Solving MDPs using the MDP package in R

Lars Relund Nielsen*

Department of Genetics and Biotechnology, University of Aarhus, P.O. Box 50, DK-8830 Tjele, Denmark, lars@relund.dk.

Anders Ringgaard Kristensen

Department of Large Animal Sciences, University of Copenhagen, Groennegaardsvej 2, DK-1870 Frederiksberg C, Denmark.

July 26, 2009

1 Introduction

The MDP package in R is a package for solving Markov decision processes (MDPs) with discrete time-steps, states and actions. Both ordinary [4] and hierarchial MDPs [1] can be solved. In this paper we use the term MDP for both types of MDPs.

Generating and solving an MDP is done in two steps. First, the MDP is generated and saved in a set of binary files. Next, you load the MDP into memory from the binary files and solve it.

The package uses algorithms based on the *state-expanded directed hypergraph* of the MDP [3] which are all implemented in `C++` for fast running times. Under development is also support for MLHMP which is a Java implementation of algorithms for solving MDPs [2]. A hypergraph representing an MDP with time-horizon $N = 5$ is shown in Figure 1. Each node corresponds to a specific state in the MDP and a directed hyperarc is defined for each possible action. For instance, node $v_{2,1}$ corresponds to a state number 1 at stage 2. The two hyperarcs with head in node $v_{3,0}$ show that two actions are possible given state number 0 at stage 3. Action `mt` corresponds to a deterministic transition to state number zero at stage 4 and action `nmt` corresponds to a transition to state number 0 or 1 at stage 4 with a certain probability greater than zero.

States and actions can be identified using either an unique id or index vector v. In an ordinary MDP the index vector consists of the stage and state number, i.e. state corresponding to node $v_{3,1}$ in Figure 1 is uniquely identified using $v = (n, s) = (3, 1)$. Similar, action `buy` is uniquely identified using $v = (n, s, a) = (0, 0, 0)$. Note that **index always start from zero**.

*Corresponding author

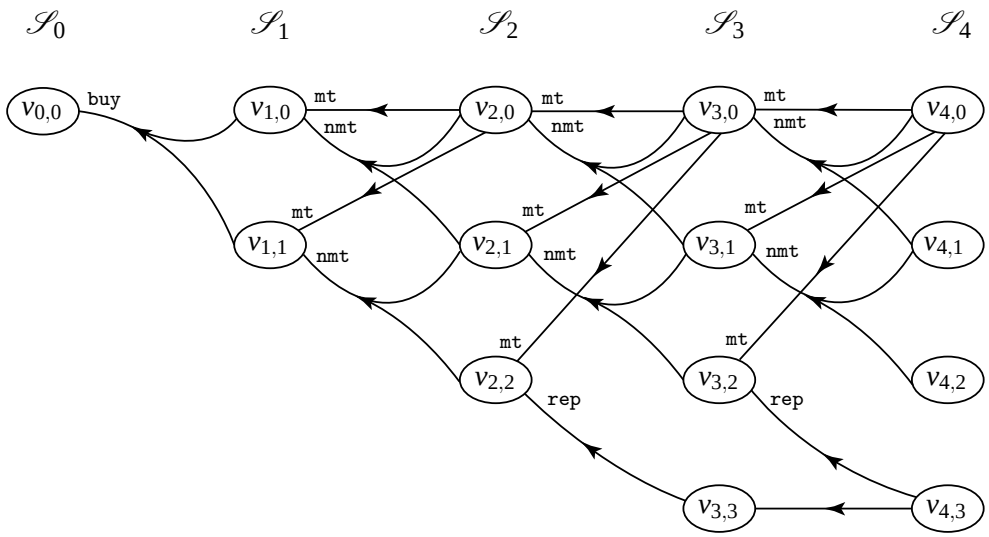

Figure 1: A state-expanded hypergraph for an MDP with time horizon $N = 5$. At stage n each node $v_{n,i}$ corresponds to a state in $\mathscr{S}_n$. The hyperarcs correspond to actions, e.g. if the system at stage 3 is in state number 1 then there are two possible actions. Action `mt` results in a deterministic transition to state zero (because there is only one tail) at stage 4 and `nmt` results in a transition to either state number 1 or 2 with a certain probability.

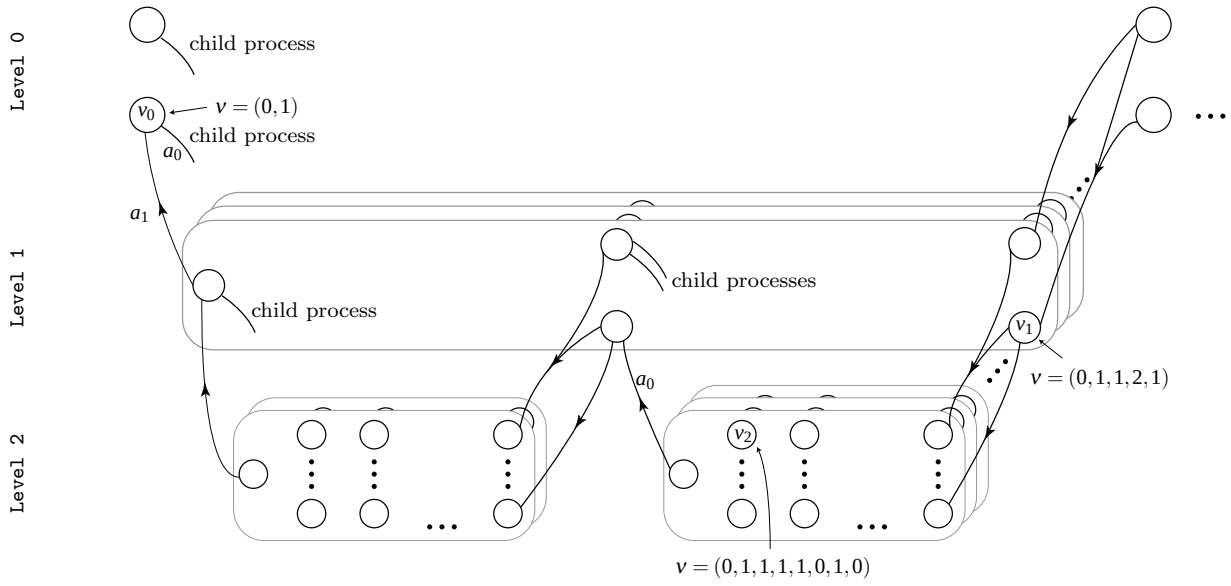

Figure 2: A hypergraph representation of the first stage of a hierarchical MDP. Level 0 indicate the founder level, and the nodes indicates states at the different levels. A child process (oval box) is represented using its state-expanded hypergraph (hyperarcs not shown) and is uniquely defined by a given state and action of its parent process.

