POMDPStressTesting.jl Example: Walk1D

Robert J. Moss MOSSR@CS.STANFORD.EDU
Computer Science, Stanford University

Abstract

In this self-contained tutorial, we define a simple problem for adaptive stress testing (AST) to find failures. This problem, called Walk1D, samples random walking distances from a standard normal distribution $\mathcal{N}(0, 1)$ and defines failures as walking past a certain threshold (which is set to ± 10 in this example). AST will either select the seed which deterministically controls the sampled value from the distribution (i.e. from the transition model) or will directly sample the provided environmental distributions. These action modes are determined by the seed-action or sample-action options. AST will guide the simulation to failure events using a notion of distance to failure, while simultaneously trying to find the set of actions that maximizes the log-likelihood of the samples.

1. Gray-box Simulator and Environment

The simulator and environment are treated as gray-box because we need access to the state-transition distributions and their associated likelihoods.

Parameters. First, we define the parameters of our simulation.

```
@with_kw mutable struct Walk1DParams
    startx::Float64 = 0 # Starting x-position
    threshx::Float64 = 10 # +- boundary threshold
    endtime::Int64 = 30 # Simulate end time
end
```

Simulation. Next, we define a `GrayBox.Simulation` structure.

```
@with_kw mutable struct Walk1DSim <: GrayBox.Simulation
    params::Walk1DParams = Walk1DParams() # Parameters
    x::Float64 = 0 # Current x-position
    t::Int64 = 0 # Current time ±
    distribution::Distribution = Normal(0, 1) # Transition distribution
end
```

1.1 GrayBox.environment

Then, we define our `GrayBox.Environment` distributions. When using the `ASTSampleAction`, as opposed to `ASTSeedAction`, we need to provide access to the sampleable environment.

```
GrayBox.environment(sim::Walk1DSim) = GrayBox.Environment(:x => sim.distribution)
```

1.2 GrayBox.transition!

We override the transition function from the `GrayBox` interface, which takes an environment sample as input. We apply the sample in our simulator, and return the log-likelihood.

```
function GrayBox.transition!(sim::Walk1DSim, sample::GrayBox.EnvironmentSample)
    sim.t += 1 # Keep track of time
    sim.x += sample[:x].value # Move agent using sampled value from input
    return logpdf(sample)::Real # Summation handled by ‘logpdf()‘
end
```

2. Black-box System

The system under test, in this case a simple single-dimensional moving agent, is always treated as black-box. The following interface functions are overridden to minimally interact with the system, and use outputs from the system to determine failure event indications and distance metrics.

2.1 BlackBox.initialize!

Now we override the `BlackBox` interface, starting with the function that initializes the simulation object. Interface functions ending in `!` may modify the `sim` object in place.

```
function BlackBox.initialize!(sim::Walk1DSim)
    sim.t = 0
    sim.x = sim.params.startx
end
```

2.2 BlackBox.distance

We define how close we are to a failure event using a non-negative distance metric.

```
BlackBox.distance(sim::Walk1DSim) = max(sim.params.threshx - abs(sim.x), 0)
```

2.3 BlackBox.isevent

We define an indication that a failure event occurred.

```
BlackBox.isevent(sim::Walk1DSim) = abs(sim.x) >= sim.params.threshx
```

2.4 BlackBox.isterminal

Similarly, we define an indication that the simulation is in a terminal state.

```
BlackBox.isterminal(sim::Walk1DSim) =
    BlackBox.isevent(sim) || sim.t >= sim.params.endtime
```

2.5 BlackBox.evaluate!

Lastly, we use our defined interface to evaluate the system under test. Using the input sample, we return the log-likelihood, distance to an event, and event indication.

```
function BlackBox.evaluate!(sim::Walk1DSim, sample::GrayBox.EnvironmentSample)
    logprob::Real = GrayBox.transition!(sim, sample) # Step simulation
    d::Real = BlackBox.distance(sim) # Calculate miss distance
    event::Bool = BlackBox.isevent(sim) # Check event indication
    return (logprob::Real, d::Real, event::Bool)
end
```

3. AST Setup and Running

Setting up our simulation, we instantiate our simulation object and pass that to the Markov decision proccess (MDP) object of the adaptive stress testing formulation. We use Monte Carlo tree search (MCTS) with progressive widening on the action space as our solver. Hyperparameters are passed to `MCTSPWSolver`, which is a simple wrapper around the POMDPs.jl implementation of MCTS. Lastly, we solve the MDP to produce a planner. Note we are using the `ASTSampleAction`.

```
function setup_ast(seed=0)
    # Create gray-box simulation object
    sim::GrayBox.Simulation = Walk1DSim()

    # AST MDP formulation object
    mdp::ASTMDP = ASTMDP{ASTSampleAction}(sim)
    mdp.params.debug = true # record metrics
    mdp.params.top_k = 10 # record top k best trajectories
    mdp.params.seed = seed # set RNG seed for determinism

    # Hyperparameters for MCTS-PW as the solver
    solver = MCTSPWSolver(n_iterations=1000, # number of algorithm iterations
                          exploration_constant=1.0, # UCT exploration
                          k_action=1.0, # action widening
                          alpha_action=0.5, # action widening
                          depth=sim.params.endtime) # tree depth

    # Get online planner (no work done, yet)
    planner = solve(solver, mdp)

    return planner
end
```

After setup, we *search* for failures using the planner and output the best action trace.

```
planner = setup_ast()
action_trace = search!(planner)
```

We can also *playback* specific trajectories and print intermediate x-values.

```
final_state = playback(planner, action_trace, sim->sim.x)
```

Finally, we can print metrics associated with the AST run for further analysis.

```
failure_rate = print_metrics(planner)
```

4. Solvers

The solvers provided by the POMDPStressTesting.jl package include the following.

```
# Reinforcement learning
    MCTSPWSolver
# Deep reinforcement learning
    TRPOSolver
    PPOSolver
# Stochastic optimization
    CEMSolver
# Baselines
    RandomSearchSolver
```

5. Reward Function

The AST reward function gives a reward of 0 if an event is found, a reward of negative distance if no event is found at termination, and the log-likelihood during the simulation.

$$R(p, e, d, \tau) = \begin{cases} 0 & \text{if } \tau \wedge e \\ -d & \text{if } \tau \wedge \neg e \\ \log(p) & \text{otherwise} \end{cases}$$

```
function R(p,e,d,τ)
    if τ && e
        return 0
    elseif τ && !e
        return -d
    else
        return log(p)
    end
end
```

File generated using TeX.jl: `https://github.com/mossr/TeX.jl`

Embedded Julia source file.