Summary

Using SARSOP to determine a Model Based Reflex agent policy of the Wumpus world in Julia.  Then simulate the policy 1000 times creating random scenarios and calculate the average score.  Submit the Julia code with a write-up and your average score.

Getting Julia

Julia is a programming language developed by Stanford.  You can either use the JuliaBox online Jupyter Notebook, or download it to your computer.

Windows, Mac, Linux binaries downloadable at: <https://julialang.org/downloads/>

For JuliaBox online register for free at <https://www.juliabox.com/>

Installing POMDP packages for downloaded version:

Start Julia by typing: julia

At the prompt type:

  Pkg.add("POMDPs")

  Pkg.add("POMDPModels")

  Pkg.add("POMDPToolbox")

Installing POMDP packages for JuliaBox:

Log-in to JuliaBox

In the top left corner click "Packages"

  Click "Yours" tab

  Type "POMDPs" in the Enter the package textbox (it should pop-up and allow you to select it).  Then click +

  Type "POMDPModels" in the Enter the package textbox (it should pop-up and allow you to select it).  Then click +

  Type "POMDPToolbox" in the Enter the package textbox (it should pop-up and allow you to select it).  Then click +

  Then click "Start" button

  The progress of download and installing will show, and take a while; wait until done

Example Julia file for simple Wumpus world creating a policy using SARSOP.

The following implementations builds a model of a POMDP problem featuring a simplified Wumpus world and builds a solution using the SARSOP algorithm. In the simplified problem, a mobile agent uses a set of simple actions (Move Right, Move Left, Move Up, Move Down) to search a grid-based world for a grid cell containing gold in order to collect a reward. Actions may either be deterministic and deliver the agent to the targeted next grid cell state with certainly (see first implementation) or stochastic, wherein an action is not always successful and the agent may end in a small subset of other grid-cell states (as in the second implementation.

Most critically, as a POMDP, the agent knows neither its own absolute location in the grid nor the exact location of the gold. Rather, the agent must collect observations (here, the “glitter” of the gold) to acquire information about its current state and use the distribution of observations across states to compute a policy in the form of a finite state controller that determines the best action over a distribution of states (a “belief”) in order to reach the cell containing gold.

Defining the POMDP model requires specification of states, actions, a transition function that describes how the agent moves from one state to the next across timesteps, an observation function that defines the probability (in this case) of receiving an observation of glitter (a Boolean value TRUE) or no-glitter (FALSE), and a Reward function that links glitter observations to a receiving a certain reward. To realize a fuller Wumpus world implementation, the set of possible observations and rewards may be expanded to include (for example) detection of the Wumpus stench with a negative reward by adding additional conditional branches to the appropriate functions (this expansion is described with additional details in the implementation comments).

The solution in the first implementation uses the SARSOP (Successive Approximations of the Reachable Space under Optimal Policies) algorithm to compute the solution policy for this simplified Wumpus world model. A full implementation of the algorithm is described in Kurniawati et al (2008)\*\* (available here: <http://www.comp.nus.edu.sg/~leews/publications/rss08.pdf>). SARSOP is an offline planning approach that build a solution policy before taking any action, based on the complete exploration of possible contingencies. The approach samples from the distribution of states to construct a “reachable” tree with nodes built from the conjunction of state, action, and observation and selected within bounds placed on the optimal value function. From selected nodes, information is back-propagated up the tree, which results in the pruning of some nodes that do not contribute to an optimal policy. This process of tree exploration continues to a reward at the leaf through multiple iterations until the difference between the upper and lower bound on the value function at the root of the tree (the initial state distribution) reaches a certain displacement, or until a certain time limit is reached. The result is a policy that, for every node in the tree, computes the best possible action to maximize reward.

\*\*Kurniawati H, Hsu D, Lee WS. 2008. SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. Proceedings of Robotics: Science and Systems. 4, 1-8.

**A few tips and NBs:**

The following implementations may be run in the Julia REPL by calling include(“filename”).

# represents a single-line comment

#= begins a block comment and =# ends the block.

If you run these implementations in the Julia REPL, liberally refresh the window to avoid duplicate use of previously-created instances of pomp variables and other parameters.

