Andrew Koulogeorge | HMM

Filtering

Overview of Functionality and Code

See the forward_message() function in the SensorRobot.py file. - In the SensorRobotSolver initilization we create an initial probability distribution for the given maze assuming that each location is equally likley. We consider locations with a wall to be impossible to start at and normalize the initial distribution accordingly. - We also compute the transition matrix T and the observation matricies O_R, O_G, O_B, O_Y corresponding to the different colors that the robot could observe. We compute these and store them in the global variables in the class to prevent on the fly computation and an increase in efficency! - We loop over the total number of steps that the robot is going to take, starting with the state beign equal to the initial distribution. We have the robot take a random step and use its fuzzy sensor to compute a color based on the distribution given in the assignment. Ground truth location, color observed, true color and correctness of the sensor are all cached in global variables for later testing and analysis. - Based on the color observed we select the proper observation matrix from our global cache and apply both the transition and the evidence update with matrix multiplication. We normalize the resulting vector to maintain the distributions integrity. We cache this state since we are going to need it later when we do smoothing! - After we compute the distribution given the evidence, we output information about whats going on with the robot such as robot location, color observed, correctness of sensor, the full probability distribution at this time step, the probability of being correct about where we are in the maze, and what the maze looks like with the robot in it.

Testing and Analysis

The first observation to make about how the probability distribution evolves overtime for a maze with no walls is that, due to the randomness in our sensor, we cannot be sure at any time where we are (assuming that there are more spaces to walk on then colors, but we will get to that later).

Consider the following example from no_walls.text which illistrates how the potential incorrectness of our sensor can be detrimental to the correctness of our posterior. - At the given timestep, the smoothing algorithm has nailed down the location of the robot almost to a tee. In the next step, however, the sensor reads G even though the ground truth color was R. Our liklihood of being in the correct location dropped from near certainty to complete confusion. How is this possible? - It its because this mess up from the sensor was particularly bad. Indeed, there was a 50 chance of being in the bottom right corner and having the true color observed be G. Thus, if you look at the full posterior after the robot moxed away from the bottom right, we see that we still think there is a 95.3% chance of being in the bottom right! Dam you fuzzy sensor!

```
--------------------
Here is our robots location! (3, 0)
Color we observed! G

Liklihood of location we are actually at: 0.9506829954247896

[5.53003953e-05 5.70169519e-04 2.16483733e-02 9.50682995e-01
 1.23395920e-03 5.79835892e-05 1.07809977e-03 2.16483733e-02
 1.73275525e-05 1.37268452e-03 4.72454194e-05 5.69235682e-04
 2.08310666e-06 6.10134852e-06 9.66439429e-04 4.36284034e-05]


RBGY
RGYB
GYBR
YBRx
--------------------
Here is our robots location! (3, 1)
Color we observed! G

Liklihood of location we are actually at: 0.021696231642370275

[4.26522826e-05 4.97461070e-04 2.16962524e-02 9.53019575e-01
 6.68734274e-04 9.47819268e-05 9.66817153e-04 2.16962316e-02
 5.84976692e-05 6.30512963e-05 8.88018867e-05 4.96941061e-04
 6.14705425e-07 5.22883576e-05 5.21146886e-04 3.61522348e-05]


RBGY
RGYB
GYBx
YBRG
```

Another thing to note about the previous example is how much symmetry there is in the colors of the maze. This makes can make it harder for the robot to have confidence about where it is. For example, say that is in location (1,3) on the grid and then it randomly moves and it correctly observes the color G. Well, two of the neighbors of (1,3) have that color so the robot isnt able to give a large probability mass to a single one location. For example, consider the following case where this causes the robot to become less confident of where it is in the maze, even though the sensor was correct!

```
---------------------
-> Here is our robots new location after trying to move! (1, 3)
-> Color we observed! B. This observation is True
-> Here is the updated distro:
[0.013, 0.020, 0.016, 0.031, 0.0173, 0.030, 0.0235, 0.01682,
```

```
0.0400, 0.0075, 0.0226, 0.0233, 0.02078, 0.6994, 0.0092, 0.006]

-> Here is the probability of being in the true location: 0.699441792562702

RxGY
RGYB
GYBR
YBRG


----------------------
-> Here is our robots new location after trying to move! (1, 2)
-> Color we observed! G. This observation is True
-> Here is the updated distro:
[0.001622823630679828, 0.0020265492405675207, 0.0022897873,
0.052750253005078664, 0.05553578335343107, 0.00170643033,
0.002157259656199002, 0.0023614260057727905, 0.00212434119,
0.43272533490594917 (SPLIT 1), 0.0015801596765176995, 0.00171113,
0.019379495370019588, 0.01828826956604677, 0.4026270942633809 (SPLIT 2),
0.001113855149638648]

-> Here is the probability of being in the true location: 0.43272533490594917

RBGY
RxYB
GYBR
YBRG
```

Consider the maze unique_neighbors where each square does not have two neighboring squares of the same color. If we simulate the robot walking around a maze without this symmetry of colors on the floor, we see that much more information is always gained each time the sensor is correct. For example, considder the following case where in just 2 correct sensor moves, the robot is able to gain 70 probability on its most likley state. This is because each correct senor read tells the robot a ton of information about where it is in the maze, since there is a unique color to move to from each location!

```
----------------------
-> Here is our robots new location after trying to move! (3, 1)
-> Color we observed! B. This observation is False
-> Here is the updated distro:
[0.0037, 0.0071, 0.0, 0.0035, 0.0,
0.00366, 0.07868, 0.22441, 0.00351764,
0.0018454, 0.0, 0.66, 0.00180, 0.005047, 0.00043, 0.0]

-> Here is the probability of being in the true location: 0.22441385663258984

RBG#
```

```
GY#G
#RBx
YG#R


---------------------
-> Here is our robots new location after trying to move! (2, 1)
-> Color we observed! B. This observation is True
-> Here is the updated distro:
[0.0014949, 0.0017665, 0.0, 0.019131, 0.0,
0.00743, 0.690203, 0.079176, 0.0008694, 0.00114,
0.0, 0.18091, 0.000990, 0.016350875, 0.0005, 0.0]

-> Here is the probability of being in the true location: 0.6902031633107681

RBG#
GY#G
#RxY
YG#R


---------------------
-> Here is our robots new location after trying to move! (2, 1)
-> Color we observed! B. This observation is True
-> Here is the updated distro:
[0.0001775684312121542, 0.00035400482070859494, 0.0, 0.0038791,
0.0, 0.01989835, 0.9167174, 0.0275355, 0.0001100,
0.00073285, 0.0, 0.0176652, 0.0005453, 0.011875,
0.0005085, 0.0]

-> Here is the probability of being in the true location: 0.9167174869819382

RBG#
GY#G
#RxY
YG#R
```

