Continued Exploration of the Pocket Cube

By Ryan Brown

This paper does not seek to be a full mathematical article on the topic, but instead seeks to more in-depth summarize the work done this summer as well as to serve as a reference should further work be done on this topic, or a more in-depth article written on this subject.

Initial Exploration of the Pocket Cube

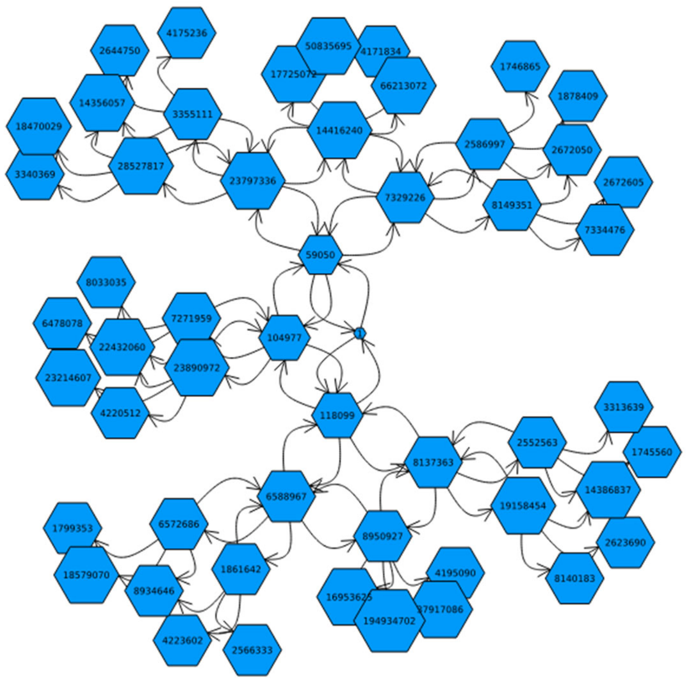
The start of this project was completed over the spring semester and the article about that can be read if desired. Throughout the semester, the software I programmed kept track of the 'state' of the cube as two 8-tuplets (two 8-digit long strings) where the first string kept track of the position of each cubie while the second string kept track of each cubie's rotations. This was partially modeled off an article proving God's Number[1] for the Rubik's Cube.[2] The benefit of this model was it allowed for enhanced use of symmetry to limit the number of elements needed to fully explore the Rubik's Group.

The summer started by fine tunning the program and continuing computation of the states. Issues started appearing, where the computer would run out of active memory. As such the program shifted to storing all calculations done in files which held information about each element. This method was able to calculate thousands of elements, however, never became robust enough to finish the job of exploring the full cube. The software can certainly be cleaned up, however other methods were used to achieve the results. Seen below is a directed

[1] The maximum, minimum number of moves needed to solve some state of the Rubik's Cube

[2] Tomas Rokicki et al. "The Diameter of the Rubik's Cube Group Is Twenty". In: SIAM Review 56.4 (2014), pp. 645–670. doi: 10.1137/140973499. eprint: https://doi.org/10.1137/140973499. url: https://doi.org/10.1137/140973499.

graph of the elements up through depth 4. Elements further out were calculated but add clutter.



During this phase to help mitigate the calculations needed, a retroactively simple result appeared. A right turn on the pocket cube is the same as a left inverse turn. Therefore, one can only achieve the full graph by using only 3 generators. The graph above also accounts for color permutation[3] and physical isometries,[4] but does not account for reflectional symmetry.[5]

[3] Colors can be rearranged, and the cube would be the same. Think of a color-blind man that sees yellow as white and white as yellow. They could still solve the cube the same as you could.
[4] A cube rotated around is the same cube.
[5] A inverse move is the same as a regular move just in opposite directions

Group Theory to the Rescue

Through this time my mentor, Daniel Glasscock, gave me resources to refresh my knowledge of group theory thinking there may lie an answer inside of that heavily studied field. The first result to note is that out of the symmetries discussed above, only physical isometries maintain the group structure of the Pocket Rubik's Group. Color Permutations and Reflectional symmetry are pseudo-symmetries which can help to lessen the number of elements to study for my purposes but would not abide by the rules of group symmetry and equivalence classes.

Glasscock also introduced GAP to me, which is a software for group theory computation. While the documentation was slim, Glasscock encouraged me to define the Pocket Rubik's Group by a set of relations and free group generators. As discussed above, we only need 3 generators to generate the full group. Through my work, I also discovered that the software worked well with cyclic generators. As such I defined the 3 generators cyclically and that allowed me to fully map out the group. By using 3 generators, the physical isometries were fixed, which allowed a 24-fold reduction of elements over the 6 generators. The 6 generators would have also given 24 different identity elements, so it was useful to not worry about that. The Code used in gap is seen below.

```
PrintTo("Elements.txt","");
PrintTo("Graph.txt","");
R:= (13,14,16,15)(10,2,19,22)(12,4,17,24);
D:= (21,22,24,23)(11,15,19,7)(12,16,20,8);
B:= (17,18,20,19)(2,5,23,16)(1,7,24,14);
G := Group(R,B,D);
GrowthFunctionOfGroup(G,14);
F := FreeGroup("R","B","D");
hom := GroupHomomorphismByImages(F,G,GeneratorsOfGroup(F),GeneratorsOfGroup(G));
G_E := Elements(G);;
for i in [1..Length(G_E)] do
        AppendTo("Elements.txt",i,":",PreImagesRepresentative( hom, G_E[i]),"\n") ;
```

```
od;
for i in [1..Length(G_E)] do
        AppendTo("Graph.txt",i,":",Position(G_E,R*G_E[i]),",",Position(G_E,B*G_E[i]),",",Position(G_E,D*G_E[i]),",",Position(G_E,R^(-1)*G_E[i]),",",Position(G_E,B^(-1)*G_E[i]),",",Position(G_E,D^(-1)*G_E[i]),"\n");
od;
```

The resulting files generated gave me two files. One which gave a name for every element (though not necessarily the simplest name), and another which gave the adjacency relations in a csv. By doing some work on each of these, each element received its true name (minimum number of moves, letters, to get each position) as well as a software which could generate a sparse matrix of a Markov chain. The element's numbers were also renumbered in order of distance from solved. The number of elements at each distance d away from solved was calculated and is seen below.