A hierarchical MDP is an MDP with parameters defined in a special way, but nevertheless in accordance with all usual rules and conditions relating to such processes [1]. The basic idea of the hierarchical structure is that stages of the process can be expanded to a so-called *child process*, which again may expand stages further to new child processes leading to multiple levels. To illustrate consider the MDP shown in Figure 2. The process has three levels. At `Level 2` we have a set of ordinary MDPs with finite time-horizon (one for each oval box) which all can be represented using a state-expanded hypergraph (hyperarcs not shown, only hyperarcs connecting processes are shown). An MDP at `Level 2` is uniquely defined by a given state s and action a of its *parent process* at `Level 1` (illustrated by the arcs with head and tail node at `Level 1` and `Level 2`, respectively). Moreover, when a child process at `Level 2` terminates a transition from a state $s \in \mathscr{S}_N$ of the child process to a state at the next stage of the parent process occur (illustrated by the (hyper)arcs having head and tail at `Level 2` and `Level 1`, respectively). Since a child process is always defined by a stage, state and action of the parent process we have that for instance a state at level 1 have an index vector $\nu = (n_0, s_0, a_0, n_1, s_1)$.

In general a state s and action a at level l can be uniquely identified using

$$\nu_s = (n_0, s_0, a_0, n_1, s_1, \ldots, n_l, s_l)$$
$$\nu_a = (n_0, s_0, a_0, n_1, s_1, \ldots, n_l, s_l, a_l).$$

The index vector of three states is illustrated in Figure 2.

Another way to identify a state or action is using an id number. Id numbers can be seen when printing information about the model i R. This will be further clarified in the example in Section 2.

Now let us have a look at package. The package can be installed from R-Forge using

```
> install.packages("MDP",repos="http://r-forge.r-project.org")
```

and afterwards loaded

```
> library(MDP)
```

Help about the package can be seen by writing

```
> ?MDP
```

We illustrate the package capabilities by some examples in the next sections.

2 Ordinary MDP with finite time-horizon

Consider the machine replacement example from Nielsen and Kristensen [3] where the machine is always replaced after 4 years. The state of the machine may be: good, average, and not working. Given the machine's state we may maintain the machine. In this case the machine's state will be good at the next decision epoch. Otherwise, the machine's state will not be better at next decision epoch. When the machine is bought it may be either in state good or average. Moreover, if the machine is not working it must be replaced.

(n,s)	$(1,0)$	$(1,1)$	$(2,0)$	$(2,1)$	$(3,0)$	$(3,1)$
reward	70	50	70	50	70	50
s'	$\{0,1\}$	$\{1,2\}$	$\{0,1\}$	$\{1,2\}$	$\{0,1\}$	$\{1,2\}$
$p_n(\cdot \mid s, \texttt{nmt})$	$\{\frac{6}{10}, \frac{4}{10}\}$	$\{\frac{6}{10}, \frac{4}{10}\}$	$\{\frac{5}{10}, \frac{5}{10}\}$	$\{\frac{5}{10}, \frac{5}{10}\}$	$\{\frac{2}{10}, \frac{8}{10}\}$	$\{\frac{2}{10}, \frac{8}{10}\}$

Table 1: Input data for the machine replacement problem given action *nmt*.

The problem of when to replace the machine can be modelled using a Markov decision process with $N = 5$ decision epochs. We use system states `good` (state 0), `average` (state 1), `not working` (state 2) and dummy state `replaced` together with actions buy (`buy`), maintain (`mt`), no maintenance (`nmt`), and replace (`rep`).

The set of states at stage zero S_0 contains a single dummy state s_0 representing the machine before knowing its initial state. The only possible action is *buy*.

The cost of buying the machine is 100 with transition probability of 0.7 to state `good` and 0.3 to state `average`. The reward (scrap value) of replacing a machine is 30, 10, and 5 in state 0, 1, and 2, respectively. The reward of the machine given action `mt` becomes 55, 40, and 30 given state 0, 1, and 2, respectively. Moreover, the system enters state 0 with probability 1 at the next stage. Finally, Table 1 shows the reward, transition states and probabilities given action `nmt`.

The state-expanded hypergraph is shown in Figure 1. It contains a hyperarc for each possible action a given stage n and state $s \in S_n$. The head node of a hyperarc corresponds to the state of the system before action a is taken and the tail nodes to the possible system states after action a is taken.

