Perkolacje

ab

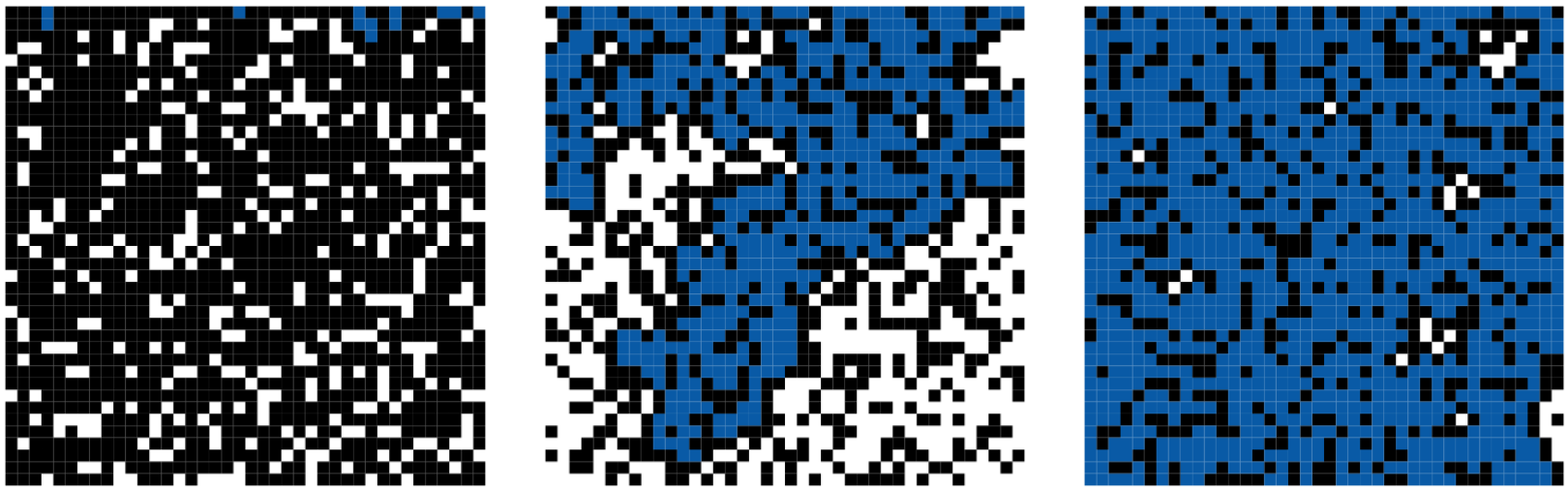
13 grudnia 2016

Tematyka

- Co to są perkolacje. Zastosowania.
- Algorytmy: wertykalny, *inferno*, algorytm *przeszukiwania w głąb* (*Depth First Search*)
- Algorytm Newmana-Ziffa
- Obserwable

Przykład I

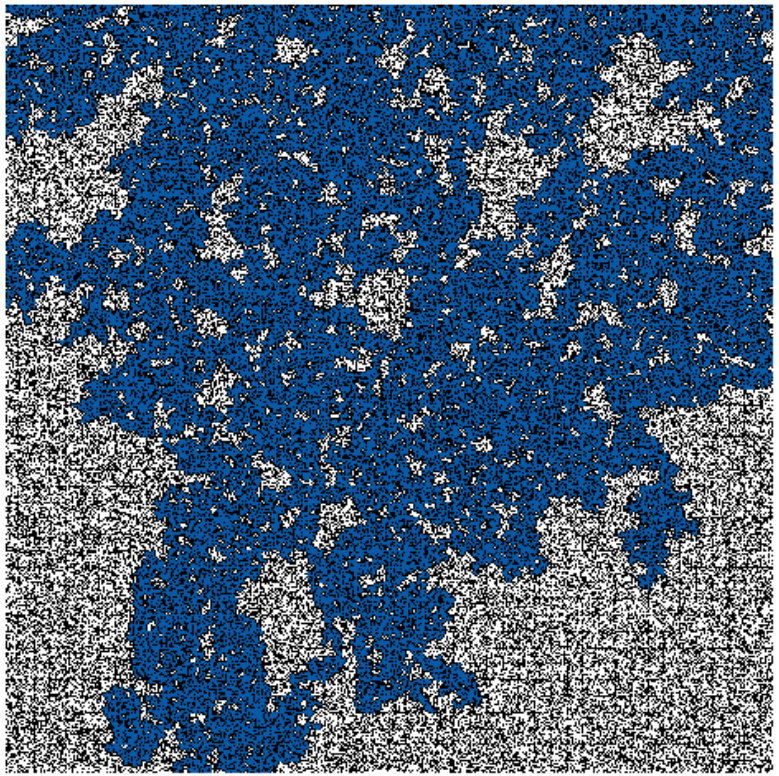
Przykład. Sieć 40×40. Prawdopodobieństwo pola pustego $= p$.



Pory (pola jasne) i przegrody (pola czarne) dla $p = 0.2$, $p = 0.6$, $p = 0.7$.
Pola niebieskie obraazują *przeciekanie*.

Przykład I

Przykład. Sieć 512×512. Prawdopodobieństwo pola pustego $p = 0.593$.



Pory (pola jasne) i przegrody (pola czarne) dla $p = 0.593$. Pola niebieskie obraazują *przeciekanie*.

Zastosowania

- substancje porowate
- przejścia Sol-Gel
- mieszanina przewodniki-izolatory
- rozprzestrzenianie się pożarów (las)
- epidemie, wirusy komputerowe
- załamania rynków finansowych
- ...

Algorytm *spalania*

Algorytm znajduje połączenie *góra-dół* przez pola obsadzone w tablicy wypełnionej zerami i jedynkami (o ile takie połączenie istnieje).