At the end of each implementation (i.e. after the model and policy files have been generated) are included commands to 1)simulate the policy and produce command-line text that presents a step through of a number of state/actions and 2) produce a visualization of the policy graph that represents the finite state controller policy. Uncomment these blocks of code to run these options. Note: the policy visualization produces a .dot file, that requires GraphViz to produce that graph itself. GraphViz may be run in the browser at <http://www.webgraphviz.com/> (paste the contents of the .dot file in the browser window) or may be downloaded (for PC and Linux!) at <https://www.graphviz.org/>

**Implementation 1: Deterministic Transitions (the agent always reaches the grid cell state it moves towards)**

#=

GridWorld: SARSOP solver with deterministic transitions: the agent always reaches the targeted cell, as long as that cell lies in the bounds of the grid; if that move is beyond the grid boundary, the agent remains in its current state.

Agent searches for cell containing gold

Presence of gold detected via boolean observation (glitter = TRUE) when gold is co-located with agent

Agent moves through grid using four actions (Right, Left, Up, Down).

Run file in Julia REPL using include("filename")

=#

#Install the packages needed for implementation first by "adding" the package and the using the "using" keyword

#It saves some time to "add" the package directly to the Julia REPL and use only the "using" statement in the file.

using POMDPs #For general POMDP functions

POMDPs.add("POMDPToolbox") #For model production some of the solving-related functions

Pkg.add("Distributions") #For use of the SparseCat distribution

Pkg.add("SARSOP") #For the solver itself (the using statement follows below)

using Distributions

using POMDPToolbox

#First, define the struct that will characterize each state. In this case, a state is defined

#with the current (x,y) tuple of the Agent's location in the grid, along with the(x,y)

#tuple of the location of the gold. Multiple reward states (i.e. multiple cells

#containing gold) could be represented with a vector of tuples.

struct GridState

xA::Int64

yA::Int64

xGold::Int64

yGold::Int64

end

#Once created, a GridState constructor is built, in this case containing a pre-set

#gold location (in cell [2,1]). Note the constructor requires specification only

#of Agent location for ease.

GridState(xA::Int64, yA::Int64) = GridState(xA,yA,2,1)

#To define the terminal state and shape observations, a helper function specifies

#when the agent "finds" the gold. Discovery is simple: the agent finds and retrieves

#gold when agent and gold are co-located.

goldFound(s1::GridState) = s1.xA==s1.xGold && s1.yA==s1.yGold

#Once the foundation of a GridState has been defined, the GridWorld problem itself

#can be characterized. Here, the problem is defined using the size of the world

#(i.e. the number of grid cells), the reward (r) and the discount factor. Note that the

#problem can be expanded by adding additional fields including other rewards

#(or costs as negative rewards), the accuracy level of observations, etc

type GridPOMDP <: POMDP{GridState, Int64, Bool}

size\_x::Int64 #Number of grid cells in the x-direction

size\_y::Int64 #Number of grid cells in the x-direction

r::Int64 #Reward for finding gold

discount::Float64 #Discount factor

end

#Once defined, a simple problem constructor specifies the default characteristics.

#The following specifies a 3x3 grid world, with a reward of +10 and a discount factor of 0.95

function GridPOMDP()

return GridPOMDP(3,3,10,0.95)

end;

#With the problem specified, a problem instance can be built

pomdp = GridPOMDP()

#A paired function establishes that finding the gold makes the state terminal.

POMDPs.isterminal(pomdp::GridPOMDP, s::GridState) = goldFound(s)

######STATES######

#For the state distribution, each state is pushed onto an array and indexed

function POMDPs.states(pomdp::GridPOMDP)

s = GridState[] #initialize array of GridWorldStates

for xA=1:pomdp.size\_x, yA=1:pomdp.size\_y, xGold=1:pomdp.size\_x, yGold=1:pomdp.size\_y

push!(s, GridState(xA,yA,xGold,yGold))

end

return s #array of states

end;

function POMDPs.state\_index(pomdp::GridPOMDP, state::GridState)

return sub2ind((pomdp.size\_x, pomdp.size\_y, 3), state.xA, state.yA, state.xGold, state.yGold)

end

#The POMDP package requires computation of the number of expected states

POMDPs.n\_states(p::GridPOMDP) = (p.size\_x\*p.size\_y)\*(p.size\_x\*p.size\_y)

######ACTIONS#######

#The next functions specify general parameters concerning the actions available to the agent.