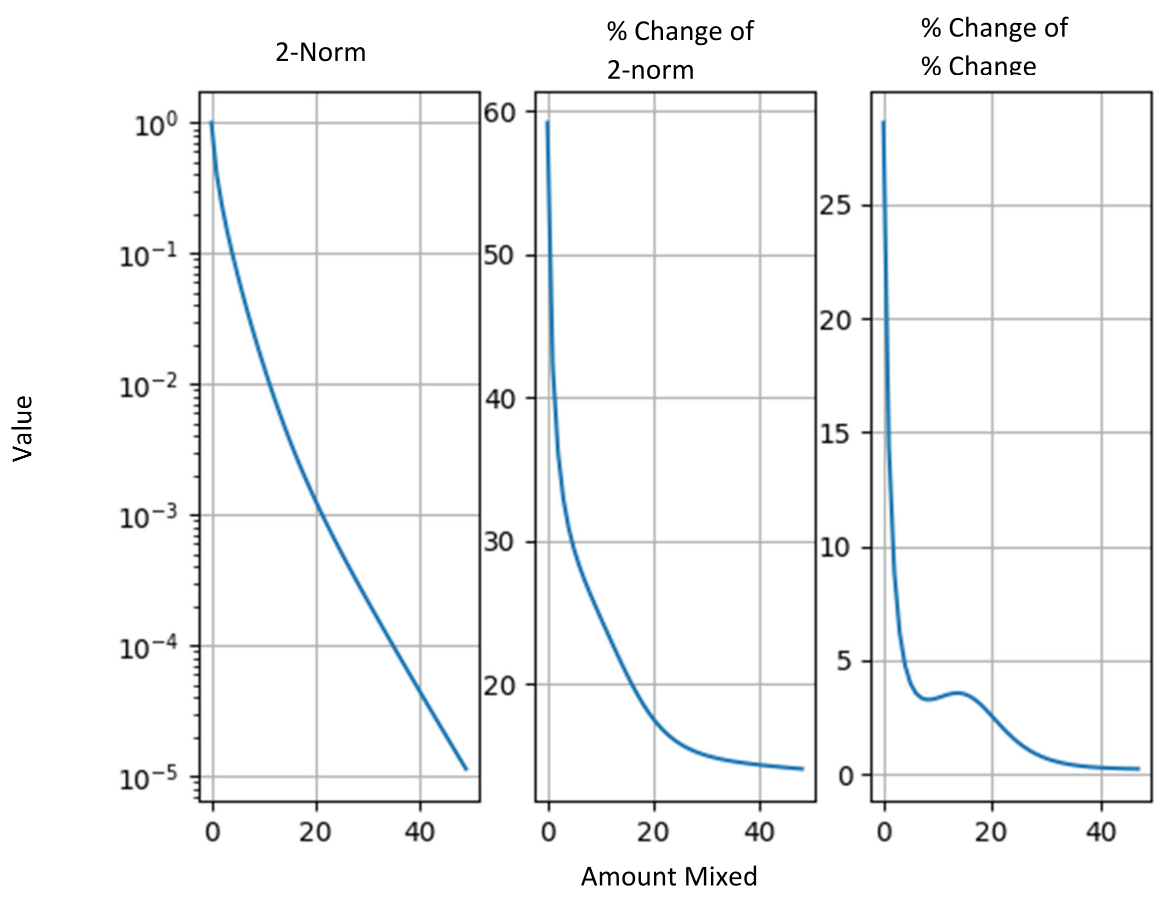
Distance to Solve	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Number of Elements	1	6	27	120	534	2256	8969	33058	114149	360508	930588	1350852	782536	90280	276

Further work was continued on in python for "light" Markov chain analysis where the probability vector after m turns was analyzed instead of the full Stochastic Matrix. The full matrix would, typically, give more powerful results and results that would apply from every starting state, however that is not needed for our study and was also too large to keep in active memory. Graphing and other computation on the chain was done in python for their sparse matrix package. Julia was initially used, but their sparse package was not sufficiently integrated with their parallel programing packages, which gave python a leg up. The python code will be at the end of the document, but the results will be discussed here.

Results of Markov Chain Analysis

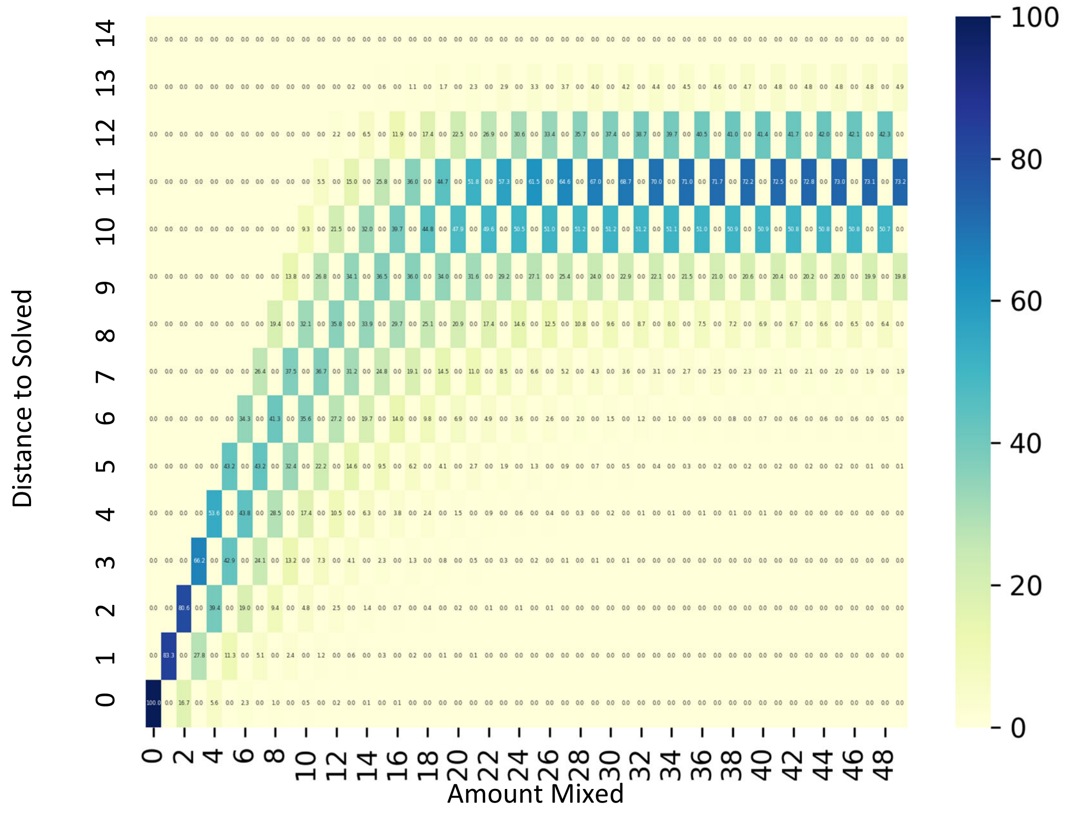
The first thing noticed is that there isn't just a steady state, but there are two. A corollary to this is that states only shared edges with states that are one closer or one further away to solved. To analyze the distance from steady, both steady states need to be calculated. By a simple counting argument one can surmise that half the elements are even, and half are odd. The easy way would be to add up all the even elements, then the odd elements and see that clearly, they each are the same. Or one can reason that the even steady state must have ***n*** elements with ***p*** probability each and the odd steady state must have ***m*** elements with ***q*** probability each. By probability $np = mq$. As the edges going from an odd state to an even, must be the same as an even state to an odd state (just the reverse edge), and each element has 6 edges from it. As discussed in the previous paper, all the 6 elements connected must be unique, therefore every odd element has $6 \times \frac{p}{6} = p$ probability, and by definition $p = q$. Then by simple algebra $n = m$.

