CSCI 154

Abdulaziz Al-Said, Kalvin Xiong, Weston Hawes, Anthony Lopez

Black Jack (Monte Carlo)

Motivation

Using simulation techniques to calculate the effectiveness of tactics/methods that would be difficult to do otherwise.

Problem

Write a Monte Carlo script to evaluate the following Monte Carlo Policies

All policies are only for the player!

- Policy 1: if your hand ≥ 17, stick. Else hit
- Policy 2: if your hand ≥ 17 and is hard, stick. Else hit unless your hand = 21
- Policy 3: Always stick
- Policy 4: Always hit
- Policy 5: if your hand is ≤ 10 and soft, convert ace into 11 points instead of 1 and stick. Else hit unless your hand = 21

Problem (cont)

Evaluate all policies for the following versions of the game:

- Infinite deck: On every run a card is drawn with equal probability.
- Single deck: One deck of cards is used. The deck is re-shuffled after every game

Related Work and Background Material

Blackjack Rules:

Goal: Each participant attempts to beat the dealer by getting a count as close to 21 as possible, without going over 21.

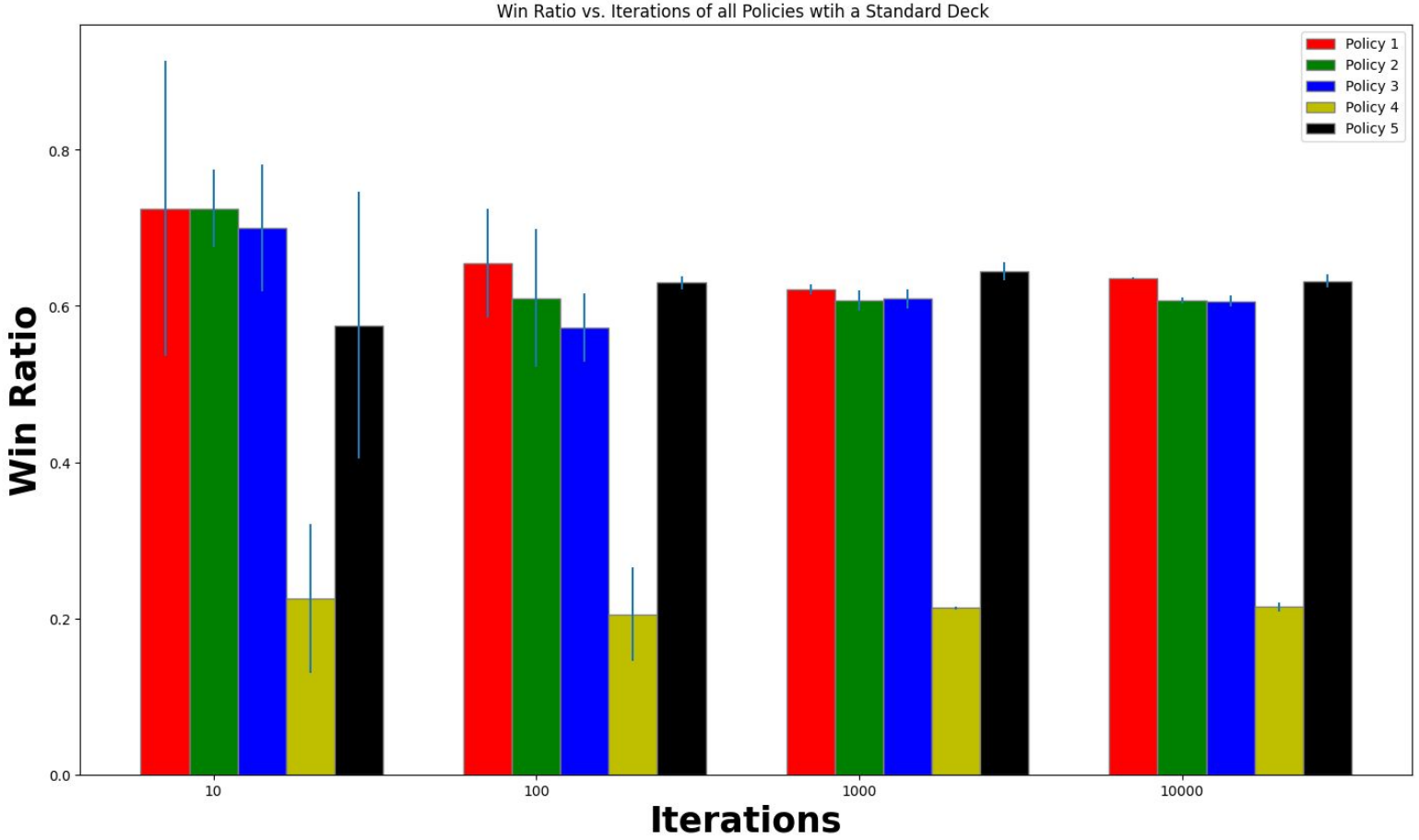
Values: it is up to each individual player if an ace is worth 1 or 11. Face cards are 10 and any other card is its pip value.