Smoothing (Extension)

Overview of Functionality and Code

Using Dynamic Programming, we run the forward pass to compute the probability distribution of the robot at each time step based on the evidence at that state. We store each of these distributions for each time step. Then, given the array of colors observed from the sensor, we can compute the backwards message for each time step (also storing these). Once we have stored the forward and backwards message for each time step, the product of these distributions will give us the smoothed distribution for each time step (updated with all the evidence from

the future time steps).

The function backward_message() loops over the evidence in reverse order and recursivley computes the backward message for each time step. In smooth(), we use the forward and backward messages to compute the updated distribution as well as several statistics about how much the smoothing improved our results.

It is intuitive that incorperating future information should make our probability distribution more likley to predict the actual location as the most probable state. But, does it always make it better? On average, does it make the distribution better?

Testing and Analysis

To analyze the benefics of smoothing, I separated out the impact that smoothing has for each time step on the probability of being in the correct state in the maze. For each distribution, I compute the difference between the forward message correct state probability with the smoothened distribution correct state probability. I aggregared all of these points and plotted them on a scatter plot with green points having a positive contribution and red points having a negative contribution. I also computed the average benefit from smoothing and found it to be overal positve, which is what we would hope for! The first plot below is for the no_walls maze and the second is for the unique_neighbors maze. The third plot is for the wizard of oz maze where all of the colors are clustered together. We see that the benefit of smoothing from this maze is much less than the other mazes. This makes complete sense since knowing the color of a state in the wizard of oz maze gives us signifcantly less information about the exact correct location of the robot.

Even though we observed different behaviors between these two mazes in the forward message computation, these two plots share many core themes- on 50k time steps, the average benefit of the smoothing was nearly identical (around 14%) and the distributions of points are very similar.

Future information can reduce the probability of the current correct state if the information from the future is bad output from the sensor. In other words, if the most likley state at the next time step is not compatible with the most likley state currently, then the update step could drop the probability!

Viterbi (Extension)

Overview of Functionality and Code

Given an inital distribution and a sequence of observations for each time step we want to know what the most likley sequence of states was for the robot. Its not as simple as just taking the max of the distribution at each time step since that does not condition on all of the steps that came before it- thats just looking at local information. Instead, we will implement an O(TKˆ2) dynamic