Now one can examine the norm of the difference of any given probability distribution and the corresponding steady state. The 2-norm is used for this purpose.

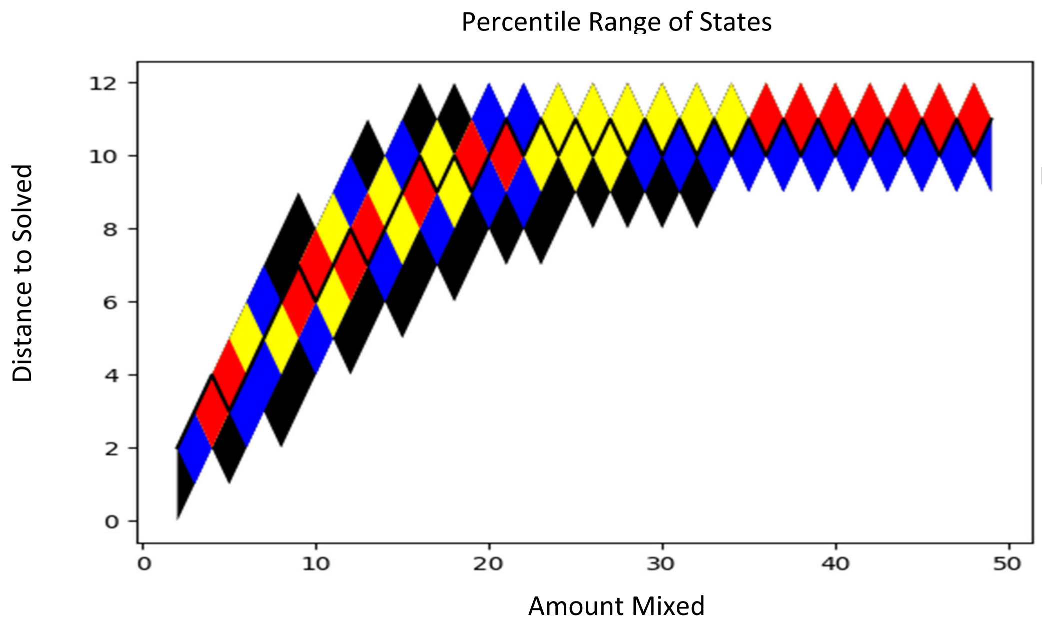


The mixing algorithm used is a random moved (out of 6, R,B,D,L,F,U), and the probability vector of state is subtracted by the steady state in order take the norm of the resulting vector. The results show that much of the distribution occurs at first, with a spike around the upper teens. Then around 30 is where the mixing stops changing all that much and the gain for a additional mix remains more or less constant and low.

Probability Heat Map



By looking at the heat map, a similar story is seen. Once mixed around 20 times the end behavior is almost reached, but around the low 30s it solidifies closer to the steady state.



Areas:

Black – 10th to 90th Percentiles

Blue – 20th to 80th Percentile

Yellow- 30th to 70th Percentile

Red – 40th to 60th Percentile

Line:

Black - Median

When looking at the percentiles it is seen that the median stops changing at around 20 moves, except for the even odd change, however at 33 and 34 the graph stops changing and it is seen that the 10th to 90th percentile is completely covered by the 40th to 60th and the 20th to 80th, meaning that with a probability of at least 80%, one is either 10 or 12 away from solved at 34 or 9 to 11 at 35.

In terms of the cube distance to solved is often described by God's Algorithm or the perfect solution. Meaning that if you were to hand a cube to God, you would only need to mix it randomly 34 times before you are fairly certain it is a less trivial activity for god than if you were to mix it less, but mixing the cube more would be unnecessary as it would gain little.

As the analysis for God's algorithm came up with the same answer as analyzing the distance from the steady state, I conjecture that for most solving algorithms for the pocket cube, it is sufficient to mix the cube randomly 34 times and mixing it more indeed tends to be near pointless.

Further questions

Some questions clearly arise. Is this conjecture true? For which solving algorithms does this conjecture fail? How does this change if one uses a different mixing algorithm? One of the biggest questions is, of course, what about the standard cube or the nxnxn cube? The amount of computation or thought needed is more due to the shear size and the increasing complexity of the cube, so that question may not be answered soon, but would be interesting to answer. Another question is does the steady state parity exists in all sizes of Rubik's Cubes, or only even, or only prime, or only in the 2x2?

