Chopsticks Reinforcement Learning and Game Analysis

W. Aaron Morris

Preface

I feel that I have really failed at this project because I don't feel confident in the answer and there is a lot of work still left to do in finishing this analysis. Initially I was disappointed in the fact I couldn't get the appropriate data; however, I am glad that I chose to analyze the game of Chopsticks. I really wanted learn how to analyze large Markov Chains that all states cannot be permutated. My thought was that I could potentially layer in the Bayesian methods previously learned to estimate the transition probabilities.

In my first attempt, I implemented the game and have a randomly simulated $40,000$ games; however, I could never wrangle the data I generated. Therefore, I decided to simpliify my project and figure out two states before expanding to more states. I was able to generate the states by looping; however, I found this was taking too long for more players and implemented this using recursion and dynamic programming; this was my first time programming really understanding recurion and dynamic programming.

The next hard part was to generate the right transition matrix. I found myself having a **super** difficult time working with MDPToolbox. In first couple of attempts, I got several erros most of which involved not obtaining an appropriate matrix shape and it being stochastic. After some attempts, I succumbed to manualling implementing the Bellman equation. Through this implementation, I realized the true source of my errors. Initially, I have the entire game-all n players-represented by the transition matrix. Therefore, I modified the probability matrix from only the viewpoint of a single player; thus including the stochastic behavior and transition probabilities given an action. However, this wasn't the only thing wrong; I notice that the shape of each action was different because some states cannot perform certain actions. While the Markov Process of the states was correct, I realized that a Markov Decision Process has n probability matrices of shape SxS; one for each action. After a lot of additional reading and watching Youtube videos, I finally found a probability matrix I was pleased with that actually worked with MDPToolbox.

After implementing, I was looking through the recommended policies and discovered some very unreasonable behavior. The policy appeared to have learned how to cheat the game. After some playing with various rewards and transitions probabilities, I think that I have arrive at something a lot more reasonable. Given my time constraint, my goal now is to analyze the various algorithms and see how different their policies are.

see how different their policies are.

I really started trying to make this a formal paper about the best policy, the implementation of MDPs with a large number of states. However, a lot of my writing has devolved into more me trying to learn how to really think and implement reinforcement learning. This project has defnitely help solidify how to actually implement these algorithms.

I feel like I'm a little stuck, but I will probably keep hacking away at this after this class. However, I have decided to throw in a bonus project of analyzing my children's sleep patterns as a baby in hopes to make up for the lacking of my primary project.

Overview

Chopsticks is a simple game that teaches kids basic counting and mathematical operations. The game can be directly mapped into a discrete Markov process. During class, we looked at utilizing MDPToolbox to find optimal policies. I will try to apply the ideas behind Discrete Time Markov Chains and MDPToolbox to analyze and derive an appropriate policy for playing chopsticks.

Note: I originally set out to find the optimal policy given n players. However, I soon realized that this would be harder than anticipated. I think that I originally set up my data model incorrectly.

Chopsticks Game and Rules

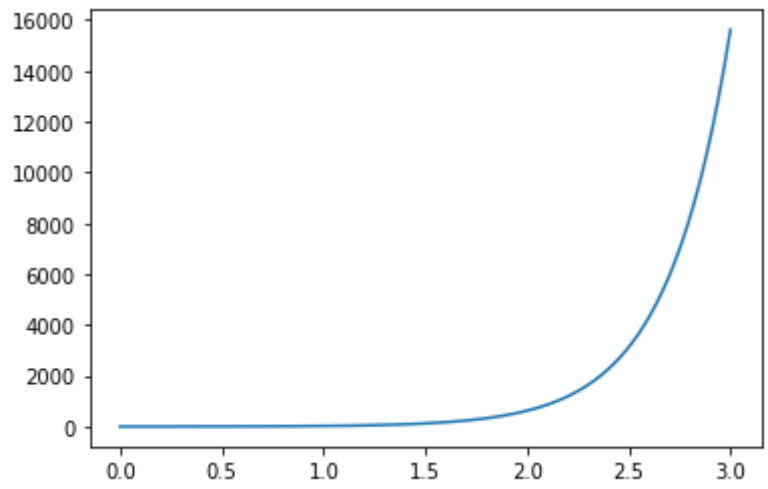
The rules are simple.