2.1 Generating the MDP

We generate the model in R using the `binaryMDPWriter`:

```
> prefix<-"machine_"
> w <- binaryMDPWriter(prefix)
> w$setWeights(c("Net reward"))
> w$process()
>   w$stage()   # stage n=0
>     w$state(label="Dummy")          # v=(0,0)
>       w$action(label="buy", weights=-100, prob=c(1,0,0.7, 1,1,0.3), end=T)
>     w$endState()
>   w$endStage()
>   w$stage()   # stage n=1
>     w$state(label="good")           # v=(1,0)
>       w$action(label="mt", weights=55, prob=c(1,0,1), end=T)
>       w$action(label="nmt", weights=70, prob=c(1,0,0.6, 1,1,0.4), end=T)
>     w$endState()
>     w$state(label="average")        # v=(1,1)
>       w$action(label="mt", weights=40, prob=c(1,0,1), end=T)
>       w$action(label="nmt", weights=50, prob=c(1,1,0.6, 1,2,0.4), end=T)
>     w$endState()
>   w$endStage()
>   w$stage()   # stage n=2
>     w$state(label="good")           # v=(2,0)
>       w$action(label="mt", weights=55, prob=c(1,0,1), end=T)
>       w$action(label="nmt", weights=70, prob=c(1,0,0.5, 1,1,0.5), end=T)
```

```
>     w$endState()
>     w$state(label="average")         # v=(2,1)
>       w$action(label="mt", weights=40, prob=c(1,0,1), end=T)
>       w$action(label="nmt", weights=50, prob=c(1,1,0.5, 1,2,0.5), end=T)
>     w$endState()
>     w$state(label="not working")     # v=(2,2)
>       w$action(label="mt", weights=30, prob=c(1,0,1), end=T)
>       w$action(label="rep", weights=5, prob=c(1,3,1), end=T)
>     w$endState()
>   w$endStage()
>   w$stage()   # stage n=3
>     w$state(label="good")            # v=(3,0)
>       w$action(label="mt", weights=55, prob=c(1,0,1), end=T)
>       w$action(label="nmt", weights=70, prob=c(1,0,0.2, 1,1,0.8), end=T)
>     w$endState()
>     w$state(label="average")         # v=(3,1)
>       w$action(label="mt", weights=40, prob=c(1,0,1), end=T)
>       w$action(label="nmt", weights=50, prob=c(1,1,0.2, 1,2,0.8), end=T)
>     w$endState()
>     w$state(label="not working")     # v=(3,2)
>       w$action(label="mt", weights=30, prob=c(1,0,1), end=T)
>       w$action(label="rep", weights=5, prob=c(1,3,1), end=T)
>     w$endState()
>     w$state(label="replaced")        # v=(3,3)
>       w$action(label="Dummy", weights=0, prob=c(1,3,1), end=T)
>     w$endState()
>   w$endStage()
>   w$stage()   # stage n=4
>     w$state(label="good", end=T)         # v=(4,0)
>     w$state(label="average", end=T)      # v=(4,1)
>     w$state(label="not working", end=T) # v=(4,2)
>     w$state(label="replaced", end=T)     # v=(4,3)
>   w$endStage()
> w$endProcess()
> w$closeWriter()
```

```
Statistics:
  states : 14
  actions: 18
  weights: 1

Closing binary MDP writer.
```

A set of binary files (all with prefix `machine_`) containing the model have now been generated. Note how the model is generated in a hierarchical way. A process contain stages which contain states which again contain actions. An action is defined by a set of weights (in this case the net reward) and a set of transition probabilities. The probabilities are defined using a vector of the form $(q_0, i_0, p_0, \ldots, q_r, i_r, p_r)$ stating that r transitions are possible. Each triple (q_j, i_j, p_j) define a transition. The number $q_j \in \{0,1,2\}$ is the scope of the transition. If $q_j = 0$ then we make a transition to the next stage in the parent process, if $q_j = 1$ we make a transition to the next stage in the current process and if $q_j = 2$ we make a transition to the first stage in the child process. The number i_j define which state index we consider at the next stage, e.g. if $i_j = 0$ we consider the state with index 0 (remember index start from zero). Finally, p_j is the probability. For instance, $(q_j, i_j, p_j) = (1,3,0.2)$ specify that we have a transition with probability 0.2 to the state with index 3 at the next stage of the current process.

2.2 Getting an overview

Various information about the whole model can be seen in R:

```
> stateIdxDf(prefix)      # states of the MDP with labels returned as a data frame
```

```
   sId n0 s0       label
1    0  0  0       Dummy
2    1  1  0        good
3    2  1  1     average
4    3  2  0        good
5    4  2  1     average
6    5  2  2 not working
7    6  3  0        good
8    7  3  1     average
9    8  3  2 not working
10   9  3  3    replaced
11  10  4  0        good
12  11  4  1     average
13  12  4  2 not working
14  13  4  3    replaced
```

```
> actionInfo(prefix)      # all action information of the MDP returned in a single data
frame
```

```
   aId sId scp0 idx0 pr0 scp1 idx1 pr1 label
1    0   0    1    0 0.7    1    1 0.3   buy
2    1   1    1    0 1.0   NA   NA  NA    mt
3    2   1    1    0 0.6    1    1 0.4   nmt
4    3   2    1    0 1.0   NA   NA  NA    mt
5    4   2    1    1 0.6    1    2 0.4   nmt
6    5   3    1    0 1.0   NA   NA  NA    mt
7    6   3    1    0 0.5    1    1 0.5   nmt
8    7   4    1    0 1.0   NA   NA  NA    mt
9    8   4    1    1 0.5    1    2 0.5   nmt
10   9   5    1    0 1.0   NA   NA  NA    mt
11  10   5    1    3 1.0   NA   NA  NA   rep
12  11   6    1    0 1.0   NA   NA  NA    mt
13  12   6    1    0 0.2    1    1 0.8   nmt
14  13   7    1    0 1.0   NA   NA  NA    mt
15  14   7    1    1 0.2    1    2 0.8   nmt
16  15   8    1    0 1.0   NA   NA  NA    mt
17  16   8    1    3 1.0   NA   NA  NA   rep
18  17   9    1    3 1.0   NA   NA  NA Dummy
```

Note that the data frame for the states show both each states unique id (a single number) and index vector (the columns with names n<level> and s<level>). For the action data frame each action is given an unique id.

2.3 Finding the optimal policy

A finite-horizon MDP can be solved using value iteration. First we load the model:

```
> mdp<-loadMDP(prefix)
```

```
Cpu for reading the binary files: 0s
Cpu time for checking MDP 0s.
Cpu time for building state-expanded hypergraph 0s
```

```
> mdp
```

```
$binNames
[1] "machine_stateIdx.bin"        "machine_stateIdxLbl.bin"     "machine_actionIdx.bin"
[4] "machine_actionIdxLbl.bin"    "machine_actionWeight.bin"    "machine_actionWeightLbl.bin"
[7] "machine_transProb.bin"

$timeHorizon
[1] 5

$states
[1] 14

$founderStatesLast
[1] 4

$actions
[1] 18

$levels
[1] 1

$weightNames
[1] "Net reward"

$ptr
<pointer: 0x013a8fe8>

attr(,"class")
[1] "MDP:C++"
```