Python Code:

```
import numpy as np
import pandas as pd
import scipy.sparse as sp
import pickle as pk
from joblib import Parallel, delayed
import matplotlib.pyplot as plt
import math as mt
import seaborn as sns

## Gives the nth moment with probability vector p, and value vector v
def moment(p,v,n):
	sum = 0
	for i in range(len(p)):
		sum += pow(v[i],n)*p[i]
	return sum
##Loads the sparse matrix
def load(c,g):
	for j in range(3674160):
		for i in range(1,7):
			c[g[0][j]-1,g[i][j]-1]=1/6
		if(j%100000 == 0):
			print(j)
```

```
        m= sp.csr_matrix(c)
        return m
## Multiplies probability vector by matrix m, n times
def power(pv,m,n):
        p = pv
        for i in range(n):
                p = m*p
        return p


##Returns expectation of probability vector multiplied n times by matrix m
def power_exp(length,pv,m,i):
        return np.dot(length,power(pv,m,i))

## Returns moment of probability vector multiplied n times by matrix m
def power_moment(p,m,n,v,l):
        pv = power(p,m,n)
        return moment(pv,v,l)

##Returns the variance of a vector
def varience(p,v):
        return moment(p,v,2)-moment(p,v,1)**2

##Returns the standard deviation of a vector
def std_dev(p,v):
        return mt.sqrt(varience(p,v))

### Returns the variation of a vector multiplied n times by matrix m
def power_varience(p,v,m,n):
        pv = power(p,m,n)
        return varience(pv,v)

### Returns the standard deviation of a vector multiplied n times by matrix m
def power_std_dev(p,v,m,n):
        pv = power(p,m,n)
        return mt.sqrt(varience(pv,v))

##Gives the norm of the difference between the steady state and the probability vector
multiplied by m, n times.
##t gives the type of norm, 2 norm is useful
def power_norm(pv,m,i,even_steady_state,odd_steady_state,t):
        if(i%2 ==0):
                return np.linalg.norm(power(pv,m,i) - even_steady_state,t)
        return np.linalg.norm(power(pv,m,i) - odd_steady_state,t)
```

```
## This gives a array needed for the heat map
def prob_heat(pv):
        lst = [0.0]*15
        lst[0] = pv[0]*100
        lst[1] = sum(pv[1:6])*100
        lst[2] = sum(pv[7:33])*100
        lst[3] = sum(pv[34:153])*100
        lst[4] = sum(pv[154:687])*100
        lst[5] = sum(pv[688:2943])*100
        lst[6] = sum(pv[2944:11912])*100
        lst[7] = sum(pv[11913:44970])*100
        lst[8] = sum(pv[44971:159119])*100
        lst[9] = sum(pv[159120:519627])*100
        lst[10] = sum(pv[519628:1450215])*100
        lst[11] = sum(pv[1450216:2801067])*100
        lst[12] = sum(pv[2801068:3583603])*100
        lst[13] = sum(pv[3583604:3673883])*100
        lst[14] = sum(pv[3673884:3674159])*100
        return np.array(lst)

##Power composed with prob heat
def power_prob_heat(pv,m,n):
        return prob_heat(power(pv,m,n))

##Gives the cumulative sum
def cum_sum(pv):
        p = pv
        for i in range(1,len(pv)):
                p[i] = p[i]+p[i-1]
        return p
##Returns the values of the quartiles
def iqr(pv):
        lst = [0]*3
        p = cum_sum(prob_heat(pv))
        i = 0
        while(p[i]<25):
                i+=1
        lst[0]=i
        while(p[i]<50):
                i+=1
        lst[1]=i
        while(p[i]<75):
                i+=1
        lst[2]=i
        return np.array(lst)

## Composes powers and iqr
def power_iqr(pv,m,n):
        return iqr(power(pv,m,n))









##Gives the percentile specified of vector p
def percentiles(pv,num):
        lst = [0]*9
        p = cum_sum(prob_heat(pv))
        #print(p)
        i=0
        while(p[i]<num):
                if(i<=13):
                        i+=1
                else:
                        return i
        return i

##Composes power and percintiles
def power_pecentiles(pv,m,n,num):
        print(n)
        return percentiles(power(pv,m,n),num)

###This generates the spare stochastic matrix, only needed to run once, as it saves
#g = pd.read_csv('graph_rename.csv',header=None)
#c = sp.lil_matrix((3674160,3674160),dtype=float)
#filehandler = open('adjencency_matrix.obj', 'wb')
#pk.dump(load(c,g), filehandler)

### Opens the stochastic matrix
file_pi = open('adjencency_matrix.obj','rb')
m = pk.load(file_pi)

## Starting Probabilitity Distribution
lst = [0.0]*3674160
```

```
lst[0] = 1.0
pv = np.array(lst)

## 1837080 Elements Even & Odd Distance Away
#### Even Steady State
#lst = [0]*3674160
#lst[0] = 1/1837080
#for i in range(7,34):
#	lst[i]= 1/1837080
#for i in range(154,688):