#In this simple world, the agent can move right, left, up, or down, which are specified in the

#actions function as integers (1=right, 2=left, 3=up, 4=down). Actions can also be specified

#as strings (symbols in Julia) as follows: [:right, :left, :up, :down]. The conversion between

#action representation and action index may be updated in the action\_index function below.

#The number of actions must also be explicitly specified in the n\_actions function.

POMDPs.actions(p::GridPOMDP) = [1,2,3,4]

POMDPs.n\_actions(p::GridPOMDP) = 4

POMDPs.actions(pomdp::GridPOMDP, state::GridState) = POMDPs.actions(pomdp)

#The action\_index function enables conversion between the action representation and the

#action index that will be used to track agent location. So, if the "move right" action was

#a symbol, :right, replace a==1 with a==:right.

function POMDPs.action\_index(::GridPOMDP, a::Int64)

if a==1

return 1

elseif a==2

return 2

elseif a==3

return 3

else

return 4

end

error("invalid action: $a") #note the $ placeholder for var reference in print

end;

######TRANSITION FUNCTION######

#The transition function models how the agent moves through the grid world.

#Function isInbounds helps determine whether a targeted action is possible within

#the bounds of the world and subseuently shapes the state result of actions.

function isInbounds(pomdp::GridPOMDP, st::GridState)

if (1 <= st.xA <= pomdp.size\_x) && (1 <= st.yA <= pomdp.size\_y)

return true

end

return false

end

#The transition function proper uses the isInbounds function to determine where an action

#delivers the agent. If the targeted action is out of bounds, the

#agent rebounds into the original cell.

function POMDPs.transition(p::GridPOMDP, s::GridState, a::Int64)

x = s.xA

y = s.yA

#The neighbor array represents the possible states to which the

#agent in its current state may transition. The states correspond to

#the integer representation of each action.

neighbor = [

GridState(x+1,y), #right

GridState(x-1,y), #left

GridState(x,y+1), #up

GridState(x,y-1), #down

GridState(x,y) #original cell

]

#The target cell is the location at the index of the appointed action.

target = neighbor[a]

#If the target cell is out of bounds, the agent remains in

#the same cell. Otherwise the agent transitions to the target

#cell.

if !isInbounds(p,target)

return SparseCat([s], [1.0])

else

return SparseCat([target], [1.0])

end

end

######OBSERVATIONS######

#Like actions and states, observations are specified both through explicit parameters and

#through distributions. This simple implementation uses just one type of observation: "glitter",

#the presence of gold in the agent's current cell location; correspondingly a simple binary

#observation structure is used: an observation of glitter corresponds to a "true" observation,

#which is produced when the agent is co-located with gold. The range of observations may be

#expanded by modifying this representation using an equally simple approach, for example, using

#integers to represent the presence of each type of observation.

POMDPs.observations(::GridPOMDP) = [true, false]

POMDPs.observations(pomdp::GridPOMDP, s::GridState) = POMDPs.observations(pomdp);

POMDPs.n\_observations(::GridPOMDP) = 2

#The observation distribution establishes the likelihood

#of a true observation (glitter)

type ObservationDistribution

op\_true::Float64

end

ObservationDistribution() = ObservationDistribution(0.06)

iterator(od::ObservationDistribution) = [true, false]

#The observation function and density function maintain the observations

#received by the agent. The density function (pdf) establishes the value

#of the distribution at a particular sample. The observation function (below)

#determines the likelihood of an observation at a particular state.

function POMDPs.pdf(od::ObservationDistribution, obs::Bool)

if obs

return od.op\_true

else

return 1 - od.op\_true

end

end

#Sampling function for use in simulation

function POMDPs.rand(rng::AbstractRNG, od::ObservationDistribution)

if rand(rng) <= od.op\_true

return true

else

return false

end

end

function POMDPs.observation(pomdp::GridPOMDP, s::GridState)

od = ObservationDistribution()

if goldFound(s)

od.op\_true = 1.0

else

od.op\_true = 0.0

end

od

end

#The reward function tracks the current reward, in this case by adding to the reward sum

#if gold has been found and then returning the current total.

#The function can be expanded with other rewards/costs by including those

#additions/subtractions in other conditional branches

function POMDPs.reward(p::GridPOMDP, s::GridState, a::Int64)

r = 0.0

if goldFound(s)

r += p.r

else

r += 0.0

end

r

end

POMDPs.reward(pomdp::GridPOMDP, s::GridState, a::Int64, obs::Bool) = reward(pomdp,s,a)

POMDPs.discount(p::GridPOMDP) = p.discount

#The initial state distribution establishes the initial distribution over states.