1. All players begin with a predefined number of hands and several chopsticks per hand. Typically, two hands and one chopstick.
2. Players take turns picking an offensive hand and the opponent's (defensive) hand. The number of chopsticks in the offensive hand gets added to the defensive hand. A player may also choose to redistribute their chopsticks between two of their own hands.
3. The defensive player's hand is assessed at the end of each turn. A hand with five or more chopsticks is considered dead and can be brought back through redistribution. A more interesting variant-that this project will analyze-subtracts the number of chopsticks over 5 from 5. For example, the defensive hand has seven chopsticks after an attack. The player would be left with three chopsticks at the end of the round.
4. The last player alive wins.

States and Actions

With two players and two hands, the number of potential states can be calculated with a simple $5^4 = 625$ states. Keeping the number of hands constant, this grows by a power of 2 with each new player added. There can be some advantages taken given the symmetry of some of the more basic versions of the game. Given my desire to create an algorithm flexible enough to handle any configuration,

```
Possible States for 2 Players: 625 (Distinct Combinations 225)
Possible States for 3 Players: 15625 (Distinct Combinations 3375)
Possible States for 4 Players: 390625 (Distinct Combinations 50625)
Possible States for 5 Players: 9765625 (Distinct Combinations 759375)
```



States and Rewards

In my initial tests, we can quickly see our ability to permute the state space is diminishing. Ferguson and Pumperla state, "If you could read out a hypothetical sequence all the way to the end of the game, you'd know who'd win; it's easy to decide whether that's a good sequence. But that's not practical in any game more complex than tic-tac-toe: the number of possible variations is just too large" (Ferguson, Kevin; Pumperla, Max 2019).

The game of Chopsticks has an interesting attribute. Upon a player being eliminated, a new game starts with $n - 1$ players with a specific starting space. I'm hoping that dynamic programming can take advantage of this; however, I don't know how this will work yet.

```
2 Players States: 625
CPU times: user 1.54 ms, sys: 0 ns, total: 1.54 ms
Wall time: 1.62 ms
```

```
3 Players States Generated: 15625
CPU times: user 18.2 ms, sys: 193 µs, total: 18.4 ms
Wall time: 18.6 ms
```

```
 4 Players States Generated: 390625
CPU times: user 598 ms, sys: 21.7 ms, total: 620 ms
```

```
CPU times: user 598 ms, sys: 21.7 ms, total: 620 ms
Wall time: 895 ms
```

Chopsticks: 2 Players

Transitions and Rewards

In order to use the MDPToolbox (*Chadès, Iadine, Guillaume Chapron, et al. 2014*), I need to have a square and stochastic transition matrix of shape (Actions, States, States). We already know that there are 625 different state combinations; we easily calculate the 13 actions. We cannot simply multiple these to get the population of states because we are not interested in the next state but the next state for the player with the stochastic behavior of the other player. In permutating these combinations, $5,078,125$ state-action pairs—a number of these combinations are just to satisfy the MDPToolbox requirements.

```
State Action Pairs: 5078125
```

MDPToolbox Processing

After working through a lot of weird errors, I finally got MDPToolbox to finally accept my probability matrix. While I really want to formally compare the various outputs of MDPToolbox, I found that the policy generated by many of the algorithms didn't seem correct. I have spent a lot of hours playing with the rewards and discount to try to get something more reasonable.

Using the Policy Iteration algorithm, it is evident that the algorithm converges and in a reasonable time.