#	lst[i]= 1/1837080
#for i in range(2944,11913):
#	lst[i]= 1/1837080
#for i in range(44971,159120):
#	lst[i]= 1/1837080
#for i in range(519628,1450216):
#	lst[i]=  1/1837080
#for i in range(2801068,3583604):
#	lst[i]= 1/1837080
#for i in range(3673884,3674160):
#	lst[i]= 1/1837080
#even_steady_state = np.array(lst)
#### Odd Steady State
#lst = [0]*3674160
#for i in range(1,7):
#	lst[i]= 1/1837080
#for i in range(34,154):
#	lst[i]=1/1837080
#for i in range(688,2944):
#	lst[i]=1/1837080
#for i in range(11913,44971):
#	lst[i]=1/1837080
#for i in range(159120,519628):
#	lst[i]=1/1837080
#for i in range(1450216,2801068):
#	lst[i]=1/1837080
#for i in range(3583604,3673884):
#	lst[i]=1/1837080
#odd_steady_state = np.array(lst)



## Generates the value vector for each element, gives the distance to solved
lst = [0]*3674160
```

```
for i in range(1,7):
        lst[i]=1
for i in range(7,34):
        lst[i]=2
for i in range(34,154):
        lst[i]=3
for i in range(154,688):
        lst[i]=4
for i in range(688,2944):
        lst[i]=5
for i in range(2944,11913):
        lst[i]=6
for i in range(11913,44971):
        lst[i]=7
for i in range(44971,159120):
        lst[i]=8
for i in range(159120,519628):
        lst[i]=9
for i in range(519628,1450216):
        lst[i]=10
for i in range(1450216,2801068):
        lst[i]=11
for i in range(2801068,3583604):
        lst[i]=12
for i in range(3583604,3673884):
        lst[i]=13
for i in range(3673884,3674160):
        lst[i]=14

length = np.array(lst)




##This gives the graph of the percentiles from 10 to 90 for the probability vector run between 2
and 50 times through the stochastic matrix
range_1 = 2
range_2 = 50

r_10 = Parallel(n_jobs=2)([delayed(power_pecentiles)(pv,m,i,10) for i in
range(range_1,range_2)])
r_20= Parallel(n_jobs=2)([delayed(power_pecentiles)(pv,m,i,20) for i in
range(range_1,range_2)])
```

```
r_30 = Parallel(n_jobs=2)([delayed(power_pecentiles)(pv,m,i,30) for i in
range(range_1,range_2)])
r_40 =  Parallel(n_jobs=2)([delayed(power_pecentiles)(pv,m,i,40) for i in
range(range_1,range_2)])
r_50 = Parallel(n_jobs=2)([delayed(power_pecentiles)(pv,m,i,50) for i in
range(range_1,range_2)])
r_60  = Parallel(n_jobs=2)([delayed(power_pecentiles)(pv,m,i,60) for i in
range(range_1,range_2)])
r_70 = Parallel(n_jobs=2)([delayed(power_pecentiles)(pv,m,i,70) for i in
range(range_1,range_2)])
r_80 = Parallel(n_jobs=2)([delayed(power_pecentiles)(pv,m,i,80) for i in
range(range_1,range_2)])
r_90 = Parallel(n_jobs=2)([delayed(power_pecentiles)(pv,m,i,90) for i in
range(range_1,range_2)])

x = range(range_1,range_2)

plt.fill_between(x,r_10,y2=r_90)
plt.fill_between(x,r_20,y2=r_80)
plt.fill_between(x,r_30,y2=r_70)
plt.fill_between(x,r_40,y2=r_60)
plt.plot(r_50)

plt.savefig('percentiles.png')



##To give expectation +/- std dev graph
#std_dev_results = Parallel(n_jobs=-2)([delayed(power_std_dev)(pv,length,m,i) for i in
range(50)])
#exp_results = Parallel(n_jobs=-2)([delayed(power_exp)(length,pv,m,i) for i in range(50)])

#plt.plot(std_dev_results)
#plt.savefig('std_dev_results.png')

#plt.fill_between(x,[a-3*b for a,b in zip(exp_results,std_dev_results)],y2=[a+3*b for a,b in
zip(exp_results,std_dev_results)])
#plt.fill_between(x,[a-2*b for a,b in zip(exp_results,std_dev_results)],y2=[a+2*b for a,b in
zip(exp_results,std_dev_results)])
#plt.fill_between(x,[a-b for a,b in zip(exp_results,std_dev_results)],y2=[a+b for a,b in
zip(exp_results,std_dev_results)])
#plt.plot(exp_results)
#plt.savefig('std_dev.png')
```

```
### Gives percentage change of std dev and exp
#dif_exp = [100*(exp_results[i] - exp_results[i-1])/exp_results[i-1] for i in range(2,50)]
#dif_std_dev =  [100*(std_dev_results[i] - std_dev_results[i-1])/exp_results[i-1] for i in
range(2,50)]
#
#plt.subplot(121)
#plt.plot(range(2,50),dif_exp)
#plt.subplot(122)
#plt.plot(range(2,50), dif_std_dev)
#plt.savefig('std_dev_percent_change.png')

##Graphs IQR
#results = Parallel(n_jobs=-2)([delayed(power_iqr)(pv,m,i) for i in range(50)])
#plt.plot([item[0] for item in results]) #25
#plt.plot([item[1] for item in results]) #50
#plt.plot([item[2] for item in results]) #75
#plt.xlabel('Moves')
#plt.ylabel('God\'s number')
#plt.savefig('iqr.png')

##Graphs Heat Map
#results = Parallel(n_jobs=-2)([delayed(power_prob_heat)(pv,m,i) for i in range(50)])
#ax =
sns.heatmap(np.transpose(results),cmap="YlGnBu",yticklabels=False,annot=True,fmt="0.1f",an
not_kws={"fontsize":2})
#plt.savefig('test.png',dpi=300)
```

