary. For example, in queueing systems $r(i)$ might represent the number of entities in the system,or the number of busy servers, when the state is i. The expected reward rate at time t is computed according to

$$E\big(r(X(t)\big) = \mathbf{a}\mathbf{P}(t)\mathbf{r},$$

where the row vector $\mathbf{a}$ has the initial conditions of the process (i.e., $a_i = P\left\{X(0) = i\right\}, i \in \mathcal{S}$). Similarly, for an irreducible CTMC, the long run rate at which the system receives rewards is calculated as

$$\lim_{t\to\infty} \frac{1}{t}\int_0^t E\big(r(X(s)\big)ds = \boldsymbol{\pi}\mathbf{r}.$$

As we will see, jMarkov provides mechanisms to define this type of rewards and can compute both, transient and steady state MOPs. There are other type of rewards, like expected time in the system, which can be easily computed using Little law.

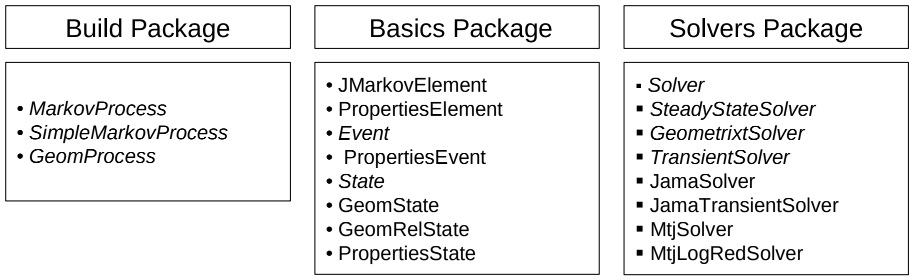


Figure 2: Class classification

3 Framework Design

In this section, we give a brief description of jMarkov's framework architecture. We start by describing object-oriented programming and then describe the three packages that compose jMarkov.

3.1 Java and Object Oriented Programming

Java is a programming language created by Sun Microsystems [12]. The main characteristics that Sun intended to have in Java are: Object-Oriented, robust, secure, architecture neutral, portable, high performance, interpreted, threaded and dynamic.

Object-Oriented Programming (OOP) is not a new idea. However, it did not have an increased development until recently. OOP is based on four key principles: abstraction, encapsulation, inheritance and polymorphism. An excellent explanation of OOP and the Java programming language can be found in [13].

The abstraction capability is the one that interests us most. Java allows us to define abstract types like `MarkovProcess`, `State`, etc. We can also define *abstract* functions like `active`, and `dests`. We can program the algorithm in terms of these abstract objects and functions and the program works independently of the particular implementation of the aforementioned elements. All the user has to do is to *implement* the abstract functions. What is particularly nice is that if a function is declared as abstract, then the compiler itself will force the user to implement it before she attempts to run the model.

3.2 Build Package

The build package is the main one in jMarkov since it contains the classes that take care of building the state space and transition matrices. The main classes are `MarkovProcess`, `SimpleMarkov-Process`, and `GeomProcess` (see Figure 3). Whereas the first two allow to model general Markov processes, `GeomProcess` is used for Quasi-Birth and Death Processes (QBD) and its description is given in Section 5.3 below.

The class `SimpleMarkovProcess` represents a Markov chain process, and contains three abstract methods that implement the three aforementioned functions in the algorithm BuildRS: `active`, `dests`, and `rate`. In order to model a problem the user has to extend this class and implement the three functions. An example is given in Section 5.4. The class `MarkovProcess` is the main class in the module, and provides a more general mechanism to describe the dynamics of the system. It also contains tools to communicate with the solvers to compute steady state and transient solutions, and print them in a diverse array of ways. For details, see [9].

3.3 Basic Package

This package contains the building blocks needed to describe a Markov Chain. It contains classes such as `State`, and `Event`, which allow the user to code a description of the states and events,

