Statistische Mechanik ungeordneter Systeme, SoSe 2005

Timo Aspelmeier, Alexander K. Hartmann, Universität Göttingen

27. April 2005

3 Perkolation

3.2 Cluster Algorithmus

Repräsentation eines d-dim Systems im Computer:
z.B. durch ein d-im Array `site`, z.B. 3-dim `site[x][y][z]`. Auch höhere Dimensionen von theoretischem Interesse, also `site [x1][x2][x3][x4][x5][x6][x7]` → zu kompliziert, zu unflexibel (andere Gitterstrukturen).

Lösung: verwende 1-dimensionales (!!) Array `site`:
`site[t]=1` falls Gitterplatz `i` besetzt.
Realisierung der Gitterstruktur Variable '`num_n`' (=Anzahl der Nachbarn) und durch Array `next`.
Jeder Gitterplatz '`t` hat hier `num_n` Nachbarn, in `next[t*num_n]`... `next[(t+1)*num_n-1]` gespeichert.
Reihenfolge: +x,-x,-y,-y,... Richtung.
Zugriff bequem über Makros:

```
#define INDEX(t, r, nn) ((t)*(nn)+r)
#define NEXT(t, r) next[INDEX(t, r, num_n)]
```

Achtung: immer wenn das Makro verwendet wird, müssen die Variablen `next` und `num_n` mit genau diesen Namen verfügbar sein. Kann man aber immer sicherstellen, wenn es nur lokal verwendet wird. Durch die Makros ist es ein wenig unflexibler, aber viel bequemer und besser lesbar.

Man braucht das Array `next[]` nur einmal am Anfang zu initialisieren, und kann es dann immer verwenden. Hier: periodische Randbedingungen:
Beispiel: $L \times L$ Quadratgitter für $L = 10$. Gitterplatz $i = 48$ hat die Nachbarn $i + 1 = 49$ (+x), $i - 1 = 47$ (-x), $i + L = 58$ (+y) und $i - L = 38$. Gitterplatz $i = 1$ hat die Nachbarn 2 (+x), 10 (-x), 11 (+y), 91 (-y).
Hauptfrage der Perkolation: gibt es einen *perkolierenden* Cluster der durch das ganze System läuft:
Zuerst *Cluster* = zusammenhängende Bereiche, bestimmen (s.u.). Dann:

a) Direktes Nachschauen, ob es einen Cluster gibt, der von einer Seite zur anderen geht.

b) Wenn das System perkoliert, muss der Größte Cluster linear mit der Systemgröße wachsen →: Anteil der Punkte am größten Cluster ausrechnen (für verschiedene Größen).

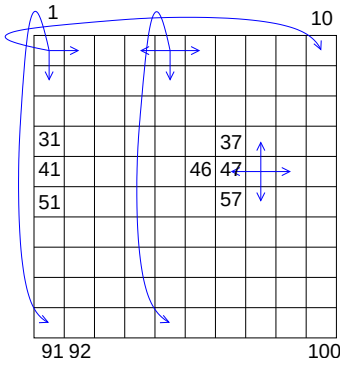


Abbildung 1: Ein 10x10 Quadratgitter mit Periodischen Randbedingungen. Die Pfeile zeigen zu den Nachbarn.

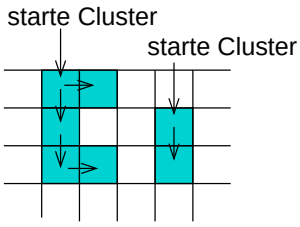
Hier: zunächst Methode b), weil einfacher zu realisieren.

