Union-FindからのNewman-Ziff

Newman, Mark EJ, and Robert M. Ziff. "Fast Monte Carlo algorithm for site or bond percolation." *Physical Review E* 64.1 (2001): 016706.

(実装例)

https://colab.research.google.com/drive/157gvVrEHktmlsFs-8ZMqpRt_2FKgscUT?usp=sharing

Masaki Chujyo

2023/2/24@zemi

Newman-Ziff algorithmの実装を目指して

パーコレーションは、各リンク(bond) or ノード(site)が確率1-pで除去される過程.

リンクとノードの二種類を考えられるが、本発表では簡単のためリンク(bond)を扱う.

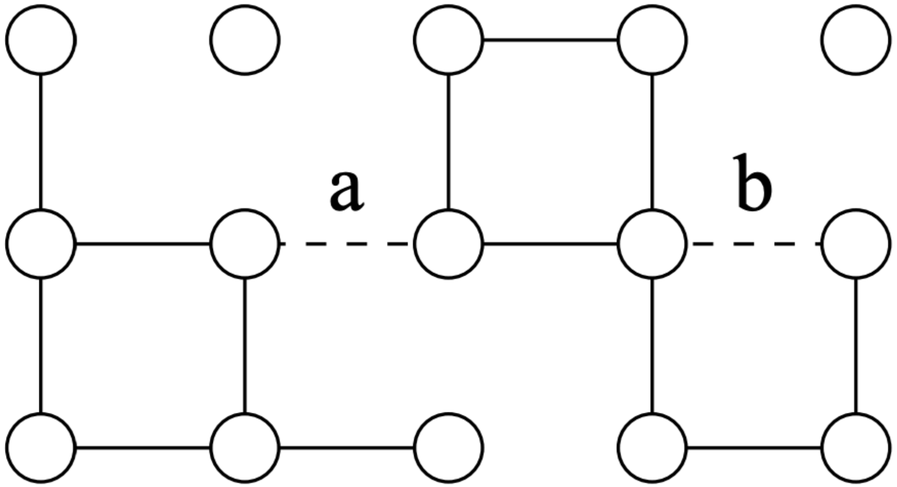
Newman-Ziffアルゴリズムはパーコレーションを数値シミュレーションする方法.

素集合データを扱うデータ構造の**Union-Find Tree**を使用することで高速に計算可能.

リンク除去の逆過程を考えることで、つなぐ(Union)操作を行っていく.

リンクを1本だけ繋いだ時の連結成分サイズは「増える」か「変化しない」のが肝.

👆This is the idea at the heart of our algorithm.



algorithm	time in seconds
depth-first search	4 500 000
unweighted relabeling	16 000
weighted relabeling	4.2
tree-based algorithm	2.9

Union-Find Tree

Union-Find Treeは、素な集合を高速に扱うためのデータ構造.

2集合の統合(Union)と**要素が含まれる集合の発見(Find)**の2つの操作からなる.

競技プログラミングなどでは典型的な手法となっている.

そのため資料も例題もネットや書籍などにめっちゃある.