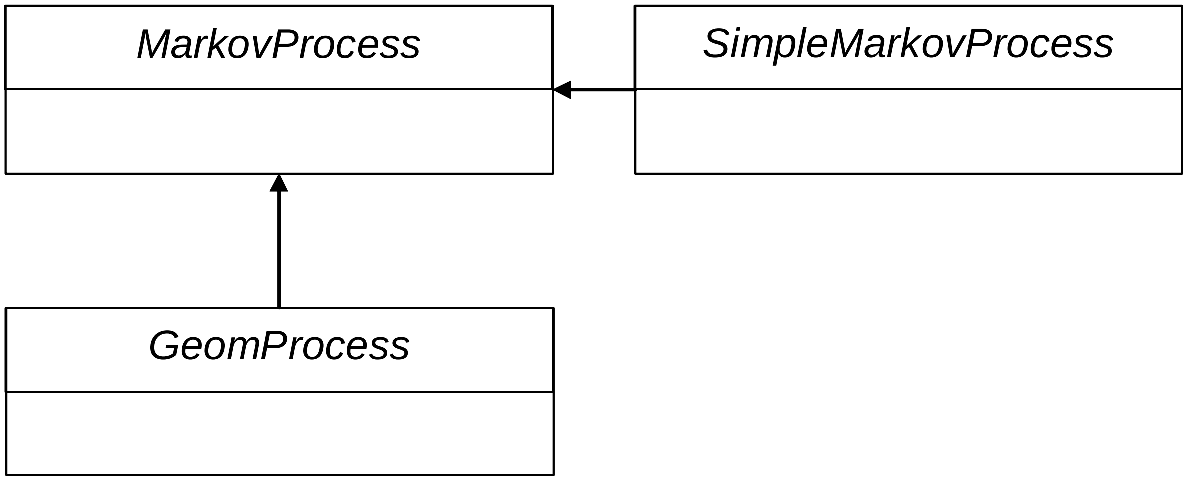


Figure 3: Class diagram build module

respectively (see Figure 4). The user has freedom to choose any particular coding that best describes the states in her model, like any combination of integers, strings, etc. However, she must establish a complete ordering among the elements since, for efficiency, jMarkov works with ordered sets. For simplicity, however, a built-in class is provided, called `PropertiesState`, that describes the state with an array of integers, something which is quite appropriate for many applications. Similarly, there is an analogous class called `PropertiesEvent`. The package also contains the classes `States` and `Events` that are used to describe collections of states and events. These are fairly general classes, since all that is required from the user is to provide a mechanism to "walk through" the elements of the set, taking advantage of Java iterator mechanism. This implies that, for large sets, there is no need to generate (and store) all the elements in the set. For convenience, the package provides implementations of these set classes based on sorted sets classes available in Java.

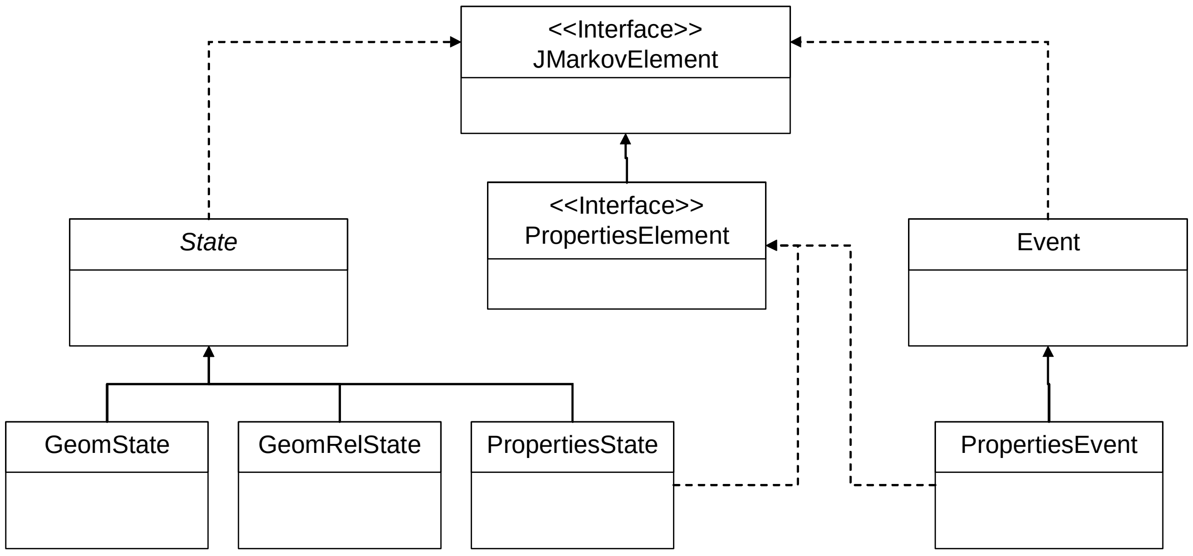


Figure 4: Class diagram for the basic package

3.4 The Solvers Package

As stated above, jMarkov separates modeling from solving. Various solvers are provided to find steady-state and transient probabilities (see Figure 5). If the user does not specify the solver to use, one is provided by default. However, the architecture is flexible enough to allow an interested user to choose a different solver, or, if she desires, to implement her own. The basic class is called `Solver`, that has two sub-classes called `SteadyStateSolver`, `TransientSolver`, and `GeomSolver` (see Figure 5). As the names indicate, the first two provide solvers for steady state and transient probabilities, whereas the latter is used for QBDs, as explained in section 5. The implementations provided relay on two popular Java packages to handle matrix operations JAMA [3] and MTJ [2], for dense and sparse matrices, respectively.