Idee für Bestimmung der Cluster:
Ausgangspunkt: ein besetzter Gitterpunkt.
Alle besetzten Nachbarn gehören zum gleichen Cluster.
Alle deren besetzte Nachbarn gehören auch zum gleichen Cluster etc.
Realisierung: Die Nachbarn werden in einem Stack gespeichert und dann nach und nach verarbeitet. Man muss nur dafür Sorgen tragen, dass kein Gitterplatz doppelt behandelt wird.

algorithm Cluster
begin
  while es gibt unbehandelten Gitterplatz t **do**
  begin
    lege t auf Stack
    Starte neuen Cluster mit t
    while Stack ist nicht leer **do**
    begin
      nehme einen Gitterplatz c vom Stack
      for alle Nachbarn n von c **do**
        if n noch nicht behandelt **then**
          Legen n auf Stack und füge zum Cluster hinzu
    end
  end
end

Da jeder Gitterplatz nur einmal behandelt wird: Laufzeit $O(N)$.

Beispiel:



Unterprogramm für Clusterbestimmung:

```
/****************** percol_cluster() ******************/
/** Calculates the connected clusters of the lattice   **/
/** 'site'. Occupied sites have 'site[i]=1' (i=1..N).  **/
/** Neigbours of occupied sites form clusters.         **/
/** For each site, in 'cluster[i]' the id of the       **/
/** cluster (starting at 0) is stored                  **/
/** PARAMETERS: (*)= return-paramter                   **/
/**       num_n: number of neighbours                  **/
/**           N: number of sites                       **/
/**        next: gives neighbours (0..N)x(0..num_n-1)  **/
/**              0 not used here. Use NEXT() to access **/
/**        site: tells whether site is occupied        **/
/** (*) cluster: id of clusters sites are contained in **/
/** RETURNS: number of clusters                        **/
/*******************************************************/
int percol_cluster(int num_n, int N, int *next,
                   short int *site, int *cluster)
{
  int num_clusters=0;
  int t, r;       /* loop counters over sites, directions */
  int current, neighbour;                        /* sites */
  lstack_t *members;      /* stack of members for cluster */
  for(t=1; t<=N; t++)    /* initialise all clusters empty */
    cluster[t] = -1;
  members = lstack_new(N, sizeof(int));
  for(t=1; t<=N; t++)              /* loop over all sites */
  {
    if((site[t] == 1)&&(cluster[t]==-1)) /* new cluster ? */
    {
      lstack_push(members, &t);          /* start cluster */
      cluster[t] = num_clusters;
      while(!lstack_is_empty(members))  /* extend cluster */
      {
        lstack_pop(members, &current);
        for(r=0; r<num_n; r++)   /* loop over neighbours */
        {
          neighbour = NEXT(current, r);
          if((site[neighbour]==1)&&(cluster[neighbour]==-1))
          {          /* neighbour belongs to same cluster */
            lstack_push(members, &neighbour);
            cluster[neighbour] = num_clusters;
          }
        }
      }
      num_clusters++;
    }
  }
  lstack_delete(members);
  return(num_clusters);
}
```

Beispiellauf für 2 Dimensionen, Seitenlänge $L = 10$, $p = 0.5$). Ausgabe der Nummern der resultierenden Cluster:

```
0  0  0  0  0  0  0  0     0
0                 0     0  0  0
                  0        0
   1        0  0  0  0
   1     2        0  0     3
   1     2     4        5
      6     4  4  4  4
0                       0  0
0  0              7     0  0
0  0     0  0  0     0  0
```

3.3 Ergebnisse

Viele Programmaufrufe mittels Script `run_percol.scr`:

```
#!/bin/bash
L=$1
for p in 0.1 0.2 0.3 0.4 0.45 0.5 0.52 0.54 0.56 0.57 0.58 0.59 \
         0.60 0.61 0.62 0.64 0.66 0.68 0.7 0.8 0.9 1.0
do
  percolation1 $L $p
done
```

Ggf. muss das Script per Hand ausführbar gemacht werden: `chmod u+x run_percol.scr`. Ergebnis für 2 Dimensionen, $L = 10, \ldots, 200$:

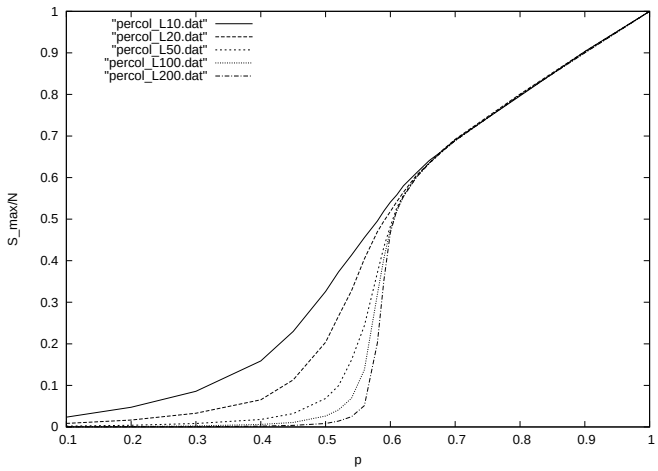


Abbildung 2: Relative Größe des maximalen Clusters als Funktion der Wahrscheinlichkeit p für Quadratgitter.

3.4 Newman-Ziff Algorithmus [1]

N: Anzahl besetztbarer Plätze
Q: Messgröße, z.B. Größe des größten Clusters.
Q_n: Mittelwert bei fester Zahl n besetzter Plätze ("mikrokanonisches Ensemble")

$Q(p)$: Mittewert bei Besetzungswahrscheinlichkeit p ("kanonisches Ensemble")

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

naiver Ansatz:

algorithm microcanonical()
begin
  for all $0 \leq n \leq N$ **do**
  begin
    Erzeuge zufällige Konfiguration mit n Gitterplätzen
    Berechne Cluster $\rightarrow$ Q_n
  end
end

$\rightarrow$ Laufzeit $O(N^2)$
besser:

algorithm Newman-Ziff()
begin
  Erzeuge zufällige Liste L von Plätzen
  for alle $s \in L$ **do**
  begin
    Besetzte Platz s
    Verbinde s und alle angrenzenden Cluster zu einem $\rightarrow$ Q_n
  end
end

Essentiell: Darstellung der Cluster als Bäume:

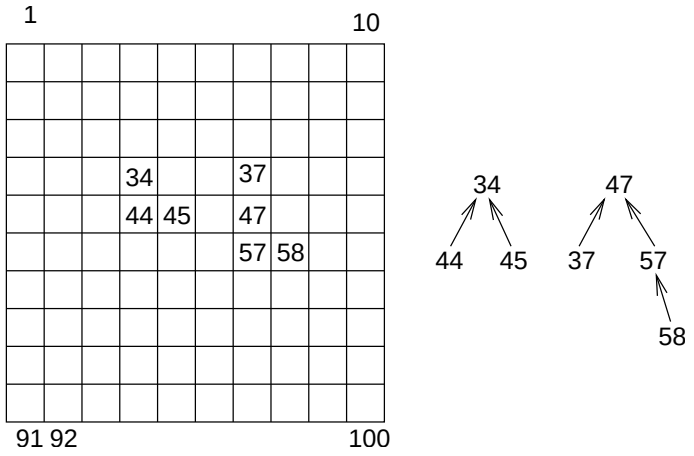


Abbildung 3: Zwei Cluster im Quadratgitter als Bäume

Datenstruktur:

```
typedef struct p_node
{
  int id;                                /* to identify node */
  struct p_node *parent;             /* points to predecessor */
  int              size; /* of subtree with this node as root */
  int            *delta; /* difference vector to site at root */
  int              *pos; /* position of site                  */
} percol_node_t;
```

Darstellung: nicht eindeutig, entsteht dynamisch:
Verbindung von Clustern:
Hänge kleineren Baum an die Wurzel des größeren Baums.
Neue besetzter Gitterplatz: zunächst Baum der Größe 1.