The object is a list containing basic information about the model and a pointer to the C++ object containing the model. Next, we solve the MDP using value iteration:

```
> iW<-0                          # index of the weight we want to optimize
> scrapValues<-c(30,10,5,0)      # scrap values of replacing the machine (the values of the
4 states at stage 4)
> valueIte(mdp, iW, termValues=scrapValues)
```

```
Run value iteration using quantity 'Net reward' under expected reward criterion. Finished (0s).
```

The MDP has now been optimized. The optimal policy can be extracted using:

```
> policy<-getPolicy(mdp, labels=TRUE)      # optimal policy for each sId
> states<-stateIdxDf(prefix)               # information about the states
> policy<-merge(states,policy)             # merge the two data frames
> policyW<-getPolicyW(mdp, iW)             # the optimal rewards of the policy
> policy<-merge(policy,policyW)            # add the rewards
> policy
```

```
   sId n0 s0       label aLabel    w0
1    0  0  0       Dummy    buy 102.2
2    1  1  0        good    nmt 208.5
3    2  1  1     average     mt 187.5
4    3  2  0        good    nmt 147.5
5    4  2  1     average     mt 125.0
6    5  2  2 not working     mt 115.0
7    6  3  0        good     mt  85.0
8    7  3  1     average     mt  70.0
9    8  3  2 not working     mt  60.0
10   9  3  3    replaced  Dummy   0.0
11  10  4  0        good         30.0
12  11  4  1     average         10.0
13  12  4  2 not working          5.0
14  13  4  3    replaced          0.0
```

2.4 Modifying the MDP

It is possible to do some manipulations to the MDP already stored in memory. You may remove some actions from the MDP. For instance assume that it is not possible to maintain the machine at stage 1. We remove the `mt` actions at stage 1:

```
> removeAction(mdp, sId=1, iA=0)   # remove action 0 at the state with sId=1
> removeAction(mdp, sId=2, iA=0)
```

Next, we try to optimize the MDP:

```
> valueIte(mdp, iW, termValues=scrapValues)
```

```
Run value iteration using quantity 'Net reward' under expected reward criterion. Finished (0s).
```

```
> policy<-getPolicy(mdp, labels=TRUE)      # optimal policy for each sId
> states<-stateIdxDf(prefix)               # information about the states
> policy<-merge(states,policy)             # merge the two data frames
> policyW<-getPolicyW(mdp, iW)             # the optimal rewards of the policy
> policy<-merge(policy,policyW)            # add the rewards
> policy
```

```
   sId n0 s0       label aLabel     w0
1    0  0  0       Dummy    buy  97.25
2    1  1  0        good    nmt 208.50
3    2  1  1     average    nmt 171.00
4    3  2  0        good    nmt 147.50
5    4  2  1     average     mt 125.00
6    5  2  2 not working     mt 115.00
7    6  3  0        good     mt  85.00
8    7  3  1     average     mt  70.00
9    8  3  2 not working     mt  60.00
10   9  3  3    replaced  Dummy   0.00
11  10  4  0        good         30.00
12  11  4  1     average         10.00
13  12  4  2 not working          5.00
14  13  4  3    replaced          0.00
```

We could also have removed the `mt` actions by fixing the `nmt` actions:

```
> fixAction(mdp, sId=1, iA=1)  # remove all actions at state sId=1 except action 1
> fixAction(mdp, sId=2, iA=1)
```

You reset the MDP again with:

```
> resetActions(mdp)   # reset the MDP such that all actions are used
```

It is possible to modify the weights of an action, e.g. assume that the cost of buying the machine is 50 instead of 100:

```
> setActionWeight(mdp, w=-50, sId=0, iA=0, iW=0)
```

The solution now becomes:

```
Run value iteration using quantity 'Net reward' under expected reward criterion. Finished (0s).
```

```
   sId n0 s0       label aLabel    w0
1    0  0  0       Dummy    buy 152.2
2    1  1  0        good    nmt 208.5
3    2  1  1     average     mt 187.5
4    3  2  0        good    nmt 147.5
5    4  2  1     average     mt 125.0
6    5  2  2 not working     mt 115.0
7    6  3  0        good     mt  85.0
8    7  3  1     average     mt  70.0
9    8  3  2 not working     mt  60.0
10   9  3  3    replaced  Dummy   0.0
11  10  4  0        good         30.0
12  11  4  1     average         10.0
13  12  4  2 not working          5.0
14  13  4  3    replaced          0.0
```

2.5 Evaluating a specific policy

We may evaluate a certain policy, e.g. the policy always to maintain the machine:

```
> setActionWeight(mdp, w=-100, sId=0, iA=0, iW=0)      # set weight to original
> policy<-data.frame(sId=states$sId,iA=0)
> policy<-as.matrix(policy)
> setPolicy(mdp, policy)
```

If the policy matrix does not contain all states then the actions from the previous optimal policy are used. Now let us calculate the expected reward of that policy:

```
> calcWeights(mdp, iW, termValues=scrapValues)
> policy<-getPolicy(mdp, labels=TRUE)      # optimal policy for each sId
> states<-stateIdxDf(prefix)               # information about the states
> policy<-merge(states,policy)             # merge the two data frames
> policyW<-getPolicyW(mdp, iW)             # the optimal rewards of the policy
> policy<-merge(policy,policyW)            # add the rewards
> policy
```

```
   sId n0 s0       label aLabel    w0
1    0  0  0       Dummy    buy  90.5
2    1  1  0        good     mt 195.0
3    2  1  1     average     mt 180.0
4    3  2  0        good     mt 140.0
5    4  2  1     average     mt 125.0
6    5  2  2 not working     mt 115.0
7    6  3  0        good     mt  85.0
8    7  3  1     average     mt  70.0
9    8  3  2 not working     mt  60.0
10   9  3  3    replaced  Dummy   0.0
11  10  4  0        good         30.0
12  11  4  1     average         10.0
13  12  4  2 not working          5.0
14  13  4  3    replaced          0.0
```

3 Ordinary MDP with infinite time-horizon

For a sow it is relevant to consider at regular time intervals whether to keep the sow for a period more or replace it by a new sow. Let a stage denote the time between two litters. At the time of a stage we observe the state of the sow which in this simple example is the current litter size `small`, `average` or `big`.

