Computerphysik Programmiertutorial 13

Prof. Dr. Matteo Rizzi und Dr. Markus Schmitt – Institut für Theoretische Physik, Universität zu Köln

ILIAS: https://www.ilias.uni-koeln.de/ilias/goto_uk_crs_3862489.html

Github: https://github.com/markusschmitt/computphys2021

Inhalt dieses Notebooks: Cluster finden mit Rekursion, Verkehrssimulation, Herangehensweise für numerische Experimente

Cluster finden mit Rekursion

Wir wollen in einem zufällig besetzten Quadratgitter den größten zusammenhängenden Cluster finden. Am Ende wollen wir die Clustergröße als Funktion der Besetzungswahrscheinlichkeit untersuchen.

Erzeugen wir also zunächst ein Quadratgitter:

In [3]:
```
using PyPlot, StatsLinx, Random
```

In [4]:
```
function get_configuration(N, p)
    return reshape((rand(1:p) for j in 1:N^2), (N,N))
end

G=get_configuration(10,0.4)

imshow(G)
axis('off');
```

Die Aufgabe zusammenhängende Cluster zu finden eignet sich gut für eine rekursive Implementierung.

Wir arbeiten dafür mit einem weiteren Array `C`, das mit Null initialisiert wird. Dann werden wir die einzelnen Cluster suchen und durchnummerieren. Am Ende soll in `C` auf jedem Gitterplatz, der zu einem Cluster gehört die Nummer des Clusters stehen, oder Null, falls der Platz zu keinem Cluster gehört.

Wir schreiben dafür eine Funktion `find_cluster!`, die überprüft ob ein Gitterplatz `i,j` besetzt ist und ob er bereits zu einem Cluster gehört (also ob `C[i,j]==0`). Falls der Platz besetzt ist, wird die Rekursion betreten. Falls der Gitterplatz nicht besetzt ist, wird er dem aktuellen Cluster hinzugefügt und `find_cluster!` wird für die benachbarten Gitterplätze aufgerufen.

Die Funktion soll außerdem direkt die Clustergröße bestimmen, daher zählen wir mit wie viele Gitterplätze hinzugefügt werden und geben die Zahl zurück.

In [5]:
```
function find_cluster!(clusterIdx, i,j,C,G,step=1)
    N = size(G)[1]

    if !G[i,j] || C[i,j] > 0
        return step-1
    end

    C[i,j] = clusterIdx

    if i>1
        step = find_cluster!(clusterIdx, i-1, j, C, G, step+1)
    end
    if j>1
        step = find_cluster!(clusterIdx, i, j-1, C, G, step+1)
    end
    if i<N
        step = find_cluster!(clusterIdx, i+1, j, C, G, step+1)
    end
    if j<N
        step = find_cluster!(clusterIdx, i, j+1, C, G, step+1)
    end

    return step
end
```

Out[5]:
```
find_cluster! (generic function with 2 methods)
```

Test der Funktion:

In [6]:
```
C=zeros(Int, 10,10)

find_cluster!(17,1,1,C,G)
```

Out[6]:
```
4
```

In [7]:
```
C
```

Out[7]:
```
10×10 Matrix{Int64}:
 17  17  0  0  0  0  0  0  0  0
 17  17  0  0  0  0  0  0  0  0
 17  17  0  0  0  0  0  0  0  0
  0   0  0  0  0  0  0  0  0  0
  0   0  0  0  0  0  0  0  0  0
  0   0  0  0  0  0  0  0  0  0
  0   0  0  0  0  0  0  0  0  0
  0   0  0  0  0  0  0  0  0  0
  0   0  0  0  0  0  0  0  0  0
  0   0  0  0  0  0  0  0  0  0
```

Um den größten Cluster zu finden, starten wir `find_cluster!` auf jedem Gitterplatz um alle Cluster zu identifizieren. In der Schleife merken wir uns gleich welche die maximale Clustergröße ist.

In [18]:
```
function find_largest_cluster(G)
    N = size(G)[1]
    Nk = size(G)[1]
    C = zeros(Int, N, N)

    k = 1
    for i in 1:N
        for j in 1:N
            new_size = find_cluster!(k, i, j, C, G)
            if new_size > 0
                k+=1
            end
            if new_size > max_size
                max_size = new_size
            end
        end
    end
    return max_size, C
end

maxSize, C = find_largest_cluster(G)
```

Out[18]:
```
(44, [1 1 … 0 3; 1 0 … 3 3; … ; 9 9 … 3 3; 0 9 … 3 3])
```

In [19]:
```
imshow(C)
axis('off');
```

Jetzt berechnen wir für eine Reihe von Besetzungswahrscheinlichkeiten $p \in [0,1]$ die maximalen Clustergrößen und tragen sie im Plot auf:

In [121]:
```
Random.seed!(42)

N=100
p=range(0,1,step=0.05)
mean_cluster_sizes=[]

for p in p
    tmp_sizes=[]
    for j in 1:100
        G=get_configuration(N,p)
        s, C = find_largest_cluster(G)
        push!(tmp_sizes, s)
    end
    push!(mean_cluster_sizes, mean(tmp_sizes))
end

plot(p, mean_cluster_sizes/N^2)
xlabel("Besetzungswahrscheinlichkeit")
ylabel("Normierte Clustergröße");
```

Verkehrssimulation

Wir wollen den Verkehrsfluss auf einer einspurigen Straße simulieren und dabei die Abhängigkeit von der Verkehrsdichte ρ untersuchen.