Approach

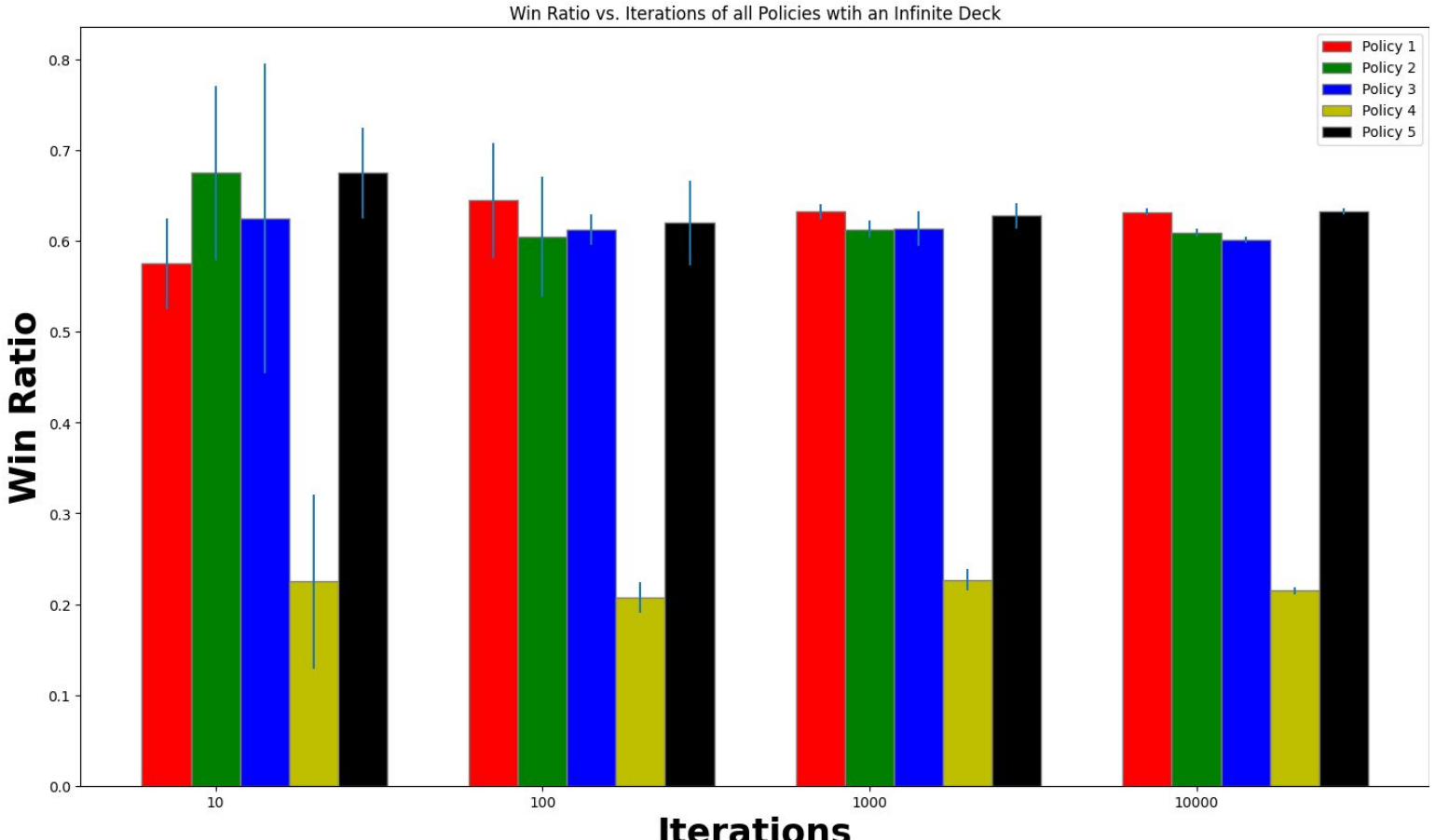
- We created an array to represent the 52 card deck
- Deal out the hands by randomly generating an index and removing that card from the deck
    - 2 cards for both the player and the dealer