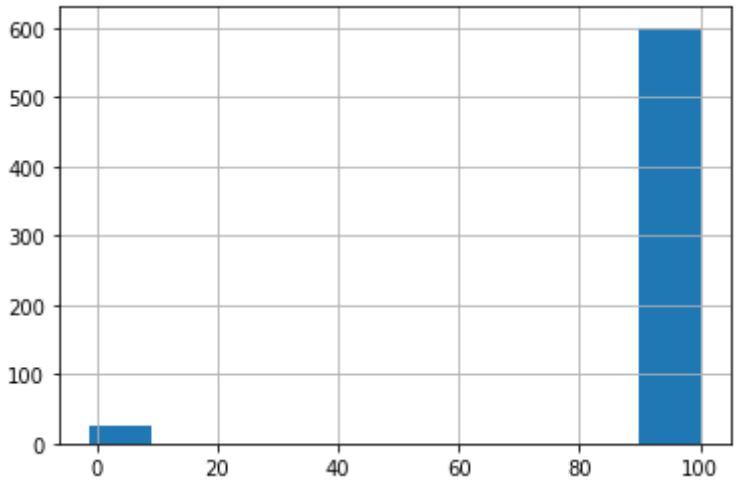
	state	policy_str	value
462	[(3,3),(2,2)]	send,0,0	98.040497
200	[(1,3),(0,0)]	g,a,m,e,_,o,v,e,r	99.990058
397	[(3,0),(4,2)]	send,0,1	97.607869
75	[(0,3),(0,0)]	g,a,m,e,_,o,v,e,r	99.990058
259	[(2,0),(1,4)]	send,0,1	97.776924
260	[(2,0),(2,0)]	split,0,1,1	98.031200
513	[(4,0),(2,3)]	send,0,1	97.596598
427	[(3,2),(0,2)]	send,0,1	99.990058
376	[(3,0),(0,1)]	split,0,1,1	97.839531
157	[(1,1),(1,2)]	send,0,0	97.400688

```
policy[policy['state']=='[(1,1),(1,1)]']['policy_str'].iloc[0].split(",")
```

```
['send', '0', '0']
```

```
policy.to_json("2p.json")
```

```
<AxesSubplot:>
```



What Move Should I take?

The MDP policy really doesn't like defensive moves when playing with 2 players. When in doubt always attack your opponent.

MDP Against Random CPU

When I used the policy against my random CPU, the MDP won 99 of the game regardless of which player took the first turn. When the Policy plays against itself, a stale-mate occurs. This is where I would like to add adjust of the explore vs exploitation of my agent. It makes complete sense that the MDP would lose on occasion to a random player. However, I think this project could be improved many ways including but not limited to: 1) adjusting the reward functions, 2) applying different algorithms, 3), allowing the algorithm to choose moves probabilistically (ie doesn't always take the best move). The most astonishing thing was that the best policy from Wikihow article on *How to Always Win Chopsticks* doesn't match the policy generated by the MDP algorithm. I'll be looking into this and understand why.

In this project, I originally set out to come up with a generic policy that could be used on n players and m hands. I wanted to learn more about the kd tree search algorithm that AlphaGo to help aid in large state spaces. Yet, I found myself really struggling with wrangling the right transition matrix, the right state space, the sampling of the state space. In a lot of ways I feel like I totally bombed this project, yet, I feel like the struggle has allowed me to progress and puch through. That said, I feel like I failure in my initial goals and objects; however, I feel much more confident in implementing and utilizing the various MDP algorithms.

3 Players

I realize that I have to modify the function to get my transition matrix to get all the interactions between an n number of states. This isn't an easy change and I'm running out time. I'm going to try to focus on evaluating this using my simulations; however, I don't feel that confident in my approach to getting all the states. My goal is to try to use something like the kd tree that AlphaGo uses. However, it is taking me a while to get through the book to try to build that out.

```
ValueError: too many values to unpack (expected 2)
```

Random Simulation

After building the game, a series of random simulations were used to test the code. Instead of throwing these simulated games away, I am going to try to use them to establish baseline probabilities to try to jump start the reinforcement algorithm.

Initial Observation given Random Simulations

Given the fairly large sample of random states from testing, we can clearly see that the majority of the states have negative values. Initial observations yield that all states possess a negative value. This makes sense given two reasons: 1) each player has a 2/3 chance of losing and 2) there are significantly more losing states than winning states.

3 Players

This was my first attempt at this whole project; however, I couldn't quite get the data and the algorithm appropriate wrangled together. From my brief-potentially wrong- analysis. The game changes from 2 to 3 player from an offensive game to a more defensive. With 3 players, it appears that the goal is to survive and the likelihood of losing is $\frac{2}{3}$ at base. That said, my initial thought was to switch to a 2 player game when a player was eliminated. But I think that I am going to have settle for comparing some of the values of states for two people.