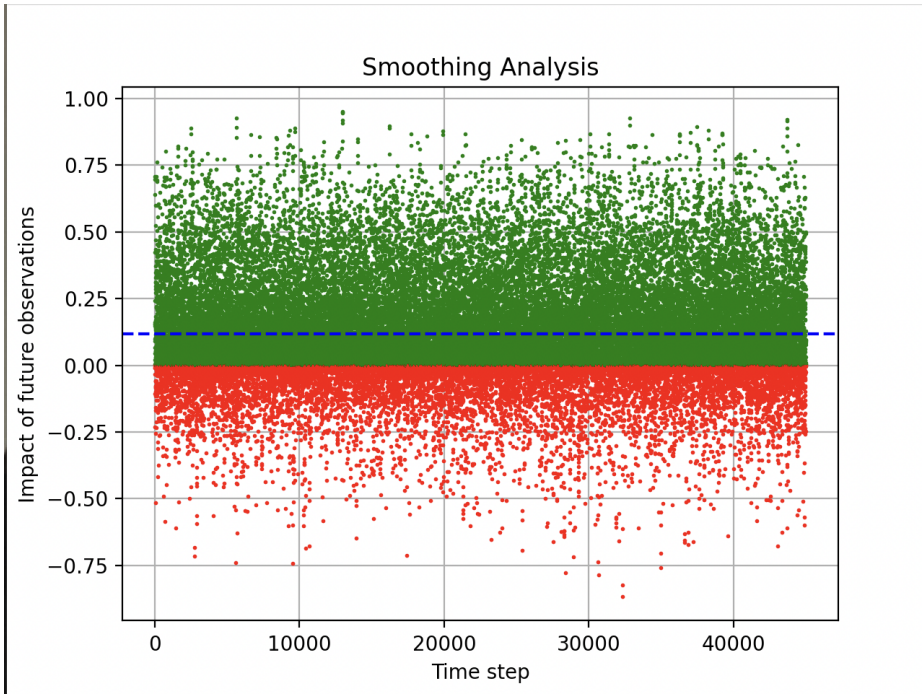

Figure 1: Smoothing.png

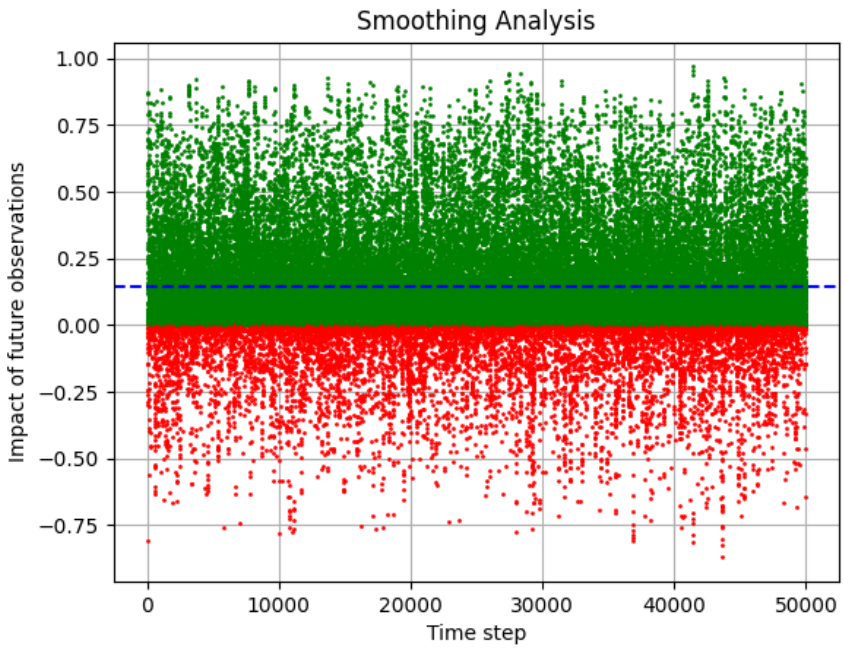

Figure 2: Smoothing.png

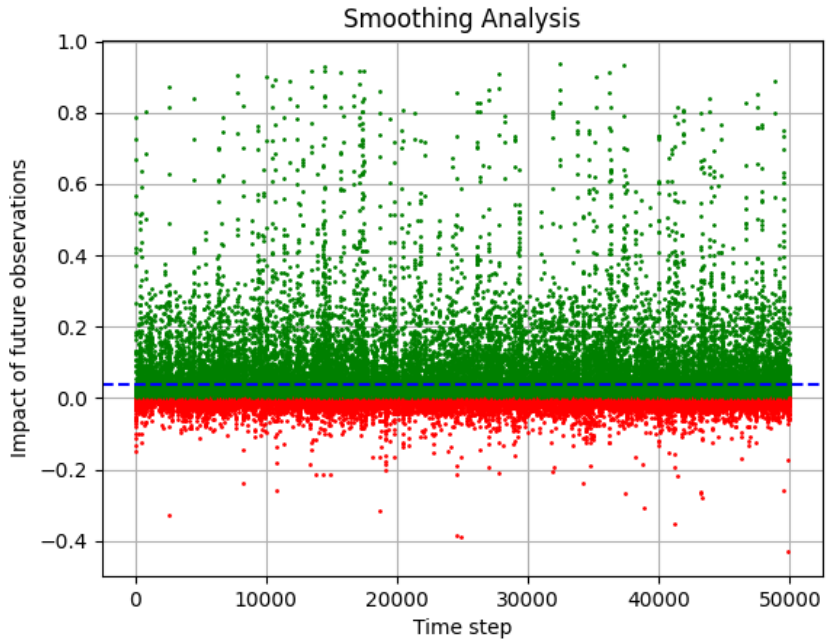

Figure 3: Smoothing.png

programming algorithm which computes the most likley sequence of locations the robot was in.

- We will construct two matrices T_1 and T_2; T_1 will be used to compute the most likley path through the search space of possible states that the robot can be in. T_2 will be used to take the most likley final state and backtrack through each time step and reconstruct the most likley sequence. More formally:
  - $T_1(i, j)$ stores the probability of the most likley path up until the jth time step where the value of the jth state is i
  - $T_2(i, j)$ stores the most likley state at the $j - 1$ time step for the most likley path up until the jth time step
- We construct the emission matrix E which stores a row for each possible value of the state and a column for each color that the sensor could observe. E(i,j) is the probability of seeing color j if we are at state i.
- We loop over each of the time steps and compute the value of T_1(i,j) for each i ranging across each of the possible state values by computing the most likley state which transitioned from time step j-1 to state i in the time current time step j. After looking over all previous states in the j-1 time stepm, we store this maximum probability as well as which state we transition to in the T_1 and T_2 tables .
- Also note that the viterbi algorithm has numerical underflow errors for deep time steps. I worked around this problem with the classic log trick! Because of the monotonocity of the log function, we know that if x < y then log(x) < log(y). However, because of the walls in the maze, some of the states will have probability 0 and the log function is not defined at 0. We get around this by checking before hand if the product of our probability terms is 0 and if it is then we set the probability score to be the largest negative integer.
- Once we have completed the forward pass, we compute what the most likley final state was by taking the argmax of the final column in our T_1 matrix. Once we have the value of our finals state, we use our T_2 matrix to backtrack through the most likley state at time step T conditioned on the most likley state at time step T + 1. Backtracking through the entire graph will yield the most likley path!

Testing and Analysis

Consider the no_walls maze. We observe a very similar problem when we run the viterbi algorithm on this maze than we did when we were running the forward filtering. This maze has such a uniform structure that even though the sensor was almost nearly perfect, our most likley sequence was able to get thrown off. It got thrown off because there was more than one correct next state that was B! This threw off our most likley sequence.