- Code simulates players turn based on policy chosen
- Next, code simulates dealers turn (which is the universal policy of hitting once if their hand is <21)
- Compare hands to find if the player won
- Store wins in local variable and increment based on the number of iterations(10,100,1000,10000)
- Create and display graphs

Standard Deck



Infinite Deck



Results

- Standard deck
  - Policy 1 & 5 have consistently good results
- Infinite deck
  - Policy 5 has better results on average with policy 1 being a strong policy
- Both
  - Policy 2, 3, and 4 are extremely similar across iterations and deck types

Conclusion

- Policy 1 & 5 are consistently better than the other policies across both decks, while policy 4 is the consistent worse. Overall policy 5 seems to be slightly more effective than policy 1
- Policy 4 seems to be the least effective across both deck types

Game of Life

Motivation

Observe how complex behavior can emerge from simple rules using cellular automata.

Problem Statement

Implement the game in Python and answer the following:

- What are the rules followed?
- What are some interesting properties?
- What are some interesting objects?

Related Work

Eugene M. Izhikevich et al. (2015) Game of Life. Scholarpedia, 10(6):1816.

Play John Conway's Game of Life. Play John Conway's. (n.d.). Retrieved April 6, 2023, from https://playgameoflife.com/

Rules

Rules for cells that are alive:

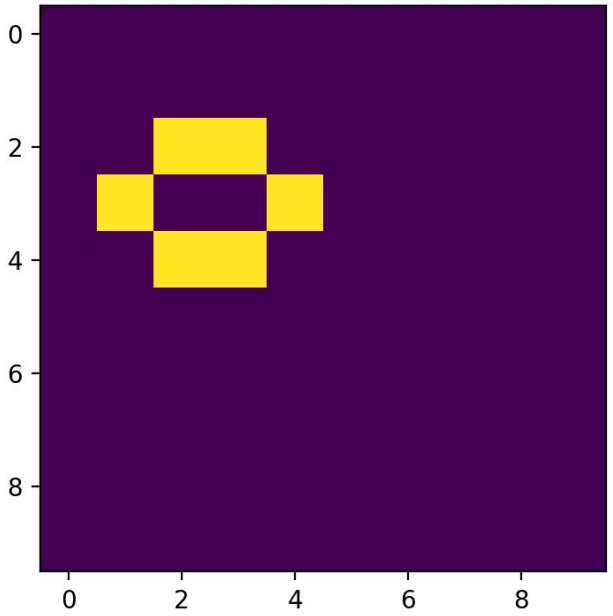
1. Each cell that has 0 or 1 neighboring cells Dies.
2. Each cell with 4+ neighboring cells Dies.
3. Each cell with 2 or 3 neighboring cells Survives.

Rule for dead cells:

4. If the cell is empty, then Each cell with only 3 neighboring cells becomes populated.

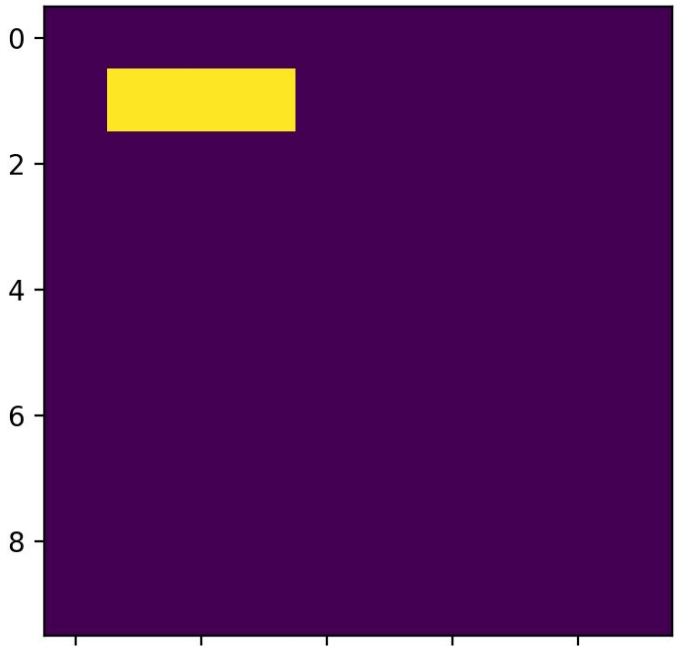
Interesting objects (Still Life)