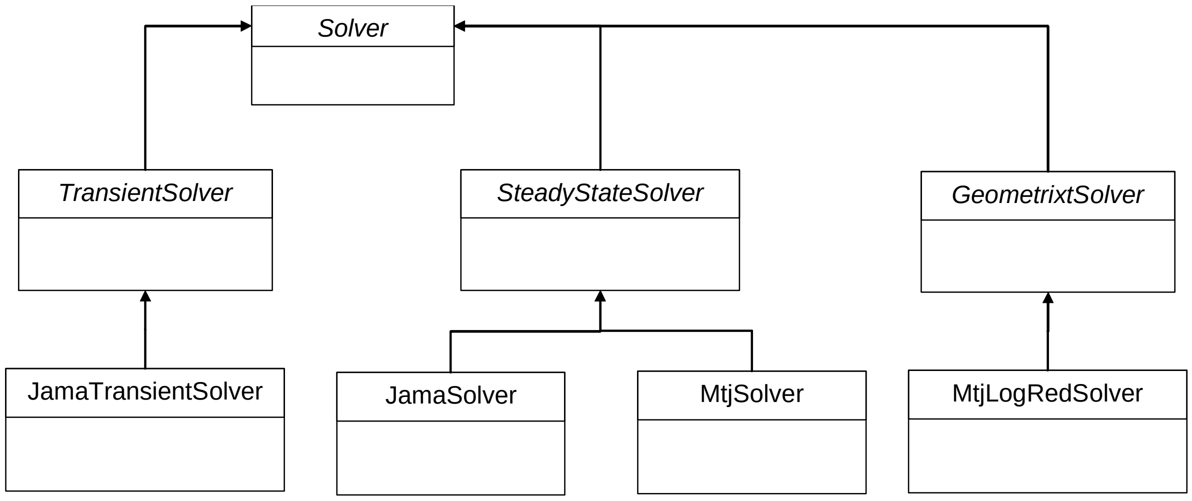


Figure 5: Class diagram of the solvers package

4 Examples

4.1 Example: An M/M/2/N with different servers

Assume that a system has Poisson arrivals with rate λ. There are two exponential servers with rates μ_1 and μ_2 respectively. There is a maximum of N customers in the system. An arriving customer that finds the system empty will go to server 1 with probability α. Otherwise he will pick he first available server, or join a single FCFS queue. If there are N in the system the customer goes away.

4.1.1 The model

We model this system with the triple $\mathbf{X}(t) = (X(t), Y(t), Z(t))$, where $X(t)$ and $Y(t)$ represents the status of the server (1 if busy 0 otherwise) and $Z(t)$ represents the number in queue, which is a number from 0 to $N-2$. There are $2 \times 2 \times N - 2$ potential states, however not all combinations of X, Y and Z are possible. For example the state $(0, 1, 2)$ is not acceptable since we assume that a server will not be idle if there are people in the queue. The set of states will be of the form

$$\mathcal{S} = \{(0,0,0), (0,1,0), (1,0,0)\} \cup \{(1,1,k) : k = 0, 1, \ldots, N-2\}$$

The transition matrix will have the form

	000	010	100	110	111	112	...	1,1,N-3	1,1,N-2
000		$\lambda\alpha$	$\lambda(1-\alpha)$						
010	μ_2			λ					
100	μ_1			λ					
110		μ_1	μ_2		λ				
111						λ			
112					$\mu_1+\mu_2$				
$\vdots$									
1,1,N-3									λ
1,1,N-2								$\mu_1+\mu_2$	

4.1.2 Class MM2dNState

Our characterization of each state fits nicely as a particular case of the PropertiesState class with three properties. To model the State we begin by creating a constructor that assigns x, y, and z to the properties. We provide methods to access the three properties and a method to check whether the system is empty. We also implement the method label to override the one in the class PropertiesState. We provide and illustration of the implementation of the State for this example in the code snippet below. The detailed source code for this class may be explored inside the file QueueMM2dN.java.

```java
class MM2dNState extends PropertiesState {
    MM2dNState(int x, int y, int z) {
        super(3); // Creates a PropertiesState with 3 properties.
        this.prop[0] = x;
        this.prop[1] = y;
        this.prop[2] = z;
    }

    @Override
    public void computeMOPs(MarkovProcess mp) {
        setMOP(mp, "Status Server 1", getStatus1());
        setMOP(mp, "Status Server 2", getStatus2());
        setMOP(mp, "Queue Length", getQSize());
        setMOP(mp, "Number in System", getStatus1() +
                    getStatus2() + getQSize());
    }
    public int getStatus1() {
        return prop[0];
    }
    public int getStatus2() {
        return prop[1];
    }
    public int getQSize() {
        return prop[2];
    }
    boolean isEmpty() {
        return (getStatus1() + getStatus2() + getQSize() == 0);
    }
    @Override
    public String label() {
        String stg = "0";
        if ((getStatus1() == 1) && (getStatus2() == 0))
            stg = "1A";
        if ((getStatus2() == 1) && (getStatus1() == 0))
            stg = "1B";
        if ((getStatus2() == 1) && (getStatus1() == 1))
            stg = "" + (2 + getQSize());
        return stg;
    }

}
```