#A SparseCat sparse array is used given the few states that have a non-zero

#likelihood of occupancy.

function POMDPs.initial\_state\_distribution(pomdp::GridPOMDP)

return SparseCat([GridState(1,1)], [1.0])

end;

#####Model Constructed######

#############Applying SARSOP solver . . .#########

#This first section of the solver generates and saves the

#pomdpx model file (model.pomdpx) that stores parameters

#instantiating the problem model and can be initialized

#from file for repeated use

pomdp = GridPOMDP()

#SARSOP solver

println("SARSOP")

#The using keyword initializes use of the SARSOP Julia solver

using SARSOP

#Initializing the solver loads the model file

solver = SARSOPSolver()

#Running the solve function creates and saves the policy file (out.policy)

#that determines an action for each state and observation

policy = solve(solver, pomdp)

#As desired, the solving package also includes functions for printing the alpha values

alphas(policy)

#=

# To run a simulation of the policy:

#Establish the initial prior across states

b = uniform\_belief(pomdp);

a = action(policy, b)

using POMDPToolbox # for simulation

pomdp = GridPOMDP() # initialize problem

init\_dist = initial\_state\_distribution(pomdp) # initialize distribution over state

up = updater(policy) # belief updater for the policy

hr = HistoryRecorder(max\_steps=14, rng=MersenneTwister(1)) # history recorder that keeps track of states, observations and beliefs

hist = simulate(hr, pomdp, policy, up, init\_dist)

#Print out each packet of simulated information

for (s, b, a, r, sp, op) in hist

println("s: $s, b: $(b.b), action: $a, obs: $op")

end

println("Total reward: $(discounted\_reward(hist))")

=#

######Visualize policy graph######

#=

To visualize the policy graph, uncomment the following codeblock that produces a .dot file. Note use of the pomdp model file model.pomdpx that was just constructed in the previous implementation. To build the graph itself, this dot file must be processed by GraphViz visualization software. GraphViz may be run in the browser at <http://www.webgraphviz.com/> (paste the contents of the .dot file in the browser window) or may be downloaded (for PC and Linux!) at <https://www.graphviz.org/>

=#

#=

pomdp = POMDPFile(pomdp, "model.pomdpx")

import SARSOP.polgraph

import SARSOP.PolicyGraphGenerator

import SARSOP.\_get\_options\_list

const EXEC\_POLICY\_GRAPH\_GENERATOR = Pkg.dir("SARSOP", "deps", "polgraph")

graphgen = PolicyGraphGenerator("Grid.dot")

function polgraph(graphgen::PolicyGraphGenerator, pomdp::SARSOPFile, policy::SARSOPPolicy)

options\_list = \_get\_options\_list(graphgen.options)

run(`$EXEC\_POLICY\_GRAPH\_GENERATOR $(pomdp.filename) --policy-file $(policy.filename) $options\_list`)

end

polgraph(graphgen, pomdp, policy)

=#

**Implementation 2: Stochastic Transitions (the agent only reaches the grid cell state it moves towards with a certain probability (p=0.8). If the action is unsuccessful, it rebounds to one of the two cells perpendicular to the direction of movement. So, if the agent moves Left, but that move is unsuccessful, it will end up either in the cell above the current cell (with p=0.1) or in the cell below the current cell (with p=0.1).**

#=

GridWorld: SARSOP solver with Stochastic transitions:

Agent reaches intended cell with p=0.8. If action is unsuccessful, the agent ends in one of the two cells perpendicular to the direction of desired action immediately surrounding the original cell, at p = 0.1, for each cell.

When the desire action transports the agent beyond the grid boundaries, it remains in its current cell with p = 0.8 or moves to the previously-described rebound cells with p =0.1 each. If the rebound cells are out of bounds, the probability accumulates on the target or original cell (so that, if one rebound cell is out of bounds, the agent may move to its target with p = 0.9).

Agent searches for cell containing gold.

Presence of gold detected via boolean observation when gold is co-located with agent

Agent moves through grid using four actions (Right, Left, Up, Down) with transitions that are successful at a rate of 80%.

When agent transition is unsuccessful, the agent rebounds to one of the two

cells located perpendicularly to the intended direction of movement.

