Michael Schniepp
PSTAT 160A

# The Racers Dilemma: A Markov Decision Process Experiment

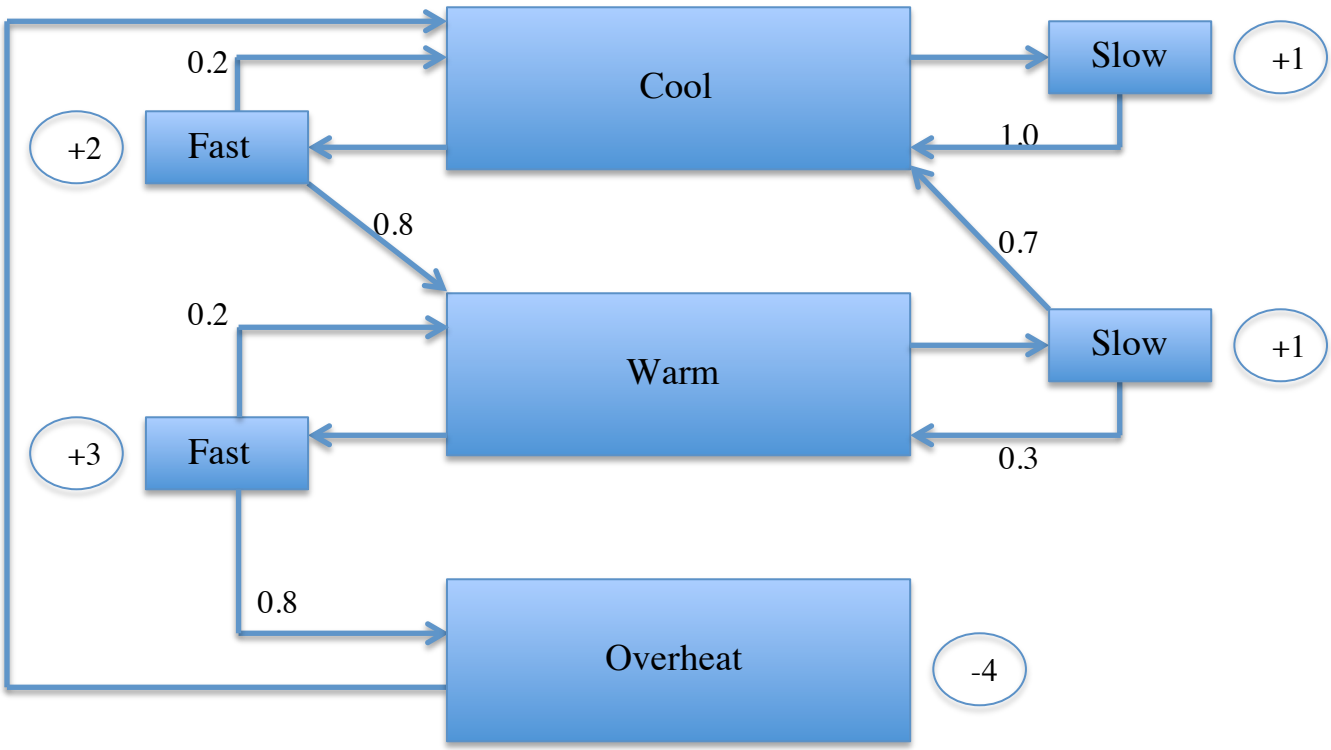
**Summary:** Our experiment's aim was to solve the racers dilemma of completing a 'race' in the fewest steps possible with the dilemma being how often to go fast or slow, each with its own reward and risk associated. Our intention was to determine if a risk-loving decision-making policy would be statistically significantly more efficient. By designing and implementing a Markov Decision Process in Python we were able to effectively construct a model and test our strategy against a risk-averse control strategy.

**Motivation:** We designed a race-like game where the objective was for the driver to reach a specified distance by making various decisions to increase his distance. Our motivation for the experiment was to design and implement a Markov Decision Process to test the effectiveness of various strategy types and observe each policies effects as the driver made his way to the destination distance.

**Methods:** The setup of the race is as follows: The racer will continually exist in one of three states: cool engine temperature, warm engine temperature, or overheated engine temperature. Decisions are made in the cool or warm states, while the overheated state only serves as the primary risk factor by existing solely to deduct distance points. When the overheat state is reached then the driver is sent to the cool state to restart the process. While in the cool or warm states the driver can make a decision between driving fast or slow. Driving fast will provide more distance points bringing him closer to his destination but also run the risk of bringing him to the overheated state. Choosing the slow option provided no risk as the cost of offering minimal reward.

Our risk-averse control strategy was designed to select the 'slow' option each step, providing him with plus one distance points each step, and thus his expected number of steps to the destination was equal to the destination value each time guaranteed. The risk-loving policy we formulated always chose the slow option while in the first 90% of reaching the goal value, and when above 90% of the goal he only chooses the fast option. To get a more clear idea of the model refer to the chart below.

