Imperial College
London

COURSEWORK 1

IMPERIAL COLLEGE LONDON

DEPARTMENT OF MATHEMATICS

M3A29 - Theory of Complex Systems

Author:
Tudor Trita Trita (CID: 01199397)

Date: February 19, 2019

1 Write Up

1.1 Part A

We can define the branching probability P_k as the probability that k number of active sites are spawned from the original active site. In the case of the paper on p.4071, the branching probabilities are:

$$\begin{aligned} P_0 &= 1 - p, \\ P_2 &= p, \\ P_k &= 0, \textit{ for all other } k. \end{aligned}$$

1.2 Part B

Using the generation function in the course lecture notes and substituting in our branching probabilities calculated in Part A:

$$\begin{aligned} g_x(s) &= \sum_{n=0}^{\infty} P_x(n)s^n, \\ &= P_0 + P_2 \cdot s^2, \\ &= (1-p) + p \cdot s^2, \\ &= 1 + p(s^2 - 1). \end{aligned}$$

The average branching ratio (found in the lecture notes) from the generator after substituting our values of P_k is:

$$\begin{aligned} \mu &= \sum_k k p_k \\ &= 2p \end{aligned}$$

The process is critical when the mean $\mu = 1$, as found in the lecture notes, and this occurs when $p = p_c = 1/2$. This agrees with the paper attached, as it explicitly says that $p_C = 1/2$ for this process.

1.3 Part C

From the lecture notes, we know that for $\mu < 1 : P(\tau > n) \propto \mu^n$ for large n.

If we set $n_0 = \frac{-1}{\ln \mu}$:

$$
\begin{aligned}
n_0 &= \frac{-1}{\ln \mu}, \\
\Rightarrow \mu &= \exp\left(\frac{-1}{n_0}\right), \\
\Rightarrow \mu^n &= \exp\left(-\frac{n}{n_0}\right), \\
\Rightarrow P(\tau > n) \propto \mu^n &= \exp\left(-\frac{n}{n_0}\right),
\end{aligned}
$$

as required. The variable $n_0 > 0$ as $\mu < 0$, thus the condition in the coursework is satisfied.

The case $\mu = 1$ gives the probability in non-exponential form, and the case $\mu > 1$ involves a non-zero probability of infinite avalanches, so are not considered in this part.

1.4 Part D

Let the average branching ration be of the form $\mu = 1 - \Delta$.

We know from Part C that $n_0 = \frac{-1}{\ln \mu}$.

Therefore, by using a Taylor expansion:

$$
\begin{aligned}
n_0 &= \frac{-1}{\ln \mu}, \\
&= \frac{-1}{\ln (1 - \Delta)}, \\
&= \frac{-1}{-\Delta - O(\Delta^2)}, \\
&= \frac{1}{\Delta + O(\Delta^2)}.
\end{aligned}
$$

As $\Delta \to 0^+$, we see that terms of higher order $O(\Delta^2) \to 0^+$ fast, and so:

$$
n_0 \propto \frac{1}{\Delta} = \Delta^{-1}.
$$

By inspection, we can determine that $a = 1$. We also see that $n_0 \to \infty$ algebraically as $\Delta \to 0^+$, which is what we want.

1.5 Part E

Eq. (1) in the attached paper can be seen as a reasonably effective relationship as it takes into account the number of active sites at generation n, and it gives a good description of the global dynamics of the SOBP. From the paper, we know that if $\sigma_n = 0$, then the probability increases, due to the increase of energy in the system without an output. However, if $\sigma_n > 0$, then p decreases as energy is in fact leaving the system. This is accounted for in the equation, due to the sigma term, and it is normalised by diving by N.

If we assume that each boundary site lost two units of energy instead of one, Eq. (3) in the attached paper is modified to:

$$\frac{dp}{dt} = \frac{1 - 2(2p)^n}{N}.$$

To find the fixed point this time, we set this equation equal to 0 and solve:

$$\begin{aligned} \frac{1 - 2(2p)^n}{N} &= 0, \\ \Rightarrow 1 - 2(2p)^n &= 0, \\ \Rightarrow (2p)^n &= \frac{1}{2}, \\ \Rightarrow p &= \frac{1}{2(2^{\frac{1}{n}})}. \end{aligned}$$

Therefore, the fixed point value, which corresponds to the critical probability is

$$p_c = \frac{1}{2(2^{\frac{1}{n}})}.$$

If we let $n \to \infty$, then $2^{\frac{1}{n}} \to 1$, and then the value of $p_c = 1/2$.