Run file in Julia REPL using include("filename")

=#

#Install the packages needed for implementation first by "adding" the package and the using the "using" keyword

#It saves some time to "add" the package directly to the Julia REPL and use only the "using" statement in the file.

using POMDPs #For general POMDP functions

POMDPs.add("POMDPToolbox") #For model production some of the solving-related functions

Pkg.add("Distributions") #For use of the SparseCat distribution

Pkg.add("SARSOP") #For the solver itself (the using statement follows below)

using Distributions

using POMDPToolbox

#First, define the struct that will characterize each state. In this case, a state is defined

#with the current (x,y) tuple of the Agent's location in the grid, along with the(x,y)

#tuple of the location of the gold. Multiple reward states (i.e. multiple cells

#containing gold) could be represented with a vector of tuples.

struct GridState

xA::Int64

yA::Int64

xGold::Int64

yGold::Int64

end

#Once created, a GridState constructor is built, in this case containing a pre-set

#gold location (in cell [2,1]). Note the constructor requires specification only

#of Agent location for ease.

GridState(xA::Int64, yA::Int64) = GridState(xA,yA,2,1)

#To define the terminal state and shape observations, a helper function specifies

#when the agent "finds" the gold. Discovery is simple: the agent finds and retrieves

#gold when agent and gold are co-located in the same cell.

goldFound(s1::GridState) = s1.xA==s1.xGold && s1.yA==s1.yGold

#Once the foundation of a GridState has been defined, the GridWorld problem itself

#can be characterized. Here, the problem is defined using the size of the world

#(i.e. the number of grid cells), the reward (r),the probability of successfully

#transitioning to a new state (tProb), and the discount factor. Note that the

#problem can be expanded by adding additional fields including other rewards

#(or costs as negative rewards), the accuracy level of observations, etc

type GridPOMDP <: POMDP{GridState, Int64, Bool}

size\_x::Int64 #Number of grid cells in the x-direction

size\_y::Int64 #Number of grid cells in the x-direction

r::Int64 #Reward for finding gold

tProb::Float64 #Probability that transition to a new state is successful

discount::Float64 #Discount factor

end

#Once defined, a simple problem constructor specifies the default characteristics.

#The following specifies a 3x3 grid world, with a reward of +10, a transition

#probability of 0.8 and a discount factor of 0.95

function GridPOMDP()

return GridPOMDP(3,3,10,0.8,0.95)

end;

#With the problem specified, a problem instance can be built

pomdp = GridPOMDP()

#A paired function establishes that finding the gold makes the state terminal.

POMDPs.isterminal(pomdp::GridPOMDP, s::GridState) = goldFound(s)

######STATES######

#For the state distribution, each states is pushed onto an array and indexed

function POMDPs.states(pomdp::GridPOMDP)

s = GridState[] #initialize array of GridWorldStates

for xA=1:pomdp.size\_x, yA=1:pomdp.size\_y, xGold=1:pomdp.size\_x, yGold=1:pomdp.size\_y

push!(s, GridState(xA,yA,xGold,yGold))

end

return s #array of states

end;

function POMDPs.state\_index(pomdp::GridPOMDP, state::GridState)

return sub2ind((pomdp.size\_x, pomdp.size\_y, 3), state.xA, state.yA, state.xGold, state.yGold)

end

#The POMDP package requires computation of the number of expected states

POMDPs.n\_states(p::GridPOMDP) = (p.size\_x\*p.size\_y)\*(p.size\_x\*p.size\_y)

######ACTIONS#######

#The next functions specify general parameters concerning the actions available to the agent.

#In this simple world, the agent can move right, left, up, or down, which are specified in the

#actions function as integers (1=right, 2=left, 3=up, 4=down). Actions can also be specified

#as strings (symbols in Julia)like this: [:right, :left, :up, :down]. The conversion between

#action representation and action index may be updated in the action\_index function below.

#The number of actions must also be explicitly specified in the n\_actions function.

POMDPs.actions(p::GridPOMDP) = [1,2,3,4]

POMDPs.n\_actions(p::GridPOMDP) = 4

POMDPs.actions(pomdp::GridPOMDP, state::GridState) = POMDPs.actions(pomdp)

#The action\_index function enables conversion between the action representation and the

#action index that will be used to track agent location. So, if the "move right" action was

#a symbol, :right, replace a==1 with a==:right.

function POMDPs.action\_index(::GridPOMDP, a::Int64)

if a==1

return 1

elseif a==2

return 2

elseif a==3

return 3

else

return 4

end

error("invalid action: $a") #note the $ placeholder for var reference in print

end;

######TRANSITION FUNCTION######

#The transition function models how the agent moves through the grid world, using the

#transition probability to determine the success of targeted actions.

