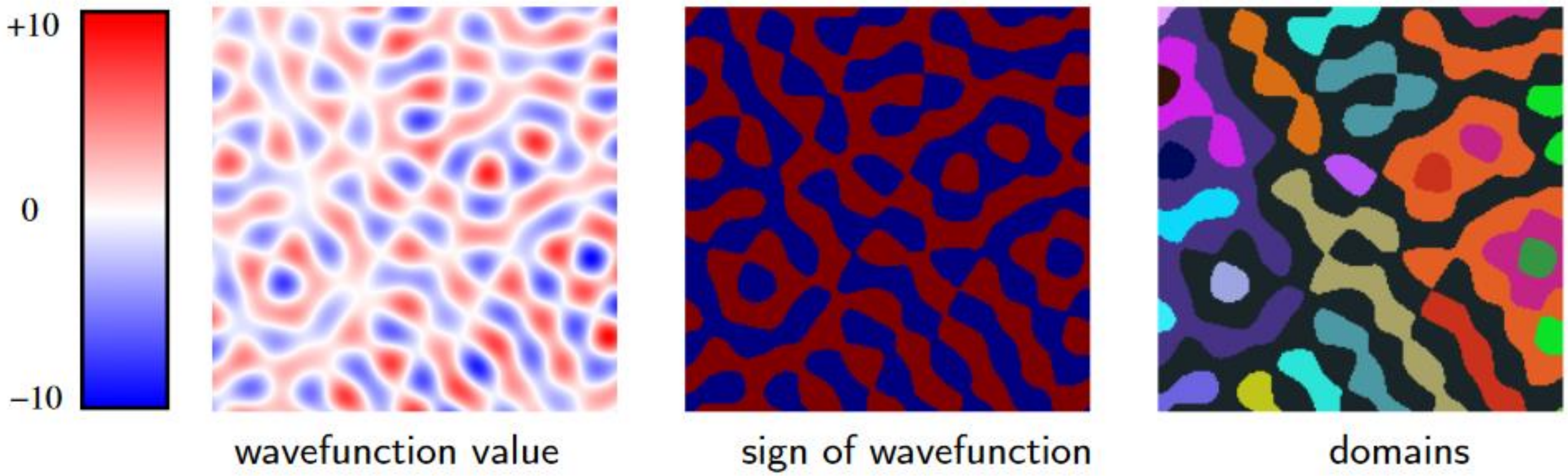
A Brief Introduction to Hoshen-Kopelman Algorithm For Solving Percolation Problem

Final Project of Computational Physics Course

Kuo Ting-Kai, Department of Economics , NCCU

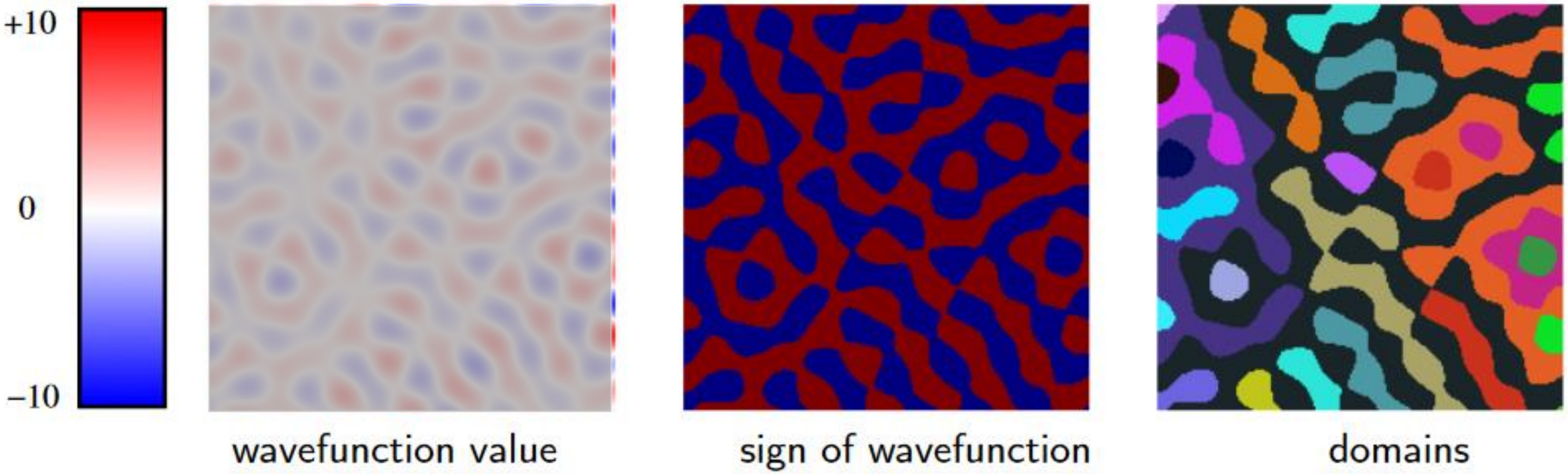
July 4th,2018

Application: Nodal Domains



Application: Nodal Domains

Percolation Problem is similar to these two cases (discret value)

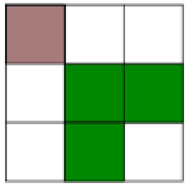


Definitions in H-K Algorithm

- Grid of m*n sites : m is the length of grid,n is the width of grid. For simplicity, set m equal to n.
- Value at these sites may be:

  1.Percolation clusters: only 1(occuplied site) and 0 (free site) in the grid in the problem which I focused on.

  2.Spin value: +1(spin up) or -1(spin down), it also can be applied to Swendsen-Wang algorithm, an efficient variation of Metropolis- Hasting algorithm for solving ground state in Ising model.



normal HK

- Cluster Criterion: only consider above, bottom, left-side and right-side neighbors, exclude diagonal ones.
- Percolation rate(denoted by “p”): the probability each site be occupied. The percolation rates of every sites are mutually independent.