**Transition Diagram**

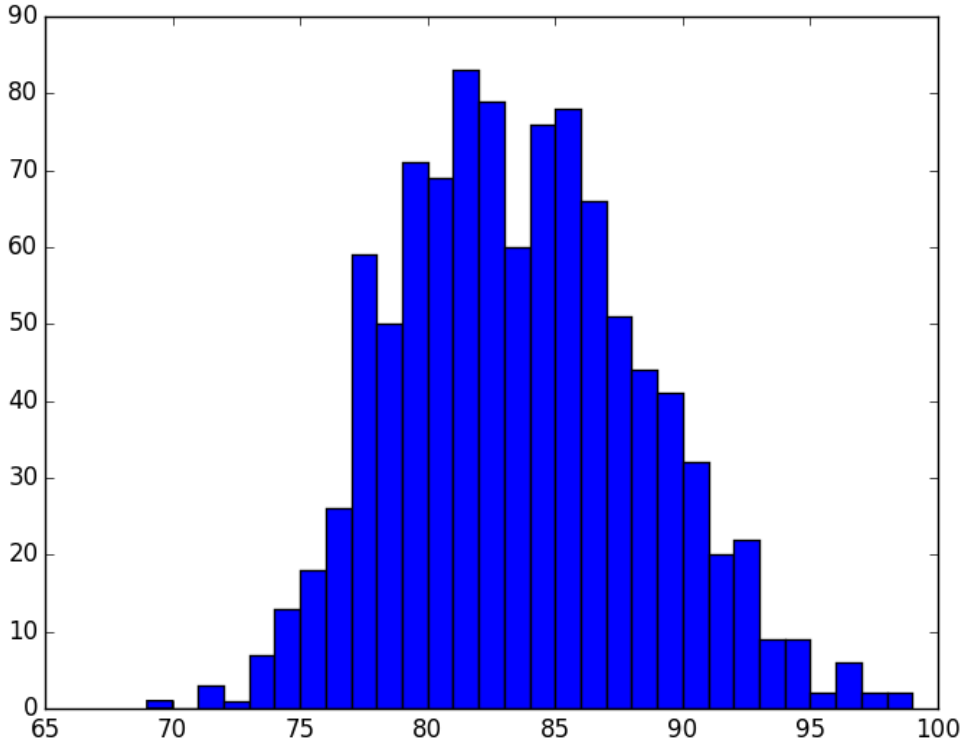



Legend: Cool, Warm, & Overheat: States
Values: i.e. 1.0, 0.7, 0.3, 0.2, 0.8: transition probabilities
Circle Numbers: Reward values

**Results:** For our testing we used an objective distance of 100 points, thus the control would be 100 steps if the driver selected slow each time. The first risk-loving strategy we employed used the slow option in the first 50% of the goal and the fast in the second 50% of the goal. This strategy took on far too much risk and ended up generating excessive loss with average runtimes of 150 steps to reach the goal, 50% longer than the risk-averse strategy. With some tuning we found choosing slow in the first 90% and fast in the last 10% to provide greater efficiency. The idea was that in the first 90% our driver could spend some time accumulating distance points and then in the final 10% he could take the riskier options to quickly finish the race. To test the mean time to finish the race with this policy we ran the simulation 1000 times and computed the mean. The strategy yielded a mean of 83 steps to finish versus the control of 100 steps. The output is shown below:

```
('Mean', 83.174000000000007)
('Variance', 23.903724000000004)
('Standard Deviation', 4.8891434832698462)
```

To better visualize the results we generated a histogram of the resulting number of steps it took to reach the goal of 100. Clearly you can see over 1000 trials the resulting distribution was approximately normal.




To conclude our results, we found it to be beneficial and more effective to have risk-loving tendencies.

**Discussion:** For this project we wanted to construct a model that would utilize a Markov Decision Process to produce varying results based on our own decision making policies that would reflect strategies that we felt would be most efficient. We found that too much risky behavior would result in too much loss and would reduce effectiveness and thus the amount of risk to take must be calibrated carefully to match the system. While the system we constructed is based off of a car race this type of system can be applied to a variety of other situations involving a risk/reward-based system, in economics, finance or some other situation where risk must be managed. In the future we would like to explore more in-depth methods and techniques to better maximize our efficiency and develop more complex models with a more practical application.

Python code for reference:

```
'''
This is an implementation and expirimentation with Markov Decsion
Processes
Title: Race Car Game
'''

import numpy as np
import matplotlib.pyplot as plt

# Transition Probabilities
cool_fast = [0.2,0.8]
cool_slow = [1.0]
warm_fast = [0.2,0.8]
warm_slow = [0.3,0.7]
overheat = [1.0]

states = [1,2,3]

def race():
      # Transition Probabilities
      cool_fast = [0.2,0.8]
      cool_slow = [1.0]
      warm_fast = [0.2,0.8]
      warm_slow = [0.3,0.7]
      overheat = [1.0]

      states = [1,2,3]

      # initial fortune
      distance = 0
      goal = 100

      def random_pick(statesList, transMatrix): # state chooser
            x = np.random.uniform(0, 1)
            cumulative_probability = 0.0
            for state, ij_prob in zip(statesList, transMatrix):
                  cumulative_probability += ij_prob
                  if x < cumulative_probability: break
            return state
      n = 100
      steps = 0
      state = [1]
      endSteps = []
      while distance < goal+1:
            if state[0] == 1:
            state[0] = random_pick(states,cool_fast)
            distance += 2
            elif state[0] == 2 and distance <= (0.90)*goal:
            state[0] = random_pick(states,warm_slow)
            distance += 1
            elif state[0] == 2 and distance > (0.10)*goal:
            state[0] = random_pick(states[1:],warm_fast)
            distance += 3
            elif state[0] == 3:
            state[0] = 1
            distance += -4
```

```
      steps += 1
return steps

endSteps = []
for i in range(0,1000):
      endSteps.append(race())

xbar = np.mean(endSteps)
std = np.std(endSteps)
print('Mean',xbar)
print('Variance',np.var(endSteps))
print('Standard Deviation',std)

plt.hist(endSteps, bins = 30)
plt.show()
```