Manual Bellman Equation Implementation

Long Term Probabilities

I remember telling one of the first babysitters, " If Addie Mae gets upset you have a 1/3 chance that she needs sleep, food or a diaper change." While this statement is true about the reason for the problem-especially for an infant, the probabilities might not neccessarily be truly a third.

After looking at the long-run probabilities, I can quickly give a babysitter the order in which I would try to calm our upset child. These results have flabbergasted my wife and I, we would have never expected that my second child would have a higher probability of being in a state of nursing. My wife makes several comments about how it seems like she was always feeding our first because of the uncertainty that is partnered with being a first time parent. After some careful consideration, I think it would be wise to condition the probability on the child's age. Since my youngest hasn't hit the same stages where she is eating 3 or 4 distinct meals, I think it would be more insightful to see this conditioned on the age of the child.

Transition Probability Matrix

	next_activity	diaper	nursing	sleep
Baby	activity			
1	diaper	0.143791	0.856209	0.000000
	nursing	0.079684	0.181265	0.739051
	sleep	0.000000	0.695702	0.304298
2	diaper	0.111111	0.648148	0.240741
	nursing	0.026391	0.226047	0.747562
	sleep	0.001235	0.810994	0.187770

Long Term Probabilities

	Child 1	Child 2
diaper	0.043178	0.015803
nursing	0.463956	0.510062
sleep	0.492865	0.474136