https://www.slideshare.net/chokudai/union-find-49066733

https://algo-method.com/descriptions/132

https://atcoder.jp/contests/atc001/tasks/unionfind_a

https://qiita.com/ofutonton/items/c17dfd33fc542c222396

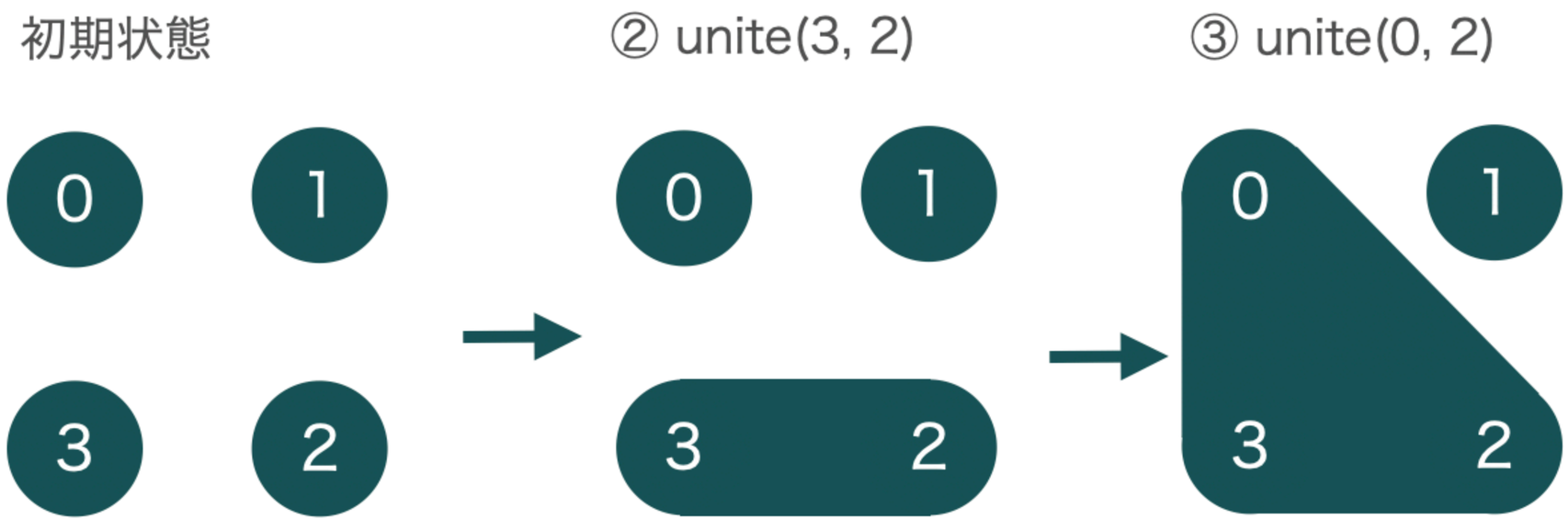
アルゴ式 Beta

アルゴ式によるUnion-Find Tree解説

したいこと: 素な集合を統合する(Union)と自身が含まれる集合の発見(Find)

Findは二つの要素が同じ集合に含まれているかの判定とも言える.

“集合からある要素を取り出す操作”は考えない.