We see in the example below that the backtracking algorithm messed up on the 7th time step even though the sensor was correct!

```
RBGY
RGYB
GYBR
YBRx

This is how many times the sensor got it right out of 50: 49

Colors that the robot saw:
['R', 'G', 'G', 'G', 'R', 'R', 'B', 'Y',
'B', 'R', 'R', 'B', 'R', 'G', 'R', 'B',
'Y', 'G', 'Y', 'Y', 'B', 'B',
 'Y', 'G', 'B', 'B', 'R', 'R', 'B', 'B',
 'B', 'G', 'G', 'G', 'G', 'G', 'B', 'B', 'R', 'R', 'R', 'R', 'R', 'R',
 'B', 'G', 'B', 'B', 'G', 'Y']
-------------
Where did the sensor lead us wrong?
[True, True, True, True, True, True, True,
True, True, True, True, True, True, True,
True, True, True, True,
True, True, True, True, True, True, True,
True, True, True, True, True, True, True,
True, True, False, True,
True, True, True, True, True, True, True,
True, True, True, True, True, True, True]

Predicted:
[2, 3, 3, 3, 2, 2, 1 (FIRST ERROR), 0, 1, 2, 2, 1, 2, 3, 2, 1,
0, 4, 0, 0, 1, 1, 5, 9, 13, 13, 12, 12, 13, 13,
 13, 14, 14, 14, 14, 14, 13, 13, 12, 12, 12, 12,
 12, 12, 13, 9, 13, 13, 9, 5]

Ground Truth:
[2, 3, 3, 3, 2, 2, 6 (FIRST ERROR), 5, 6, 7, 7, 6, 7, 3, 7, 11,
10, 14, 15, 15, 11, 11, 10, 9, 13, 13, 12, 12,
 13, 13, 13, 14, 14, 14, 13, 9, 13, 13, 12, 12,
  12, 8, 12, 12, 13, 9, 13, 13, 14, 10]
```

Now lets run our backtracking algorithm on the maze unique_walls. Since the expected number of correct sensor reads is 42, I ran the algorithm several time searching for a "good" run where we got a lot of the colors correct. This is the starting position of the maze. We see that our viterbi algorithm is working very well! We were able to construct the EXACT path for 50 time steps! This is because when we are correctly getting the information from the sensor, that information tells us a LOT about where we are in the maze because there isnt several locations that we could be in after observing the correct color. This is is a great sanity check that our algorithm is correct that we were able to perfectly

reconstruct the sequence.

```
RBG#
GY#G
#RBY
YG#x

This is how many times the sensor got it right out of 50: 49

Colors that the robot saw:
['R', 'R', 'Y', 'B', 'B', 'B', 'R', 'R', 'Y', 'R',
'B', 'B', 'B', 'Y', 'B', 'B', 'B', 'R', 'G', 'G',
'Y', 'Y', 'G', 'Y', 'G', 'G', 'G', 'R', 'Y', 'B',
'Y', 'R', 'B', 'B', 'B', 'B', 'Y', 'Y', 'Y', 'R',
'R', 'R', 'Y', 'G', 'G', 'Y', 'Y', 'R', 'R', 'G']
-------------

Where did the sensor lead us wrong?
[True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True,
True, True, True,True, True, True, True, True, True,
True, True, True, True, False]

Predicted:
[3, 3, 7, 6, 6, 6, 5, 5, 9, 5, 6, 6, 6, 7,
6, 6, 6, 5, 1, 1, 0, 0, 1, 0, 1, 1, 1, 5, 9, 13,
9, 5, 6, 6, 6, 6, 7, 7, 7, 3, 3, 3, 7, 11, 11, 7,
7, 3, 3, 3]

Ground Truth:
[3, 3, 7, 6, 6, 6, 5, 5, 9, 5, 6, 6, 6, 7,
6, 6, 6, 5, 1, 1, 0, 0, 1, 0, 1, 1, 1, 5, 9, 13,
9, 5, 6, 6, 6, 6, 7, 7, 7, 3, 3, 3, 7, 11, 11, 7,
7, 3, 3, 3]
```

Perfect Sensor Analysis (Extension)

A logical question to ask is how the dynamics of this problem change when we remove the randomness in the sensor. I simply added a class variable that will construct the observation matricies accordingly.

We see in an example from no_walls that now it is possible to have certainty about where we are in the maze. This could be achived in several different ways. For example, if we run into the bonudry onece and the color near the boundary is not a legal move then we know for vcertain we are on that square. However,

the pattern seen in the prior two experiments also can re-introduce uncertainty for the robot where there is more than one color to move to. See the below example for this exact case! There is no B square that is next to the square that the robot is currently at so when it bumps into the walll and sees B, it knows where it is!

```
---------------------
-> Here is our robots new location after trying to move! (1, 3)
-> Color we observed! B. This observation is True
-> Here is the updated distro:
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0]

-> Here is the probability of being in the true location: 1.0

RxGY
RGYB
GYBR
YBRG


---------------------
-> Here is our robots new location after trying to move! (2, 3)
-> Color we observed! G. This observation is True
-> Here is the updated distro:
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0, 0.5, 0.0]

-> Here is the probability of being in the true location: 0.5

RBxY
RGYB
GYBR
YBRG
```

Now lets look at a much more fun example. Consider the wizard_of_oz maze where there are no walls and the 4 quadrants of the maze are broken up into solid color blocks. When we let the robot walk around this maze with a random sensor, it isnt able to get confidence about where it is exactly. This makes a lot of sense because reading a color only gives it very coarse information about where it is in the maze. If the robot reads R R R R, then it really has no clue where it is in the maze exactly but it is more likley that it is in the R quadrant.

My question is the following: if we take the randomness away from the sesnor and we let the robot walk around randomly, will it converge to certainty about its location at any point? If it does, how? On average, how many steps does it take to converge?