HK Algorithm in C-like Pseudo Code

```
Raster Scan and Labeling on the Grid
largest_label = 0;
for x in 0 to n_columns {
    for y in 0 to n_rows {
        if occupied[x,y] then
            left = occupied[x-1,y];
            above = occupied[x,y-1];
            if (left == 0) and (above == 0) then /* Neither a label above nor to the left. */
                largest_label = largest_label + 1; /* Make a new, as-yet-unused cluster label. */
                label[x,y] = largest_label;
            else if (left != 0) and (above == 0) then /* One neighbor, to the left. */
                label[x,y] = find(left);
            else if (left == 0) and (above != 0) then /* One neighbor, above. */
                label[x,y] = find(above);
            else /* Neighbors BOTH to the left and above. */
                union(left,above); /* Link the left and above clusters. */
                label[x,y] = find(left);
    }
}
```

HK Algorithm in C-like Pseudo Code

```
Union
void union(int x, int y)  {
    labels[find(x)] = find(y);
}
```

```
Find
int find(int x)  {
    int y = x;
    while (labels[y] != y)
        y = labels[y];
    while (labels[x] != x)  {
        int z = labels[x];
        labels[x] = y;
        x = z;
    }
return y;
}
```

Demo

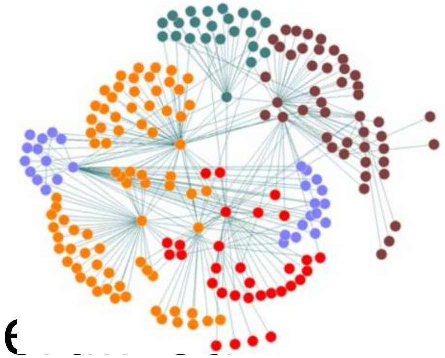
```
Original Graph:
0 1 0 1 0 1 1 1 0 1

1 1 0 0 0 1 0 0 1 0

1 0 0 0 1 1 1 0 1 1

1 1 1 1 0 0 1 0 0 0

1 1 1 1 0 0 0 1 0 1

1 0 0 1 0 1 1 0 0 0

0 0 1 1 0 1 1 0 0 1

1 1 0 1 0 1 0 0 1 0

0 0 0 0 1 1 1 1 1 1

0 0 0 1 1 1 1 1 0 0
```

Demo

```
Labeled Graph:
  2   W   W W W   )

2 2       W     2

2       W W W   2 2

2 2 2 2     W

2 2 2 2       O   2

2     2   W W

    2 2   W W     e

f f   2   W     W

        W W W W W W

      W W W W W
```

Given p=0.5,n=10
Max. Cluster: W
Size: 26

Our goal



- Main algorithm : label the clusters on a grid (a grid can be gene
  to a network, or we can say, a graph)
- Given p and n, find the maximum size of cluster(denote by “mc”) on a grid (In detail, the average sizes of maximum clusters in the grids with same p and n generated in random)
- Then derive the relationship between p and mc given some n.

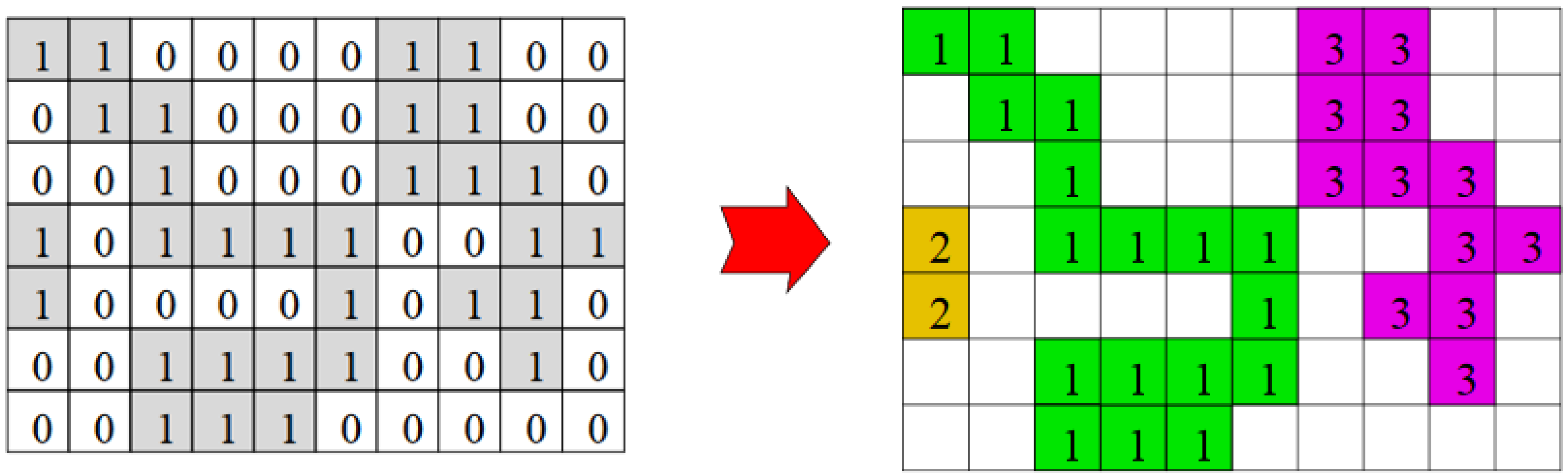
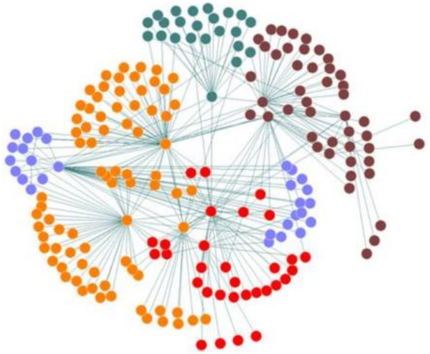