https://algo-method.com/descriptions/132

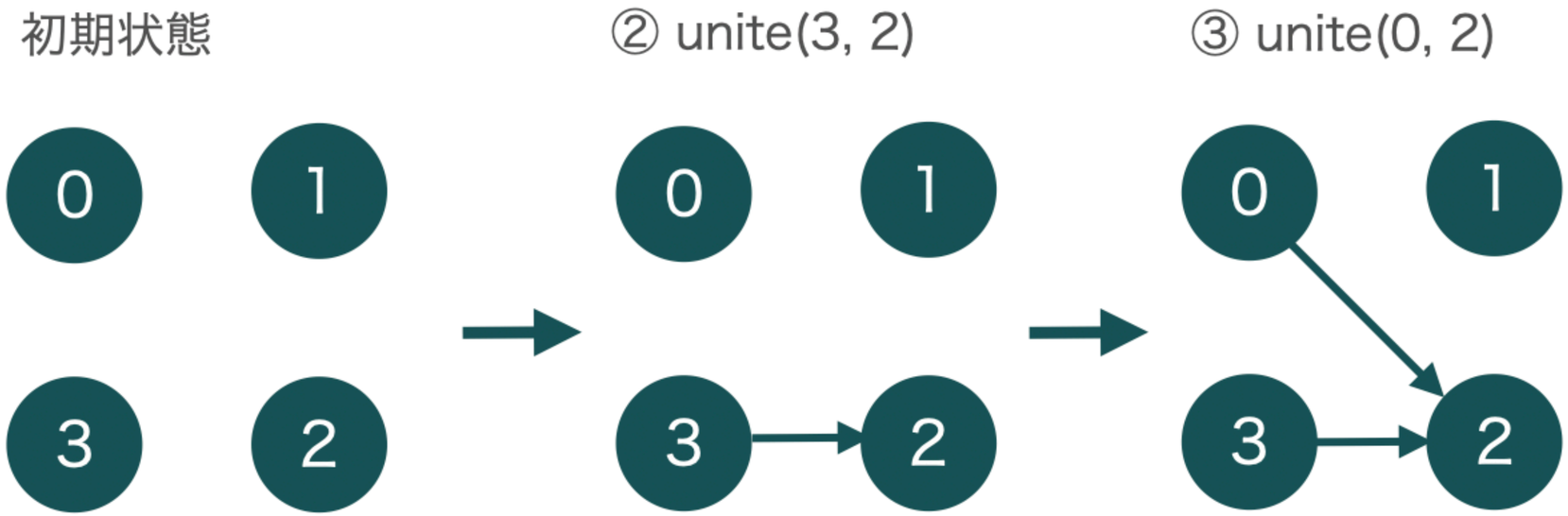
アルゴ式によるUnion-Find Tree解説

各集合を有向リンクによる根つき有向木 (森)で表現する.

あるノードから出ている有向辺→先のノードを親ノードと呼ぶ.

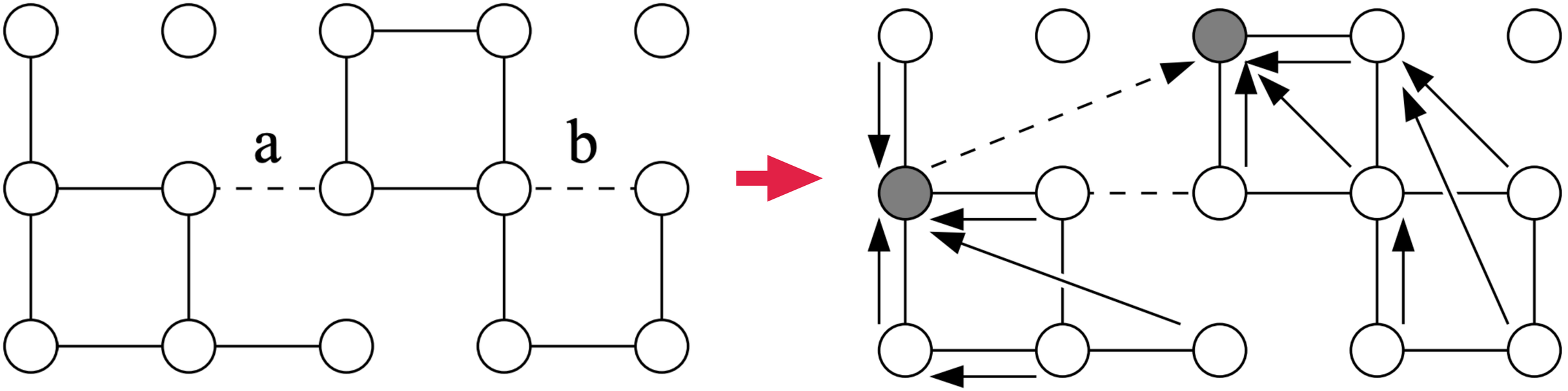
要素が含まれる集合を発見する(**Find**): 有向辺を辿って到達する最終点(根ノード)を代表とする.

素な集合を統合する(**Union**): 根ノードの片方からもう一方へ有向辺を作る.



https://algo-method.com/descriptions/132

ネットワークでのUnion-Find Treeの例



ここまでのPython実装

各ノードに親ノードを設定して

根つき有向グラフを作る.