1. Zaznacz pola zajete w górnym wierszu tablicy pól markerem $t = 2$
2. w kroku iteracyjnym $t + 1$
   - przejdź przez wszystkie pola i znajdź pola z markerem t
   - dla każdego pola z markerem t
     - sprawdź czy sąsiednie pola (północ, wschód, południe, zachód) są zajęte i *nie płoną* (marker 1)
     - oznacz znalezione pola markerem $t + 1$
3. Powtarzaj krok 2 (z $t = t + 1$) do momentu gdy nie ma już sąsiednich pól do *spalenia* lub gdy został osiągnięty ostatni wiersz – w tej sytuacji, ostatni marker zmniejszony o 1 jest najkrótszą drogą łączącą brzegi (górny i dolny) sieci

Algorytm *przeszukiwania w głąb*

Algorytm znajdowania klastra przeciekającego (z góry na dół w tablicy zer i jedynek $A[N][N]$).

1. przepisz tablicę $A[N][N]$ do $B[N][N]$
2. przejdź przez pierwszy wiersz tablicy A i znajdź pola zajęte (1)
3. Dla pola zajętego:
   - 3.1 zaznacz to pole jedynką w B
   - 3.2 znajdź w A zajęte pola sąsiednie (lewe, górne, prawe, dolne)
   - 3.3 wykonaj krok 3
4. przejdź przez pola ostatniego wiersza w B. Znaleziona w ostatnim wierszu tablicy B jedynka oznacza istnienie klastra przeciekającego.

Program: przeszukiwanie w głąb I

```
// PRZECIEKANIE
// znajduje zacieki w 'a', zapisując to w 'b'
boolean[][] zacieki(boolean[][] a){
    int N = b.length;
    boolean[][] b = new boolean[N][N];
    for(int j=0;j<N;j++)
        cieknie(a, b, 0, j);
    return b;
}
// przeszukuje 'a' w głąb
void cieknie(boolean[][] a, boolean[][] b, int i, int j){
    // koniec
    int N = a.length;
    if(i<0 || i>=N) return;
    if(j<0 || j>=N) return;
    if(!a[i][j]) return;
    if(b[i][j]) return;
    // zaznacz i,j ze cieknie
    b[i][j] = true;
    cieknie(a, b, i+1, j);  // dolny
    cieknie(a, b, i, j+1);  // prawy
```

Program: przeszukiwanie w głąb II

```
    cieknie(a, b, i-1, j);  // gorny
    cieknie(a, b, i, j-1);  // lewy
}
// zwraca true jeśli isnieje klaster przeciekający
boolean przecieka(boolean[][] a){
    int N = a.length;
    boolean[][] b = przecieki(a);
    for(int j=0;j<N;j++)
        if (b[N-1][j]) return true;
    return false;
}
```

Program I

```
/**
 *  Perkolacje: generowanie i wizualizacja obsadzen sieci L x L z
 *            zadanym prawdopodobienstwem p; rysowane są również
 *            'zacieki'. Metoda "przeszukiwanie w~głąb"
 * Kompilacja:
 *    javac Perkolacje.java
 * Uzycie:
 *    java 40 .4 10
 *    java 30 .5 20
 */
import java.awt.*;
import java.awt.image.BufferedImage;
import javax.swing.*;
import java.util.Random;

public class VisualP extends JPanel implements Runnable {
    int size = 512;       // rozmiar okna, px
    private Thread animator;
    private volatile boolean running = false;
    private Graphics dbg;
    private Image dbImage = null;
```

Program II

```
int L;                  // rozmiar sieci L x L
double p;               // prawdopodobienstwo obsadzania miejsc sieci
boolean[][] b;          // tablica obsadzen (true || false)
Random rand;            // generator liczb pseudolosowych

int licznik;            // bieżąca konfiguracja
int liczbaKonf;         // liczba wszystkich konfiguracji

boolean[][] cieki;      // zacieki z gory do dołu
boolean PRZECIEKA;      // true jeżeli jest klaster przeciekający

public VisualP(int L, double p, int liczbaKonf){
    this.L = L;
    this.p = p;
    this.liczbaKonf = liczbaKonf;
    licznik = 0;
    rand  = new Random();
    cieki = new boolean[L][L];
    JFrame okno = new JFrame("Perkolacje ... ");
    okno.getContentPane().add(this);
    okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    okno.setSize(size,size+20);
```

Program III

```
        okno.setBackground(Color.white);
        okno.setVisible(true);
        dbImage = new BufferedImage(size, size,
                    BufferedImage.TYPE_INT_ARGB);
    }
    void percStart() {
        if(animator == null || !running){
            animator = new Thread(this);
            animator.start();
        }
    }
    @Override
    public void run() {
        running = true;
        while (running) {
            timeStep();
            percRender();
            repaint();
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException ie) {}
            licznik++;
```

Program IV

```
            if (licznik >= liczbaKonf) break;
        }
    }
    @Override
    public void paintComponent(Graphics g) {
        //Graphics2D g2 = (Graphics2D)g;
        super.paintComponent(g);
        if(dbImage != null)
            g.drawImage(dbImage, 0, 0, null);
    }
    /**
       Rysuje konfiguracje poza ekranem
     */
    private void percRender() {
        if (dbImage == null) {
            System.out.println("dbImage is null.");
            return;
        } else {
            // get graphics context for drawing to off-screen images.
            dbg = dbImage.getGraphics();
        }
        // clear the background
        dbg.setColor(Color.white);
```