#Function isInbounds helps determine whether a targeted action is possible within

#the bounds of the world and subsequently shapes the state result of actions.

function isInbounds(pomdp::GridPOMDP, st::GridState)

if (1 <= st.xA <= pomdp.size\_x) && (1 <= st.yA <= pomdp.size\_y)

return true

end

return false

end

#The transition function proper uses the isInbounds function with the

#transition probability to determine whether a particular action

#successfully deposits the agent in the expected grid cell. Otherwise, the

#agent rebounds in one of the two perpendicular grid cells. If the

#targeted action is out of bounds, the agent might rebound into the

#original cell, with an adjusted transition probability.

function POMDPs.transition(p::GridPOMDP, s::GridState, a::Int64)

x = s.xA

y = s.yA

#The neighbor array represents the possible states to which the

#agent in its current state may transition. The states correspond to

#the integer representation of each action.

neighbor = [

GridState(x+1,y), #right

GridState(x-1,y), #left

GridState(x,y+1), #up

GridState(x,y-1), #down

GridState(x,y)

]

#The target state is the state at the index of the appointed action.

target = neighbor[a]

#The bounce array holds the possible rebound states

if a == 1 || a == 2

bounce = [3,4]

else

bounce = [1,2]

end

perCellProb = (1 - p.tProb) / 2 #0.1

#The probab array holds the likelihood of transitioning to each

#possible neighbor state. Most of these states will remain at 0.0,

#so a sparse array is used to maintain the state distribution

probab = [

0.0,

0.0,

0.0,

0.0,

0.0

]

#If the target cell is out of bounds, the agent may remain in

#the same cell, or rebound into the remaining bounce cell(s).

#Otherwise the agent may transition to the target cell with

#probability p.tProb.

#Probabilities are stored in the probab array at the index correspondingly

#to the state in the neighbor array

if !isInbounds(p,target)

if !isInbounds(p, neighbor[bounce[1]])

probab[bounce[2]] = perCellProb #prob =0.1

probab[5] = p.tProb + perCellProb #prob = 0.9

elseif !isInbounds(p, neighbor[bounce[2]])

probab[bounce[1]] = perCellProb #prob =0.1

probab[5] = p.tProb + perCellProb #prob =0.9

else

probab[bounce[1]] = perCellProb #prob =0.1

probab[bounce[2]] = perCellProb #prob =0.1

probab[5] = p.tProb #prob =0.8

end

else

if !isInbounds(p, neighbor[bounce[1]])

probab[bounce[2]] = perCellProb #prob =0.1

probab[a] = p.tProb + perCellProb #prob =0.9

elseif !isInbounds(p, neighbor[bounce[2]])

probab[bounce[1]] = perCellProb #prob =0.1

probab[a] = p.tProb + perCellProb #prob =0.9

else

probab[bounce[1]] = perCellProb #prob =0.1

probab[bounce[2]] = perCellProb #prob =0.1

probab[a] = p.tProb #prob =0.8

end

end

#Since the likelihood that the agent enters most of the neighboring

#cells is 0, a sparse array from the Julia distributions package (SparseCat)

#is used to store the set of state probabilities, each associated with the

#corresponding neighbor state.

return SparseCat(neighbor, probab)

end

######OBSERVATIONS######

#Like actions and states, observations are specified both through explicit parameters and

#through distributions. This simple implementation uses just one type of observation: "glitter",

#the presence of gold in the agent's current cell location; correspondingly a simple binary

#observation structure is used: an observation of glitter corresponds to a "true" observation,

#which is produced when the agent is co-located with gold. The range of observations may be

#expanded by modifying this representation using an equally simple approach, for example, using

#integers to represent the presence of each type of observation.