First, lets analyze some of the dynamics of the probability distribution over time for the robot moving around the wizard of oz maze with perfect information. We see that when its just walking around, it is still very uncertain about where

it exactly. For example, consider this time step where the robot is towards the center of the yellow quadrant. It knows for certain that its somewhere in one of the bottom right squares, but no clue exactly where:

```
-> Here is our robots new location after trying to move! (6, 3)
-> Color we observed! Y. This observation is True
-> Here is the updated distro:
[0.0, 0.0, 0.0, 0.0, 0.0, 0.111, 0.113809, 0.
050215, 0.008722, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.1027, 0.
10459, 0.04614, 0.00801, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
085, 0.086, 0.038, 0.0066, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0611060, 0.0621756, 0.027433, 0.0047658, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.03184, 0.032401, 0.01429, 0.00248366, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0]

-> Here is the probability of being in the true location: 0.06217560733587193

RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
GGGGGYYYYY
GGGGGYxYYY
GGGGGYYYYY
GGGGGYYYYY
GGGGGYYYYY
```

The robot does gain a lot of information, however, when it crosses over the lines into a different region. This is because since we have removed the randomness from the sensor, the robot knows that it can only be in one of 5 locations once it has crossed over from one region to another. This can be seen in the following example:

```
-> Here is our robots new location after trying to move! (4, 4)
-> Color we observed! G. This observation is True
-> Here is the updated distro:
[0.065016, 0.059749, 0.049641, 0.
0355121, 0.018505, 0.0, 0.0, 0.0, 0.0, 0.0, 0.066, 0.0615, 0.
05113, 0.0365802, 0.019062, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0657, 0.
060457, 0.0502300, 0.035933, 0.0187, 0.0, 0.0, 0.0, 0.0, 0.0,
0.055008, 0.05055, 0.04200017, 0.030045, 0.015657, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.03184, 0.029265, 0.0243143, 0.0173937,
```

```
0.0090640393675255, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

-> Here is the probability of being in the true location: 0.0090640393675255

RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
GGGGxYYYYY
GGGGGYYYYY
GGGGGYYYYY
GGGGGYYYYY
GGGGGYYYYY


----------------------
-> Here is our robots new location after trying to move! (5, 4)
-> Color we observed! Y. This observation is True
-> Here is the updated distro:
[0.0, 0.0, 0.0, 0.0, 0.0, 0.228 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.23, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.2311,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.1932, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.11, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0]

-> Here is the probability of being in the true location: 0.11188230459356747

RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
GGGGGxYYYY
GGGGGYYYYY
GGGGGYYYYY
GGGGGYYYYY
```

```
GGGGGYYYYY
```

It is possible for the robot to gain 100% certainty about where it is in the maze? Yes! This is because of the uniqueness of the color reading when the robot is towards the center of the maze. For example, given the sequence of colors observed being R B Y, the robot can only be in a single square! This is because any time the robot sees the color sequence R B, it know it is in one of the 5 locations on the boarder between the R and B region. When it then sees Y, there is only 1 B tile on the boarder between R and B that also is next to a Y, thus the robot knows with certainty where it is! This example was found and I have illustrated it below!

```
-> Here is our robots new location after trying to move! (4, 4)
-> Color we observed! G. This observation is True
-> Here is the updated distro:
[0.0, 0.0, 0.0, 0.0, 0.28470339925899435, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.2612892051638434, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.21765717019392553,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.15514642371481158, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.08120380166842515, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

-> Here is the probability of being in the true location: 0.08120380166842515

RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
GGGGxYYYYY
GGGGGYYYYY
GGGGGYYYYY
GGGGGYYYYY
GGGGGYYYYY


----------------------
-> Here is our robots new location after trying to move! (5, 4)
-> Color we observed! Y. This observation is True
-> Here is the updated distro:
[0.0, 0.0, 0.0, 0.0, 0.0, 0.2847033992589943, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.2612892051638434, 0.0, 0.0, 0.0, 0.0, 0.0,
```

```
 0.0, 0.0, 0.0, 0.0, 0.2176571701939255, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.15514642371481155, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.08120380166842514, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

-> Here is the probability of being in the true location: 0.08120380166842514

RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
GGGGGxYYYY
GGGGGYYYYY
GGGGGYYYYY
GGGGGYYYYY
GGGGGYYYYY


----------------------
-> Here is our robots new location after trying to move! (5, 5)
-> Color we observed! B. This observation is True
-> Here is the updated distro:
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0]

-> Here is the probability of being in the true location: 1.0

RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRBBBBB
RRRRRxBBBB
GGGGGYYYYY
```

```
GGGGGYYYYY
GGGGGYYYYY
GGGGGYYYYY
GGGGGYYYYY


----------------------
CONVERGED TO DETERMINISTC IN 56 steps!
```

We know that its possible for the robot to find out where it is and we know that it only took the robot in this case 56 steps to find its location. If we randomize the starting location of the robot and run this experiment several times, what would the results look like.

Below is several simulations for a robot doing a random walk in the wizard of oz maze. On the horizontal axis we have the starting location of the robot which represents the index of the starting location of the robot and the vertical axis is the number of steps taken to converge to a state where the robot is 100% confident in location. The red dotted line at the bottom shows which of the trials did not converge to the confident state. We see that at around 5000 steps all of the random walkers converge to the confident state reguardless of start state. The average number of steps taken for convergense appears to be around 330 steps.

It does not appear from the plots that the start location has a large impact on the convergence speed of the robot.

For each of the simulations I recorded the number of random walkers who didnt converge given the number of steps they were allowed to take. Once each of the random walkers converged, I computed the average number of steps taken for each of the robots to converge.

```
481 of the random walkers didnt converge with 10 steps around the maze our of 500 total rand

369 of the random walkers didnt converge with 100 steps around the maze our of 500 total ran

124 of the random walkers didnt converge with 500 steps around the maze our of 500 total ran

27 of the random walkers didnt converge with 1000 steps around the maze our of 500 total ran

3 of the random walkers didnt converge with 2000 steps around the maze our of 500 total rand

0 of the random walkers didnt converge with 5000 steps around the maze our of 500 total rand
Average Number of steps taken to converge: 329.684

0 of the random walkers didnt converge with 10000 steps around the maze our of 500 total ran
Average Number of steps taken to converge: 330.12
```