Wir modellieren das Problem wie folgt: Die Straße besteht aus L diskreten Positionen auf einem Ring (periodische Randbedingungen). Jede Position kann leer oder mit einem Fahrzeug besetzt sein. Die Fahrzeuge bewegen sich in diskreten Zeitschritten nach den folgenden Regeln:

1. Falls die nächsten zwei Gitterplätze vor einem Fahrzeug frei sind, rückt es um einen Platz vor.
2. Falls der nächste Gitterplatz vor einem Fahrzeug frei ist, der übernächste aber besetzt, rückt das Fahrzeug mit einer Wahrscheinlichkeit von $p = 0.5$ vor (Bremsen bzw. verzögertes Anfahren).
3. Falls der nächste Gitterplatz vor einem Fahrzeug besetzt ist, rückt es nicht vor.

In [114]:
```
function get_initial_config(rho, L)
    config = zeros(Int, L)
    num_vehicles = Int(rho*L)

    occupied_sites = randperm(L)[1:num_vehicles]

    config[occupied_sites] .= 1

    return config
end

L=100
rho=0.3
config = get_initial_config(rho, L)
```

Out[114]:
```
100-element Vector{Int64}:
 0
 1
 0
 1
 0
 0
 0
 0
 0
 0
 1
 0
 1
 1
 0
 0
 0
 1
 1
 1
 1
 0
 1
 0
 0
 0
 0
 0
```

In [115]:
```
sum(config)
```

Out[115]:
```
30
```

Jetzt implementieren wir eine Funktion, die für eine gegebenen Fahrzeugkonfiguration einen Zeitschritt entsprechend der Regeln durchführt. Gleichzeitig ermittelt die Funktion die Mittlere Geschwindigkeit, also die Zahl der vorgerückten Fahrzeuge geteilt durch die Gesamtzahl an Fahrzeugen.

In [179]:
```
idx(i,L)=i%L+(i%L<0)*L

function step(config)
    L = length(config)
    new_config = zeros(Int, L)

    v = 0
    for j in 1:L
        if config[j] == 1
            if config[idx(j+1,L)] == 0
                if config[idx(j+2,L)] == 0
                    new_config[idx(j+1,L)] = 1
                    v += 1
                else
                    if rand()<0.5
                        new_config[idx(j+1,L)] = 1
                        v += 1
                    else
                        new_config[j] = 1
                    end
                end
            else
                new_config[j] = 1
            end
        end
    end

    return new_config, v/sum(new_config)
end
```

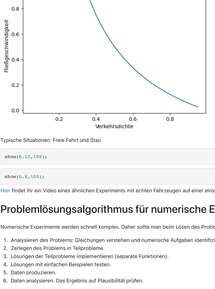
Out[179]:
```
step (generic function with 1 method)
```

Nun können wir uns einen "Film" des Verkehrsflusses ansehen:

In [139]:
```
using Printf

function show(rho, L, steps=100)
    pygui(true);
    config = get_initial_config(rho, L)

    axis('off')
    for t in 1:steps
        config, v = step(config)
        imshow(reshape(config,(1,L)))
        title(@sprintf("Geschwindigkeit: %.2f", v))
        sleep(0.2)
    end
    pygui(false);
    return nothing
end

show(0.4,100, 100)
```

Bestimme die mittlere Fließgeschwindigkeit in Abhängigkeit von der Verkehrsdichte ρ:

In [221]:
```
Random.seed!(123)

rhos=range(0.05,0.95,step=0.05)

L=1000

v_means=[]

for rho in rhos
    config = get_initial_config(rho, L)
    v_sum = 0
    for t in 1:10000
        config, v = step(config)
        v_sum += v
    end
    push!(v_means, v_sum/10000)
end

plot(rhos, v_means);
xlabel("Verkehrsdichte");
ylabel("Fließgeschwindigkeit");
```



Typische Situationen: Freie Fahrt und Stau

In [143]:
```
show(0.15,100);
```

In [249]:
```
show(0.8,100);
```

Hier findet ihr ein Video eines ähnlichen Experiments mit echten Fahrzeugen auf einer einspurigen Fahrbahn, das zeigt wie ein Stau aus dem Nichts entsteht.

Problemlösungsalgorithmus für numerische Experimente

Numerische Experimente werden schnell komplex. Daher sollte man beim Lösen des Problems schrittweise vorgehen um den Überblick zu behalten und Fehler frühzeitig zu identifizieren:

1. Analysieren des Problems: Gleichungen verstehen und numerische Aufgaben identifizieren.
2. Zerlegen des Problems in Teilprobleme.
3. Lösungen der Teilprobleme implementieren (separate Funktionen).
4. Lösungen mit einfachen Beispielen testen.
5. Daten produzieren.
6. Daten analysieren. Das Ergebnis auf Plausibilität prüfen.