This is an object which no changes will occur throughout following iterations.



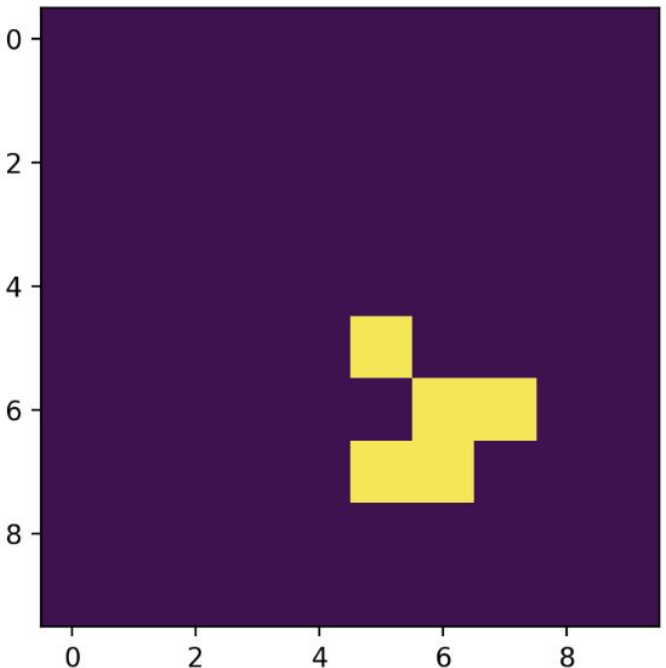
Interesting objects (Oscillator)

This object category consists of objects that are confined between certain boundaries and repeat themselves after a certain number of iterations.



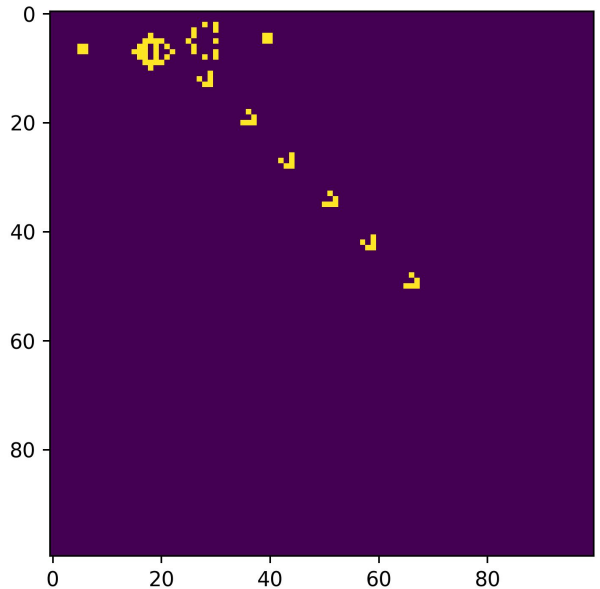
Interesting objects (Traveler)

Objects belonging to the traveler category are ones that will traverse the evolution space without boundaries (indefinitely). Represents a simple example of emergence.



Interesting objects (Glider Gun)

This object creates glider objects from the same initial state and thus each glider created travels in the same pattern and direction. The importance of this object lies in its ability to endlessly create gliders in a stable way.



Interesting properties

Zero player game

- The game is deterministic based off of the initial state.