POMDPs.observations(::GridPOMDP) = [true, false]

POMDPs.observations(pomdp::GridPOMDP, s::GridState) = POMDPs.observations(pomdp);

POMDPs.n\_observations(::GridPOMDP) = 2

#The observation distribution constructs the establishes the likelihood

#of a true observation (glitter)

type ObservationDistribution

op\_true::Float64

end

ObservationDistribution() = ObservationDistribution(0.06)

iterator(od::ObservationDistribution) = [true, false]

#The observation function and density function maintain the observations

#received by the agent. The density function (pdf) establishes the value

#of the distribution at a particular sample. The observation function (below)

#determines the likelihood of an observation at a particular state.

function POMDPs.pdf(od::ObservationDistribution, obs::Bool)

if obs

return od.op\_true

else

return 1 - od.op\_true

end

end

#Sampling function for use in simulation

function POMDPs.rand(rng::AbstractRNG, od::ObservationDistribution)

if rand(rng) <= od.op\_true

return true

else

return false

end

end

function POMDPs.observation(pomdp::GridPOMDP, s::GridState)

od = ObservationDistribution()

if goldFound(s)

od.op\_true = 1.0

else

od.op\_true = 0.0

end

od

end

#The reward function tracks the current reward, in this case by adding to the reward sum

#if gold has been found and then returning the current total.

#The function can be expanded with other rewards/costs by including those

#additions/subtractions in other conditional branches

function POMDPs.reward(p::GridPOMDP, s::GridState, a::Int64)

r = 0.0

if goldFound(s)

r += p.r

else

r += 0.0

end

r

end

POMDPs.reward(pomdp::GridPOMDP, s::GridState, a::Int64, obs::Bool) = reward(pomdp,s,a)

POMDPs.discount(p::GridPOMDP) = p.discount

#The initial state distribution establishes the initial distribution over states.

#A SparseCat sparse array is used given the few states that have a non-zero

#likelihood of occupancy.

function POMDPs.initial\_state\_distribution(pomdp::GridPOMDP)

return SparseCat([GridState(1,1)], [1.0])

end;

#####Model Constructed######

#############Applying SARSOP solver . . .#########

#This first section of the solver generates and saves the

#pomdpx model file (model.pomdpx) that stores parameters

#instantiating the problem model and can be initialized

#from file for repeated use

pomdp = GridPOMDP()

#SARSOP solver

println("SARSOP")

#The using keyword initializes use of the SARSOP Julia solver

using SARSOP

#Initializing the solver loads the model file

solver = SARSOPSolver()

#Running the solve function creates and saves the policy file (out.policy)

#that determines an action for each state and observation

policy = solve(solver, pomdp)

#As desired, the solving package also includes functions for printing the alpha values

alphas(policy)

#=

# To run a simulation:

#Establish the initial prior across states

b = uniform\_belief(pomdp);

a = action(policy, b)

using POMDPToolbox # for simulation

pomdp = GridPOMDP() # initialize problem

init\_dist = initial\_state\_distribution(pomdp) # initialize distribution over state

up = updater(policy) # belief updater for the policy

# history recorder that keeps track of states, observations and beliefs

hr = HistoryRecorder(max\_steps=14, rng=MersenneTwister(1))

hist = simulate(hr, pomdp, policy, up, init\_dist)

#Print out each packet of simulated information

for (s, b, a, r, sp, op) in hist

println("s: $s, b: $(b.b), action: $a, obs: $op")

end

println("Total reward: $(discounted\_reward(hist))")

=#

######Visualize policy graph######

#=

To visualize the policy graph, uncomment the following codeblock that produces a .dot file. Note use of the pomdp model file model.pomdpx that was just constructed in the previous implementation. To build the graph itself, this dot file must be processed by GraphViz visualization software. GraphViz may be run in the browser at <http://www.webgraphviz.com/> (paste the contents of the .dot file in the browser window) or may be downloaded (for PC and Linux!) at <https://www.graphviz.org/>

=#

#=

pomdp = POMDPFile(pomdp, "model.pomdpx")

import SARSOP.polgraph

import SARSOP.PolicyGraphGenerator

import SARSOP.\_get\_options\_list

const EXEC\_POLICY\_GRAPH\_GENERATOR = Pkg.dir("SARSOP", "deps", "polgraph")

graphgen = PolicyGraphGenerator("Grid.dot")

function polgraph(graphgen::PolicyGraphGenerator, pomdp::SARSOPFile, policy::SARSOPPolicy)

options\_list = \_get\_options\_list(graphgen.options)

run(`$EXEC\_POLICY\_GRAPH\_GENERATOR $(pomdp.filename) --policy-file $(policy.filename) $options\_list`)

end

polgraph(graphgen, pomdp, policy)

=#