4.1.3 Class QMM2dNEvent

There are two basic events that may occur: arrivals and service completions. We have to distinguish two types of service completions depending on whether the server that finishes is 1 or 2, which are labeled as DEPARTURE1 and DEPARTURE2. Also, when the system is empty we have to distinguish between arrivals that go to server 1 and those that go to server 2, which are labeled as ARRIVAL1 and ARRIVAL2. Finally, when at most one server is available an arrival is simply labeled ARRIVAL. Hence, in total we have five events. We provide and illustration of the implementation of the Event class for this example in the code snippet below. The detailed sorce code for this class may be explored inside the file QueueMM2dN.java.

```java
class QMM2dNEvent extends Event {
    /** Event types */
    public enum Type {
        /** An arrival */
        ARRIVAL,
        /** Arrival to server 1 (only for emtpy system) */
        ARRIVAL1,
        /** Arrival to server 2 (only for emtpy system) */
        ARRIVAL2,
        /** departure from server 1 */
        DEPARTURE1,
        /** departure from server 2 */
        DEPARTURE2;
    }

    private Type type;

    /**
     * @param type
     */
    public QMM2dNEvent(Type type) {
        super();
        this.type = type;
    }

    /**
     * @return Returns the type.
     */
    public final Type getType() {
        return type;
    }

    /**
     * @return the set of all events.
     */
    public static EventsSet<QMM2dNEvent> getAllEvents() {
        EventsSet<QMM2dNEvent> evSet = new EventsSet<QMM2dNEvent>();
        for (Type type : Type.values())
            evSet.add(new QMM2dNEvent(type));
        return evSet;
    }
}
```

4.1.4 Class QueueMM2dN

For this example we extend `SimpleMarkovProcess`. In the following code illustrate the system implementation with the class `QueueMM2dN`.

```java
public class QueueMM2dN extends SimpleMarkovProcess<MM2dNState, QMM2dNEvent> {
    // Events
    final int ARRIVAL = 0;
    final int ARRIVAL1 = 1; // only for empty system
    final int ARRIVAL2 = 2; // only for empty system
    final int DEPARTURE1 = 3;
    final int DEPARTURE2 = 4;
    private double lambda;
    private double mu1, mu2, alpha;
    private int N;
    public QueueMM2dN(double lambda, double mu1,
                double mu2, double alpha, int N) {
        super((new MM2dNState(0, 0, 0)), //
                QMM2dNEvent.getAllEvents()   ); // num Events
        this.lambda = lambda;
        this.mu1 = mu1;
        this.mu2 = mu2;
        this.alpha = alpha;
        this.N = N;
    }
     */
    public @Override boolean active(MM2dNState i, QMM2dNEvent e) {
        boolean result = false;
        switch (e.getType()) {
```

```
        case ARRIVAL:
            result = ((i.getQSize() < N - 2) && (!i.isEmpty()));
            break;
        case ARRIVAL1:
            result = i.isEmpty();
            break;
        case ARRIVAL2:
            result = i.isEmpty();
            break;
        case DEPARTURE1:
            result = (i.getStatus1() > 0);
            break;
        case DEPARTURE2:
            result = (i.getStatus2() > 0);
            break;
        }
        return result;
    }

    public @Override States<MM2dNState> dests(MM2dNState i, QMM2dNEvent e) {
        int newx = i.getStatus1();
        int newy = i.getStatus2();
        int newz = i.getQSize();

        switch (e.getType()) {
        case ARRIVAL:
            if (i.getStatus1() == 0) {
                newx = 1;
            } // serv 1 desocupado
            else if (i.getStatus2() == 0) {
                newy = 1;
            } // serv 2 desocupado
            else { // ambos ocupados
                newz = i.getQSize() + 1;
            }
            break;
        case ARRIVAL1:
            newx = 1;
            break;
        case ARRIVAL2:
            newy = 1;
            break;
        case DEPARTURE1:
            if (i.getQSize() != 0) {
                newx = 1;
                newz = i.getQSize() - 1;
            } else {
                newx = 0;
            }
            break;
        case DEPARTURE2:
            if (i.getQSize() != 0) {
                newy = 1;
                newz = i.getQSize() - 1;
            } else {
                newy = 0;
            }
            break;
        }
        return new StatesSet<MM2dNState>( new MM2dNState(newx, newy, newz));
    }

    public @Override double rate(MM2dNState i,MM2dNState j, QMM2dNEvent e) {
        double res = 0;
        switch (e.getType()) {
        case ARRIVAL:
            res = lambda;
            break;
        case ARRIVAL1:
            res = lambda * alpha;
            break;
        case ARRIVAL2:
            res = lambda * (1 - alpha);
            break;
        case DEPARTURE1:
            res = mu1;
            break;
        case DEPARTURE2:
            res = mu2;
            break;
        }
        return res;
     */
    public static void main(String[] a) {
        String stg;
        BufferedReader rdr = new BufferedReader(
                new InputStreamReader(System.in));
        try {
            System.out.println("Input rate ");
            stg = rdr.readLine();
            double lda = Double.parseDouble(stg);
            System.out.println("Service rate 1  ");
            stg = rdr.readLine();
            double mu1 = Double.parseDouble(stg);
            System.out.println("Service rate 2  ");
            stg = rdr.readLine();
            double mu2 = Double.parseDouble(stg);
            System.out.println("Provide alpha  ");
            stg = rdr.readLine();
            double alpha = Double.parseDouble(stg);
            System.out.println("Max in the system ");
            stg = rdr.readLine();
            int N = Integer.parseInt(stg);
            QueueMM2dN theQueue = new QueueMM2dN(lda, mu1, mu2, alpha, N);
            theQueue.showGUI();
            theQueue.printAll();
        } catch (IOException e) {
        }
        ;
    }

} // class end
```