Undecidable

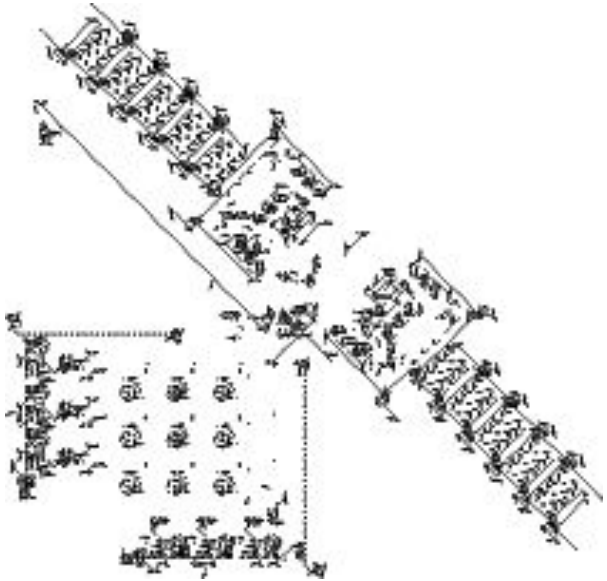
- There is not an algorithm that exists that can tell if a specific pattern will ever appear in later iterations with a given initial state and a specific pattern to look for.

Turing Complete

- Any function that can be computed by a turing machine can be computed by the Game of Life.

Turing Complete

. If multiple glider guns along with other objects are placed in the initial state in specific ways, complex systems can be formed and thus making Conway’s Game of Life Turing Complete.



https://conwaylife.com/w/images/6/63/Turingmachine.png

Approach

- Create a matrix to represent the evolution space.
- Create a function to validate all 4 rules.
- Create a function to handle a surrounded condition (8 cells surrounding the current cell) all 8 boundary conditions (current cell is not surrounded by 8 cells.
- Hardcode patterns to be plotted on evolution space.
- Use the command line to pass arguments that can select between plotting different initial patterns and to determine matrix size.

Python implementation (Evolution space)

- Matrix with boundary conditions:

```
def neighbors(map, row, col):
    sum = 0
    if col == 0 and (row!= 0 and row != size - 1):
        sum = map[row+1][col] + map[row-1][col] + map[row-1][col+1] + map[row+1][col+1] + map[row][col+1]
    elif col == 0 and (row == size - 1):
       sum =  map[row-1][col] + map[row-1][col+1] + map[row][col+1]
    elif (col != 0 and col != size-1)and row == size-1 :
        sum =  map[row-1][col-1] + map[row-1][col] + map[row-1][col+1] + map[row][col+1] + map[row][col-1]
    elif col == size - 1 and row == size-1:
        sum =  map[row-1][col-1] + map[row-1][col] + map[row][col-1]
    elif col == size - 1 and ( row!= size - 1 and row!= 0 ):
        sum =  map[row-1][col-1] + map[row-1][col] + map[row][col-1] + map[row+1][col-1] + map[row+1][col]
    elif col == size - 1 and row== 0:
        sum = map[row][col-1] + map[row+1][col-1] + map[row+1][col]
    elif  row== 0 and (col != 0 and col != size - 1):
       sum = map[row][col+1] + map[row][col-1] + map[row+1][col-1] + map[row+1][col] + map[row+1][col+1]
    elif col == 0 and row== 0:
        sum = map[row][col+1] + map[row+1][col] + map[row+1][col+1]
    else:
        sum = map[row-1][col-1] + map[row-1][col] + map[row-1][col+1] + map[row][col+1] + map[row][col-1] + map[row+1][col-1] + map[row+1][col] + map[row+1][col+1]
    return sum
```