Program V

```
        dbg.fillRect(0,0,size,size);
        // rysowanie pól obsadzonych i zacieków
        rysuj_mat(dbg, b,     new Color(100,100,170));
        rysuj_mat(dbg, cieki, new Color(10,10,170));
        // liczba konfiguracji i przeciekanie
        int licz = licznik+1;
        dbg.setColor(new Color(200,100,100));
        dbg.drawString("L="+L, 5, 12);
        dbg.drawString("p="+p, 60, 12);
        dbg.drawString("# "+licz+"/"+liczbaKonf, 120, 12);
        if( PRZECIEKA ) dbg.drawString("PRZECIEKA", 200, 12);
    }
    /**
       Maluje tablicę boolowską mat[][] w kolorze col
     */
    void rysuj_mat(Graphics g, boolean[][] mat, Color col){
        // obliczanie wymiarów elementów graficznych
        Dimension dim = getSize();
        int size = dim.height;
        int bok = size/L-1;
        if (bok <= 0) bok=1;
        int del = (size-bok*L)/2;
        g.setColor(col);
```

Program VI

```
        for(int x=0; x<L; x++)
            for(int y=0; y<L; y++) {
                if (mat[y][x]) g.fillRect(del+x*bok,del+y*bok+1,bok,bok);
            }
    }
    /**
     * Generuje konfiguracje, wypelniając tablicę b[][].
     * Znajduje zacieki (cieki[][]). Sprawdza przeciekanie.
     */
    private void timeStep() {
        // obsadzanie miejsc wg. prawdopodobienstwa p
        b = new boolean[L][L];
        for(int x=0; x<L; x++)
            for(int y=0; y<L; y++)
                b[y][x] = rand.nextDouble() < p;

        // Analiza konfiguracji; Obliczenia obserwabli
        // dla konfiguracji zapisanej w b[][] ...

        // znajdz tylko zacieki;
        //cieki = zacieki(b);

        // znajdź zacieki i sprawdź czy przecieka
```

Program VII

```
    PRZECIEKA = przecieka(b);
}
/**
 *  PRZECIEKANIE -- algorytm przeszukiwania w głąb
 *  @param a tablica kwadratowa o wartościach true/false
 *  @return tablica kwadratowa o wartosciach true/false
 *  zawierająca true w miejscach, która zaciekają
*/
boolean[][] zacieki(boolean[][] a){
    int N = b.length;
    boolean[][] b = new boolean[N][N];
    for(int j=0;j<N;j++)
        cieknie(a, b, 0, j);
    return b;
}
/**
 * Algorytm przeszukiwania w głąb
 * @param a tablica true/false
 * @param b wynik przeszukiwania (zacieki od góry: true)
 * @param ij przeszukiwanie od miejsca a[i][j]
 */
void cieknie(boolean[][] a, boolean[][] b, int i, int j){
    // koniec
```

Program VIII

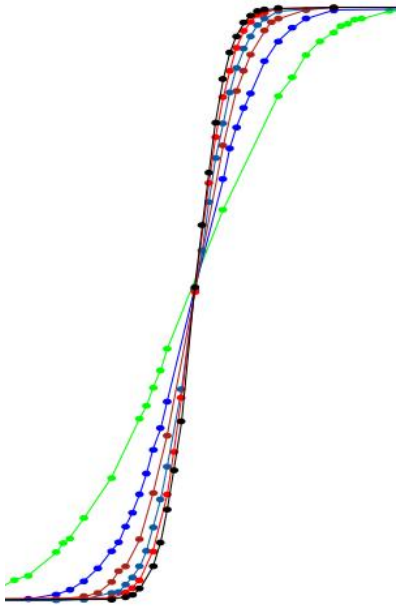
```
    int N = a.length;
    if(i<0 || i>=N) return;
    if(j<0 || j>=N) return;
    if(!a[i][j]) return;
    if( b[i][j]) return;
    // zaznacz i,j (true) jeśli cieknie
    b[i][j] = true;
    cieknie(a, b, i+1, j);  // dolny
    cieknie(a, b, i, j+1);  // prawy
    cieknie(a, b, i-1, j);  // gorny
    cieknie(a, b, i, j-1);  // lewy
}
/**
 * @param a tablica [][] true/false (do sprawdzenia)
 * @return true/false przecieka / nie przecieka
 */
boolean przecieka(boolean[][] b){
    int N = b.length;
    cieki = zacieki(b);   // cieki[][] - zdefiniowana wcześniej
    for(int j=0;j<N;j++)
        if (cieki[N-1][j]) return true;
    return false;
}
```

Program IX

```
    public static void main(String[] args) {
        // dane opcjonalne
        int L = 64;             // rozmiar konfiguracji LxL
        double p = 0.59;        // praedopodobienstwo obsadzania
        int liczbaKonf = 10;    // liczba konfiguracji
        // obsluga wiersza polecen (jesli podano: L p liczbaKonf)
        if (args.length >= 3) {
            L = Integer.parseInt(args[0]);            // rozmiar liniowy L
            p = Double.parseDouble(args[1]);          // prawdopodobienstwo
            liczbaKonf = Integer.parseInt(args[2]);   // liczba konfiguracji
        }
        // obliczenia
        VisualP vp = new VisualP(L, p, liczbaKonf);
        vp.percStart();
    }
}
```

Prawdopodobieństwo krytyczne (przeciekania)

Dla pewnego $p = p_c$ – prawdopodobieństwo krytyczne, pojawiają sie klastry przeciekające z góry na dół (lub z prawa na lewo). Badając konfiguracje dla różnych p (zespoły konfiguracji dla różnych p; tutaj liczba konfiguracji dla danego p wynosiła 10000) można wyznaczyć prawdopodobieństwo krytyczne. Dla sieci kwadratowej $p_c \approx 0.592746$. Dla innych typów sieci p_c jest inne (zależy od liczby sąsiadujących pól).