1.6 Part F

Note: The code I have is written in Python 3.7. and imports numpy and matplotlib.pyplot as:

```
import numpy as np
import matplotlib.pyplot as plt
```

To generate the branching process I have used the following function, which returns the number of active sites at generation n:

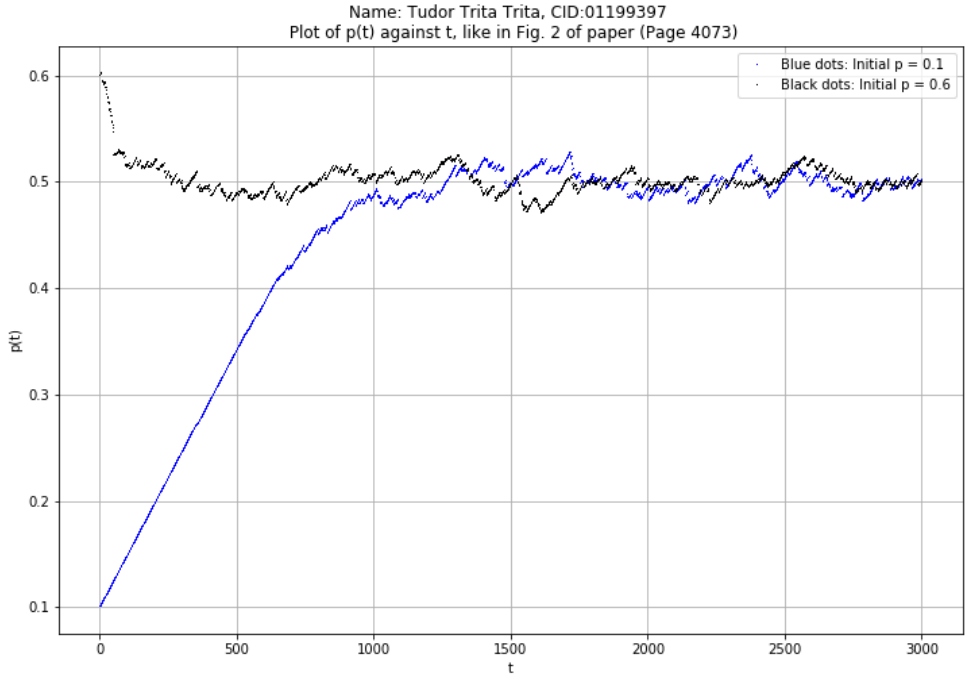
```
def tree(prob, n):
    """Function to simulate active sites.
    Input: Probability (p), Number of generations (n).
    Returns: Number of active sites at generation n (active).
    Variable active is the same as sigma in the coursework notes.
    """

    active = 1  # We start with just 1 active site at the beginning

    i = 1
    while active > 0 and i < n:  # Main loop for each generation

        # Generating branching outcomes from probabilities:
        prob_array = np.random.binomial(1, prob, active)

        # Counting number of active sites at next generation:
        active = 2*sum(prob_array)
        i += 1

    return active
```

The code used for generating curves similar to Fig.(2) in the attached paper is here:

```
# Code for part f

#Initial parameters
n = 10
N = 2**(n+1) - 1
t = 3000
times = range(t)
parray1 = [0.1] # Starting probability 1
parray2 = [0.6] # Starting Probability 2

# Main loop for generating probabilities at each timestep
for i in range(t-1):
  sigma1 = tree(parray1[i], n)
  sigma2 = tree(parray2[i], n)

  # Using formulas in page 4072 of paper:
  flt1 = parray1[i] + (1 - sigma1)/N
  flt2 = parray2[i] + (1 - sigma2)/N

```

```
    parray1.append(flt1)
    parray2.append(flt2)

plt.figure(figsize=(12,8))
plt.plot(times, parray1, 'b,',
         times, parray2, 'k,')
plt.xlabel('t')
plt.ylabel('p(t)')
plt.legend(('Blue dots: Initial p = 0.1', 'Black dots: Initial p = 0.6'))
plt.title('Name: Tudor Trita Trita, CID:01199397 \n Plot of p(t) against
    t, like in Fig. 2 of paper (Page 4073)')

plt.grid(True)
plt.show()
```

The previous code produces the following figure:



This figure is similar to Fig. (2) in the paper, as it is simulating the same process and has the same starting probabilities. The plots change every time the code is run, as there is a stochastic element to the process. To get rid of the stochastic element, we can work with the average of sigma, which is given in the paper as $(2p)^n$. I will be exploring this further in the next part.