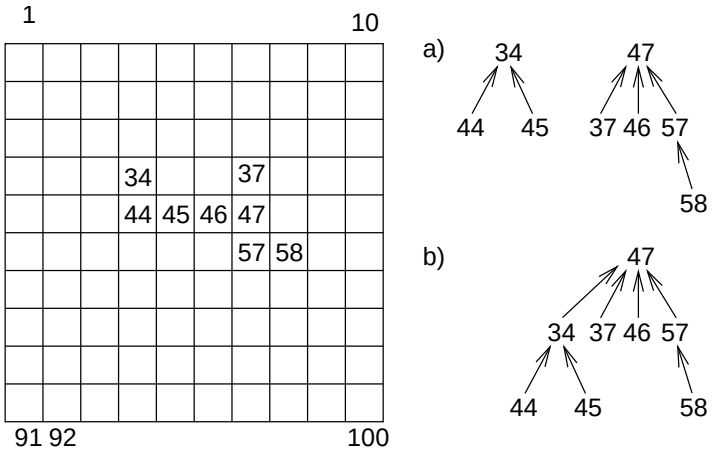


Abbildung 4: Besetzen von Gitterplatz 46: Wird mit Nachbarclustern verbunden → beide Bäume werden in zwei Schritten a),b) zu einem vereinigt.

```
/****************** percol_join_trees() ****************/
/** Merges two trees into one. The smaller is attached **/
/** to the larger one.                                 **/
/** The merging occurs, because two sites node1,node2  **/
/** become adjacent.                                   **/
/** Also the delta displacments vectors are updated    **/
/** PARAMETERS: (*)= return-paramter                   **/
/**    root1: root of 1st tree                         **/
/**    diff1: vector from node1 to root1               **/
/**    root2: root of 2nd tree                         **/
/**    diff1: vector from node1 to root1               **/
/** diff_loc: vector from node1 to node2               **/
/**     glob: pointer to global data                   **/
/** RETURNS:                                           **/
/**       pointer to common root                       **/
/*******************************************************/
percol_node_t *percol_join_trees(percol_node_t *root1, int *diff1,
```

```
                                    percol_node_t *root2, int *diff2,
                                    int *diff_loc, percol_glob_t *glob)
{
  int d;
  if(root1->size > root2->size)
  {
    root2->parent = root1;                /* attach tree */
    root1->size += root2->size;         /* updated sizes */
    if(root1->size > glob->largest)   /* update largest */
      glob->largest = root1->size;
    for(d=0; d<glob->dim; d++)
      root2->delta[d] = -diff2[d]-diff_loc[d]+diff1[d];
    return(root1);                    /* return new root */
  }
  else
  {
    root1->parent = root2;                /* attach tree */
    root2->size += root1->size;         /* updated sizes */
    if(root2->size > glob->largest)   /* update largest */
      glob->largest = root2->size;
    for(d=0; d<glob->dim; d++)
      root1->delta[d] = -diff1[d]+diff_loc[d]+diff2[d];
    return(root2);                    /* return new root */
  }
}
```

Hilfsfunktion:
Test ob zwei Plätze s_1, s_2 im gleichen Baum:

algorithm same-tree(s_1, s_2)
begin
  while parent(s_1) $\neq 0$ **do**
    s_1 =parent(s_1);
  while parent(s_2) $\neq 0$ **do**
    s_2 =parent(s_2);
  if $s_1 = s_2$ **then**
    return(same tree);
  else
    return(different trees);
end

Verwende *Pfadkomprimierung*:
Immer wenn man von s zur Wurzel läuft:
alle Zeiger werden auf Wurzel umgesetzt.

$\rightarrow$ Laufzeit $O(N)$ [1]
Feststellung ob Perkolation (Cluster läuft um System herum) vorliegt:

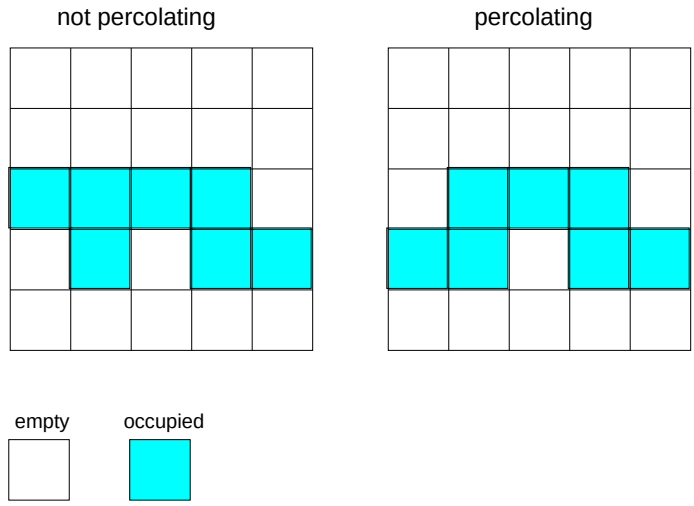


Abbildung 5: Nicht perkolierender (links) und perkolierenden (rechts) Cluster.