Two actions are possible, namely, `keep` or `replace`. Given an action 3 weights are defined the duration, net reward and the number of piglets. The weights and transition probabilities of an action are specified explicit when we generate the MDP:

```
> prefix="sow_"
> w<-binaryMDPWriter(prefix)
> w$setWeights(c("Duration", "Net reward", "Piglets"))
> w$process()
>   w$stage()
>     w$state(label="Small litter")
>       w$action(label="Keep",weights=c(1,10000,8),prob=c(1,0,0.6, 1,1,0.3, 1,2,0.1))
>       w$endAction()
>       w$action(label="Replace",weights=c(1,9000,8),prob=c(1,0,1/3, 1,1,1/3, 1,2,1/3))
>       w$endAction()
>     w$endState()
>     w$state(label="Average litter")
>       w$action(label="Keep",weights=c(1,12000,11),prob=c(1,0,0.2, 1,1,0.6, 1,2,0.2))
>       w$endAction()
>       w$action(label="Replace",weights=c(1,11000,11),prob=c(1,0,1/3, 1,1,1/3, 1,2,1/3))
>       w$endAction()
>     w$endState()
>     w$state(label="Big litter")
>       w$action(label="Keep",weights=c(1,14000,14),prob=c(1,0,0.1, 1,1,0.3, 1,2,0.6))
>       w$endAction()
>       w$action(label="Replace",weights=c(1,13000,14),prob=c(1,0,1/3, 1,1,1/3, 1,2,1/3))
>       w$endAction()
>     w$endState()
>   w$endStage()
> w$endProcess()
> w$closeWriter()
```

```
Statistics:
  states : 3
```

```
    actions: 6
    weights: 3

  Closing binary MDP writer.
```

Note that since we only have one stage at the founder level (level 0) the MDP have an infinite time-horizon. That is, the MDP model a sow and all it successors (when a sow is replaced, a new is always inserted).

Let us have a overview over the model

```
> stateIdxDf(prefix)       # states of the MDP with labels returned as a data frame
```

```
  sId n0 s0          label
1   0  0  0   Small litter
2   1  0  1 Average litter
3   2  0  2     Big litter
```

```
> actionInfo(prefix)       # all action information of the MDP returned in a single data
frame
```

```
  aId sId Duration Net reward Piglets scp0 idx0       pr0 scp1 idx1       pr1 scp2 idx2       pr2   label
1   0   0        1      10000       8    1    0 0.6000000    1    1 0.3000000    1    2 0.1000000    Keep
2   1   0        1       9000       8    1    0 0.3333333    1    1 0.3333333    1    2 0.3333333 Replace
3   2   1        1      12000      11    1    0 0.2000000    1    1 0.6000000    1    2 0.2000000    Keep
4   3   1        1      11000      11    1    0 0.3333333    1    1 0.3333333    1    2 0.3333333 Replace
5   4   2        1      14000      14    1    0 0.1000000    1    1 0.3000000    1    2 0.6000000    Keep
6   5   2        1      13000      14    1    0 0.3333333    1    1 0.3333333    1    2 0.3333333 Replace
```

3.1 Finding the optimal policy under different criteria

Let us try to optimize our model under the expected discounted reward criterion. Here two optimization techniques are possible. Let us first have a look at value iteration which provide an approximate solution.

```
> mdp<-loadMDP(prefix)
```

```
Cpu for reading the binary files: 0s
Cpu time for checking MDP 0s.
Cpu time for building state-expanded hypergraph 0s
```

```
> mdp
```

```
$binNames
[1] "sow_stateIdx.bin"        "sow_stateIdxLbl.bin"     "sow_actionIdx.bin"       "sow_actionIdxLbl.bin"
[5] "sow_actionWeight.bin"    "sow_actionWeightLbl.bin" "sow_transProb.bin"

$timeHorizon
[1] Inf

$states
[1] 3

$founderStatesLast
[1] 3

$actions
[1] 6

$levels
[1] 1

$weightNames
[1] "Duration"   "Net reward" "Piglets"

$ptr
<pointer: 0x01d13728>

attr(,"class")
[1] "MDP:C++"
```

```
> ## solve the MDP using value iteration
> iW<-1                    # index of the weight we want to optimize
> iDur<-0                  # index of the duration/time
> rate<-0.1                # discount rate
> rateBase<-1              # rate base
> valueIte(mdp, iW, iDur, rate, rateBase, times = 10000, eps = 0.00001)
```

```
Run value iteration with epsilon = 1e-05 at most 10000 time(s)
using quantity 'Net reward' under expected discounted reward criterion
with 'Duration' as duration using interest rate 0.1 and rate basis equal 1.
Iterations: 211. Running time 0s.
```

```
> policy<-getPolicy(mdp, labels=TRUE)      # optimal policy for each sId
> states<-stateIdxDf(prefix)               # information about the states
> policy<-merge(states,policy)             # merge the two data frames
> policyW<-getPolicyW(mdp, iW)             # the optimal rewards of the policy
> policy<-merge(policy,policyW)            # add the rewards
> policy
```

```
  sId n0 s0          label  aLabel       w1
1   0  0  0   Small litter Replace 124363.1
2   1  0  1 Average litter    Keep 127287.7
3   2  0  2     Big litter    Keep 130836.9
```

First note that the we optimize the MDP for a specific interest rate which according to a rate basis, i.e. if the rate is 0.1 and the rate base is 4 then the discount rate over one time unit is $\exp(-0.1/4) = 0.9753$. The discount rate over t time units then becomes

$$\delta(t) = \exp(-\texttt{rate}/\texttt{rateBase})^t.$$

Second, the parameter `times` denote an upper bound on the number of iterations. Finally, the parameter `eps` denote the ε for stopping the algorithm. If the maximum difference between the expected discounted reward of 2 states is below ε then the algorithm stops, i.e the policy becomes epsilon optimal (see [4] p161).