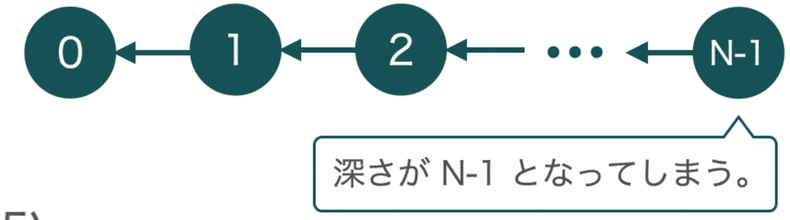
あとはさっきの説明をそのまま実装.

```python
class SimpleUnionFind():
    def __init__(self, n):
        # n ノード
        self.n = n
        # 親ノードのリスト, -1は自身が根ノードであることを示す.
        self.parents = [-1] * n

    def find(self, x):
        # 根である場合は、そのidを返す
        if self.parents[x] < 0:
            return x
        # 再帰で親ノードを遡って根ノードを返す.
        else:
            return self.find(self.parents[x])

    def union(self, x, y):
        # それぞれの集合の根ノードを探す
        root_x = self.find(x)
        root_y = self.find(y)
        # 同じ集合なら何もしない
        if root_x == root_y:
            return
        # 異なる集合なら、片方の根ノードの親をもう一方の根ノードにする.
        else:
            self.parents[root_y] = root_x
```

Union-Find Treeをもっと効率的に

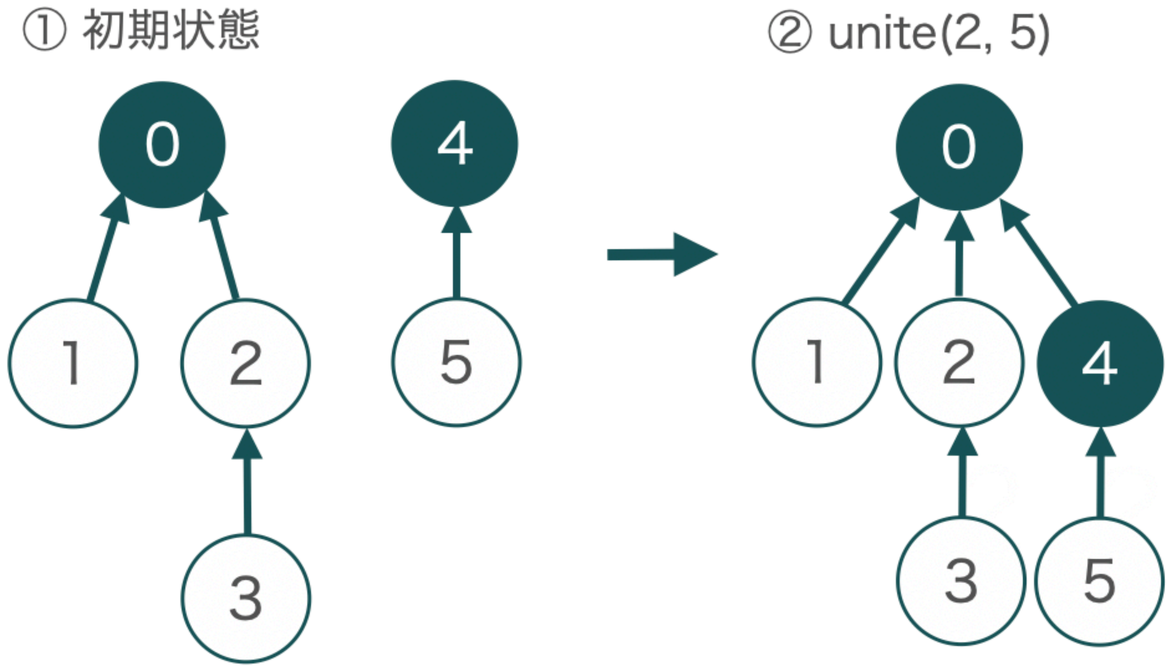
Union-Find Treeをそのまま作成すると、高さの長い木が生成されてしまうことがある.