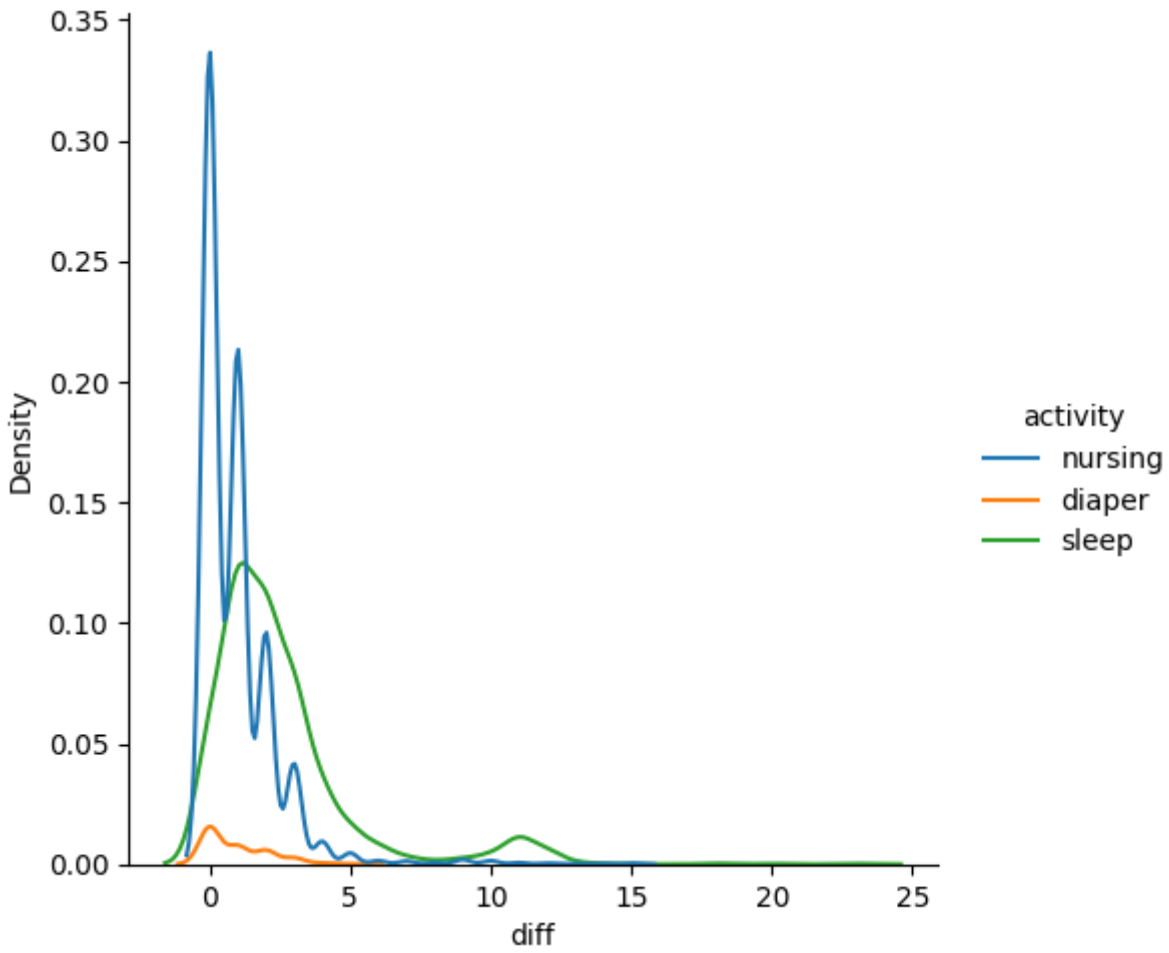
Renewal Theory

We have established that the inter-arrival times in a state is exponential; however, I feel that there is more that I can do with this data particularly using renewal theory and Semi-Markov processes. I would really like to try to identify changes in sleep patterns as both my children age out of the infant and toddler phases.