Let us have a look at how value iteration performs for each iteration.

```
> termValues<-c(0,0,0)
> iterations<-1:211
> df<-data.frame(n=iterations,a1=NA,V1=NA,D1=NA,a2=NA,V2=NA,D2=NA,a3=NA,V3=NA,D3=NA)
> for (i in iterations) {
+   valueIte(mdp, iW, iDur, rate, rateBase, times = 1, eps = 0.00001, termValues)
+   a<-getPolicy(mdp, labels=T)
+   w<-getPolicyW(mdp, iW)
+   res<-rep(NA,10)
+   res[1]<-i
+   res[2]<-a[1,2]
+   res[3]<-round(w[1,2],2)
+   res[4]<-round(w[1,2]-termValues[1],2)
+   res[5]<-a[2,2]
+   res[6]<-round(w[2,2],2)
+   res[7]<-round(w[2,2]-termValues[2],2)
+   res[8]<-a[3,2]
+   res[9]<-round(w[3,2],2)
+   res[10]<-round(w[3,2]-termValues[3],2)
+   df[i,]<-res
+   termValues<-w[,2]
+ }
> df[c(1:3,51:53,151:153,210:211),]
```

	n	a1	V1	D1	a2	V2	D2	a3	V3	D3
1	1	Keep	10000	10000	Keep	12000	12000	Keep	14000	14000
2	2	Keep	19953.21	9953.21	Keep	22858.05	10858.05	Keep	25762.89	11762.89
3	3	Replace	29682.82	9729.61	Keep	32682.82	9824.77	Keep	35997.02	10234.13
51	51	Replace	123583.65	81.97	Keep	126508.29	81.97	Keep	130057.51	81.97
52	52	Replace	123657.82	74.17	Keep	126582.47	74.17	Keep	130131.69	74.17
53	53	Replace	123724.93	67.11	Keep	126649.58	67.11	Keep	130198.8	67.11
151	151	Replace	124363.03	0	Keep	127287.67	0	Keep	130836.89	0
152	152	Replace	124363.03	0	Keep	127287.68	0	Keep	130836.9	0
153	153	Replace	124363.03	0	Keep	127287.68	0	Keep	130836.9	0
210	210	Replace	124363.06	0	Keep	127287.71	0	Keep	130836.93	0
211	211	Replace	124363.06	0	Keep	127287.71	0	Keep	130836.93	0

Note value iteration converges very slowly to the optimal value.

Another optimization technique is policy iteration which finds an optimal policy. Let us solve the MDP under the expected discount criterion.

```
> policyIteDiscount(mdp, iW, iDur, rate, rateBase)
```

```
Run policy iteration using quantity 'Net reward' under discounting criterion
with 'Duration' as duration using interest rate 0.1 and a rate basis equal 1.
Iteration(s): 1 2 3 finished.
```

```
> policy<-getPolicy(mdp, labels=TRUE)
> sIdx<-stateIdxDf(prefix)
> policy<-merge(sIdx,policy)
> policyW<-getPolicyW(mdp, iW)
> policy<-merge(policy,policyW)
> rpo<-calcRPO(mdp, iW, iA=0, criterion="discount", iDur=iDur, rate=rate,
rateBase=rateBase)
> policy<-merge(policy,rpo)
> policy$w1<-round(policy$w1,0)
> policy$rpo<-round(policy$rpo,0)
> policy
```

```
  sId n0 s0          label  aLabel     w1  rpo
1   0  0  0   Small litter Replace 124363 -455
2   1  0  1 Average litter    Keep 127288  925
3   2  0  2     Big litter    Keep 130837 2474
```

First, note that policy iteration converges fast only 3 iterations are needed. Second, we also here try to calculate the *retention payoff* (*RPO*) or opportunity cost with respect to action `keep` (action index 0). The RPO is the discounted gain of keeping the sow until her optimal replacement time instead of replacing her now. For instance if we consider a sow with a big litter we loose 2474 by replacing the sow instead keeping her to her until her optimal replacement time. That is, if the RPO is positive the optimal decision is to keep the sow and if the RPO is negative the optimal decision is to replace the sow.

Other criteria can also be optimized using policy iteration. For instance we can maximize the average reward over time:

```
> g<-policyIteAve(mdp, iW, iDur)
```

```
Run policy iteration under average reward criterion using
reward 'Net reward' over 'Duration'. Iterations (g):
1 (12000) 2 (12187.5) 3 (12187.5) finished.
```

```
> policy<-getPolicy(mdp, labels=TRUE)
> policy<-merge(sIdx,policy)
> policyW<-getPolicyW(mdp, iW)
> policy<-merge(policy,policyW)
> rpo<-calcRPO(mdp, iW, iA=0, criterion="average", iDur = iDur, g=g)
> policy<-merge(policy,rpo)
> policy$w1<-round(policy$w1,0)
> policy$rpo<-round(policy$rpo,0)
> policy
```

```
  sId n0 s0          label  aLabel    w1  rpo
1   0  0  0   Small litter Replace -6688 -656
2   1  0  1 Average litter    Keep -3813  875
3   2  0  2     Big litter    Keep     0 2688
```

Here g is the average reward pr time unit and the weights are relative values compared to the `big litter` state.

We may also maximize the average reward over piglets:

```
> iDur<-2
> g<-policyIteAve(mdp, iW, iDur=iDur)
```

```
Run policy iteration under average reward criterion using
reward 'Net reward' over 'Piglets'. Iterations (g):
1 (1090.91) 2 (1095.81) 3 (1095.81) finished.
```

```
> policy<-getPolicy(mdp, labels=TRUE)
> policy<-merge(sIdx,policy)
> policyW<-getPolicyW(mdp, iW)
> policy<-merge(policy,policyW)
> rpo<-calcRPO(mdp, iW, iA=0, criterion="average", iDur = iDur, g=g)
> policy<-merge(policy,rpo)
> policy$w1<-round(policy$w1,0)
> policy$rpo<-round(policy$rpo,0)
> policy
```

```
  sId n0 s0          label  aLabel   w1  rpo
1   0  0  0   Small litter    Keep 4772 2198
2   1  0  1 Average litter    Keep 2251  964
3   2  0  2     Big litter Replace    0 -189
```

Here g is the average reward pr piglet and the weights are relative values compared to the `big litter` state.