そこで計算が簡単な木にするテクニックが二つある。



1. Union by size

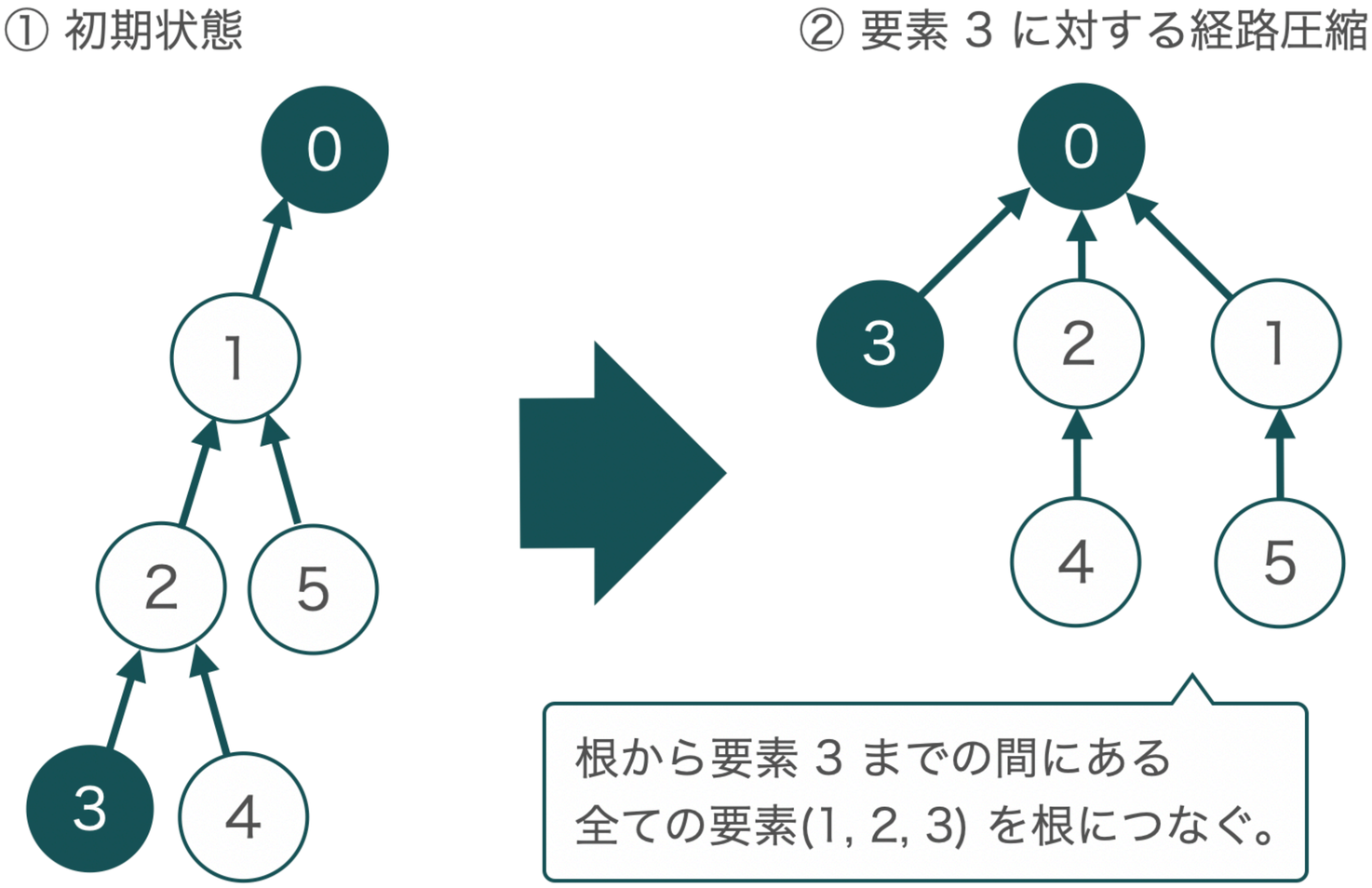
低い木につなぐ



https://algo-method.com/descriptions/133

Union-Find Treeをもっと効率的に

2. 経路圧縮: Findする時に根ノードにそのまま繋げる.



https://algo-method.com/descriptions/133

ここまでのPython実装

さっきのコードをちょっとだけ変える.