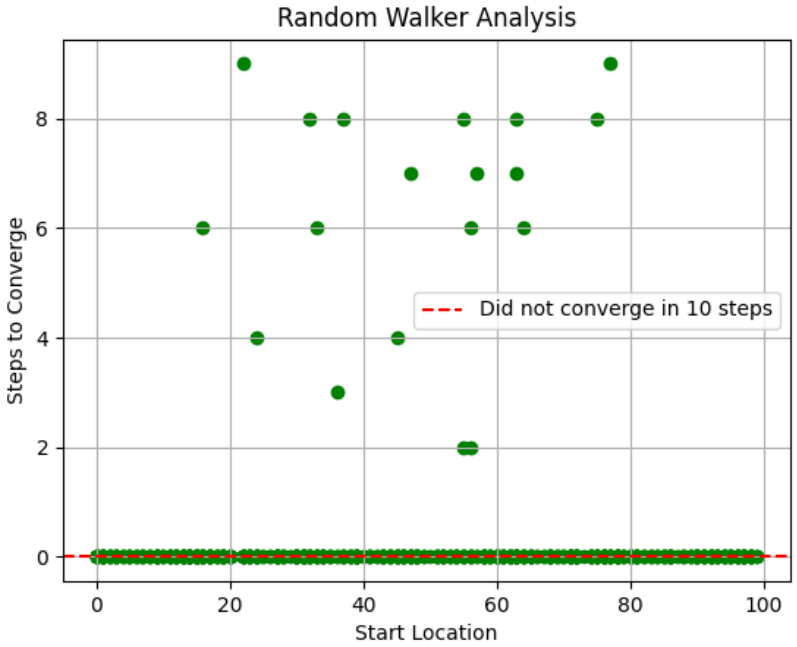

Figure 4: RandomWalker.png

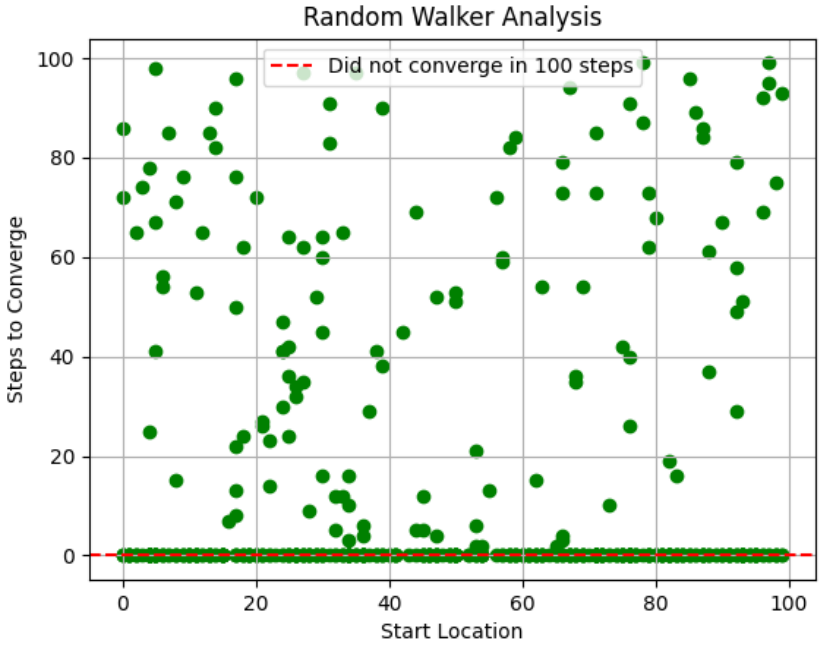

Figure 5: RandomWalker.png

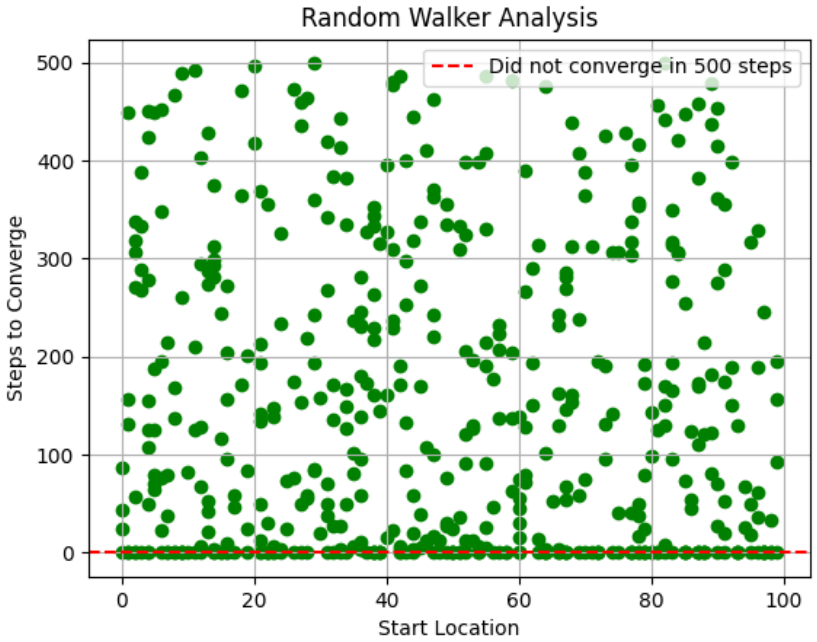

Figure 6: RandomWalker.png

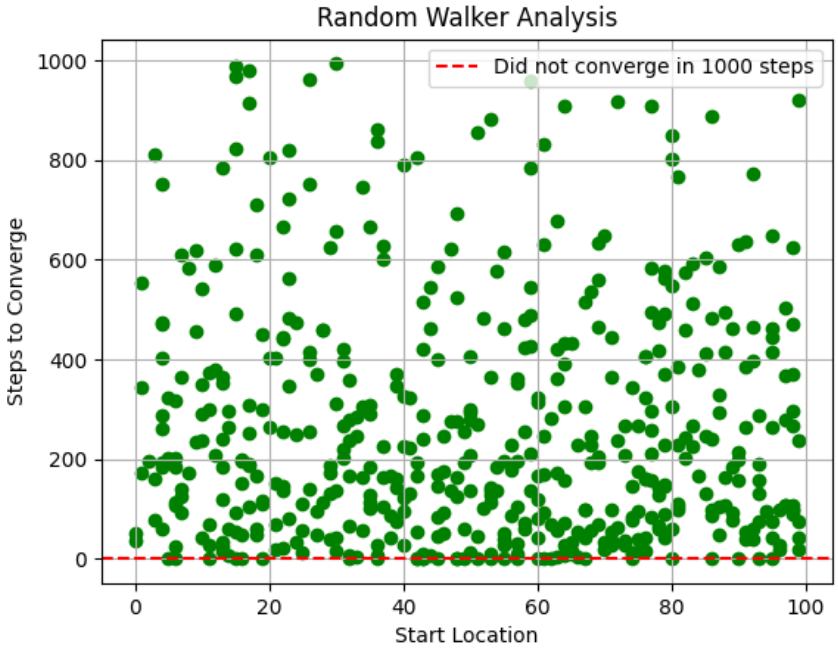

Figure 7: RandomWalker.png

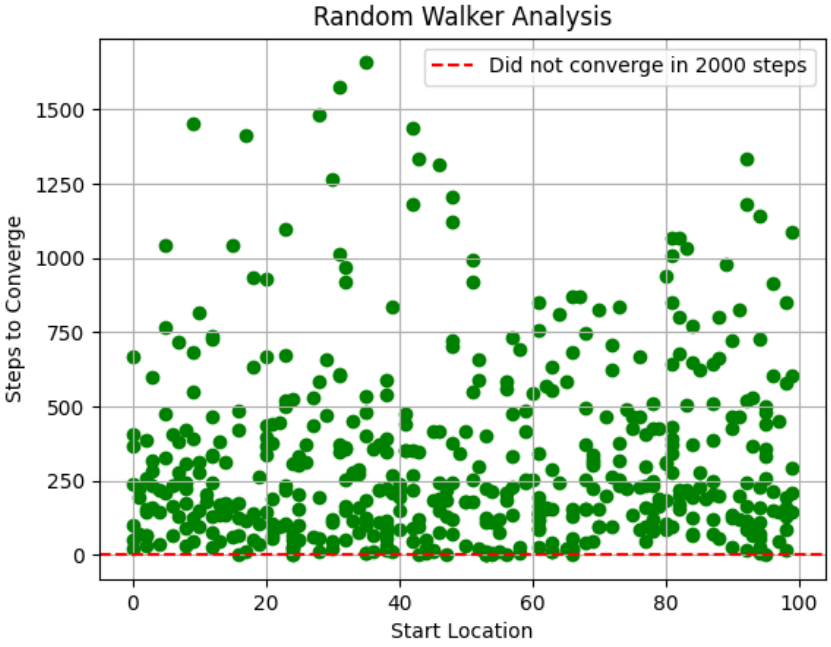

Figure 8: RandomWalker.png

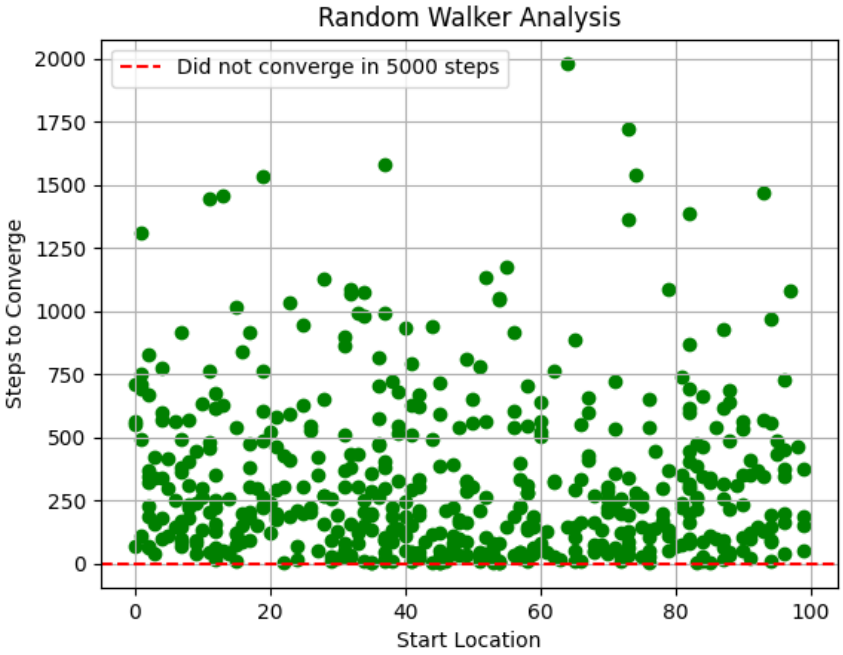

Figure 9: RandomWalker.png

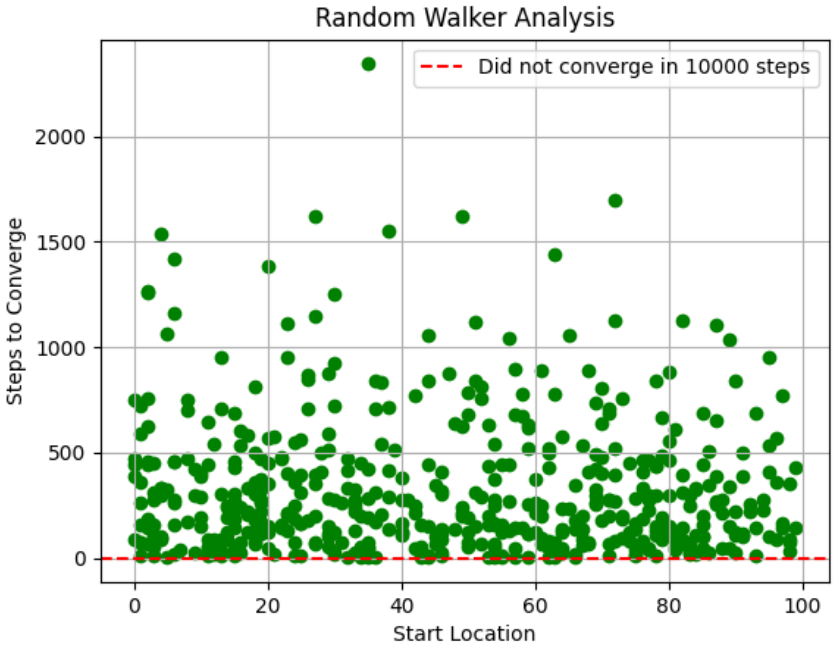

Figure 10: RandomWalker.png

Page Rank Paper

In Extrapolation Methods for Accelerating PageRank Computations by Kamvar et al., they propose a faster algorithm for computing the rank of web pages on the internet. The key assumption behind the PageRank algorithm is that the importance of a webpage can be modeled with a random walker on the internet, where the internet is modeled as a mathematical graph with nodes as web pages and links as directed edges on the graph. Given a random walker on this graph and a large number of time steps, the importance of a webpage is the probability that given a random time step T that the random walker is on that webpage.

There are a couple hurdles that the authors needed to adress due to the nature of the web graph. First, some nodes on the graph are sinks. These are web pages which do not have any out edges. If the random walker were to ever walk to one of these pages, then they would be stuck. They get around this by adding a random noise matrix D which is the outer product between the uniform distribution vector and a one hot encoded vector for each of the web pages which is a sink. This creates the behavior that if the random walker ever lands on a sink, then we randomly "teleports" to another web page on the internet.

The other hurdle comes in the form of the application of the Ergodic Theorem for Markov chains, a property which the authors want to impose on the transition matrix A. This theorem says that a unique steady state distribution exists which is not a function of the initial state if A is both aperiodic and irreducible. In order to incorperate irreducibility, the authors add the