Ułamek liczby przeciekajacych klastrów w funkcji p. Kolejne krzywe odpowiadają $N = 8, 16, 32, 64$ i 128 (im większa sieć (N) tym większe nachylenie krzywej). Punkt przegięcia odpowiada w przybliżeniu $p \sim 0.593$. $p \in (.4, .8)$

Parametr porządku

Rozważmy przypadki $p > p_c$, dla których (prawie zawsze) pojawia się klaster przeciekający (duże sieci).

Ile pól zawiera największy klaster?
Definiujemy ułamek $P(p)$ równy stosunkowi liczby pól w największym klastrze do wszystkich obsadzonych pól sieci. Wielkość ta nosi nazwę parametru porządku.

$$P(p) = \frac{\text{Liczba pól w największym klastrze}}{\text{Liczba zajętych pól sieci}} \tag{1}$$

Okazuje się, że

$$P(p) \sim (p - p_c)^{\beta},$$

gdzie β zależy od wymiaru. Dla sieci 2-wymiarowej $\beta = \frac{5}{36}$ i $\beta \sim 0.41$ dla sieci 3-wymiarowej.

Rozkład gęstości wielkości klastrów

Jaki jest rozkład gęstości klastrów zależnie od ich wielkości?

$$n_p(s) = \frac{\text{Liczba klastrów wielkości } s}{\text{Liczba pól sieci}}.$$

Tutaj s jest wielkością klastra – *masą*, a p jest prawdopodobieństwem obsadzania sieci (nie licząc klastra przeciekającego).

Prawdopodobieństwo, że zajęte pole należy do klastra s jest równe

$$w_s(p) = \frac{sn_s}{\sum_s sn_s}$$

Średnia wielkość klastra

Średnia wielkość klastra zdefiniowana jest jako

$$S(p) = \sum_s s w_s = \frac{s^2 n_s}{\sum_s s n_s}$$

gdzie nie uwzględnia się klastrów przeciekających. W granicy $N \to \infty$

$$S(p) \sim \frac{1}{|p - p_c|^\gamma} .$$

Dla sieci 2D $\gamma = 43/18$.

Zakres korelacji

Dwa pola sieci są skorelowane jeśli są zajęte i należą do jednego klastra. Zakres korelacji $\xi(p)$ jest średnią odległością między obsadzonymi polami (bez klastra przeciekającego). W punkcie krytycznym wielkość ta jest rozbieżna w pobliżu p_c

$$\xi(p) = \frac{1}{|p - p_c|^{\nu}}$$

gdzie $\nu = 4/3$ na sieci 2-wymiarowej.

Prawa skalowania

Istnieją teoretyczne i praktyczne powody, dla których uważ się, że wykładniki krytyczne nie są niezależne lecz są związane prawami skalowania. W układach z przejściami fazowymi typu perkolacji

$$2\beta + \gamma = \nu d$$

gdzie d jest wymiarem przestrzennym. W przypadku 2-wymiarowym

$$2 \times \frac{5}{36} + \frac{43}{18} = \frac{4}{3} \times 2\,.$$

Teoria skalowania w układach skończonych

Przejścia fazowe, osobliwości i wykładniki krytyczne oraz prawa skalowania są słuszne dla układów nieskończonych. Aby je zbadać w doświadczeniach lub obliczeniach komputerowych należy rozpatrywać ciągi układów skończonych i następnie ekstrapolować otrzymane wyniki. Podstawową wielkością jest zakres korelacji ξ. W układach nieskończonych ξ jest rozbieżne w punkcie krytycznym. W układach o wymiarze L zakres ξ jest ograniczony do L. W małym zakresie wartości parametru p

$$\xi(p) \sim L \sim \frac{1}{|p - p_c|^{\nu}}, \quad \lim_{L\to\infty} p_c(L) = p_c,$$

gdzie $p_c(L)$ dąży do krytycznego p_c układu nieskończonego. Stąd

$$|p - p_c| \sim \frac{1}{L^{1/\nu}}.$$

Wielkość parametru porządku $P(p)$ i średnią wielkość klastrów w układach skończonych można oszacować na podstawie wielkości dla układów skończonych:

$$P(p) \sim (p - p_c)^{\beta} \sim L^{-\beta/\nu}$$

$$S(p) \sim |p - p_c|^{-\gamma} \sim L^{\gamma/\nu}.$$

Rozkład gęstości wielkości klastrów

Dobrym narzędziem do badania rozkładu wielkości klastrów jest algorytm Hoshena-Kopelmana (1976).

Algorytm Hoshena-Kopelmana

$N[L \times L]$ – element N_{ij} przechowuje numer klastra k;
M_k – *masa* klastra k, równa liczbie pól w klastrze;

- $k \leftarrow 2\,, M_k \leftarrow 1$ (numer klastra)
- dla wszystkich i, j w N_{ij}
  1. jeżeli *górny* i *dolny* są puste (lub nie istnieją): $k \leftarrow k+1\,, N_{ij} \leftarrow k\,, M_k \leftarrow 1$
  2. jeśli w jednym z nich jest k_0: $N_{ij} \leftarrow k_0\,, M_{k_0} \leftarrow M_{k_0} 1$
  3. jeśli oba zawierają różne k_1 i k_2: wybieramy np. k_1 i podstawiamy $N_{ij} \leftarrow k_1$, $M_{k_1} \leftarrow M_{k_1} + M_{k_2} + 1$, $M_{k_2} \leftarrow -k_1$
  4. jeśli oba zawierają k_1: podstawiamy $N_{ij} \leftarrow k_1$, $M_{k_1} \leftarrow M_{k_1} + 1$
  5. jeżeli rozważanemu k odpowiada ujemne M_k: znajdujemy odpowiadający oryginalny klaster i używamy zamiast tego numer i wagę klastra oryginalnego.
- dla $k \leftarrow 2 \ldots k_{max}$ wykonujemy:
  jeżeli $M_k > 0$ wówczas $n(M_k) \leftarrow n(M_k) + 1$