Updated Julia Code

```
##########################
# Adding Packages #
##########################
#import Pkg
#Pkg.add("Combinatorics")
#Pkg.add("BenchmarkTools")
#Pkg.add("LoopVectorization")
#Pkg.add("LinearAlgebra")
#Pkg.add("SparseArrays")
#Pkg.add("JLD2")
#Pkg.add("Logging")
##########################
# Includes #
##########################
using Combinatorics
using BenchmarkTools
using LoopVectorization
using LinearAlgebra
using JLD2
using SparseArrays
using Base.Threads: @threads, @sync, @spawn
using GraphRecipes, Plots
using Graphs
using Dates
using Logging
#########################
# Guido Dipietro - 2020 #
#########################
cp = (1,2,3,4,5,6,7,8) # like,speffz
co = (0,0,0,0,0,0,0,0) # 1 = clockwise
initial = (cp, co)
# Atoms
const Rp((a,b,c,d,e,f,g,h)) = (a,c,f,d,e,g,b,h)
const Ro(a) = @. (a + (0,1,2,0,0,1,2,0)) % 3
const Up((a,b,c,d,e,f,g,h)) = (d,a,b,c,e,f,g,h)
const xp((a,b,c,d,e,f,g,h)) = (d,c,f,e,h,g,b,a)
const xo(a) = @. (a + (2,1,2,1,2,1,2,1)) % 3
# Movedefs
const R((p,o)) = Rp(p), (Ro∘Rp)(o)
const U((p,o)) = Up(p), Up(o)
const x((p,o)) = xp(p), (xo∘xp)(o)
const D = x ∘ x ∘ U ∘ x ∘ x
const y = U ∘ D ∘ D ∘ D   ## Test if composition is saved, or matrix algebra done
const F = x ∘ x ∘ x ∘ U ∘ x
const B = y ∘ y ∘ F ∘ y ∘ y
const z = F ∘ B ∘ B ∘ B
const L = y ∘ y ∘ R ∘ y ∘ y
#########################
# Ryan Brown- 2022 #
#########################
#Log
io = open("log.txt","w+")
logger = SimpleLogger(io)
global_logger(logger)
mutable struct DIRECTED_GRAPH
   from::Int
   to::Vector{Int}
   times::Vector{Int8}
end
```

```
##Cube
mutable struct CUBE
   c1::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c2::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c3::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c4::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c5::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c6::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c7::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c8::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c9::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c10::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c11::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c12::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c13::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c14::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c15::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c16::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c17::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c18::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c19::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c20::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c21::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c22::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c23::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
   c24::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
end

##Initializes the cube
function INIT_CUBE()
   return
CUBE(initial,(transforms[1],(0,0,0,0,0,0,0,0)),(transforms[2],(0,0,0,0,0,0,0,0)),(transforms[3],(0,0
,0,0,0,0,0,0)),(transforms[4],(2,1,2,1,2,1,2,1)),(transforms[5],(2,1,2,1,2,1,2,1)),(transforms[6],(2,
1,2,1,2,1,2,1)),(transforms[7],(2,1,2,1,2,1,2,1)),(transforms[8],(1,2,1,2,1,2,1,2)) ,
(transforms[9],(1,2,1,2,1,2,1,2)) , (transforms[10],(1,2,1,2,1,2,1,2)),
(transforms[11],(1,2,1,2,1,2,1,2)) , (transforms[12],(2,1,2,1,2,1,2,1)),
(transforms[13],(2,1,2,1,2,1,2,1)), (transforms[14],(2,1,2,1,2,1,2,1)),
(transforms[15],(2,1,2,1,2,1,2,1)), (transforms[16],(1,2,1,2,1,2,1,2)),
(transforms[17],(1,2,1,2,1,2,1,2)), (transforms[18],(1,2,1,2,1,2,1,2)),
(transforms[19],(1,2,1,2,1,2,1,2)), (transforms[20],(0,0,0,0,0,0,0,0)),
(transforms[21],(0,0,0,0,0,0,0,0)), (transforms[22],(0,0,0,0,0,0,0,0)),
(transforms[23],(0,0,0,0,0,0,0,0)))
end
```

```
##Does the turns to each cube
function turns(before::CUBE,move)
    after = INIT_CUBE()
    after.c1 = (move)(before.c1)
    after.c2 = (move)(before.c2)
    after.c3 = (move)(before.c3)
    after.c4 = (move)(before.c4)
    after.c5 = (move)(before.c5)
    after.c6 = (move)(before.c6)
    after.c7 = (move)(before.c7)
    after.c8 = (move)(before.c8)
    after.c9 = (move)(before.c9)
    after.c10 = (move)(before.c10)
    after.c11 = (move)(before.c11)
    after.c12 = (move)(before.c12)
    after.c13 = (move)(before.c13)
    after.c14 = (move)(before.c14)
    after.c15 = (move)(before.c15)
    after.c16 = (move)(before.c16)
    after.c17 = (move)(before.c17)
    after.c18 = (move)(before.c18)
    after.c19 = (move)(before.c19)
    after.c20 = (move)(before.c20)
    after.c21 = (move)(before.c21)
    after.c22 = (move)(before.c22)
    after.c23 = (move)(before.c23)
    after.c24 = (move)(before.c24)
    return after
end

##Orientates the cube, unnecessary when only using 3 generators
function orientate_CUBE(cube::CUBE)
    c = INIT_CUBE()
    c.c1 = orientate_cube(cube.c1)
    c.c2 = orientate_cube(cube.c2)
    c.c3 = orientate_cube(cube.c3)
    c.c4 = orientate_cube(cube.c4)
    c.c5 = orientate_cube(cube.c5)
    c.c6 = orientate_cube(cube.c6)
    c.c7 = orientate_cube(cube.c7)
    c.c8 = orientate_cube(cube.c8)
    c.c9 = orientate_cube(cube.c9)
    c.c10 = orientate_cube(cube.c10)
    c.c11 = orientate_cube(cube.c11)
    c.c12 = orientate_cube(cube.c12)
    c.c13 = orientate_cube(cube.c13)
    c.c14 = orientate_cube(cube.c14)
    c.c15 = orientate_cube(cube.c15)
    c.c16 = orientate_cube(cube.c16)
    c.c17 = orientate_cube(cube.c17)
    c.c18 = orientate_cube(cube.c18)
    c.c19 = orientate_cube(cube.c19)
    c.c20 = orientate_cube(cube.c20)
    c.c21 = orientate_cube(cube.c21)
    c.c22 = orientate_cube(cube.c22)
    c.c23 = orientate_cube(cube.c23)
    c.c24 = orientate_cube(cube.c24)
    return c
end

#Constants
##To Print
color_table =
[['y','g','o'],['y','r','g'],['y','b','r'],['y','o','b'],['w','b','o'],['w','r','b'],['w','g','r'],['w','o','g']]
const transforms = [
    ##Not Shown Y->Y, B->B
    (4,1,2,3,6,7,8,5), ##Y->Y B->O
    (3,4,1,2,7,8,5,6), ## B->G
    (2,3,4,1,8,5,6,7), ## B->R
    (6,5,4,3,2,1,8,7), ## Y->B, B->Y
    (5,8,1,4,3,2,7,6), ##B->O
    (8,7,2,1,4,3,6,5), ##B->W
    (7,6,3,2,1,4,5,8), ## B->R
    (1,8,7,2,3,6,5,4), ## Y->O, B->B
    (2,7,6,3,4,5,8,1), ## B->Y
    (3,6,5,4,1,8,7,2), ## B->G
    (4,5,8,1,2,7,6,3), ## B->W
    (2,1,8,7,6,5,4,3), ## Y->G, B->W
    (3,2,7,6,5,8,1,4), ##B->O
    (4,3,6,5,8,7,2,1), ## B->Y
    (1,4,5,8,7,6,3,2), ## B-> R
    (8,1,4,5,6,3,2,7), ## Y->R,B->B
    (7,2,1,8,5,4,3,6), ## B -> Y
    (5,4,3,6,7,2,1,8), ## B -> G
    (6,3,2,7,8,1,4,5),  ## B -> W
    (7,8,5,6,3,4,1,2), ## Y->w, B->B
    (6,7,8,5,4,1,2,3), ## B->O