先ほど根かどうかを判定していた負の値を

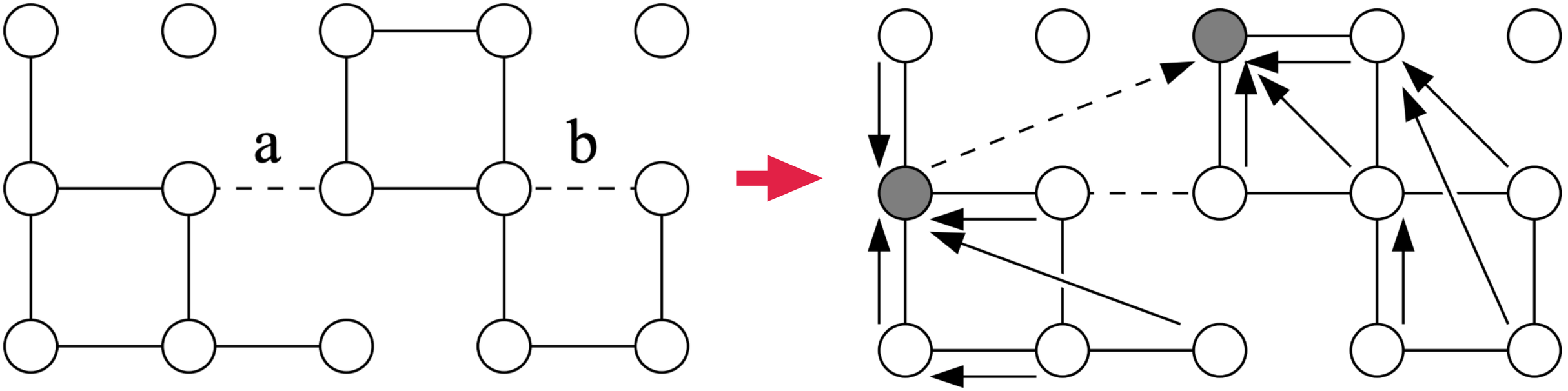
集合のサイズとしている.

つまり、これが連結成分サイズとなる.

```python
class UnionFind():
  def __init__(self, n):
    self.n = n
    # ここで負の値は、その根ノードが含まれる集合のサイズとする.
    self.parents = [-1] * n

  def find(self, x):
    if self.parents[x] < 0:
      return x
    else:
      # 経路圧縮の処理
      # 再帰で見つかった根ノードを親ノードに設定する
      self.parents[x] = self.find(self.parents[x])
      return self.parents[x]

  def union(self, x, y):
    x = self.find(x)
    y = self.find(y)
    if x == y:
      return
    # Union by sizeの処理
    # 距離が短い方を根にしてつなぐ
    if self.parents[x] > self.parents[y]:
      x, y = y, x
    self.parents[x] += self.parents[y]
    self.parents[y] = x
```

Newman-Ziff algorithmでしたいこと

1本のリンクが追加された時の連結成分の変化をUnion-Find Treeで表現する.