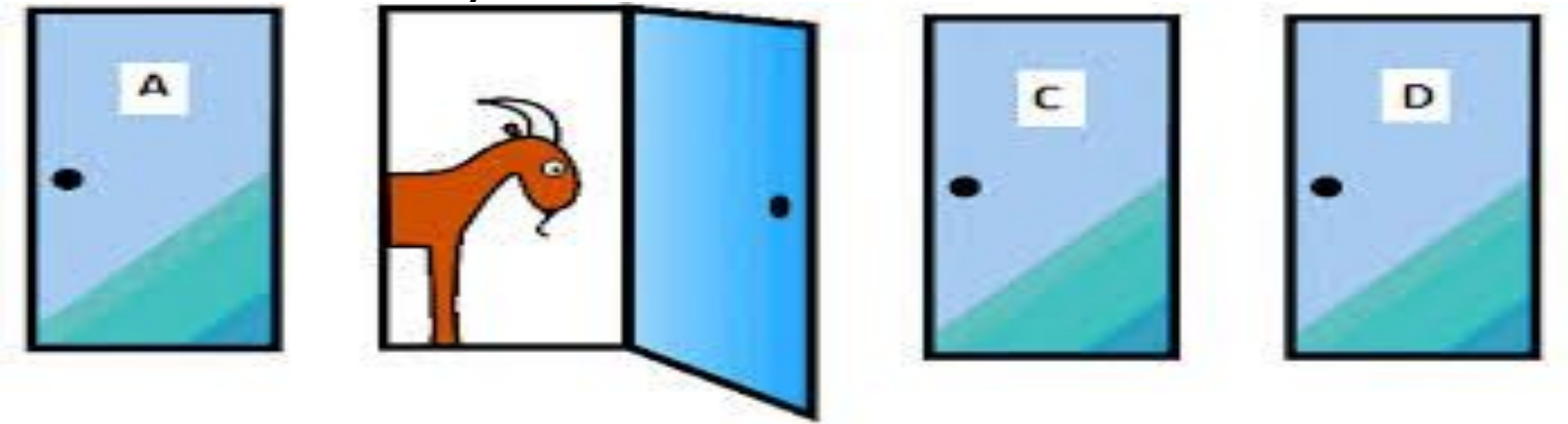
Python implementation (Rules)

- For loop to check the set of rules for each cell:

```
def update(frame, img, map):
    tmp = map.copy()
    for row in range(0, size - 1):
            for col in range(0, size - 1):
            #put neigth
                if (neighbors(map, row, col) == 1):
                    tmp[row][col] = 0
                elif neighbors(map, row, col) >= 4:
                    tmp[row][col] = 0
                elif (neighbors(map, row, col) == 2 or neighbors(map, row, col) == 3) and map [row][col] == 1:
                    tmp[row][col] = 1
                elif neighbors(map, row, col) == 3 and map[row][col] == 0:
                    tmp[row][col] = 1
                else:
                    tmp[row][col] = 0
    img.set_data(tmp)
    map[:] = tmp[:]
    return img,
```

Monty Hall

Intro to the Monty Hall Problem

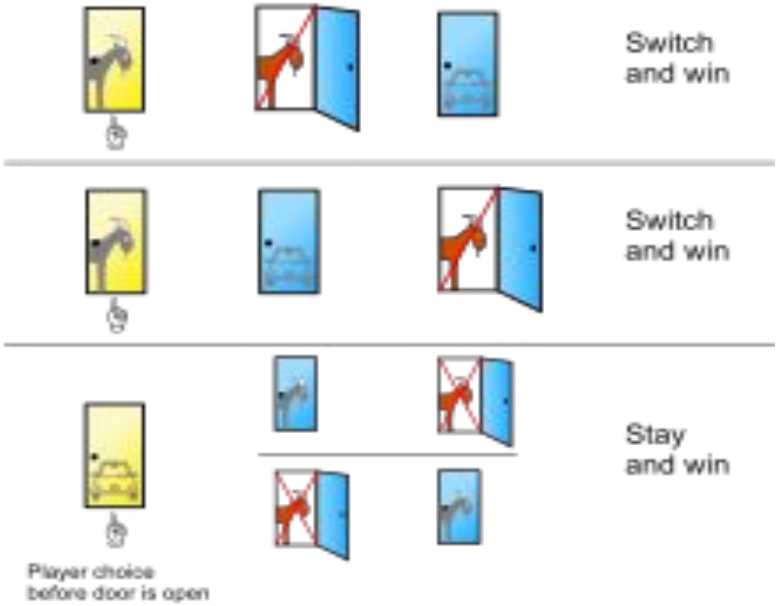


One prize hidden among a number of doors while the rest are bad prizes(goat).

Pick one door next the host opens the rest and offers you a choice of switching or staying. What do you pick? What are your chances?

Motivation

Using the power of simulation to accurately calculate the probability of counterintuitive problems to humans.



Problem Statement

Write a Monte Carlo script to estimate the probability of winning in the classic Monty Hall Problem with three policies contestant randomly decides to switches, never switches, or always switches.

• How the approximation improves with the number of iterations for each case?

• How is the probability of winning for each policy affected by the number of iterations the simulation is run?