3.2 Calculating other key figures for the optimal policy

Consider the optimal policy under the expected discounted reward criterion:

```
> policyIteDiscount(mdp, iW, iDur, rate, rateBase)
```

```
Run policy iteration using quantity 'Net reward' under discounting criterion
with 'Piglets' as duration using interest rate 0.1 and a rate basis equal 1.
Iteration(s): 1 2 finished.
```

```
> policy<-getPolicy(mdp, labels=TRUE)
> sIdx<-stateIdxDf(prefix)
> policy<-merge(sIdx,policy)
> policyW<-getPolicyW(mdp, iW)
> policy<-merge(policy,policyW)
> rpo<-calcRPO(mdp, iW, iA=0, criterion="discount", iDur=iDur, rate=rate,
rateBase=rateBase)
> policy<-merge(policy,rpo)
> policy$w1<-round(policy$w1,0)
> policy$rpo<-round(policy$rpo,0)
> policy
```

```
  sId n0 s0          label aLabel    w1  rpo
1   0  0  0   Small litter   Keep 18161  964
2   1  0  1 Average litter   Keep 18047  974
3   2  0  2     Big litter   Keep 18524 1025
```

Since other weights are defined for each action we can calculate the average number of piglets per time unit under the optimal policy:

```
> g<-calcWeights(mdp, iW=2, criterion="average", iDur = 0)
> g
```

```
[1] 11
```

or the average reward per piglet:

```
> g<-calcWeights(mdp, iW=1, criterion="average", iDur = 2)
> g
```

```
[1] 1090.909
```

4 Hierarchical MDP with infinite time-horizon

We consider a cow replacement problem where we want to represent the age of the cow, i.e. the lactation number of the cow. During a lactation a cow may have a high, average or low yield. We assume that a cow is always replaced after 4 lactations.

In addition to lactation and milk yield we also want to take the genetic merit into account which is either bad, average or good. When a cow is replaced we assume that the probability of a bad, average or good heifer is equal.

We formulate the problem as a hierarchical MDP with 2 levels. At level 0 the states are the genetic merit and the length of a stage is a life of a cow. At level 1 a stage describe a lactation and states describe the yield. Decisions at level 1 are `keep` or `replace`.

Note the MDP runs over an infinite time-horizon at the founder level where each state (genetic merit) define an ordinary MDP at level 1 with 4 lactations.

4.1 Generating the MDP

To generate the MDP we need to know the weights and transition probabilities which are provided in a csv file. To ease the understanding we provide 2 functions for reading from the csv:

```
> cowDf<-read.csv("cow.csv")
> head(cowDf)
```

```
  s0 n1 s1   label Duration Reward Output scp0 idx0       pr0 scp1 idx1       pr1 scp2 idx2       pr2
1  0  0  0   Dummy        0      0      0    1    0 0.3333333    1    1 0.3333333    1    2 0.3333333
2  0  1  0    Keep        1   6000   3000    1    0 0.6000000    1    1 0.3000000    1    2 0.1000000
3  0  1  0 Replace        1   5000   3000    0    0 0.3333333    0    1 0.3333333    0    2 0.3333333
4  0  1  1    Keep        1   8000   4000    1    0 0.2000000    1    1 0.6000000    1    2 0.2000000
5  0  1  1 Replace        1   7000   4000    0    0 0.3333333    0    1 0.3333333    0    2 0.3333333
6  0  1  2    Keep        1  10000   5000    1    0 0.1000000    1    1 0.3000000    1    2 0.6000000
```

```
> lev1W<-function(s0Idx,n1Idx,s1Idx,a1Lbl) {
+   r<-subset(cowDf,s0==s0Idx & n1==n1Idx & s1==s1Idx & label==a1Lbl)
+   return(as.numeric(r[5:7]))
+ }
> lev1W(2,2,1,'Keep')      # good genetic merit, lactation 2, avg yield, keep action
```

```
[1]     1 14000  7000
```

```
> lev1Pr<-function(s0Idx,n1Idx,s1Idx,a1Lbl) {
+   r<-subset(cowDf,s0==s0Idx & n1==n1Idx & s1==s1Idx & label==a1Lbl)
+   return(as.numeric(r[8:16]))
+ }
> lev1Pr(2,2,1,'Replace') # good genetic merit, lactation 2, avg yield, replace action
```

```
[1] 0.0000000 0.0000000 0.3333333 0.0000000 1.0000000 0.3333333 0.0000000 2.0000000 0.3333333
```

```
> lblS0<-c('Bad genetic level','Avg genetic level','Good genetic level')
> lblS1<-c('Low yield','Avg yield','High yield')
> prefix<-"cow_"
> w<-binaryMDPWriter(prefix)
> w$setWeights(c("Duration", "Net reward", "Yield"))
> w$process()
>   w$stage()   # stage 0 at founder level
>     for (s0 in 0:2) {
+       w$state(label=lblS0[s0+1])   # state at founder
+         w$action(label="Keep", weights=c(0,0,0), prob=c(2,0,1))   # action at founder
+           w$process()
+             w$stage()   # dummy stage at level 1
+                w$state(label="Dummy")
+                 w$action(label="Dummy", weights=c(0,0,0), prob=c(1,0,1/3, 1,1,1/3,
1,2,1/3))
+                 w$endAction()
+                w$endState()
+             w$endStage()
+             for (d1 in 1:4) {
```

```
+                 w$stage()   # stage at level 1
+                   for (s1 in 0:2) {
+                     w$state(label=lblS1[s1+1])
+                       if (d1!=4) {
+                         w$action(label="Keep", weights=lev1W(s0,d1,s1,"Keep"),
prob=lev1Pr(s0,d1,s1,"Keep"))
+                         w$endAction()
+                       }
+                       w$action(label="Replace", weights=lev1W(s0,d1,s1,"Replace"),
prob=lev1Pr(s0,d1,s1,"Replace"))
+                       w$endAction()
+                     w$endState()
+                   }
+                 w$endStage()
+               }
+             w$endProcess()
+           w$endAction()
+         w$endState()
+       }
>    w$endStage()
> w$endProcess()
> w$closeWriter()
```

```
  Statistics:
    states : 42
    actions: 69
    weights: 3

  Closing binary MDP writer.
```