constraint that the web graph be implicitly complete- that is, there exists a connection between each node in the graph. They made this probability very small as to not change the behavior too much but to ensure that the graph is strongly connected. This creates the behavior that at any node the random walker can jump to another node in the graph. This is the behavior we added to the sinks now applied to all nodes in the graph.

When digging into the linear algebra, we see that the key mathematical insight of the algorithm comes from the eigenvalues of the A. It can be shown that all eigenvalues of the A are stricly smaller than 1 except for the first eigenvalue. Assuming that the inital distribution lies in the span of the A, we know that the initial distribution can be written as a linear combination of the eigenvectors of the A:

$$x^1 = Ax^0 = 1u_1 + \alpha_1\lambda_2 u_2 + ... + \alpha_m\lambda_m u_m$$

where α_i is the coefficent for the linear combination expansion of x by the eigenvectors of A and $\lambda_m u_i$ is the i th eigenvalue of A. If we repeat the multiplication of the transition matrix to A we get:

$$x^n = A^n x^0 = 1u_1 + \alpha_1\lambda_2^n u_2 + ... + \alpha_m\lambda_m^n u_m$$

Since we know that $\lambda_i < 1 \implies \lambda_i^n$ goes to zero as $n \to$ inf. Thus, the probaility distribution of the x approaches the principal eigenvector u_1

Thus, the convergence of x^n is a function of how quickly we can get the rest of the terms in the eigenvector expansion to go to zero. This is a very slow convergence if λ_2 is close to 1. It can be shown that λ_2 is close to 1 exactly when the hyperparameter, c, which determines the probability the random walker does a teleportation, is low. When $c = 1$ the random walker does no random teleporation and when $c = 0$ the random walker teleports aeach step. We would like to be able to model this computation when c is closer to 1 since when $c = 0$, the ranking of each page becomes uniform.