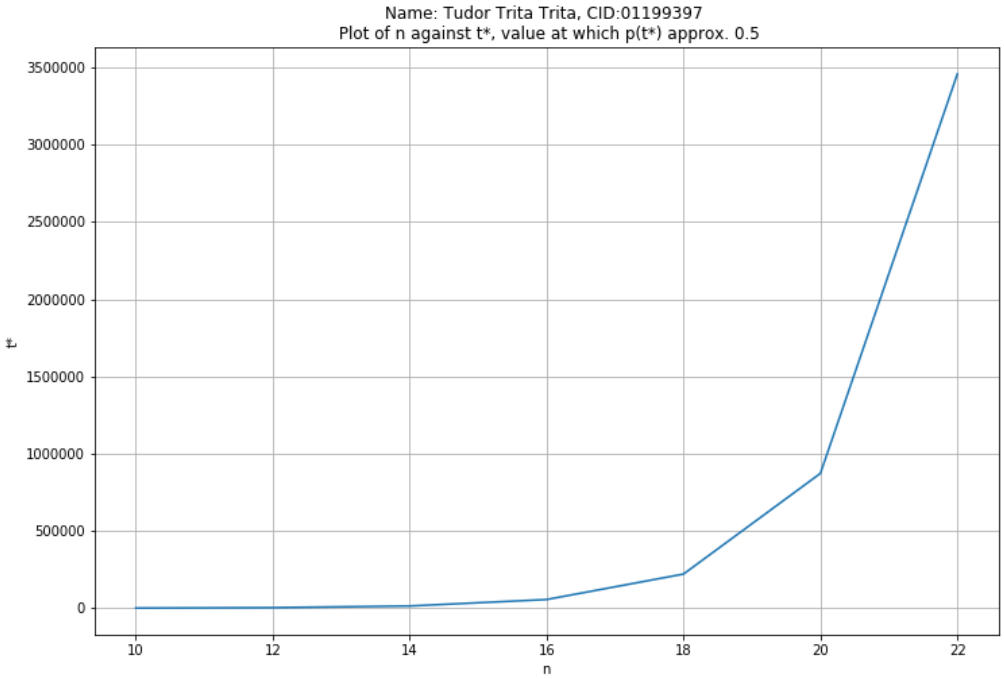
1.7 Part G

The code used in this section is here:

```
# Code for part g

# For this part, I am using the average of sigma.
# We can achieve this by taking sigma = (2*p)**n,
# as shown in the paper that this formula is correct for
# the average of sigma.

#Initial parameters
narray = [10,12,14,16,18,20,22]
t = 10000000
times = []
gradarray = []
# Main loop for generating probabilities at each timestep

for k in range(len(narray)):
  n = narray[k]
  N = 2**(n+1) - 1
  parray = [0.1]  # Starting probability equal to 0.1

  for j in range(t-1):
    sigma1 = (2*parray[j])**n  # Getting average of sigma for each
    probability.

    # Using formulas in page 4072 of paper:
    flt1 = parray[j] + (1 - sigma1)/N

    parray.append(flt1)

    # Checking if current probability has reached 0.5, ie. the critical
    value.
    if abs(parray[j]-0.5)<0.01:
      break

  times.append(j)  # Storing location at which condition above holds.
  gradarray.append((parray[4]-0.1)/5) # Estimating gradient at p(0).



plt.figure(figsize=(12,8))
plt.plot(narray, times)
plt.xlabel('n')
plt.ylabel('t*')
plt.title('Name: Tudor Trita Trita, CID:01199397 \n Plot of n against t*,
    value at which p(t*) approx. 0.5')
plt.grid(True)
plt.show()
```

To demonstrate numerically the time $t*$ at which $p(t)$ reaches the asymptotic value $p(t*) \simeq 1/2$, I used the average of sigma as in the paper i.e. $\sigma_n(p,t) = (2p)^n$, and recorded the steps it took for the probability to be within 0.01 of 1/2 when it started at 0.1.

