Vikas Natesh

Game Project

I implemented the Paul/Carole game in python. Given n alternating Paul/Carole levels in the game and the values for all 2^n terminal nodes, I back-calculated the values for all $2^n - 1$ previous nodes by performing depth-first search across the game's DAG. Since the player faces a binary 0/1 choice at each game level, the DAG is actually a binary tree and DFS corresponds to in-order traversal on the tree. This binary tree structure is stored as an array and in-order traversal relies on the same indexing scheme used in binary heaps ($parent(i) = i/2$, $left(i) = 2i$, $right(i) = 2i + 1$). Running tests for large n, such as $n = 27$, lead to large memory ($\approx 2GB$) and processing requirements for the array. In order to run many game trials for large n in a reasonable amount of time, the code was parallelized across 4 cores. I ran several simulations to estimate final payout (VALUE(e)), its distribution, and its dependence on the even/odd parity of n. The results are discussed below:

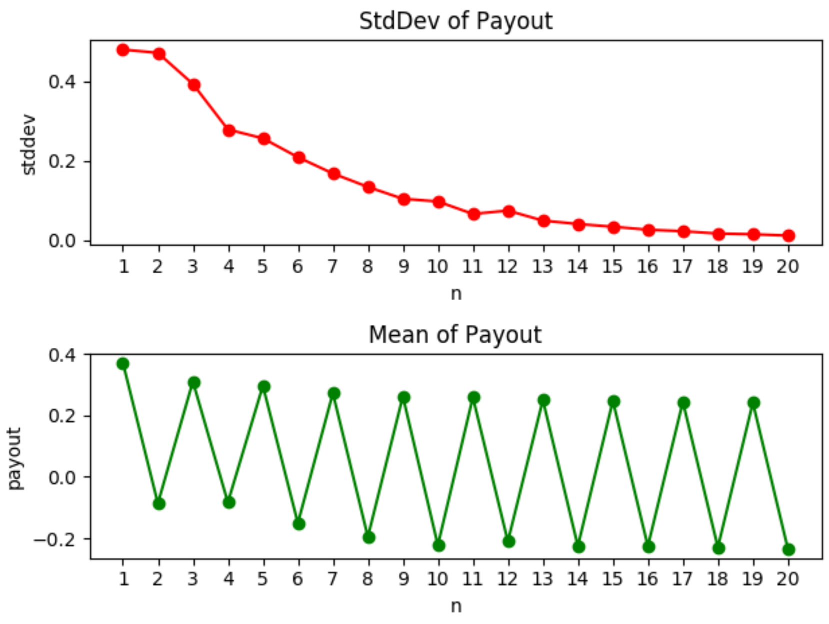

Figure 1

The mean payout and standard deviation over 100 trials for $n = 1, 2, ..., 20$ is shown above. Interestingly, the payout alternates b/w positive and negative values for odd and even numbered n, respectively (Figure 1). Specifically, when Paul is the first to move, he wins a positive amount only when the number of turns (n) is an odd number. This is because on odd n, Paul gets the last move, allowing him to pick the larger of the two payouts at the last turn (leaves of the tree). Likewise, when Paul moves first on even n, the payout is negative i.e. Carole is the winner since she gets the last move and picks the smaller of the two payouts. At the leaves of the tree, any pair of leaves has 4 sign pairs namely, +/+, -/-, +/-, and -/+. Each pair has an equal 1/4 probability of occurring since the 2^n leaf payouts are chosen from a uniform distribution [-1,1] and the sampled payout has an equal

1/2 probability of being positive or negative. If n is odd, Paul get's the last turn and will pick the larger number, which will be positive 3/4 of the time (+/+, +/-, -/+). If n is even, Carole gets the last turn and will pick the smaller number, which will be negative 3/4 of the time (-/-, +/-, -/+). On average over many trials, the 3/4 probability dominates and Paul will win a positive payout on odd n. Likewise, Carole wins with a negative payout on even n.