4.2 Finding the optimal policy

We find the optimal policy under the expected discounted reward criterion the MDP using policy iteration:

```
> ## solve under discount criterion
> mdp<-loadMDP(prefix)
```

```
Cpu for reading the binary files: 0.03s
Cpu time for checking MDP 0s.
Cpu time for building state-expanded hypergraph 0.016s
```

```
> iW<-1                    # index of the weight we want to optimize (net reward)
> iDur<-0                  # index of the duration/time
> rate<-0.1                # discount rate
> rateBase<-1              # rate base, i.e. given a duration of t the rate is
> sIdx<-stateIdxDf(prefix)
> policyIteDiscount(mdp, iW, iDur, rate, rateBase)
```

```
Run policy iteration using quantity 'Net reward' under discounting criterion
with 'Duration' as duration using interest rate 0.1 and a rate basis equal 1.
Iteration(s): 1 2 3 4 finished.
```

```
> policy<-getPolicy(mdp, labels=TRUE)
> policy<-merge(sIdx,policy)
> policyW<-getPolicyW(mdp, iW)
> policy<-merge(policy,policyW)
> rpo<-calcRPO(mdp, iW, iA=0, criterion="discount", iDur=iDur, rate=rate,
rateBase=rateBase)
> policy<-merge(policy,rpo)
> policy
```

	sId	n0	s0	a0	n1	s1	label	aLabel	w1	rpo
1	0	0	0	NA	NA	NA	Bad genetic level	Keep	115594.1	0.00000
2	1	0	0	0	0	0	Dummy	Dummy	115594.1	0.00000
3	2	0	0	0	1	0	Low yield	Replace	113594.1	-2095.39618
4	3	0	0	0	1	1	Avg yield	Replace	115594.1	-1190.55876
5	4	0	0	0	1	2	High yield	Replace	117594.1	-285.72134
6	5	0	0	0	2	0	Low yield	Replace	115594.1	-2095.39618
7	6	0	0	0	2	1	Avg yield	Replace	117594.1	-1190.55876
8	7	0	0	0	2	2	High yield	Replace	119594.1	-285.72134
9	8	0	0	0	3	0	Low yield	Replace	115594.1	-3000.23360
10	9	0	0	0	3	1	Avg yield	Replace	117594.1	-2095.39618
11	10	0	0	0	3	2	High yield	Replace	119594.1	-1190.55876
12	11	0	0	0	4	0	Low yield	Replace	114594.1	0.00000
13	12	0	0	0	4	1	Avg yield	Replace	116594.1	0.00000
14	13	0	0	0	4	2	High yield	Replace	118594.1	0.00000
15	14	0	1	NA	NA	NA	Avg genetic level	Keep	118982.8	0.00000
16	15	0	1	0	0	0	Dummy	Dummy	118982.8	0.00000
17	16	0	1	0	1	0	Low yield	Keep	115675.2	81.05822
18	17	0	1	0	1	1	Avg yield	Keep	118946.8	1352.67520
19	18	0	1	0	1	2	High yield	Keep	122326.4	2732.26494
20	19	0	1	0	2	0	Low yield	Replace	117594.1	-229.70139
21	20	0	1	0	2	1	Avg yield	Keep	120325.3	731.15596
22	21	0	1	0	2	2	High yield	Keep	123454.2	1860.07314
23	22	0	1	0	3	0	Low yield	Replace	117594.1	-1190.55876
24	23	0	1	0	3	1	Avg yield	Replace	119594.1	-285.72134
25	24	0	1	0	3	2	High yield	Keep	122213.2	619.11607
26	25	0	1	0	4	0	Low yield	Replace	116594.1	0.00000
27	26	0	1	0	4	1	Avg yield	Replace	118594.1	0.00000
28	27	0	1	0	4	2	High yield	Replace	120594.1	0.00000
29	28	0	2	NA	NA	NA	Good genetic level	Keep	125468.3	0.00000
30	29	0	2	0	0	0	Dummy	Dummy	125468.3	0.00000
31	30	0	2	0	1	0	Low yield	Keep	121968.9	4374.75204
32	31	0	2	0	1	1	Avg yield	Keep	125468.3	5874.15939
33	32	0	2	0	1	2	High yield	Keep	128967.7	7373.56674
34	33	0	2	0	2	0	Low yield	Keep	122087.6	2493.51826
35	34	0	2	0	2	1	Avg yield	Keep	125401.8	3807.72105
36	35	0	2	0	2	2	High yield	Keep	128716.0	5121.92385
37	36	0	2	0	3	0	Low yield	Keep	120213.2	619.11607
38	37	0	2	0	3	1	Avg yield	Keep	123118.1	1523.95349
39	38	0	2	0	3	2	High yield	Keep	126022.9	2428.79091
40	39	0	2	0	4	0	Low yield	Replace	118594.1	0.00000
41	40	0	2	0	4	1	Avg yield	Replace	120594.1	0.00000
42	41	0	2	0	4	2	High yield	Replace	122594.1	0.00000

4.3 Visual view of the hierarchical structure of the MDP

The program MLHMP is a Java implementation of some algorithms for solving MDPs [2]. It have a graphical user interface where the hierarchical structure of the MDP can be visualized. A model can be loaded into MLHMP using the HMP format which is an XML

file containing the model. The MDP package contain a function for converting the binary files to the HMP format:

```
> convertBinary2HMP(prefix)
```

```
       Model saved to file: cow_converted.hmp
Converted binary files to hmp format.
   user  system elapsed
   0.46    0.00    0.47
```

The function create the file `cow_converted.hmp` which can be opened by MLHMP.

References

[1] A. R. Kristensen and E. Jørgensen. Multi-level hierarchic Markov processes as a framework for herd management support. *Annals of Operations Research*, 94:69–89, 2000. doi:10.1023/A:1018921201113.

[2] A.R. Kristensen. A general software system for Markov decision processes in herd management applications. *Computers and Electronics in Agriculture*, 38(3):199–215, 2003. doi:10.1016/S0168-1699(02)00183-7.

[3] L.R. Nielsen and A.R. Kristensen. Finding the K best policies in a finite-horizon Markov decision process. *European Journal of Operational Research*, 175(2):1164–1179, 2006. doi:10.1016/j.ejor.2005.06.011.

[4] M.L. Puterman. *Markov Decision Processes*. Wiley Series in Probability and Mathematical Statistics. Wiley-Interscience, 1994.