Consider a variant where once you have selected one of the doors, the host

slips on a banana peel and accidentally pushes open another door. I.e random chance of losing at the start of game

Because the host can accidentally open the winning door, ending the game.

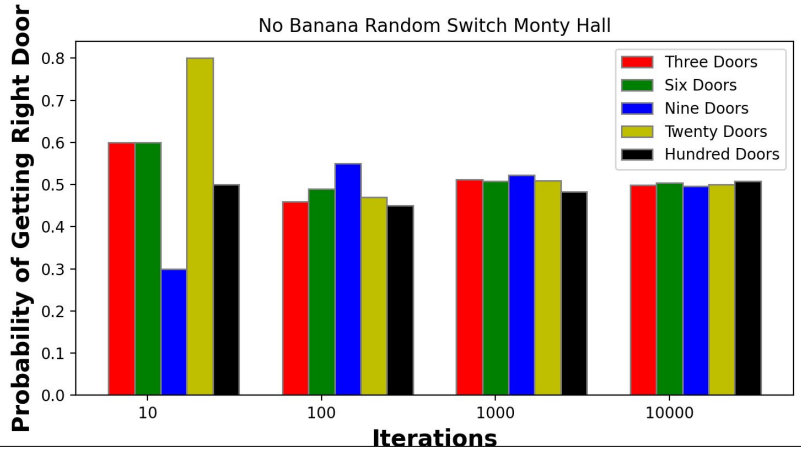
• What are the new probabilities of winning for the policies and number of doors considered?

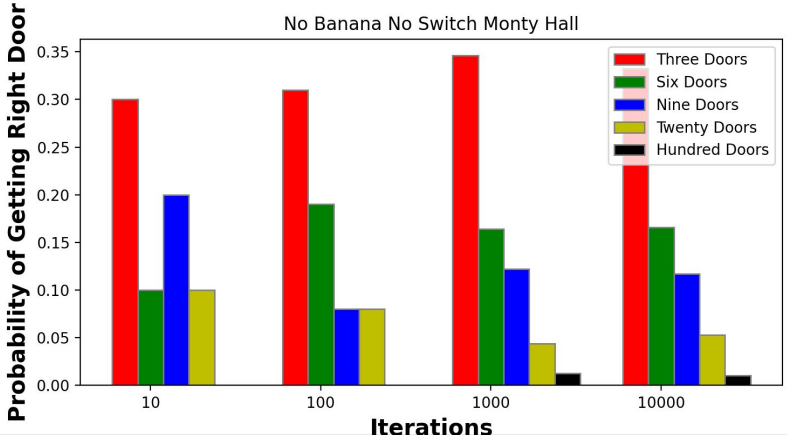
• How the probabilities change as you move to more doors?

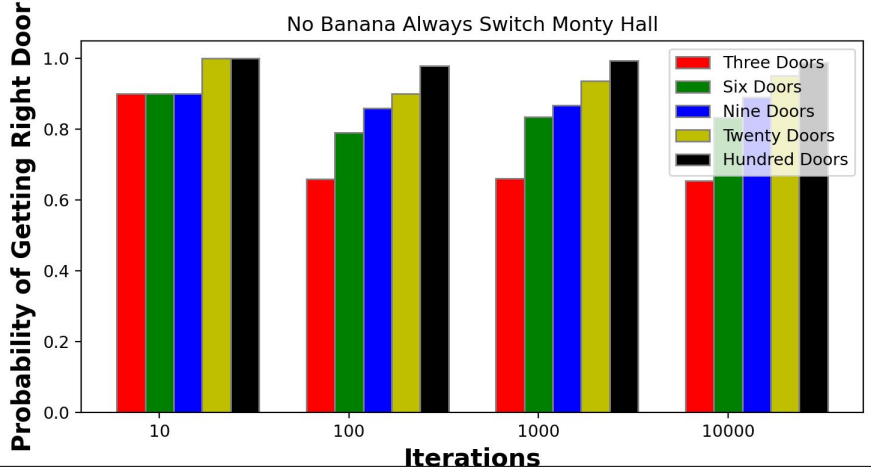
Approach

- Based on players selection of 2 game variations and 3 player policies, modify Monty Hall rules for this run of simulation.
- Create array to represent empty doors.
- Choose winning door randomly in array of doors.
- Simulate a player choice by randomly choosing a door in the array of doors including winning door.
- Open all doors except winning and players chosen door, if they are the same, leave one empty door closed.
- Determine winner based off monty hall rules.
- Collect results run and then over 10, 100, 1000, 10000 iterations
- Plot data as a graph of iteration vs probability of winning.