When Carole is the first to move, the same logic from the 'Paul first' case can be applied, except the results are flipped i.e. the payout is negative (win) for odd n (since Carole gets the last turn) and positive (loss) for even n (since Paul gets the last turn), on average. This shows that there is no advantage to playing first, since the first-moving player loses whenever n is even. However, there is an advantage to playing last, since the player that ends up with the last turn always wins, regardless of n's parity.

As n grows, the variance in payout decreases, appearing to converge to 0 (Figure 1). In addition, the coefficient of variation (stdev relative to the mean for particular n) decreases as n increases (Figure 2). This suggests that the alternating payouts reach a steady state with large n i.e. as more game levels are played. To estimate payout behavior as $n \to \infty$, the absolute value of the mean payout over 100 trials is plotted for $n = 1, 2, ...27$ (Figure 3). Payout appears to reach a steady state of ≈ 0.237.

In Figure 4, thousands of trials were performed for $n = 12, 15, 18$ and the distribution of payouts was plotted. Naturally, the distribution is approximately normal by the central limit theorem, since the payout for each trial is an i.i.d random variable.

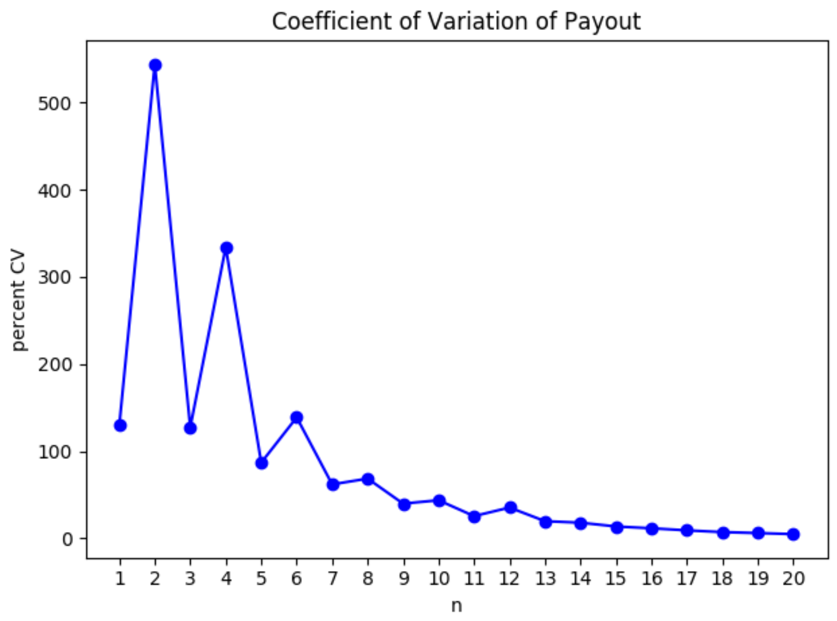

Figure 2

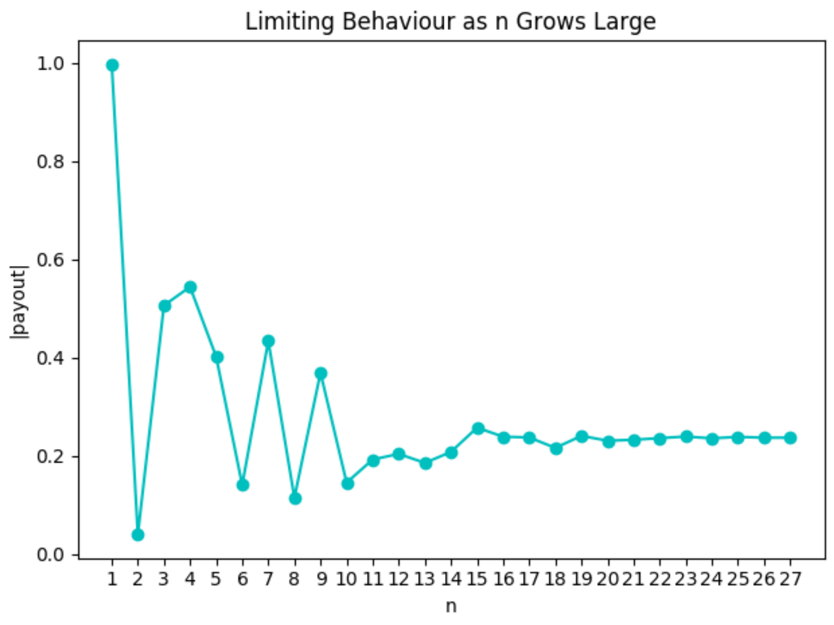

Figure 3

The convergence of average $| payout |$ to 0.237 is also apparent here, as the payout distributions get narrower (variance decreases) as n increases from 12 to 15 to 18.

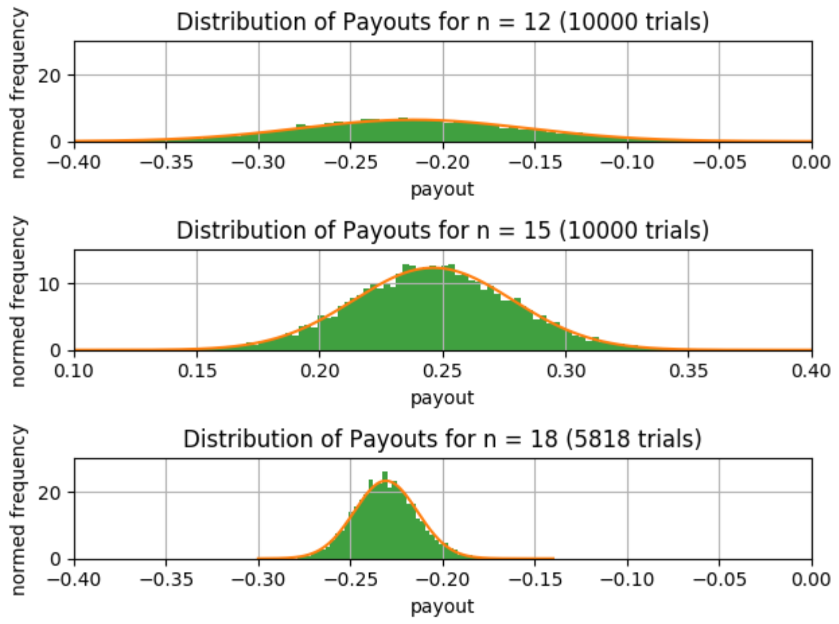

Figure 4