4.2 Multiple Server Queue

In this example we generalize what we did in the previous example. Assume that a system has exponential arrivals with exponential arrivals. There are K distinct servers with service rates $\mu_1, \mu_2, \ldots, \mu_K$. A customer that finds all servers busy joins a single FCFS queue, with capacity $N - K$ (so there will be at most N customers in the system). A customer that finds all servers idle will choose among the idle servers according to relative intensities α_k, i.e., he will choose server k with probability

$$\beta_k = \frac{\alpha_k}{\sum_{\ell \in \mathcal{I}} \alpha_\ell}, \qquad k \in \mathcal{I}$$

where $\mathcal{I}$ is the set of available servers.

4.2.1 The model

For this model we characterize each state by $X(t) = (S(t), Q(t))$, where $S(t) = (S_1(t), \ldots, S_K(t))$, where $S_k(t) = 1$ if k-th server is busy and 0 otherwise. The events that can occur are arrivals and departures. However we have to distinguish two type of arrivals. If there is no idle server the arriving customer joins the queue, and we will call this a non-directed arrival. Otherwise we call it a directed arrival. We also make part of the event description the server where the arrival is directed. In order to represent this event we need a more sophisticated structure, so instead of just numbering the events we rather extend the class Event, creating an object with two integer fields (components): the type and the server. Then it is very easy to implement the functions `active`, `dest` and `rate` just by querying the values of the type and server associated with the state.

5 Modeling Quasi-Birth and Death Processes

In this section we give a brief description of Quasi-Birth and Death Processes (QBD), and explain how they can be modeled using jMarkov. QBD are Markov Processes with an infinite space state, but with a very specific repetitive structure that makes them quite tractable.

5.1 Quasi-Birth and Death Processes

Consider a Markov process $\{X(t) : t \geq 0\}$ with a two dimensional state space $\mathcal{S} = \{(n, i) : n \geq 0, 0 \leq i \leq m\}$. The first coordinate n is called the *level* of the process and the second coordinate i is called the *phase*. We assume that the number of phases m is finite. In applications, the level usually represents the number of items in the system, whereas the phase might represent different stages of a service process.

We will assume that, in one step transition, this process can go only to the states in the same level or to adjacent levels. This characteristic is analogous to a Birth and Death Process, where the only allowed transitions are to the two adjacent states (see, e.g [4]). Transitions can be from state (n, i) to state (n', i') only if $n' = n$, $n' = n - 1$ or $n' = n + 1$, and, for $n \geq 1$ the transition rate is independent of the level n. Therefore, the generator matrix, $\mathbf{Q}$, has the following structure

$$\mathbf{Q} = \begin{bmatrix} \mathbf{B}_{00} & \mathbf{B}_{01} & & & \\ \mathbf{B}_{10} & \mathbf{A}_1 & \mathbf{A}_0 & & \\ & \mathbf{A}_2 & \mathbf{A}_1 & \mathbf{A}_0 & \\ & & \ddots & \ddots & \ddots \end{bmatrix},$$

where, as usual, the rows add up to 0. An infinite Markov Process with the conditions described above is called a Quasi-Birth and Death Process (QBD).

In general, the level zero might have a number of phases $m_0 \neq m$. We will call these first m_0 states the *boundary states*, and all other states will be called *typical states*. Note that matrix $\mathbf{B}_{00}$ has size $m_0 \times m_0$, whereas $\mathbf{B}_{01}$ and $\mathbf{B}_{10}$ are matrices of sizes $(m_0 \times m)$ and $(m \times m_0)$, respectively. Assume that the QBD is an ergodic Markov Chain. As a result, there is a steady state distribution $\boldsymbol{\pi}$ that is the unique solution $\boldsymbol{\pi}$ to the system $\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}$, $\boldsymbol{\pi}\mathbf{1} = 1$. Divide this $\boldsymbol{\pi}$ vector by levels, analogously to the way $\mathbf{Q}$ was divided, as

$$\boldsymbol{\pi} = [\boldsymbol{\pi}_0, \boldsymbol{\pi}_1, \ldots].$$

Then, it can be shown that a solution exist that satisfy

$$\boldsymbol{\pi}_{n+1} = \boldsymbol{\pi}_n \mathbf{R}, \qquad n > 1,$$

where $\mathbf{R}$ is a constant square matrix of order m [6]. This $\mathbf{R}$ is the solution to the equation

$$\mathbf{A}_0 + \mathbf{R}\mathbf{A}_1 + \mathbf{R}^2\mathbf{A}_2 = \mathbf{0}.$$