Image Resource :
Christian Joas‘s
lecture note,
Institut f˙ur theoretische
Physik, FU Berlin

Our goal



- Then derive the relationship between p and mc given some n.
- Find the point of phase transition ,p*:

  p<p*: size of maximum cluster is small

  p>p*: size of maximum cluster become big

1	1	0	0	0	0	1	1	0	0
0	1	1	0	0	0	1	1	0	0
0	0	1	0	0	0	1	1	1	0
1	0	1	1	1	1	0	0	1	1
1	0	0	0	0	1	0	1	1	0
0	0	1	1	1	1	0	0	1	0
0	0	1	1	1	0	0	0	0	0

1	1					3	3		
	1	1				3	3		
		1				3	3	3	
2		1	1	1	1			3	3
2					1		3	3	
		1	1	1	1			3	
		1	1	1					

Image Resource :
Christian Joas‘s
lecture note,
Institut f ̈ur theoretische
Physik, FU Berlin

The implementation of Hoshen-Kopelman algorithm in Python

- Let’s display the components with hierarchical order, which comprised my implementation of HK algorithm, named “HK.py”:

```
HK.py
    |__1. class QuickUnion ( UnionFindBase ):  for operating the algorithm
          |__ 1.a function “__init__”: initialize unite labels() and grid
          |__ 1.b function “root”  : find the root of a occupied site (all sites with
                same root have a same label )
          |__ 1.c function “find” :  check whether two occupied site have same root
          |__ 1.d function “unite” : unite two site in a same cluster
```

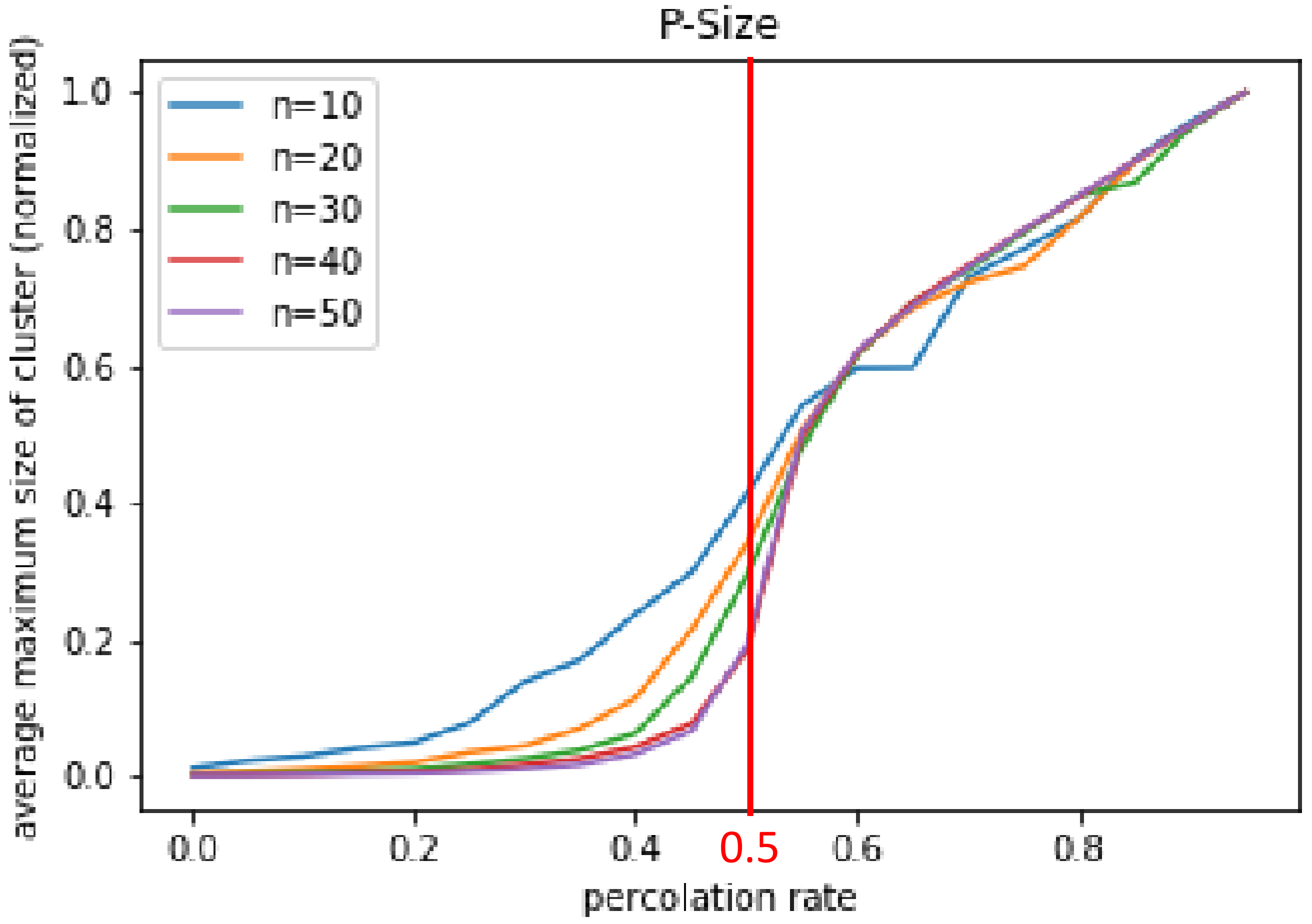
The implementation of Hoshen-Kopelman algorithm in Python

(continue)

|__2. function “original graph”: plot the intitial grid
|__3. function “labeled graph”: show the grid with labels
|__4. function “max_cluster”: show the maximum cluster
|__5. function “hoshenKopelman”: core algorithm
|__6. main routine : show the relationship between p and mc
given some n.