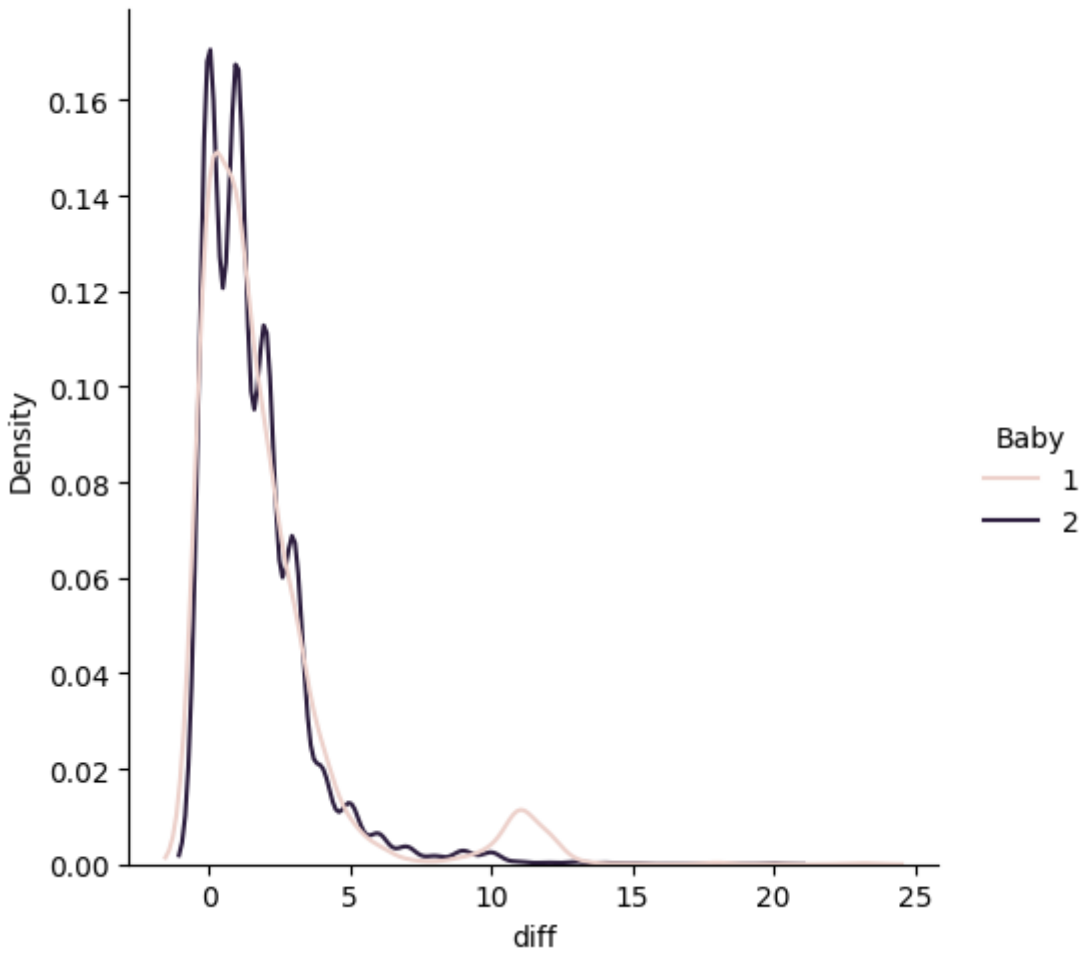
Inter-arrival Times

Using the eyeball test, the inter-arrival times betweena activities appears to match to a geometric distribution. However, we can also tell that there is some divergence from the distribution. My intial guess is that this occurs over-night; once the child begins sleeping through the night. I wonder what would happen if I fully made this a renewal process with something like an end of day state.

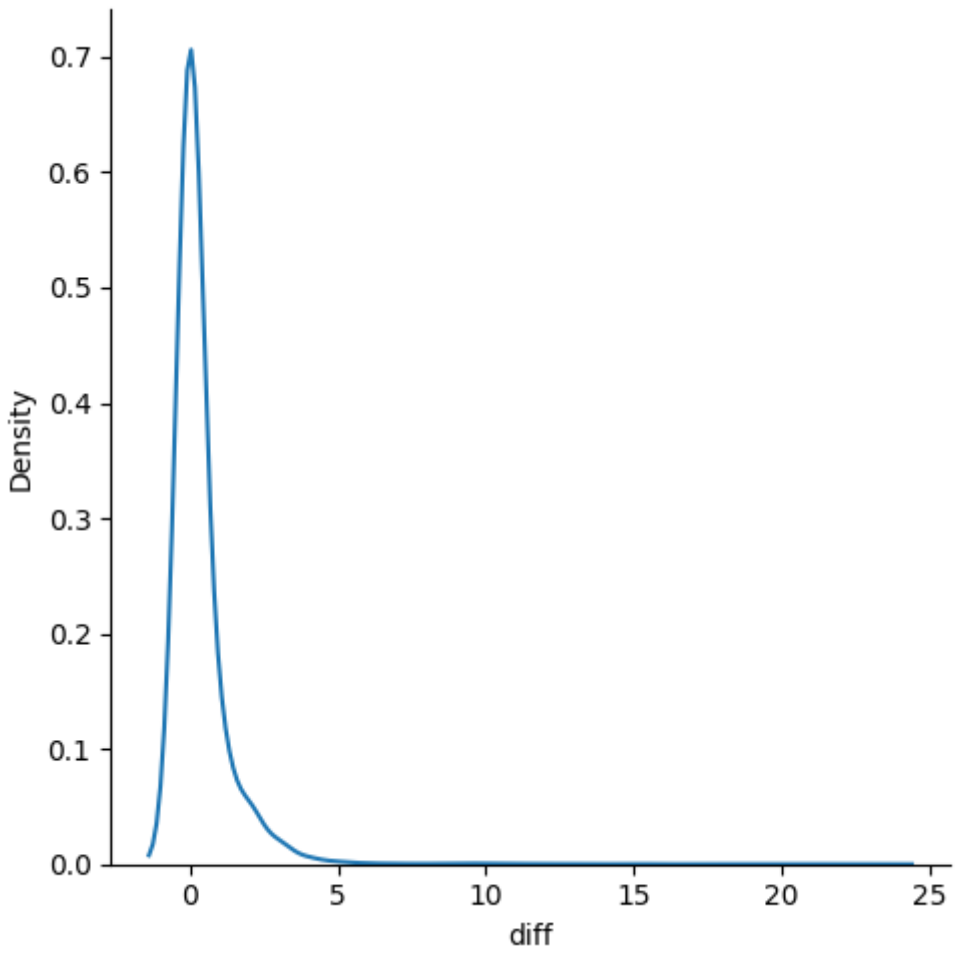
```
<seaborn.axisgrid.FacetGrid at 0x7fddc2373a00>
```




```
<seaborn.axisgrid.FacetGrid at 0x7fddc517daf0>
```




```
<seaborn.axisgrid.FacetGrid at 0x7fddc1e57490>
```


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Density
0
5
10
15
20
25
diff