    (5,6,7,8,1,2,3,4), ## B ->g
    (8,5,6,7,2,3,4,1)] ## B ->r
const arr_moves = [R,R∘R∘R,D,D∘D∘D,B,B∘B∘B]
  const Arr1 = collect(permutations((1,2,3,4,5,6,7,8)));

  ##Position is:
  #0-Top/Bottom
  #1-Right of Top
  #2-Left of Top
  function color(position, cubie,orientation)
     return color_table[cubie][1+mod(orientation+position,3)]
  end

  ##To Print
  function print_cube(cube)
    print_cube(cube[1],cube[2])
  end

  ##Prints the cube
  function print_cube(cp,co)
    print("   \\  ",color(0,cp[1],co[1]),' ',color(0,cp[2],co[2]))
    print("  /","",'\n',"    \\ ",color(0,cp[4],co[4]),' ',color(0,cp[3],co[3])," / ",'\n')
    print(color(2,cp[1],co[1])," ",color(1,cp[4],co[4])," | ",color(2,cp[4],co[4]),"
  ",color(1,cp[3],co[3])," | ",color(2,cp[3],co[3])," ",color(1,cp[2],co[2]),'\n')
    print(color(1,cp[8],co[8])," ",color(2,cp[5],co[5])," | ",color(1,cp[5],co[5]),"
  ",color(2,cp[6],co[6])," | ",color(1,cp[6],co[6])," ",color(2,cp[7],co[7]),'\n')
    print("    / ",color(0,cp[5],co[5])," ",color(0,cp[6],co[6])," \\",'\n')
    print("   /  ",color(0,cp[8],co[8])," ",color(0,cp[7],co[7]),"  \\ ",'\n','\n')
    print("      ",color(1,cp[1],co[1])," ",color(2,cp[2],co[2]),'\n')
    print("      ",color(2,cp[8],co[8])," ",color(1,cp[7],co[7]))
  end

  # Orientate cube by cubie 4 in position 4, yellow ontop
  function orientate_cube(cu)
    if(cu[1][4]==4)
    elseif(cu[1][3] == 4 || cu[1][6]==4 || cu[1][5]==4)
       while(cu[1][4]!=4)
           cu = z(cu)
       end
    elseif(cu[1][1]==4 || cu[1][8]==4)
       while(cu[1][4]!=4)
        cu = x(cu)
       end
    else
       cu = y(cu)
        return orientate_cube(cu)
    end
    if(cu[2][4] == 0)
        return cu
    elseif(cu[2][4]==1)
        cu = (y∘y∘y∘x)(cu) #y inverse
    else
        cu = (x∘x∘x∘y)(cu) #x inverse
    end
    return cu
end


##CANONICAL REPRESENTATIVES

function canonical_representative(cube::CUBE)
    cube_n = orientate_CUBE(cube)
    class = Vector(undef,24)
    class[1] = cube_to_pos(cube_n.c1)+1
    class[2] = cube_to_pos(cube_n.c2)+1
    class[3] = cube_to_pos(cube_n.c3)+1
    class[4] = cube_to_pos(cube_n.c4)+1
    class[5] = cube_to_pos(cube_n.c5)+1
    class[6] = cube_to_pos(cube_n.c6)+1
    class[7] = cube_to_pos(cube_n.c7)+1
    class[8] = cube_to_pos(cube_n.c8)+1
    class[9] = cube_to_pos(cube_n.c9)+1
    class[10] = cube_to_pos(cube_n.c10)+1
    class[11] = cube_to_pos(cube_n.c11)+1
    class[12] = cube_to_pos(cube_n.c12)+1
    class[13] = cube_to_pos(cube_n.c13)+1
    class[14] = cube_to_pos(cube_n.c14)+1
    class[15] = cube_to_pos(cube_n.c15)+1
    class[16] = cube_to_pos(cube_n.c16)+1
    class[17] = cube_to_pos(cube_n.c17)+1
    class[18] = cube_to_pos(cube_n.c18)+1
    class[19] = cube_to_pos(cube_n.c19)+1
    class[20] = cube_to_pos(cube_n.c20)+1
    class[21] = cube_to_pos(cube_n.c21)+1
    class[22] = cube_to_pos(cube_n.c22)+1
    class[23] = cube_to_pos(cube_n.c23)+1
    class[24] = cube_to_pos(cube_n.c24)+1
    return minimum(class)
end
```

```
function number_to_trinary_array(num)
   array = [0,0,0,0,0,0,0,0]
   for i in 1:8
      array[i]=num%3
      num = floor(num/3)
   end
   return Tuple(array)
end

function trinary_array_to_number(array)
   num =0;
   for i in 1:8
      num += array[i]*3^(i-1)
   end
   return num
end

function pos_to_cube(pos)
  or_arr = number_to_trinary_array(pos%(3^8))
  pos_arr = Tuple(Arr1[convert(Int,floor(pos/(3^8))+1)])
  return (pos_arr,or_arr)
end

function decrement_array(array,num)
   array=collect(array)
   for i in 1:(size(array)[1])
      if(array[i]>num)
         array[i] = array[i]-1
      end
   end
   return Tuple(array)
end

function array_to_num(array)
   pos= (array[1]-findmin(array)[1])*factorial(7)
   for i in 1:6
      array = decrement_array(array[2:(9-i)],array[1])
      pos += (array[1]-findmin(array)[1])*factorial(7-i)
   end
   return pos
end

function cube_to_pos(cube)
```

```
   return array_to_num(cube[1])*3^8+trinary_array_to_number(cube[2])
end

function moves(cube)
    array_pos = Vector{Int}(undef, 12)
    @threads for i in 1:12
        array_pos[i] = cube_to_pos((arr_moves[i])(cube))+1
    end
    return array_pos
end

function add_moves(v)
    new_v = Vector(undef,12)
    @threads for i in 1:12
        new_v[i] = v∘arr_moves[i]
    end
    return new_v
end


function add_to_directed_graph(cube_pos, array_pos, dir)
    d_n = DIRECTED_GRAPH(cube_pos,unique(array_pos),[count(==(element),array_pos) for
element in unique(array_pos) ])
    push!(dir,d_n)
    return dir
end



## Recursively Adds, Took too much time, uses a lot of memory
#function recursive_add(cube,directed_graph,depth,deep)
#    cube_pos = cube_to_pos(cube);
#    if(!isempty(nonzeros(directed_graph[cube_pos+1,:])))
#        return
#    end
#    if(depth >= deep) return end
#   for i in 1:12
#        new_cube = arr_moves[i](cube)
#        directed_graph[cube_pos+1,cube_to_pos(new_cube)+1]=1
#        recursive_add(new_cube,directed_graph,depth+1,deep)
#    end
#end
#
#function generate_recursive(deep)
```

```
#    directed_graph = spzeros(264539520,264539520)
#    recursive_add(pos_to_cube(0),directed_graph,0,deep)
#    return directed_graph
#end

function generate_itterative(depth)
    been_to = BitVector(undef,264539520)
    cube = INIT_CUBE()
    been_to[1] = 1
    output = arr_moves
    output_cubes = Vector{Int}(undef,0)
    oc_multiples = Vector{Int}(undef,0)
    output_new = Vector{Any}(undef,0)
    for i in 1:6
        new = canonical_representative(turns(cube,output[i]))
        if (been_to[new] == 0)
            push!(output_cubes,new)
            push!(output_new,output[i])
            push!(oc_multiples,1)
            been_to[new] = 1
        else
            oc_multiples[findall(x->x==new,output_cubes)[1]]+=1
        end
    end
        @info "Completed 1"
            print("Completed 1\n")
            flush(io)
         jldsave("1.jld2"; output_cubes, oc_multiples, d=0)
            ouput_current = []
            i=depth
            depth =1
         jldsave("resume.jld2";been_to,ouput_current, output_new,depth)
            depth = i
    for  i in  1:depth
        if(!isempty(output_new))
            output = output_new
            output_new = Vector{Any}(undef,0)
            for j in 1:length(output)
                oc_multiples = Vector{Int}(undef,0)
                output_cubes =  Vector{Int}(undef,0)
                for k in 1:6
                    new = canonical_representative(turns(cube,output[j]∘arr_moves[k]))
                    if(been_to[new]==0)
                        push!(output_cubes,new)
```

```
                  push!(output_new,output[j]∘arr_moves[k])
                  push!(oc_multiples,1)