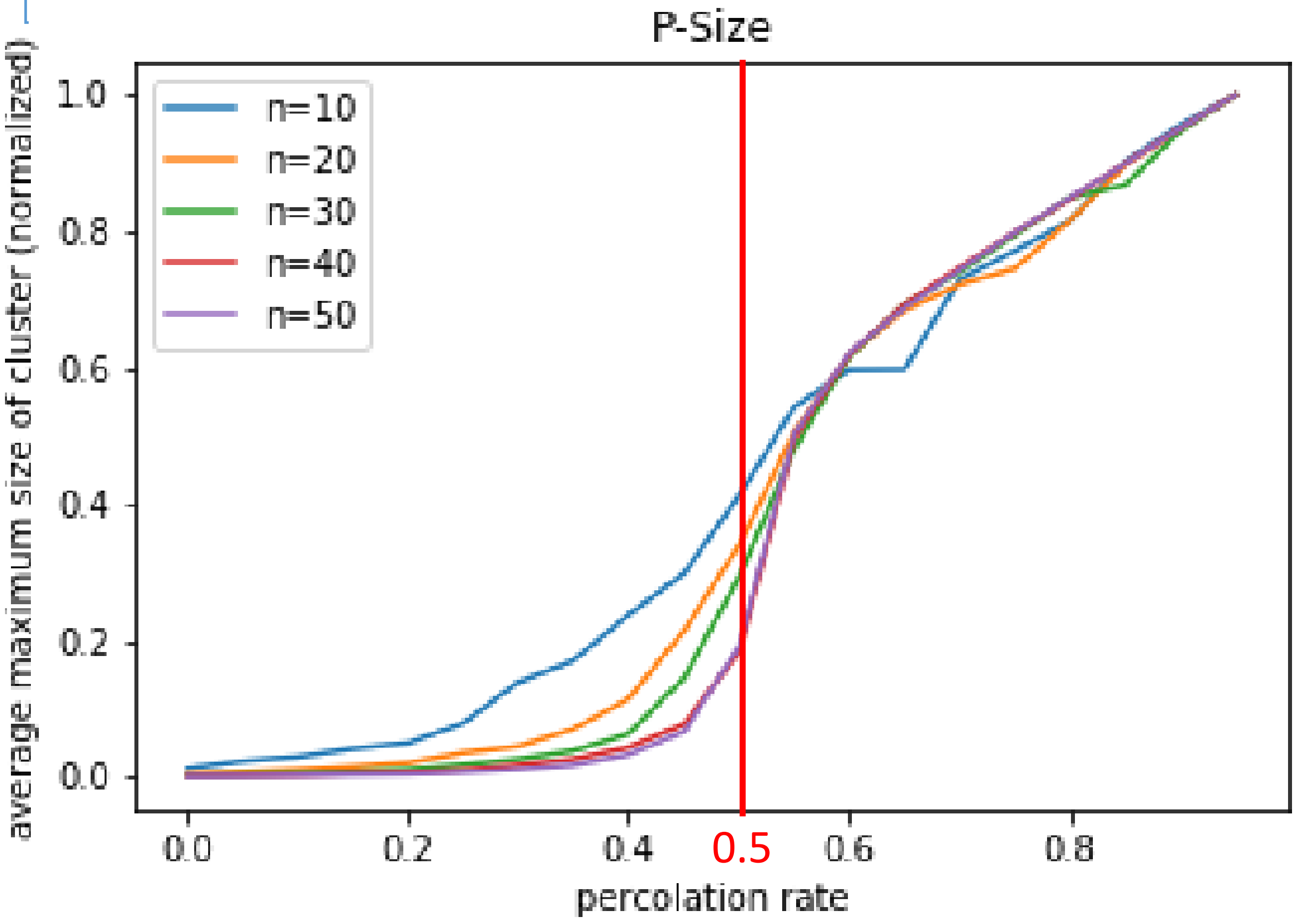
If want to know some details, see the appendix.