Thus, the authors develop the Quadratic Extrapolcation algorithm by assuming that the markov matrix A has 3 eigenvectors and they solve for an estimate of the principal eigenvector u_1. The motivation for there method is that the bottle neck of the power method algorithm is the size of the first couple eigenvalues. By making an assumption about the eigenvector being represented in closed form as the linear combination of the first eigenvectors of A, there novel algorithm periodically computes the state vector which has reduced second and third eigenvector components. There empircally showed that this algorithm, when we have very little random teleportations, dramatically boosts the performance of the PageRank algorithm.

New vocab/concepts I learned about when reading this paper:

1) Power Method: An algorithm for computing the largest eigenvalue and its corresponding eigenvector for a diagonalizable matrix A by repeated iterations of Ax where x can be any vector. Recall that a matrix A is diagonalizable if it can be expressed another matrix B which is invertible (determinant non 0) as $A = BDB^{-1}$ where D is a diaganol matrix (a matrix with non zero values at index (i,i) and 0 everywhere else).

2) Princeipal Eigenvector: the eigenvector associated with the largest eigenvalue

3) Markov Chain: Formally, a Markov Chain is a stochastic proces (a sequence of random variables) which meets the first order markov property that the probability of an event depends only on the state prior.

4) Aperiodic and Irreducible Markov Chains: Irreducible Markov chains are those in which every state can be reached from any other state. In the transition matrix, this would represent there being no zeros in any of the entries and in the mathematical graph would represent a complete graph. An aperiodic Markov Chain is one in which the behavior of the Markov chain does not begin to repeat after a certain number of steps.

Summary of Extensions

- Smoothing Implemtation + Graphical Analysis of Probability Boost on Most Likley State Across Different Mazes
- Viterbi Implementation + Analysis on Different Mazes
- Random Walk Analysis on Wizard of Oz Maze with Perfect Information

- Google Accelerated Page Rank Computation Litturature Review