                  been_to[new] = 1
               elseif (new in output_cubes)
                  oc_multiples[findall(x->x==new,output_cubes)[1]]+=1
               else
                  push!(output_cubes,new)
                  push!(oc_multiples,1)
               end
            end
            str= string("Completed ",canonical_representative(turns(cube,output[j]))," at Depth
",i, " time ", now())
                    @info str
                    print(str,'\n')
                    flush(io)
            jldsave(string(canonical_representative(turns(cube,output[j])),".jld2"); output_cubes,
oc_multiples, d=i)
            ouput_current = output[j:length(output)]
                    depth = i
            jldsave("resume.jld2";been_to, ouput_current, output_new,depth)
                    jldsave("been_to.jld2";been_to)
         end
      end
   end
end

function resume()
   cube = INIT_CUBE()
   been_to = load("resume.jld2","been_to")
   ouput_current = load("resume.jld2","ouput_current")
   output_new  = load("resume.jld2","output_new")
   depth = load("resume.jld2","depth")
   if(!isempty(ouput_current))
      output = ouput_current
       @threads for j in 1:length(output)
            oc_multiples = Vector{Int}(undef,0)
            output_cubes =  Vector{Int}(undef,0)
            for k in 1:6
               new = canonical_representative(turns(cube,output[j]∘arr_moves[k]))
               if(been_to[new]==0)
                  push!(output_cubes,new)
                  push!(output_new,output[j]∘arr_moves[k])
                  push!(oc_multiples,1)
                  been_to[new] = 1
```

```
                elseif (new in output_cubes)
                    oc_multiples[findall(x->x==new,output_cubes)[1]]+=1
                else
                    push!(output_cubes,new)
                    push!(oc_multiples,1)
                end
            end
            str= string("Completed ",canonical_representative(turns(cube,output[j]))," at Depth
",depth, " time ", now())
                    @info str
                    print(str,'\n')
                    flush(io)
            jldsave(string(canonical_representative(turns(cube,output[j])),".jld2"); output_cubes,
oc_multiples, d=depth)
            ouput_current = output[j:length(output)]
            jldsave("resume.jld2";been_to, ouput_current, output_new,depth)
                    jldsave("been_to.jld2";been_to)
        end
  end
   for  i in  (depth+1):20
      if(!isempty(output_new))
        output = output_new
        output_new = Vector{Any}(undef,0)
        @threads for j in 1:length(output)
            oc_multiples = Vector{Int}(undef,0)
            output_cubes =  Vector{Int}(undef,0)
            for k in 1:6
                new = canonical_representative(turns(cube,output[j]∘arr_moves[k]))
                if(been_to[new]==0)
                    push!(output_cubes,new)
                    push!(output_new,output[j]∘arr_moves[k])
                    push!(oc_multiples,1)
                    been_to[new] = 1
                elseif (new in output_cubes)
                    oc_multiples[findall(x->x==new,output_cubes)[1]]+=1
                else
                    push!(output_cubes,new)
                    push!(oc_multiples,1)
                end
            end
                    depth = i
            str= string("Completed ",canonical_representative(turns(cube,output[j]))," at Depth
",i, " time ", now())
                    @info str
                    print(str,'\n')
                    flush(io)
            jldsave(string(canonical_representative(turns(cube,output[j])),".jld2"); output_cubes,
oc_multiples, depth)
            ouput_current = output[j:length(output)]
            jldsave("resume.jld2";been_to, ouput_current, output_new,depth)
                    jldsave("been_to.jld2";been_to)
         end
      end
   end

end



function next_level()
   cube = INIT_CUBE()
   been_to = load("resume.jld2","been_to")
   output_new  = load("resume.jld2","output_new")
   depth = load("resume.jld2","depth")
   for  i in  depth:(depth+1)
      if(!isempty(output_new))
         output = output_new
         output_new = Vector{Any}(undef,0)
         @threads for j in 1:length(output)
            oc_multiples = Vector{Int}(undef,0)
            output_cubes =  Vector{Int}(undef,0)
            for k in 1:6
               new = canonical_representative(turns(cube,output[j]∘arr_moves[k]))
               if(been_to[new]==0)
                  push!(output_cubes,new)
                  push!(output_new,output[j]∘arr_moves[k])
                  push!(oc_multiples,1)
                  been_to[new] = 1
               elseif (new in output_cubes)
                  oc_multiples[findall(x->x==new,output_cubes)[1]]+=1
               else
                  push!(output_cubes,new)
                  push!(oc_multiples,1)
               end
            end
            @info string("Completed ",canonical_representative(turns(cube,output[j])))
            jldsave(string(canonical_representative(turns(cube,output[j])),".jld2"); output_cubes,
oc_multiples, d=i)
```

```
            ouput_current = output[j:length(output)]
            jldsave("resume.jld2";been_to, ouput_current, output_new,depth,output,j)
         end
      end
   end
end

function graph(depth)
   ##Directed Graph Setup
   directed_graph = spzeros(264539520,264539520)
   counter = 1
   nodes = [1]
   new_nodes = Vector{Int}(undef,0)
   new_weights = Vector{Int}(undef,0)
   for d in 0:depth
      for counter in counter:length(nodes)
         empty!(new_nodes)
         empty!(new_weights)
         name = string(nodes[counter],".jld2")
         new_nodes = load(name,"output_cubes")
         new_weights = load(name,"oc_multiples")
         for i in 1:length(new_nodes)
            if (!(new_nodes[i] in nodes))
               push!(nodes,new_nodes[i])
            end
            directed_graph[nodes[counter],new_nodes[i]] = new_weights[i]
         end
      end
      counter +=1
   end
      ##Reduction of directed graph
   name_vec = unique(Vcat(findnz(directed_graph)[1],findnz(directed_graph)[2]))
   count = length(name_vec)
   name_dic = Dict()
   for i in 1:count
      name_dic[name_vec[i]]=i
   end
   reduced_graph = spzeros(count,count)
   entry_count = length(findnz(directed_graph)[1])
   for i in 1:entry_count

reduced_graph[name_dic[findnz(directed_graph)[1][i]],name_dic[findnz(directed_graph)[2][i]]]
=findnz(directed_graph)[3][i]
   end
   #edgelabel_dict = Dict()
   #for i in 1:count
   #    for j in 1:count
   #        edgelabel_dict[(i, j)]= string(reduced_graph[i,j])
   #    end
   #end
   graphplot(reduced_graph,names=name_vec, shorten=0.01) #edgelabel=edgelabel_dict)
end
```