Graphic exploration and conclusion



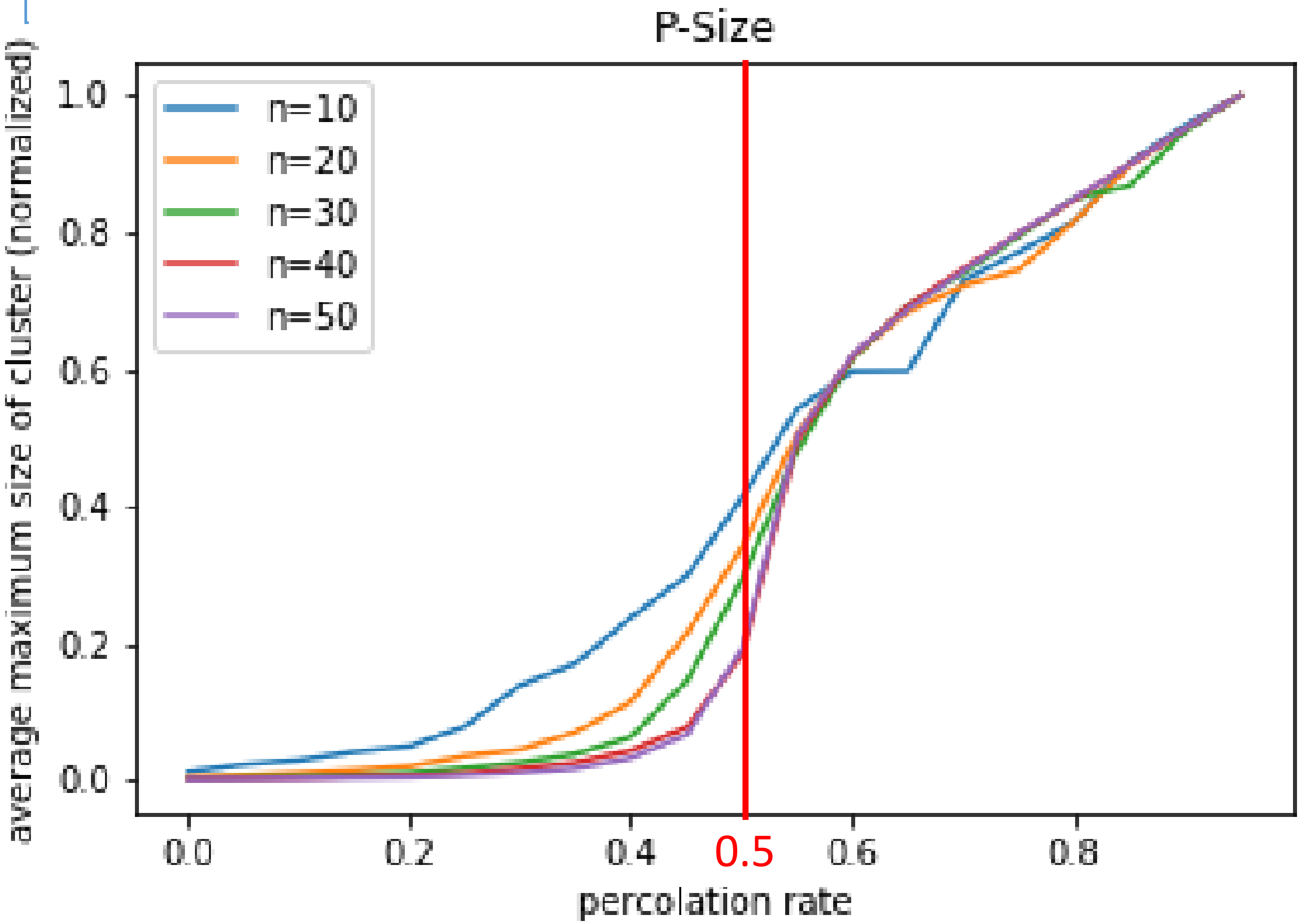
Graphic exploration and conclusion

"Normalized" means divide the "Size"(average maximum size of cluster) by n*n (scale of grid)



Graphic exploration and conclusion

“Normalized” means divide the “Size”(average maximum size of cluster) by n*n (scale of grid)

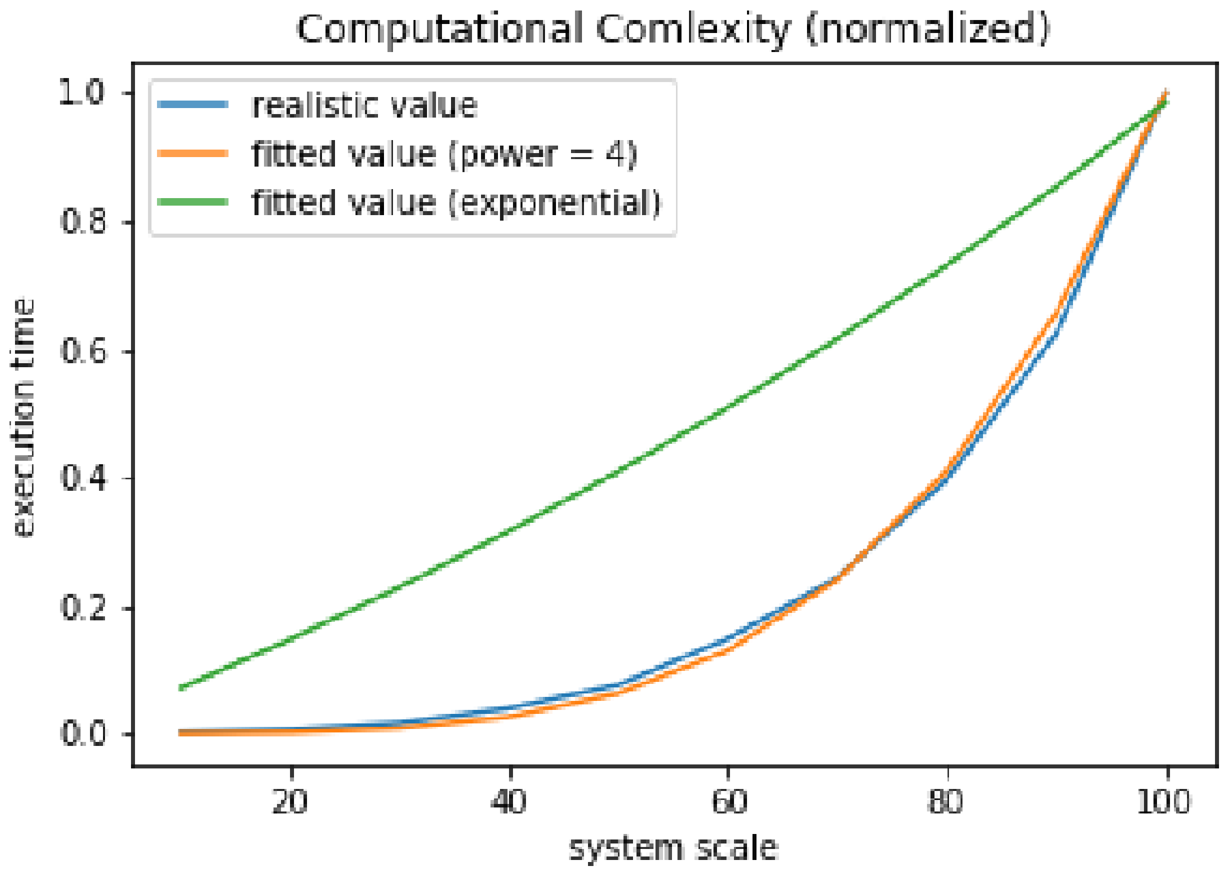


Conclusion:

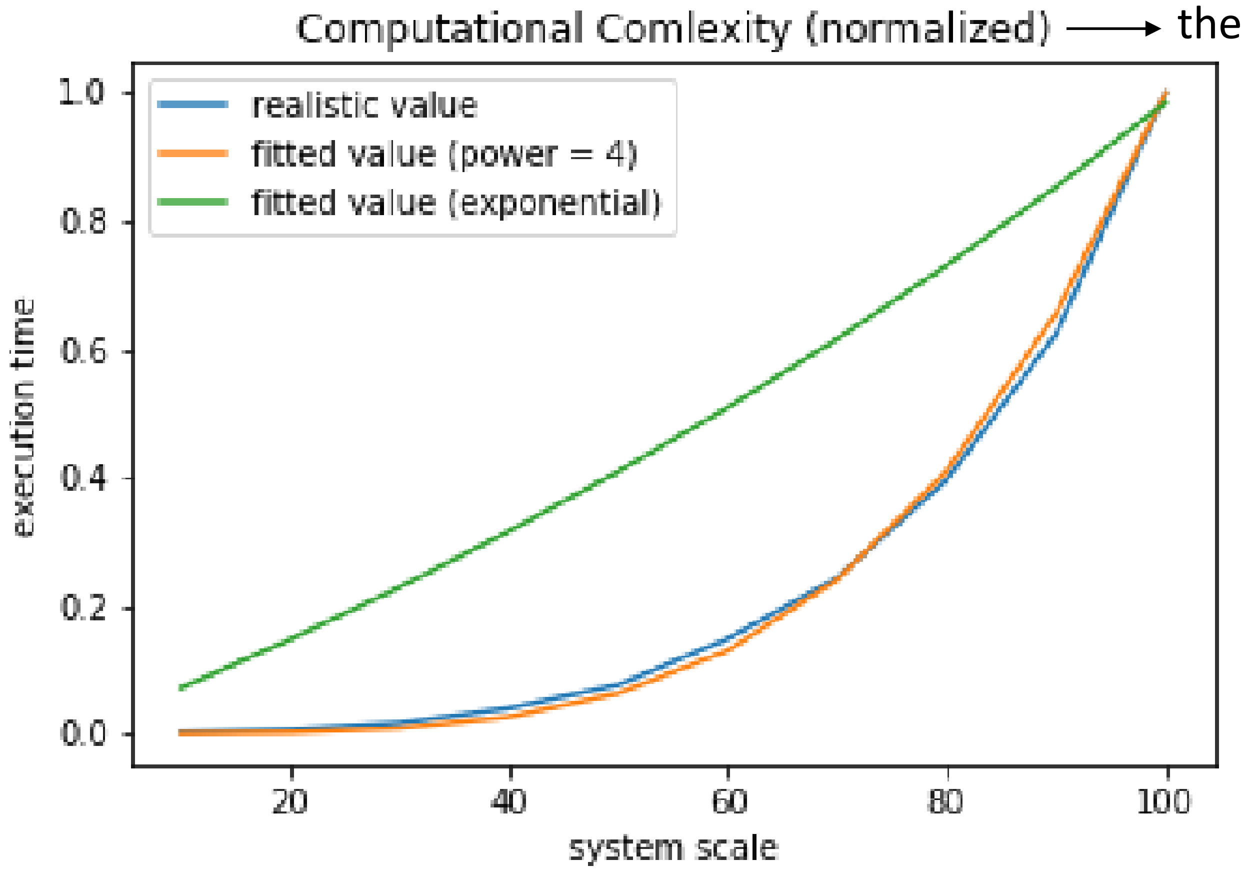
1.The point of phase transition point is at p* = 0.5. It is a vert intuitive result.

2.As n increases, the phase transition occurs more rapidly, and the behavior of curve tend to be vertical nearby the p*.