We can confirm the sign-pair probabilities stated earlier by running a simple script to average the pair counts over many trials, for a given n value. For example, running the function 'pairProbs' on a game of $n = 13$ levels over 1000 trials returns the expected $\approx 1/4$ probability for each possible pair of leaves.

```
>>> pairProbs(13,1000)
(+,+) 0.250229492188
(-,-) 0.250345214844
(+,-) 0.249734375
(-,+) 0.249690917969
```

Consider the game where $n = 5$ levels and Carole moves first. This means there are $2^n = 32$ leaves in the game DAG. Since n is odd, Carole also moves last and will realize a value at level 5, as a result of choosing the minimum value leaf among each of the 16 pairs. This level contains $2^{n-1} = 16$ nodes, of which $3/4 * 16 = 12$ have negative value and $1/4 * 16 = 4$ have positive value. Because the majority of values at this level are negative, it was stated earlier that this gave rise to the observed negative average final payout when Carole moves first on odd n. However, even though we were able to partition the space of leaf pairs into 4 sign-pair groups, we must propagate this analysis for the $n - 1$ earlier levels in order to find an analytical solution to the final payout VALUE(e). Notably, the value of nodes at the next level (4) is dependent on the sign ordering of values (chosen by Carole) in the level 5 nodes. There are 1820 possible sign orderings of the 12 negative and 4 positive level 5 values, as defined by the multinomial coefficient $\binom{n}{n_1,n_2} = \binom{16}{12,4} = \frac{16!}{12!*4!} = 1820$. At level 4, it is Paul's move and he chooses the maximum value out of the each of the 8 pairs (16 total nodes) available. Paul may 'get lucky' and be faced with a best-case set of pairs such as:

$$(+,-),(+,-),(+,-),(+,-),(-,-),(-,-),(-,-),(-,-)$$

In this situation, all 4 positive value nodes get paired with a negative value node, allowing all positive values to 'survive' and propagate to the next level 4 i.e. in level 4, there will be 4 positive value nodes and 4 negative value nodes as a result of Paul's max choice. On the other hand, Paul may be faced with an 'unlucky' worst-case set of pairs such as:

$$(+,+),(+,+),(-,-),(-,-),(-,-),(-,-),(-,-),(-,-)$$

In this situation, the 4 positive values only get paired with each other, allowing only half of them to 'survive' and propagate to the next level 4 i.e. in level 4, there will be 2 positive value nodes and 6 negative value nodes as a result of Paul's max choice. However, we could assume an 'average ordering' that allows 3/4 of the positive values to propagate to level 4, a compromise between all and half the positive values propagating. This would cause level 4 to have 3/8 positive and 5/8 negative value nodes.

This analysis quickly becomes more complicated as n gets larger. Although not shown here, we could try computing averages for the node values themselves (as opposed to just the sign), based on the mean and variance of the uniform distribution the leaf values are selected from. We could then conduct the analysis above and propagate these average values upward to the $n - 1$ levels by accounting for the different conditional outcomes based on sign pair. This may allow us to confirm the steady state final $\mid payout \mid \approx 0.237$. Overall, it appears that confirming this steady state via a closed-form expression is fundamentally difficult due to the discrete nature of the max/min choice Paul/Carole make at each level.