Newman-Ziff algorithm その1

Bond Percolationとして、リンクを一本ずつランダムに除去することを考える。

これを何回か繰り返し行う。そこで次のようにリンクの除去順リストを作成する。

除去リンクの順序リストの作成:

1. リンクのidを並べたリストを作る
2. リストの先頭リンクi=1を選択
3. ランダムにリンクj(>i)を選択
4. リンクiとリンクjを入れ替える
5. i+=1
6. 2-5をくり返す

```
def make_remove_list(M):
  remove_list=[e for e in range(M)]
  for i in range(M-1):
    rand_id=np.random.choice([k for k in range(i+1, M)])
    if(rand_id==i):
      pass
    else:
      remove_list[i], remove_list[rand_id]=remove_list[rand_id], remove_list[i]

  #  一応Permutationで減ってないかをチェック.
  if(len(list(set(remove_list))) != len(remove_list)):
    print("permutation Error.", set(remove_list)^set([e for e in range(M)]))

  return remove_list
```

Newman-Ziff algorithm その2

除去順リストの後ろから、逆にリンクを追加し、unionしていくだけ.

NZ alg.

1. 初期ノードは全てが自分自身が根の孤立ノードとする. 各サイズは1 (parents[i]=-1).

2. 除去リストの後ろから一本追加する.

3. 追加されて繋がったノードが互いに含まれている根ノードをfindする.

4. 同じ集合に含まれている時は何もしない.

5. 異なる集合に含まれている時はunionする.

6. 最大連結成分サイズや平均連結成分サイズなどを計算する.

7. 2-6を繰り返し行う.

Newman-Ziff algorithm 実装

最大連結成分(最大の集合)を得るための関数を追加.

全ての根ノードを得る関数rootsと

その中から最大を得る関数GCsizeを追加.

```python
    # 根ノードを出力
    def roots(self):
        return [i for i,x in enumerate(self.parents) if x<0 ]

    # 最大連結成分サイズを得る関数
    def GCsize(self):
        roots=self.roots()
        return max([-self.parents[x] for x in roots])
```

```python
class NZ_UnionFind():
    def __init__(self, n):
        self.n = n
        # ここで負の値は、その根ノードが含まれる集合のサイズとする.
        self.parents = [-1] * n

    def find(self, x):
        if self.parents[x] < 0:
            return x
        else:
            # 経路圧縮の処理
            # 再帰で見つかった根ノードを親ノードに設定する
            self.parents[x] = self.find(self.parents[x])
            return self.parents[x]

    def union(self, x, y):
        x = self.find(x)
        y = self.find(y)
        if x == y:
            return
        # Union by sizeの処理
        # 距離が短い方を根にしてつなぐ
        if self.parents[x] > self.parents[y]:
            x, y = y, x
        self.parents[x] += self.parents[y]
        self.parents[y] = x

    # 根ノードを出力
    def roots(self):
        return [i for i,x in enumerate(self.parents) if x<0 ]

    # 最大連結成分サイズを得る関数
    def GCsize(self):
        roots=self.roots()
        return max([-self.parents[x] for x in roots])
```