Computational Complexity Analysis

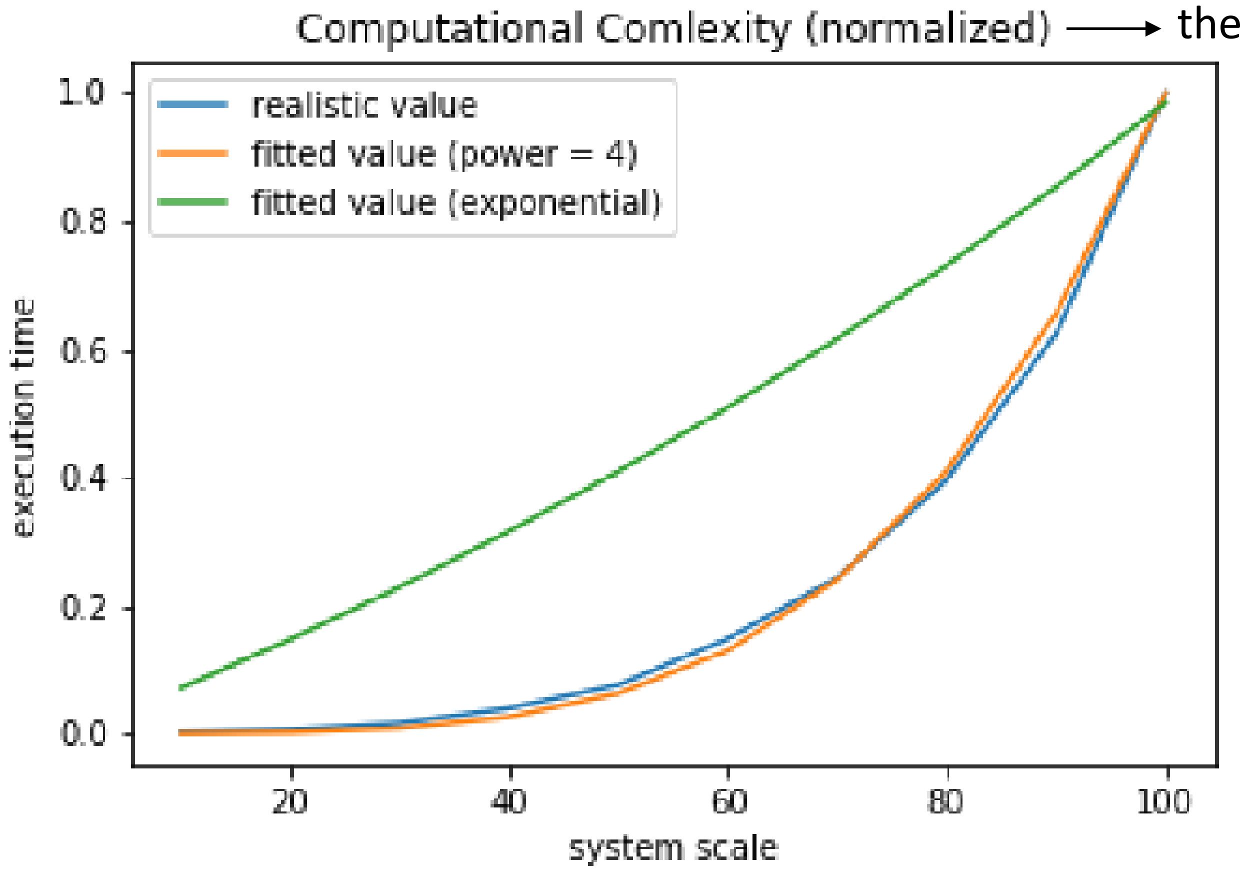


Computational Complexity Analysis



"normalized" means execution times (t) are all divided by the upper bound value at scale equals to 100 , t(n=100)

Computational Complexity Analysis



“normalized” means execution times (t) are all divided by the upper bound value at scale equals to 100 , t(n=100)

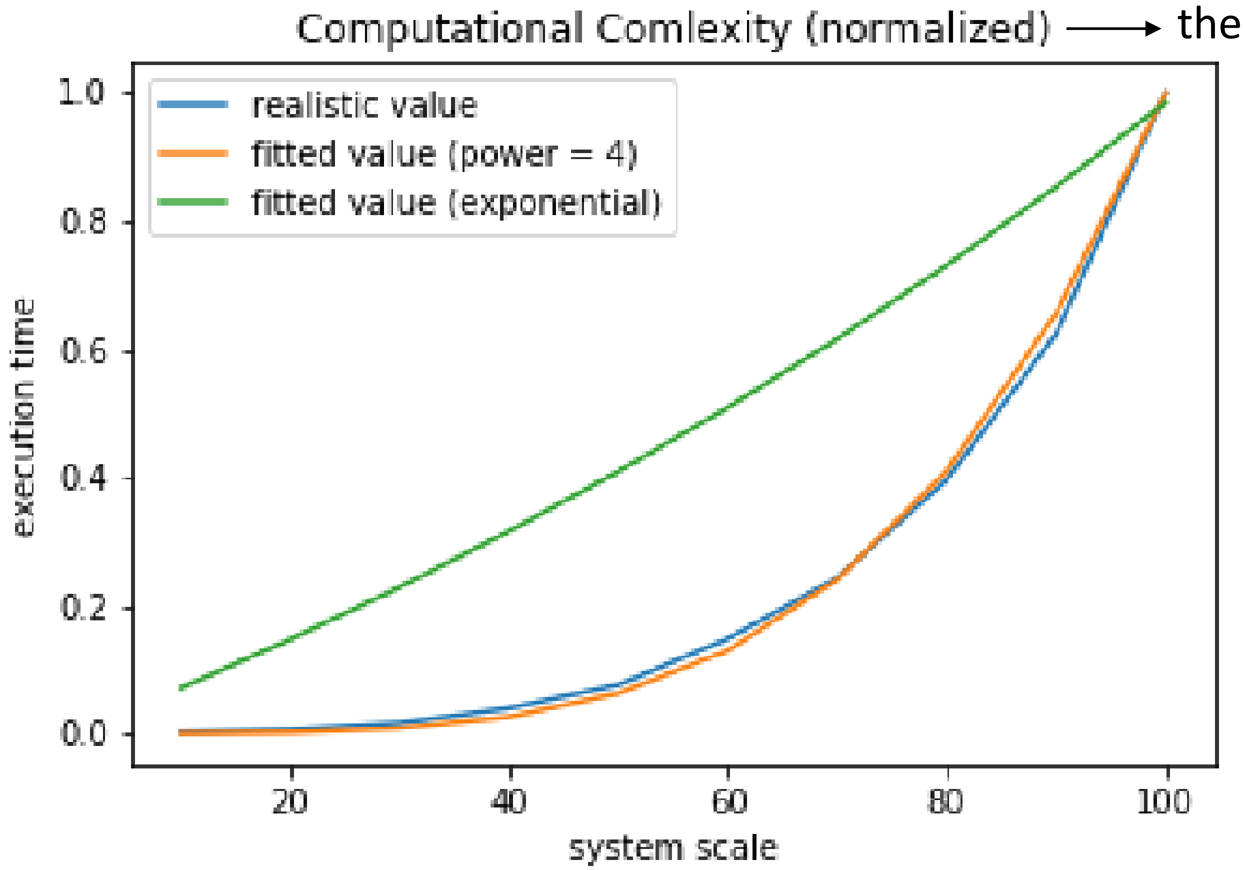
Try to fit the curve:

Green curve: f(n) = exp(n^2/(100^2*log(4.2)))-1

Orange curve: g(n) = 10^(-8)*n^4

n: system scale

Computational Complexity Analysis



"normalized" means execution times (t) are all divided by the upper bound value at scale equals to 100 , t(n=100)

Try to fit the curve:

Green curve: f(n) = exp(n^2/(100^2*log(4.2)))-1

Orange curve: g(n) = 10^(-8)*n^4

n: system scale

Observation:

1.When scale is within [0,100]:

The growth rate of execution time approximates to a quartic function g(n) ~n^4,

and less than f(n) ~exp(n^2))

2.In the whole picture (including n>100):

the growth rate is faster than f(n) ~exp(n^2)

Appendix: Source code

```python
import time
from IPython.display import clear_output
import random
import numpy as np
from matplotlib import pyplot as plt

class UnionFindBase(object):
    def __init__(self, N,p):
        self.id = [None] * N
        self.occupied = [None]*N
        for i in range(N):
            self.id[i] = i
            if random.uniform(0,1)>=p:
                self.occupied[i] = 1
            else:
                self.occupied[i] = 0

class QuickUnion(UnionFindBase):
    def __init__(self, N,p):
        super(QuickUnion, self).__init__(N,p)

    def root(self, i):
        while i != self.id[i]:
            i = self.id[i]
        return i

    def find(self, p, q):
        return self.root(p) == self.root(q)

    def unite(self, p, q): #q:變成集合的root
        i = self.root(p)
        j = self.root(q)
        self.id[i] = j

def original_graph(ni,nj,graph):
    print('Original Graph:')
    for i in range(ni):
        for j in range(nj):
            node = nj*i+j
            print(graph[node],end=' ')
        print('\n')
    print('\n\n')
```

Appendix: Source code

```python
def labeled_graph(ni,nj,graph_class,plot=False):
    labels = []
    if plot:
        print('Labeled Graph:')
    for i in range(ni):
        for j in range(nj):
            if graph_class.occupied[nj*i+j]==1:
                if plot:
                    print(chr(32+graph_class.root(nj*i+j)),end=' ')
                label = chr(32+graph_class.root(nj*i+j))
            else:
                if plot:
                    print(' ',end=' ')
                label = ' '
            labels.append(label)
        if plot:
            print('\n')
    return labels


def max_cluster(labels,isprint=False):
    max_cluster = None
    for l in list(set(labels)):
        if labels.count(l)>labels.count(max_cluster) and l != ' ':
            max_cluster = l
    if isprint:
        print('Max Cluster:',max_cluster)
    size = labels.count(max_cluster)
    if isprint:
        print('Size:',size)
    return l,size
```

```python
#core algorithm
def hoshenKopelman(ni,nj,qn):
    for i in range(ni):
        for j in range(nj):
            node = nj*i+j
            if qn.occupied[node]==1:
                left = nj*i+(j-1)%nj
                above = nj*((i-1)%ni)+j

                if qn.occupied[left] and not qn.occupied[above]:
                    qn.unite(node,left)

                if  qn.occupied[above] and not qn.occupied[left]:
                    qn.unite(node,above)

                if qn.occupied[above] and qn.occupied[left]:
                    if qn.find(left,above):
                        qn.unite(node,above)

                    elif qn.root(left)>qn.root(above):
                        qn.unite(left,above)
                        qn.unite(node,above)

                    elif qn.root(left)<qn.root(above):
                        qn.unite(above,left)
                        qn.unite(node,left)
    return qn
```

Appendix: Source code

```python
if __name__ == '__main__':


    #main routine
    ns= [10,20,30,40,50,60,70,80,90,100]
    P= np.arange(0.0,1.0,0.05)
    iteration = 30
    Sizes = []
    execution_time = []
    for n in ns:
        Size = []
        start = time.time()
        for p in P:
            mean_size = 0.0
            for t in range(iteration):
                ni,nj= n,n
                N = ni*nj
                qn = QuickUnion(N,p)
                #original_graph(ni,nj,qn.occupied)
                qn = hoshenKopelman(ni,nj,qn)
                labels = labeled_graph(ni,nj,qn)
                mcluster,size = max_cluster(labels)
                #print(mcluster,size)
                mean_size += size
            mean_size/= iteration
            mean_size/= N
            Size.append(mean_size)
        Size.reverse()
        Sizes.append(Size)
        end = time.time()
        t = end -start
        execution_time.append(t)
    print('Execution times of main routine given different n:\n',
          execution_time)
```

```python
    #p-size relation
    dn = 0
    for Size in Sizes:
        nlabel='n='+str(10+dn)
        plt.plot(P,Size,label=nlabel)
        dn+=10

    plt.title('P-Size')
    plt.xlabel('percolation rate')
    plt.ylabel('average maximum size of cluster (normalized)')
    plt.legend()
    plt.savefig('p_size_relation_2.png')
    plt.show()

#t-n relation
ns = np.array(ns)
plt.plot(ns,execution_time/np.max(execution_time),
         label = 'realistic value')
plt.plot(ns,np.power(ns,4)*0.00000001,
         label = 'fitted value (power = 4)')
plt.plot(ns,np.exp(ns**2/(100**2*np.log(4.2)))-1.0,
         label = 'fitted value (exponential)')
plt.title('Computational Comlexity (normalized)')
plt.xlabel('system scale')
plt.ylabel('execution time')
plt.legend()
plt.savefig('computational_complexity.png')
plt.show()
```

Reference

Algorithm research:

- https://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf
- https://github.com/AdamPI314/UnionFind
- https://en.wikipedia.org/wiki/Hoshen%E2%80%93Kopelman_algorithm

Image resource:

- http://rsdavis.mycpanel.princeton.edu/wp/?p=519
- Christian Joas‘s nodal domain statistics lecture note,

  Institut f˙ur theoretische Physik, FU Berlin