There are various algorithms that can be used to compute the matrix $\mathbf{R}$. For example, you can start with any initial guess $\mathbf{R}_0$ and obtain a series of $\mathbf{R}_k$ through iterations of the form

$$\mathbf{R}_{k+1} = -(\mathbf{A}_0 + \mathbf{R}_k^2\mathbf{A}_2)\mathbf{A}_1^{-1}.$$

This process is shown to converge (and $\mathbf{A}_1$ does have an inverse). More elaborated algorithms are presented in Latouche and Ramaswami [5]. Once $\mathbf{R}$ has been determined then $\boldsymbol{\pi}_0$ and $\boldsymbol{\pi}_1$ are determined by solving the following linear system of equations

$$\begin{bmatrix} \boldsymbol{\pi}_0 & \boldsymbol{\pi}_1 \end{bmatrix} \begin{bmatrix} \mathbf{B}_{00} & \mathbf{B}_{01} \\ \mathbf{B}_{10} & \mathbf{A}_1 + \mathbf{R}\mathbf{A}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \end{bmatrix}$$
$$\boldsymbol{\pi}_0\mathbf{1} + \boldsymbol{\pi}_1(\mathbf{I} - \mathbf{R})^{-1}\mathbf{1} = 1.$$

5.2 Measures of performance for QBDs

We consider two types of measures of performance that can be defined in a QBD model. The first type can be seen as a reward r_i received whenever the system is in phase i, independent of the level, for level $n \geq 1$. The long-run value for such a measure of performance is computed according to

$$\sum_{n=1}^{\infty} \boldsymbol{\pi}_n \mathbf{r} = \boldsymbol{\pi}_1 (\mathbf{I} - \mathbf{R})^{-1} \mathbf{r},$$

where $\mathbf{r}$ is an m-size column vector with components r_i. The second type of reward has the form nr_i, whenever the system is in phase i of level n. Its long-run value is

$$\sum_{n=1}^{\infty} n \boldsymbol{\pi}_n \mathbf{r} = \boldsymbol{\pi}_1 \mathbf{R} (\mathbf{I} - \mathbf{R})^{-2} \mathbf{r}.$$

5.3 Modeling QBD with jQBD

Modeling QBD with jMarkov is similar to modeling a Markov Processes. Again, the user has to code the states, the events, and then define the dynamics of the system through `active`, `dests`, and `rate`. The main difference is that special care needs to be taken when defining the destination states for the typical states. Rather than defining a new level for the destination state, the user should give a new *relative* level, which can be -1, 0, or $+1$. This is accomplished by using two different classes to define states. The current state of the system is a `GeomState`, but the destination states are `GeomRelState`. The process itself must extend the class `GeomProcess`, which in turn is an extension of `MarkovProcess`.

The building algorithm uses the information stored about the dynamics of the process to explore the graph and build only the first three levels of the system. From this, it is straightforward to extract matrices $\mathbf{B}_{00}$, $\mathbf{B}_{01}$, $\mathbf{B}_{10}$, $\mathbf{A}_0$, $\mathbf{A}_1$, and $\mathbf{A}_2$. Once these matrices are obtained, the stability condition is checked. If the system is found to be stable, then the matrices $\mathbf{A}_0$, $\mathbf{A}_1$, and $\mathbf{A}_2$ are passed to the solver, which takes care of computing the matrix $\mathbf{R}$ and the steady state probabilities vectors $\boldsymbol{\pi}_0$ and $\boldsymbol{\pi}_1$, using the formulas described above. The implemented solver (`MtjLogRedSolver`) uses the logarithmic reduction algorithm [5]. This class uses MTJ for matrices manipulations. There are also mechanisms to define both types of measures of performance mentioned above, and jQBD can compute the long run average value for all of them.

5.4 An Example

To illustrate the modeling process with jQBD, we will show the previous steps with a simple example. Consider a infinite queue with a station that has a single hiper-exponential server with n service phases, with probability α_i to reach the service phase i and with service rate μ_i at phase i, where $0 \leq i \leq n$. The station is fed from an external source according to a Poisson processes with rate λ. We will use this model as an illustrative example of a QBD process, and will show how each of the previous steps is performed for this example. Of course all measures of performnce for this system can be readilly obtained in closed form since it is a particular case of an $M/G/1$, but we chose this example bacause of its simplicity. The code below actually models any general phase-type distribution, so the hyper-geometric will be a particular case.