Algorytm Hoshena-Kopelmana I

```
/**
   Percolation program using Hoshen-Kopelman algorithm of claster
   analysis.

   Original: ...

*/
import java.util.Random;

public class Percolation_HK{

    int L;                      // linear dimension
    double p;                   // occupation probability
    boolean[][] bsite;          // random sites (true, false)

    int[][] site;               // working array
    int[] np;                   // claster labels, in HK alg

    boolean VertSpan = true;    // vertically spanning cluster
```

Algorytm Hoshena-Kopelmana II

```
boolean HorzSpan;           // similar...
boolean BothSpan;
boolean VorHSpan;
int spanningCluster;        // number of spanning cluster

// observables (see: computeObservables())
int occupiedSites;          // number of occupied sites
int finiteClusters;         // number of clusters
int nSpanning;              // sites in spanning cluster
double P_infinity;          // = nSpanning / occupiedSites
int[] n_s;                  // number of clusters of size s
int[] m_i;                  // number of sites in a cluster
double S;                   // S = sum(m*m) / sum(m), m=m_i[cluster]

Random ran;

Percolation_HK() {
    L = 16;
    p = 0.4;
    mat_init();
```

Algorytm Hoshena-Kopelmana III

```
}

// parametrized constructor
Percolation_HK(int L, double p) {
    this.L = L;
    this.p = p;
    mat_init();
}

void mat_init(){
    ran  = new Random();
    bsite = new boolean[L][L];
    site = new int[L + 2][L + 2];
    np = new int[(L + 2) * (L + 2)];
    n_s = new int[L * L];
    m_i = new int[L * L];
}

void new_config () {
    for (int y = 0; y < L; y++) {
```

Algorytm Hoshena-Kopelmana IV

```
            for (int x = 0; x < L; x++) {
                bsite[x][y] = ran.nextDouble() < p;
            }
        }

        HoshenKopelman();

    }

    // Hoshen-Kopelman algorithm

    void HoshenKopelman () {

        // initalize the array site[]
        for (int y = 1; y <= L; y++) {
            for (int x = 1; x <= L; x++) {
                site[x][y] = bsite[x-1][y-1] ? -1 : 0;
            }
        }
```

Algorytm Hoshena-Kopelmana V

```
    assign();                   // assign cluster numbers
    span();                     // check for spanning cluster
    computeObservables();

}

void assign () {

    // assign cluster numbers to occupied sites
    for (int i = 0; i < np.length; i++)
        np[i] = 0;
    int ncluster = 0;           // cluster number
    for (int y = 1; y <= L; y++) {
        for (int x = 1; x <= L; x++) {
            if (site[x][y] < 0) {
                int down = y - 1;
                int left = x - 1;
                if (site[x][down] + site[left][y] == 0) {
                    ++ncluster;                 // new cluster
                    site[x][y] = ncluster;
```

Algorytm Hoshena-Kopelmana VI

```
                    np[ncluster] = ncluster;  // proper label
                } else {
                    neighbor(x, y);
                }
            }
        }
    }

    // assign proper labels to cluster array
    for (int y = 1; y <= L; y++) {
        for (int x = 1; x <= L; x++) {
            site[x][y] = proper(site[x][y]);
        }
    }

}

void neighbor (int x, int y) {

    // determine occupancy of neighbors
```

Algorytm Hoshena-Kopelmana VII

```
        int down = y - 1;
        int left = x - 1;
        if (site[x][down] * site[left][y] > 0) {
            // both neighbors occupied
            label_min(x, y, left, down);
        } else if (site[x][down] > 0) {
            // down neighbor occupied
            site[x][y] = site[x][down];
        } else {
            site[x][y] = site[left][y];
        }

    }

    void label_min (int x, int y, int left, int down) {

        // both neighbors occupied, determine minimum cluster number
        if (site[x][y] == site[x][down]) {
            // both neighbors have same cluster label
            site[x][y] = site[left][y];
```

Algorytm Hoshena-Kopelmana VIII

```
    } else {
        // determine minimum cluster label
        int cl_left = proper(site[left][y]);
        int cl_down = proper(site[x][down]);
        int nmax = cl_left > cl_down ? cl_left : cl_down;
        int nmin = cl_left < cl_down ? cl_left : cl_down;
        site[x][y] = nmin;
        if (nmin != nmax)
            np[nmax] = nmin; // set improper label nmax = nmin
    }

}

int proper (int label) {

    // recursive function
    if (np[label] == label)
        return label;
    else
        return proper(np[label]);
```

Algorytm Hoshena-Kopelmana IX

```
}

void span () {

    // check for existence of vertically spanning cluster
    int vspan = 0;
    boolean done = false;
    for (int bottom = 1; bottom <= L; bottom++) {
        if (site[bottom][1] > 0) {
            int nbot = site[bottom][1];
            for (int top = 1; top <= L; top++) {
                if (site[top][L] > 0) {
                    int ntop = site[top][L];
                    if (ntop == nbot) {
                        vspan = proper(ntop);
                        done = true;
                        break;
                    }
                }
```

Algorytm Hoshena-Kopelmana X

```
            }
        }
        if (done)
            break;
    }

    // check for existence of horizontally spanning cluster
    int hspan = 0;
    done = false;
    for (int left = 1; left <= L; left++) {
        if (site[1][left] > 0) {
            int nleft = site[1][left];
            for (int right = 1; right <= L; right++) {
                if (site[L][right] > 0) {
                    int nright = site[L][right];
                    if (nright == nleft) {
                        hspan = proper(nright);
                        done = true;
                        break;
                    }
```