Following figure is the plot created from previous code:



The figure clearly shows an exponential relationship between n and t*, which is what we are asked to prove.
To figure out the estimate of the slope of $p(t)$ at $t = 0$, I calculate the gradient on line 33 of the code above. These gradients are given in the table below, together with the 'x intercept', which is explained in a moment.

Table 1: Gradients

n	**Est. of grad at p(0)**	x intercept
10	3.91e-04	1023
12	9.76e-05	4098
14	2.44e-05	16393
16	6.10e-06	65573
18	1.52e-06	263157
20	3.81e-07	1033592
22	9.53e-08	4197272

Table 1 has the estimated gradients at p(0) for each value of n. The last column labelled x intercept corresponds to trying to solving the equation $y + mx + c$, with $y = 0.5, c = 0.1, m =$ (grad estimate) and solving for x. These solutions (x) gives us the 'estimated time-step' at which the probability will reach 1/2. I use this linear equation, as the evolution of probability before reaching critically can be approximated by a line. We can see that the x intercept increases exponentially even though n increases linearly, thus supporting the dependence to be exponential.

1.8 Part H

For this part, I created a modified version of the previous tree function that would return the value for the avalanche:

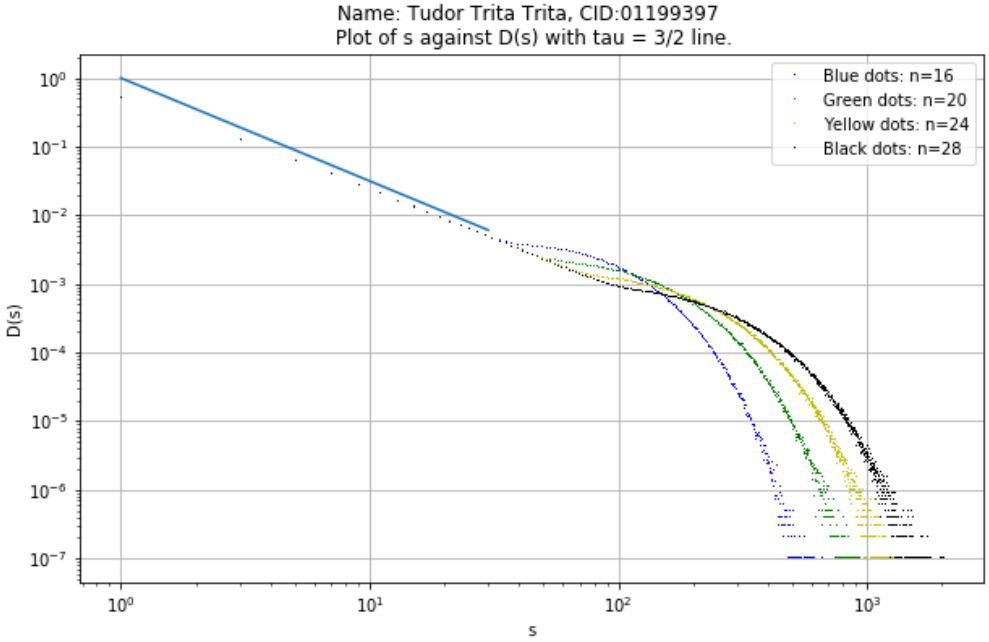
```
def avalanche(prob, n):
    """Function to simulate avalanche.
    Input: Probability (p), Number of generations (n).
    Output: Avalanches: Total number of active sites in the tree (aval).
    """
    active = 1  # We start with just 1 active site at the beginning
    aval = 1 # Avalanche at the start is simply the initial active site.

    i = 1  # Counter
    while active > 0 and i < n:  # Main loop for each generation

        # Generating branching outcomes from probabilities:
        prob_array = np.random.binomial(1, prob, active)

        # Counting number of active sites at next generation:
        active = 2*sum(prob_array)

        # Keeping track of total active sites, which is by def. the
    avalanche.
        aval += active
        i += 1

    return aval
```