- **States:** Because of the memoryless property, the state of the system is fully characterized by an integer valued vector $\mathbf{x} = (x_1, x_2)$, where $x_1 \geq 0$ represents the number of items in the system and $0 \leq x_2 \leq n$ represents the current phase of the service process.Note that, knowing this, we can know how many items are in service and how many are queuing. It is important to highlight that the computational representation uses only the phase of the system (x_2) because the level (x_1)is manged internally by the framework.

- **Events:** An event occurs whenever an item arrives to the system or finishes processing at a particular service phase $0 \leq i \leq n$. Therefore, we will define the set of possible events as $\mathcal{E} = \{a, c_1, c_2, \ldots, c_n\}$, where the event a represents an arrival to the system and an event c_i represents the completion of a service in phase i.

- **Markov Process:** We elected to implement `GeomProcess`, which implied coding the following three methods:

  - `active (i,e)`: Since the queue is an infinite QBD process the event a is always active, and the events $c_i, 0 \leq i \leq n$ are active if there is an item at workstation on service phase i. The code to achieve this can be seen in Figure 6.

  - `dests (i,e,j)`: When the event a occurs there is always an increment on the system level, but you need to consider if the server is idle or busy. When the server is idle the new costumer could start in any of the n service phases, then the system could reach anyone of the first level n states with probability α_i. On the other hand, if the server is busy on service phase i, the system will reach the next level state with the same service phase i.

    On the other hand, when the server finishes one service c_i, no matter which phase type, the level of the system is reduced by one, but you need to consider if the system is in level 1 or if it is in level 2 or above. When the level is 1, the system reach the unique state $(0, 0)$ where there are no costumer in the system and the server is idle. On the other hand, if the system level is equal or greater than 2, the system could reach any of the n states in the level below with probability α_i. The Java code can be seen in Figure 7.

  - `rate (i,e)`: The rate of occurrence of event a is given simply by λ and the rate of occurrence of an event c_i is given by μ_i. In Figure 8 you can see the corresponding code.

- **MOPs:** Using the MOPS types defined in jQBD component, we will illustrate its use calculating the expected WIP on the system.

File HiperExQueue.java

```java
    public int getCurPH() {
        if (type == ARRIVAL)
            throw new IllegalArgumentException(
                    "Current phase is not defined for event " + ARRIVAL);
        return curPH;
    }

    /**
     * @return Returns the type.
     */
    public Type getType() {
        return type;
    }
```

Figure 6: `Active` method of class HiperExQueue.java

File HiperExQueue.java

```
                // finish in phase n
                E.add(new HiperExQueueEvent(FINISH_SERVICE, n));
            }
            return E;
        }

        @Override
        public String label() {
            String stg = "";
            switch (type) {
            case ARRIVAL:
                stg = "Arrival";
                break;
            case FINISH_SERVICE:
                stg = "Ph(" + curPH + ")";
            }
            return stg;
        }
}

/**
 *  * This class define the states in the queue.
 * @author Julio Goez - German Riano. Universidad de los Andes.
 */
class HiperExQueueState extends PropertiesState {

    /**
     * We identify the states with the curPH of server in station, (1,
     * ..,n) or 0 if idle.
```

Figure 7: `dests` method of class HiperExQueue.java

Finally, the output obtained after running the model can be seen in the Graphical User Interface (GUI) in Figure 9. There is no need to use the GUI, but it is helpful to do so during the first stages of development, to make sure that all transitions are being generated as expected. All the measures of performance defined can be extracted by convenience methods defined in the API or a report printed to standard output. Such a report can be seen in Figure 10.

6 Further Development

This project is in constant development and we would appreciate all the feedback we can receive.

References

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

[2] B. Heimsund. Matrix Toolkits for Java (MTJ), December 2005. Last modified: Monday, 05-Dec-2005 09:03:23 CET.

File HiperExQueue.java

```java
    /**
     * Returns the service phase of process
     * @return Service phase
     */
    public int getSrvPhase() {
        return this.prop[0];
    }

    /**
     * @see jmarkov.basic.State#isConsistent()
     */
    @Override
    public boolean isConsistent() {
        // TODO Complete
        return true;
    }

    /**
     * Returns the service status
     * @return Service status (1 = busy, 0 = free)
     */
    public int getSrvStatus() {
        return (getSrvPhase() == 0) ? 0 : 1;
    }

    @Override
    public HiperExQueueState clone() {
        return new HiperExQueueState(getSrvPhase());
    }
```

Figure 8: `rate` method of class HiperExQueue.java

[3] J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington. JAMA: A java matrix package, July 2005. MathWorks and the National Institute of Standards and Technology (NIST).

[4] V. Kulkarni. *Modeling and analysis of stochastic systems.* Chapman & Hall., 1995.