Algorytm Hoshena-Kopelmana XI

```
                }
            }
        }
        if (done)
            break;
    }

    spanningCluster = 0;
    if (BothSpan && vspan > 0 && hspan > 0)
        spanningCluster = vspan;
    else if (VertSpan && vspan > 0)
        spanningCluster = vspan;
    else if (HorzSpan && hspan > 0)
        spanningCluster = hspan;
    else if (VorHSpan) {
        if (vspan > 0)
            spanningCluster = vspan;
        else if (hspan > 0)
            spanningCluster = hspan;
    }
```

Algorytm Hoshena-Kopelmana XII

```
}

void computeObservables () {

    occupiedSites = 0;
    nSpanning = 0;

    for (int s = 0; s < L * L; s++)
        n_s[s] = m_i[s] = 0;

    for (int y = 1; y <= L; y++) {
        for (int x = 1; x <= L; x++) {
            int cluster = proper(site[x][y]);
            if (cluster > 0) {
                ++m_i[cluster];          // count sites in this cluster
                ++occupiedSites;         // count occupied sites
            }
        }
    }
```

Algorytm Hoshena-Kopelmana XIII

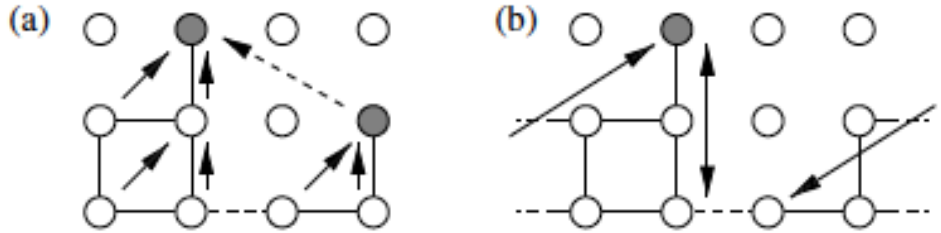
```
for (int cluster = 1; cluster < L * L; cluster++) {

    if (cluster == spanningCluster)
        nSpanning = m_i[cluster];     // sites in spanning cluster
    else if (m_i[cluster] > 0)
        ++n_s[m_i[cluster]];          // count clusters of size s

}

P_infinity = 0;
if (nSpanning > 0)
    P_infinity = nSpanning / (double) occupiedSites;

finiteClusters = 0;
double mSum = 0;
double mSquaredSum = 0;
for (int cluster = 1; cluster < L * L; cluster++) {
    if (cluster == spanningCluster)
        continue;
```

```
                int m = m_i[cluster];
                if (m > 0) {
                    mSum += m;
                    mSquaredSum += m * m;
                    ++finiteClusters;
                }
            }
            S = mSquaredSum / mSum;

        }

        // test method
        public static void main(String[] argv){
            int L =512;
            double p = 0.5;
            Percolation_HK perc = new Percolation_HK(L, p);
            perc.new_config();
            System.out.println(perc.spanningCluster);
        }
    }
```

Algorytm Newmana-Ziffa I

The basic idea behind our algorithm is the following. We start with a lattice in which no bonds are occupied, and hence every site is a separate cluster. Each of these single-site clusters is given a unique label (e.g., a positive integer) by which we identify it. We then fill in bonds on the lattice in random order. When a bond is added to the lattice it either connects together two sites which are already members of the same cluster—in which case we need do nothing—or it connects two sites which are members of two different clusters. In this second case we must change the labels of one of the clusters to reflect the fact that the new bond has amalgamated the two. In order to accomplish this efficiently, we store the clusters using a tree structure in which one site in each cluster is chosen to be the "root node" of that cluster and contains the cluster label.

Algorytm Newmana-Ziffa II



Rysunek 1: Two adjacent clusters in a bond percolation system. (a) The arrows represent pointers and the shaded sites are root nodes. (b) The difference in displacements (double-headed arrows) between the two sites and the root node of a cluster can be used as a criterion for detecting the onset of percolation.

All other sites in the cluster possess pointers which point either to the root node, or to another site in the cluster, such that by following a succession of such pointers one can get from any site to the root node. This scheme is illustrated for the case of the square lattice in Fig. 1a. (A similar scheme is used in the Hoshen-Kopelman algorithm [7].) Clusters can now be efficiently amalgamated simply by adding a pointer from the

root node of one to the root node of the other (dotted arrow in the figure), thereby making the former a subtree of the latter [12].

For site percolation, the algorithm is very similar to the one for the bond case just described. Sites are added in random order, and each one added either forms a new detached cluster in its own right, joins onto a single neighboring cluster, or joins together two or more extant clusters. The clusters are stored in a tree structure as before, and overall operation takes time $O(N)$.

Newmann, Ziff (2000).

Rozkład mikrokanoniczny zamiast kanonicznego

Obserwable kanoniczne $Q(p)$ i mikrokanoniczne Q_n obliczone dla ustalonej liczby n pól zajętych (obsadzonych) są ze sobą związane (splot Q_n i rozkładu dwumianowego):

$$Q(p) = \sum_{n}^{N} \binom{N}{n} p^n (1-p)^{N-n} Q_n \,.$$

Współczynniki dwumianowe można przybliżyć rozkładem Gaussowskim

$$\binom{N}{n} p^n (1-p)^{N-n} \approx \frac{1}{\sqrt{2\pi N p(1-p)}} \exp\left(-\frac{(n-Np)^2}{2Np(1-p)} \right) ,$$

w którym

$$\sigma_c = \sqrt{Np(1-p)} \,.$$

Na małych sieciach przybliżenie jest mało dokładne.