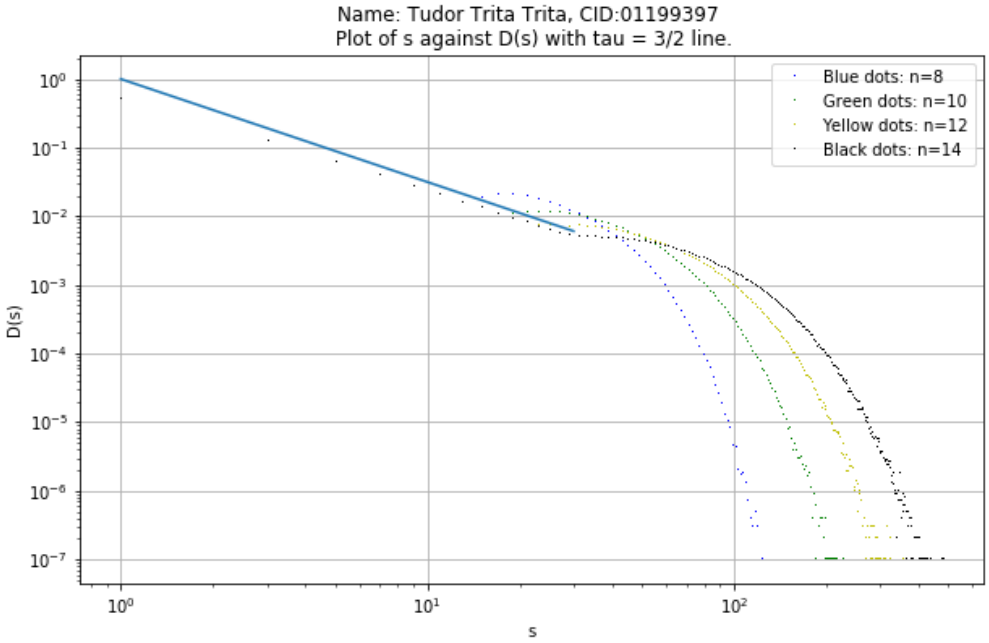
This function works simply by summing the number of active sites at each generation.

The main code for part h which generates the plots is:

```
# Code for part h

# S is number of active sites

# Want to simulate process and see the distribution of active sites.

narray = [8,10,12,14]
p = 0.5  # Probability fixed at 0.5
t = 10000000  # Max no. of repetions
colours = ['b,', 'g,', 'y,', 'k,']  # Colours for plot later
plt.figure(figsize=(10,6))

for i in range(len(narray)):
  dict = {}

  for j in range(t):

    avl = avalanche(p, narray[i])

    # To see the distribution of the avalanches, I store them in a
    dictionary:
    if avl in dict.keys():
      dict[avl] += 1
    else:
      dict[avl] = 1

  vals = []

  for k in dict.keys():
    vals.append(dict[k])

  vals = np.asarray(vals)/sum(vals)

  plt.loglog(dict.keys(), vals, colours[i])

s = np.linspace(1,30,30)
plt.loglog(s, s**(-3/2))


plt.xlabel('s')
plt.ylabel('D(s)')
plt.legend(('Blue dots: n=8', 'Green dots: n=10', 'Yellow dots: n=12', '
    Black dots: n=14'))
plt.title('Name: Tudor Trita Trita, CID:01199397 \n Plot of s against D(s
    ) with tau = 3/2 line.')
plt.grid(True)
plt.savefig('parth2.png')
plt.show()
```

Figures produced by code on previous page are: (parameters n in above code are modified for each of the plots, otherwise the code is the same)





Both of these figures are log-log plots of the avalanche distribution $D(s)$ for different system sizes, similar to Fig(3) in the paper.

A line with slope $\tau = 3/2$ is plotted for reference. We can see that, just like in the paper, for large s, the distributions fall off exponentially. Also, we can see that the distributions fall off exponentially for smaller s the smaller n is.

Explanation of why $\tau = 3/2$ is expected:

The exponent $\tau = 3/2$ is to be expected, as if the SOBP process is run with a starting probability of $1/2$, it behaves like a SOC process. The simplest approach to SOC is mean-field theory, and as the paper attached explains, mean-field exponents for SOC models have been obtained, and it turns out that all the values are the same, i.e. $\tau = 3/2$ for all of the processes discussed so far. Thus, the SOBP process with a starting probability of $1/2$ can be explained by mean-field theory with a mean-field exponent of $3/2$.