Simulation Code

```
import numpy as np
import random
import sys
import time
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
from math import *
from multiprocessing import Pool
from scipy import stats
from scipy.stats import norm
from functools import partial
from joblib import Parallel, delayed

#
def inorder(i, values, first):

  '''

  Performs in-order transversal of a binary tree, recursively
  filling in node payouts. Input root index i = 1.
  -2 used as placeholder for nodes that don't have
  all children with assigned payouts yet

  '''

  if values[i] == -2:
    left = inorder(2*i, values, first)
    right = inorder((2*i) + 1, values, first)
  else:
    return values[i]
  # paul and carole's strategy
  if first == 'Paul':
    if floor(log(i,2)) % 2:
      values[i] = min(left,right)
    else:
      values[i] = max(left,right)
  else:
    if floor(log(i,2)) % 2:
      values[i] = max(left,right)
    else:
      values[i] = min(left,right)
  return values[i]


def playGame(n):
  first = 'Paul'
  values = np.concatenate((np.array([-2 for i in xrange(1,((2**n)+1))]), \
  np.array([random.uniform(-1,1) for i in xrange(((2**n)+1),(2**(n+1))+1)])))
  return inorder(1, values, first)


def runSim(n, trials):
  return [playGame(n) for i in xrange(trials)]


def parallelSim(numTrials):
    p = Pool(processes = 4)
    prod_x = partial(runSim, trials = numTrials)
    payouts = p.map(prod_x, range(22))
    return payouts


def histData(n, numTrials):
  return Parallel(n_jobs=4)(delayed(playGame)(n) for i in range(numTrials))


def plotSim():
  payouts = parallelSim(100)
  std = [np.std(i) for i in payouts[1:]]
  mean = [np.mean(i) for i in payouts[1:]]
  cv = (np.array(std) / abs(np.array(mean)))*100
  #
  plt.figure(1)
  plt.subplot(211)
  plt.plot(range(1,22), std, '-ro')
  plt.title("StdDev of Payout")
  plt.xlabel("n")
  plt.ylabel("stddev")
  plt.tight_layout()
  plt.xticks(range(1,22))
  #
  plt.subplot(212)
  plt.plot(range(1,22), mean, '-go')
  plt.title("Mean of Payout")
  plt.xlabel("n")
  plt.ylabel("payout")
  plt.tight_layout()
  plt.xticks(range(1,22))
  plt.show()
  #
  plt.figure(1)
  plt.plot(range(1,22), cv, '-bo')
  plt.title("Coefficient of Variation of Payout")
  plt.xlabel("n")
  plt.ylabel("percent CV")
  plt.tight_layout()
  plt.xticks(range(1,22))
  plt.show()


def plotHist():
  vals12 = histData(12,10000)
  vals15 = histData(15,10000)
  vals18 = histData(18,5818)
  #
  plt.figure(1)
  plt.subplot(311)
  plt.hist(vals12, normed = True, bins = 100, facecolor='g', alpha=0.75)
  xt = plt.xticks()[0]
  xmin, xmax = min(xt), max(xt)
  lnspc = np.linspace(xmin, xmax, len(vals12))
  # fit normal distr
  m, s = stats.norm.fit(vals12) # get mean and standard deviation
  pdf_g = stats.norm.pdf(lnspc, m, s) # now get theoretical values in our
    interval
  plt.plot(lnspc, pdf_g, label="Norm")
  plt.title("Distribution of Payouts for n = 12 (10000 trials)")
  plt.xlabel("payout")
  plt.ylabel("normed frequency")
  plt.tight_layout()
  plt.axis([-0.4,0,0,30])
  plt.grid([True])
  #
  #
  plt.subplot(312)
  plt.hist(vals15, normed = True, bins = 100, facecolor='g', alpha=0.75)
  xt = plt.xticks()[0]
  xmin, xmax = min(xt), max(xt)
  lnspc = np.linspace(xmin, xmax, len(vals15))
  # fit normal distr
  m, s = stats.norm.fit(vals15) # get mean and standard deviation
  pdf_g = stats.norm.pdf(lnspc, m, s) # now get theoretical values in our
    interval
  plt.plot(lnspc, pdf_g, label="Norm")
  plt.title("Distribution of Payouts for n = 15 (10000 trials)")
  plt.xlabel("payout")
  plt.ylabel("normed frequency")
  plt.tight_layout()
  plt.axis([0.1,0.4,0,15])
  plt.grid([True])
  #
  #
  plt.subplot(313)
  plt.hist(vals18, normed = True, bins = 50, facecolor='g', alpha=0.75)
  xt = plt.xticks()[0]
  xmin, xmax = min(xt), max(xt)
  lnspc = np.linspace(xmin, xmax, len(vals18))
  # fit normal distr
  m, s = stats.norm.fit(vals18) # get mean and standard deviation
  pdf_g = stats.norm.pdf(lnspc, m, s) # now get theoretical values in our
    interval
  plt.plot(lnspc, pdf_g, label="Norm")
  plt.title("Distribution of Payouts for n = 18 (5818 trials)")
  plt.xlabel("payout")
  plt.ylabel("normed frequency")
  plt.tight_layout()
  plt.axis([-0.4,0,0,30])
  plt.grid([True])
  plt.show()

```

```
# Function to confirm expected sign-pair probabilities
def pairProbs(n, trials):
  pp=[]
  nn=[]
  pn=[]
  nps=[]
  for t in xrange(trials):
    b = np.array([random.uniform(-1,1) for i in xrange(((2**n)+1),(2**(n+1))
   +1)])
    pospos=0
    negneg=0
    posneg=0
    negpos=0
    for i in xrange(0,len(b),2):
      if b[i] > 0 and b[i+1] > 0:
        pospos+=1
      elif b[i] < 0 and b[i+1] < 0:
        negneg+=1
      elif b[i] > 0 and b[i+1] < 0:
        posneg+=1
      elif b[i] < 0 and b[i+1] > 0:
        negpos+=1
    pp.append(pospos)
    nn.append(negneg)
    pn.append(posneg)
    nps.append(negpos)
  print "(+,+) " + str(np.mean(np.array(pp)) / (2**(n-1))) + '\n' \
  + "(-,-) " + str(np.mean(np.array(nn)) / (2**(n-1))) + '\n' \
  + "(+,-) " + str(np.mean(np.array(pn)) / (2**(n-1))) + '\n' \
  + "(-,+) " + str(np.mean(np.array(nps)) / (2**(n-1)))
```