Program Newmana-Ziffa I

```
import java.util.Random;

/**
 * Site percolation on a square lattice of N = L x L sites with
 * periodic boundary conditions.
 *
 * This program prints out the size of the largest cluster on the
 * lattice as a function of number of occupied sites n for values of n
 * from 1 to N.

 * According to the paper:
 * M. E. J. Newman and R. M. Ziff, Phys. Rev. E 64, 016706 (2001);
 * http://link.aps.org/doi/10.1103/PhysRevE.64.016706,
 *
 * c code of the program:
 * http://www-personal.umich.edu/%7Emejn/percolation/index.html
 *
 * Sites are indexed with a single signed integer label for speed,
 * taking values from 0 to N-1. Note that on computers which represent
```

Program Newmana-Ziffa II

```
 * integers in 32 bits, this program can, for this reason, only be
 * used for lattices of up to 2^31, or about 2 billion sites. While
 * this is adequate for most purposes, longer labels will be needed if
 * you wish to study larger lattices.
 *
 * The array ptr[] serves triple duty: for non-root occupied sites it
 * contains the label for the site's parent in the tree (the
 * "pointer"); root sites are recognized by a negative value of ptr[],
 * and that value is equal to minus the size of the cluster; for
 * unoccupied sites ptr[] takes the value EMPTY.
 */

class Percolation_NZ {

    static final int L = 64;                    /* Linear dimension */
    static final int N = L*L;                   /* Total dimension  */
    static final int EMPTY = -N-1;              /* Empty site label */

    static int[] ptr;           /* Array of pointers (N)   */
    static int[][] nn;          /* Nearest neighbors (N,4) */
```

Program Newmana-Ziffa III

```
static int[] order;          /* Occupation order  (N)   */

static double[] Big;

Percolation_NZ() {
    ptr   = new int [N];
    nn    = new int[N][4];
    order = new int [N];
    Big   = new double[N];
    Big[0] = 0;   // there are no clusters in empty configuration
}

/**
  Now we set up the array nn[][] which contains a list of the
  nearest neighbors of each site. Only this array need be changed
  in order for the program to work with a lattice of different
  topology.
*/
void boundaries()
{
```

```
    int i;

    for (i=0; i<N; i++) {
        nn[i][0] = (i+1)%N;
        nn[i][1] = (i+N-1)%N;
        nn[i][2] = (i+L)%N;
        nn[i][3] = (i+N-L)%N;
        if (i%L==0) nn[i][1] = i+L-1;
        if ((i+1)%L==0) nn[i][0] = i-L+1;
    }
}

/**
  Now we generate the random order in which the sites will be
  occupied, by randomly permuting the integers from 0 to N-1:
*/
void permutation()
{
    int i,j;
    int temp;
```

Program Newmana-Ziffa V

```
    Random drand = new Random();

    for (i=0; i<N; i++) order[i] = i;
    for (i=0; i<N; i++) {
        j = i + drand.nextInt(N-i);
        temp = order[i];
        order[i] = order[j];
        order[j] = temp;
    }
}

/**
  We also define a function which performs the 'find' operation,
  returning the label of the root site of a cluster, as well as
  path compression.
*/
int findroot(int i)
{
    if (ptr[i]<0) return i;
    return ptr[i] = findroot(ptr[i]);
```

Program Newmana-Ziffa VI

```
}

/**
  Sites are occupied in the order specified by the array
  order[]. The function findroot() is called to find the roots of
  each of the adjacent sites. If amalgamation is needed, it is
  performed in a weighted fashion, smaller clusters being added to
  larger (bearing in mind that the value of ptr[] for the root
  nodes is minus the size of the corresponding cluster).
 */
void percolate()
{
    int i,j;
    int s1,s2;
    int r1,r2;
    int big=0;

    for (i=0; i<N; i++) ptr[i] = EMPTY;
    for (i=0; i<N; i++) {
        r1 = s1 = order[i];
```

Program Newmana-Ziffa VII

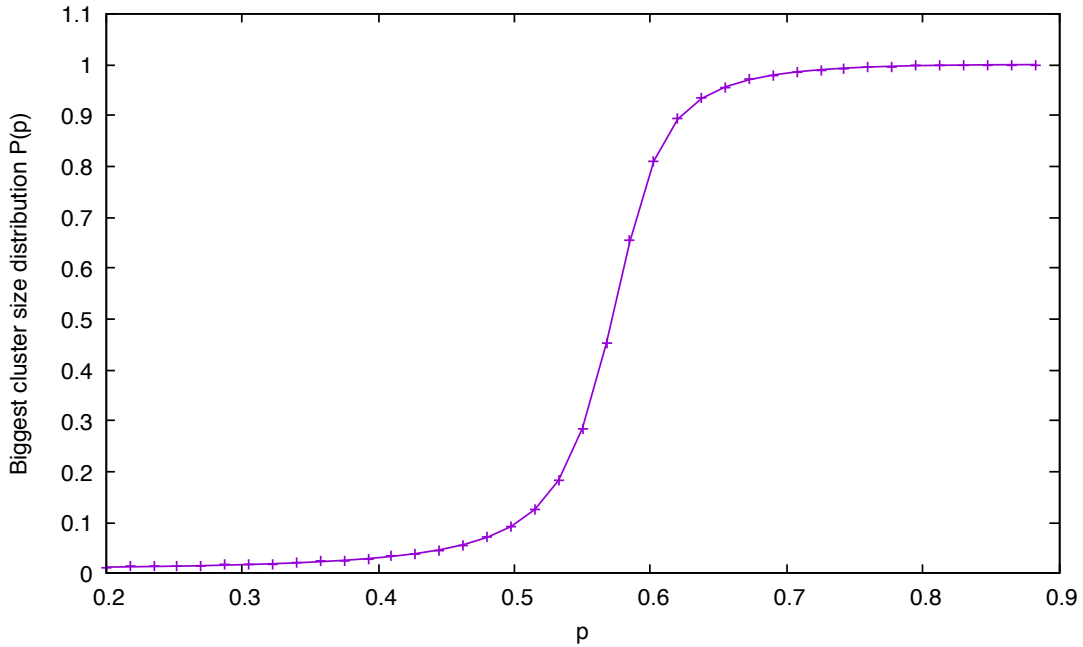
```
ptr[s1] = -1;
for (j=0; j<4; j++) {
    s2 = nn[s1][j];
    if (ptr[s2]!=EMPTY) {
        r2 = findroot(s2);
        if (r2!=r1) {
            if (ptr[r1]>ptr[r2]) {
                ptr[r2] += ptr[r1];
                ptr[r1] = r2;
                r1 = r2;
            } else {
                ptr[r1] += ptr[r2];
                ptr[r2] = r1;
            }
            if (-ptr[r1]>big) big = -ptr[r1];
        }
    }
}
//System.out.println(i+1 + " " + big);
if(i>0) Big[i] = (double)big/(double)i;
```