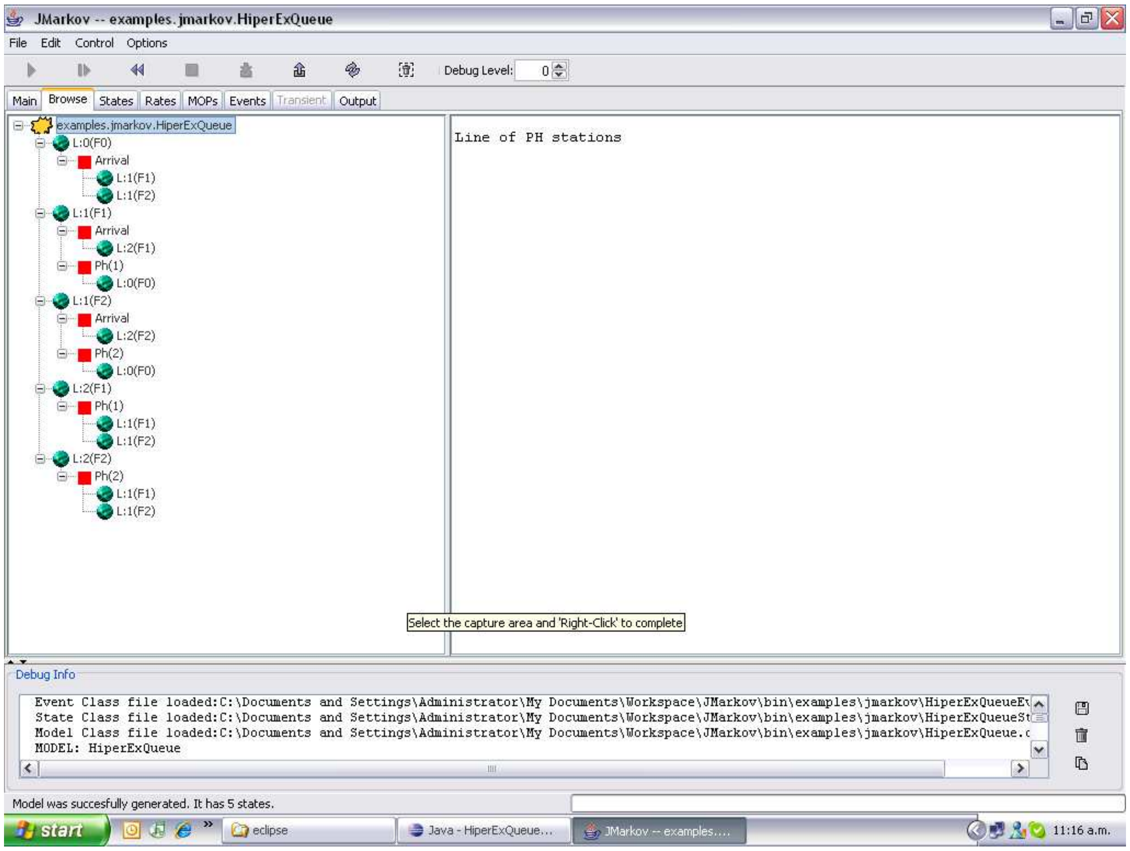
[5] G. Latouche and V. Ramaswami. *Introduction to matrix analytic methods in stochastic modeling.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1999.

[6] M. F. Neuts. *Matrix-geometric solutions in stochastic models.* The John Hopkins University Press, 1981.

[7] J. F. Pérez and G. Riaño. jPhase: an object-oriented tool for modeling Phase-Type distributions. In *SMCtools '06: Proceedings from the 2006 Workshop on Tools for Solving Structured Markov Chains*, New York, 2006. ACM Press.

[8] J. F. Pérez and G. Riaño. *jPhase User's Guide.* Universidad de los Andes, 2006.

[9] G. Riaño and J. Góez. *jMarkov User's Guide.* Industrial Engineering, Universidad de los Andes, 2005.



```
Line of PH stations
```

```
Event Class file loaded:C:\Documents and Settings\Administrator\My Documents\Workspace\JMarkov\bin\examples\jmarkov\HiperExQueueEv
State Class file loaded:C:\Documents and Settings\Administrator\My Documents\Workspace\JMarkov\bin\examples\jmarkov\HiperExQueueSt
Model Class file loaded:C:\Documents and Settings\Administrator\My Documents\Workspace\JMarkov\bin\examples\jmarkov\HiperExQueue.c
MODEL: HiperExQueue
```

Model was succesfully generated. It has 5 states.

Figure 9: GUI example of jMarkov

```
MEASURES OF PERFORMANCE

NAME                          MEAN      SDEV

Expected Level             0.14286         ?
Server Utilization         0.12500   0.33072
```

Figure 10: MOPs report of jMarkov

[10] G. Riaño and A. Sarmiento. jMDP: an object-oriented framework for modeling MDPs. Working paper. Universidad de los Andes, 2006.

[11] A. Sarmiento and G. Riaño. *jMDP User's Guide*. Industrial Engineering, Universidad de los Andes, 2005.

[12] Sun Microsystems. Java technology, Jan. 2006.

[13] P. van der Linden. *Just Java(TM) 2*. Prentice Hall, 6th edition, 2004.