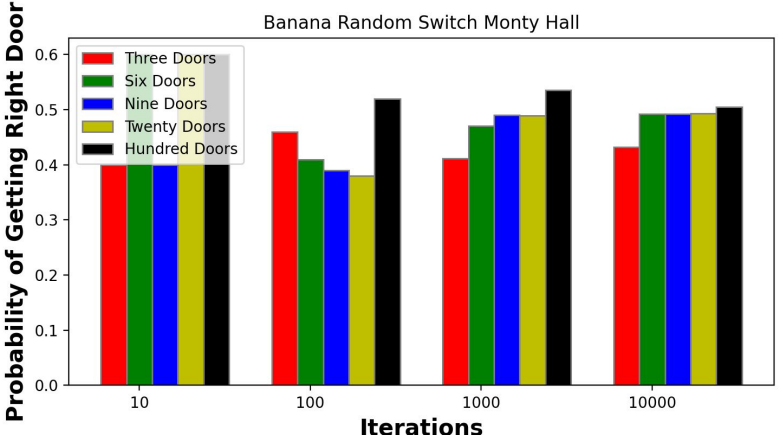
Results: (Normal Variant)

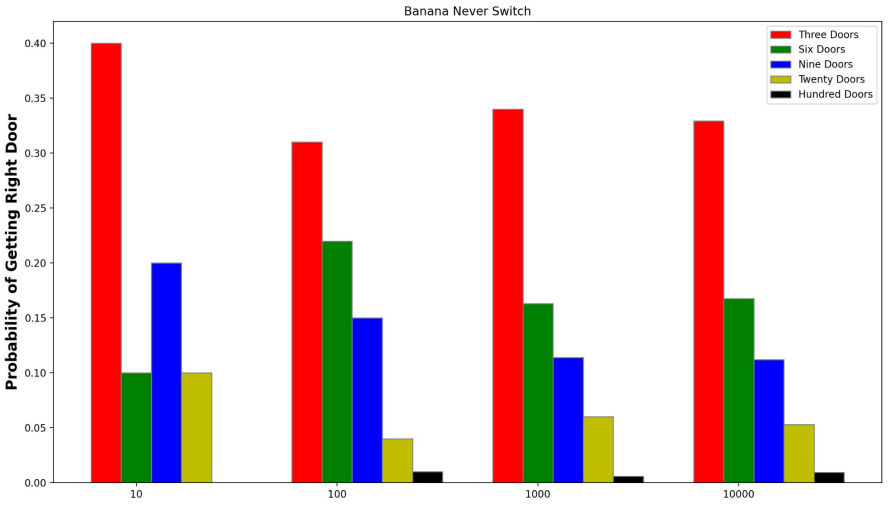


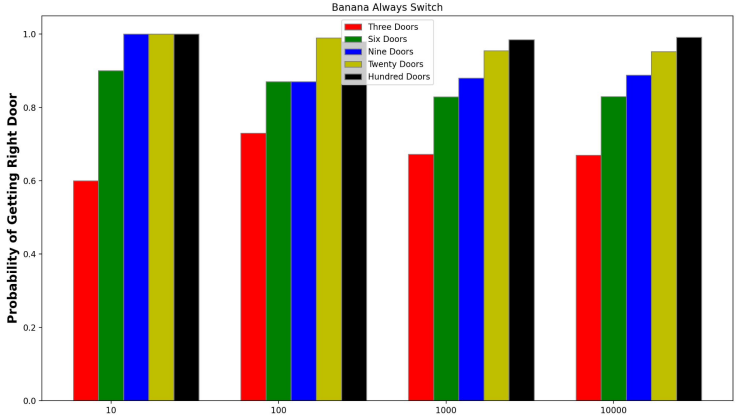




Results: (Banana Variant)







Conclusions

Our findings conclude a couple of things:

1. As we increase the number of iterations we get closer to the theoretical probability of getting the right door for each policy
2. Always switching leads to a much higher probability of getting the right door. This is compounded by the number of doors available