Newman-Ziff algorithm 実装

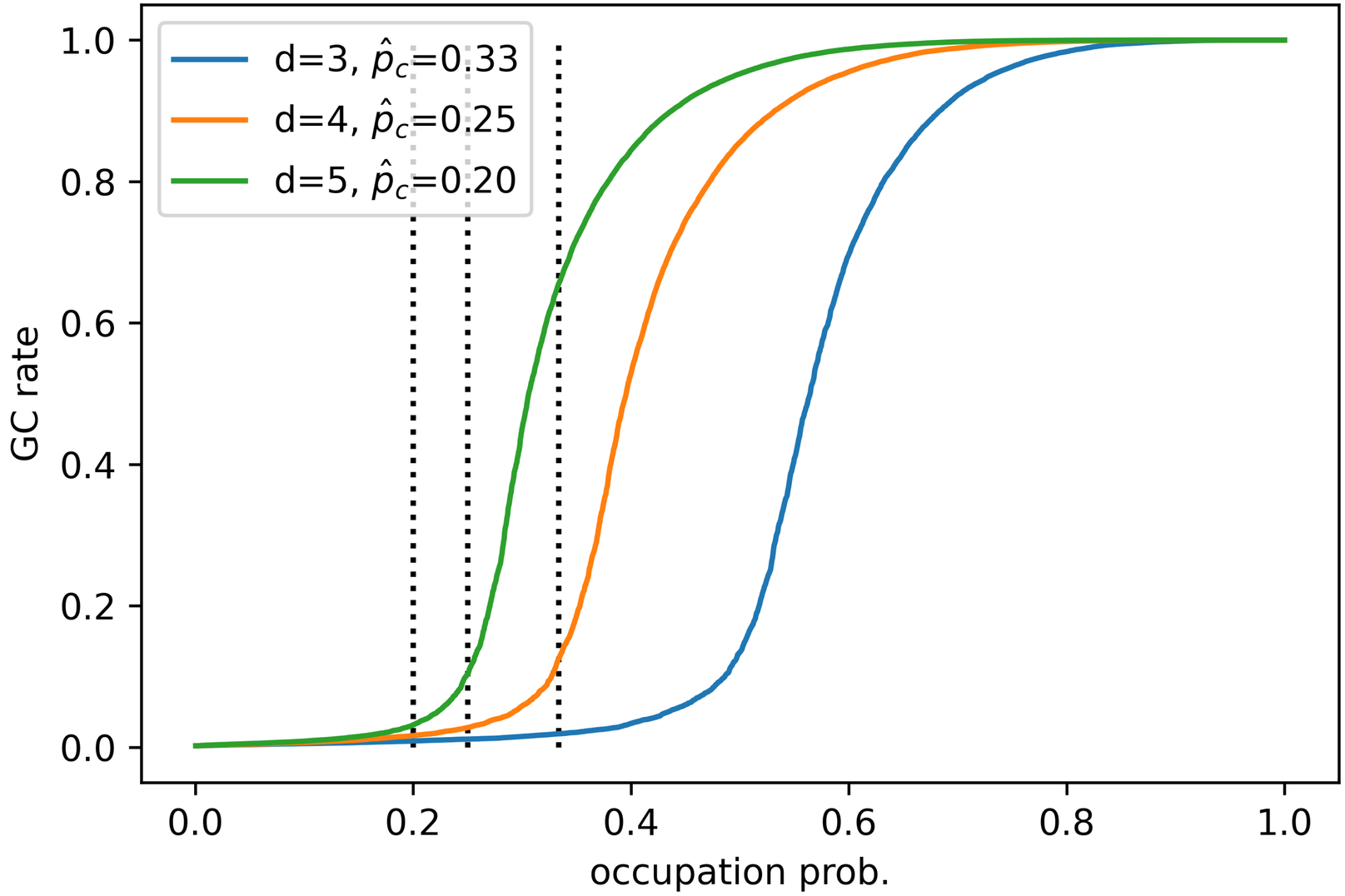
メイン部分

論文をそのまま実装してるだけ

```
def NewmanZiffAlg(G_origin):
  # 固定値
  N=G_origin.number_of_nodes()
  M=G_origin.number_of_edges()
  Edges=[[e0,e1] for e0,e1 in G_origin.edges()]
  # 結果保存用
  sn=[0]*M
  # 適当に100平均
  averaging=100

  for loops in range(averaging):
    # 除去順リスト作成
    remove_list=make_remove_list(M)
    # UnionFind tree 作成
    uf=NZ_UnionFind(N)
    # 除去リストの逆順にする
    for n_removed_link, remove_link_id in enumerate(reversed(remove_list)):
      # unionで繋げる
      uf.union(Edges[remove_link_id][0], Edges[remove_link_id][1])
      # 最大連結成分を得る
      GCsize=uf.GCsize()/float(averaging)
      sn[n_removed_link]+=GCsize

  return sn
```

Newman-Ziff algorithmのデモ

Random regular graphs(N=1000, 100平均)での結果，$\hat{p}_C$は隣接行列の固有値による推定値



Newman-Ziff algorithmのデモ

Barabasi-Albert Model(N=1000, 100平均)での結果，$\hat{p}_C$は隣接行列の固有値による推定値