Speichere für jeden Gitterplatz s den Vektor
$\underline{d}(s)$= Differenz-Vektor zum Wurzel-Platz des Cluster Baumes *entlang* besetzter Gitterplätze.

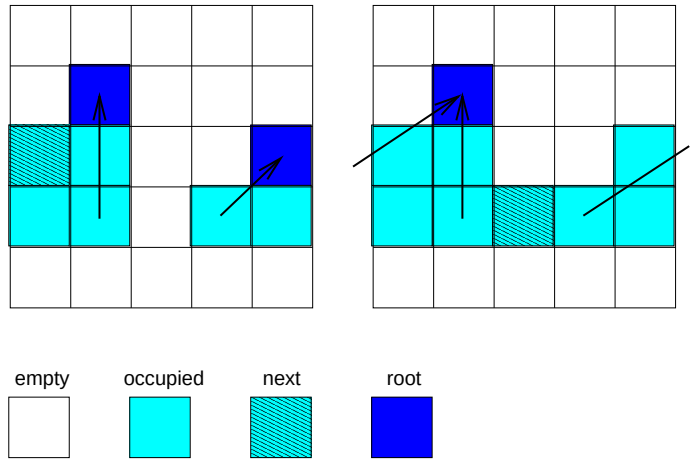


Abbildung 6: Differenzvektoren zur Feststellung der Perkolation. Nach Besetzen des Gitterplatzes links entsteht die rechte Situation. Das weiter Besetzen erzeugt einen perkolierenden Cluster.

Immer wenn zwei benachbarte Gitterplätze s_1, s_2 schon im gleichen Baum sind:
(erstmalig) Perkolation falls $\underline{d}(s_1) - \underline{d}(s_2)$ *kein* Gittervektor $(\pm 1, 0), (0, \pm 1)$ ist.

3.5 Ergebnisse2

Programmläufe

```
percolation2 1000 10 > run10.out
percolation2 1000 20 > run20.out
percolation2 1000 40 > run40.out
percolation2 1000 100 > run100.out
percolation2 1000 200 > run200.out
```

Ergebnis für 2 Dimensionen, $L = 10, \ldots, 200$:

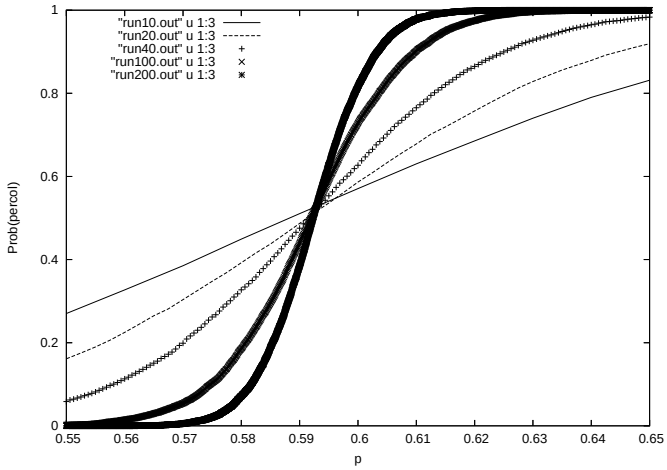


Abbildung 7: Wahrscheinlichkeit für einen in x-Richtung perkolierenden Cluster als Funktion von p.

Literatur

[1] M.E.J. Newman and R.M. Ziff, Efficient Monte Carlo Algorithm and High-Precision Results for Percolation, Physical Review Letters **85**, 4104 (2000)