Program Newmana-Ziffa VIII

```
            }
        }


        public static void main(String[] args) {
            Percolation_NZ p = new Percolation_NZ();
            p.boundaries();
            p.permutation();
            p.percolate();
        }

    }
```

Parametr porządku I

Rozkład gęstości rozmiarów największego klastra $P(p)$; (definicja (1)) dla sieci 64×64 z algorytmu Newmana-Ziffa. Narysowana jest średnia wartość $P(p)$ dla 10^4 konfiguracji.



1. Przypomnienie: przybliżenie Stirlinga dla $n!$.
2. Rozkład dwumianowy
3. Rozkład kanoniczny i mikrokanoniczny
4. Obliczyć przybliżoną wartość p_c z algorytmu N-Z?
5. Narysować rozkład gęstości wielkości klastrów $n_p(s)$ dla a) $p < p_c$ b) $p = p_c$ c) $p > p_c$ (Wykresy logarytmiczne). Sprawdzić, że

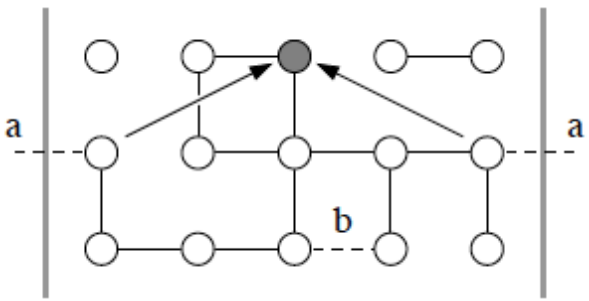
$$n_p(s) \sim \begin{cases} s^{-\theta} e^{as}\,, & p < p_c\,, \\ s^{-\tau}\,, & p = p_c\,, \\ e^{-bs^{\mathbf{1}-\mathbf{1}/d}}\,, & p > p_c\,. \end{cases}$$

gdzie wielkości $a, b, \theta, d = 2, \tau = 187/91$ są stałymi.

Detecting cluster wrapping

FIG. 7. Method for detecting cluster wrapping on periodic boundary conditions. When a bond is added (a) which joins together two sites which belong to the same cluster, it is possible, as here, that it causes the cluster to wrap around the lattice. To detect this, the displacements to the root site of the cluster (shaded) are calculated (arrows). If the difference between these displacements is not equal to a single lattice spacing, then wrapping has taken place. Conversely if the bond (b) where added, the displacements to the root site would differ by only a single lattice spacing, indicating that wrapping has not taken place.

NZ, 2001



Claster spanning

In many calculations one would like to detect the onset of percolation in the system as sites or bonds are occupied. One way of doing this is to look for a cluster of occupied sites or bonds which spans the lattice from one side to the other. Taking the example of site percolation, one can test for such a cluster by starting the lattice in the configuration shown in Fig. 6 in which there are occupied sites along two edges of the lattice and no periodic boundary conditions. (Notice that the lattice is now not square.) Then one proceeds to occupy the remaining sites of the lattice one by one in our standard fashion. At any point, one can check for spanning simply by performing „find" operations on each of the two initial clusters, starting for example at the sites marked ×. If these two clusters have the same root site then spanning has occurred. Otherwise it has not.

(*NZ, 2001*)

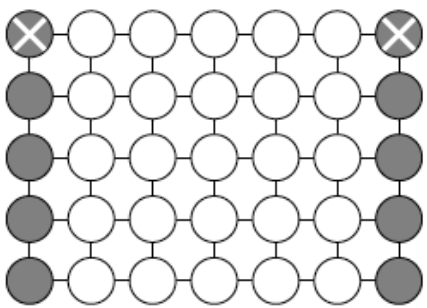


FIG. 6. Initial configuration of occupied (gray) and empty (white) sites for checking for the presence of a spanning cluster. In this example there are no periodic boundary conditions on the lattice. As the empty sites are filled up during the course of the run, the two sites marked × will have the same cluster root site if and only if there is a spanning cluster on the lattice.

Literatura

- Introduction to Computational Physics. Lecture of Prof. H. J. Herrmann. Swiss Federal Institute of Technology ETH, Zürich, Switzerland Script by Dr. H. M. Singer, Lorenz Müller and Marco - Andrea Buchmann Computational Physics, IfB, ETH Zürich.
- Computational Physics. Richard J. Gonsalves. http://www.physics.buffalo.edu/gonsalves/
- Project in Computational Physics. PERCOLATION. Jan Hasenbusch and Matthias Wilhelm. 2010/2011 (Internet).
- M. E. J. Newman, R. M. Ziff. Phys. Rev Lett. 0031-9007/00/85(19)/4104(4)
- A fast Monte Carlo algorithm for site or bond percolation M. E. J. Newman, R. M. Ziff (arXiv:cond-mat/0101295; DOI:10.1103/PhysRevE.64.016706)

Thank you!