Institut für Informatik

Lehrstuhl für Technische Informatik
Prof. Dr. Björn Scheuermann

Peer-to-Peer-Systeme WS 2015/16

Übungsblatt 3a

Besprechung am 3. Dezember 2015

Aufgabe 9

Watts und Strogatz haben einen Algorithmus vorgeschlagen, um Small-World-Graphen mit vorgegebener Knoten- und Kantenzahl zu erzeugen.

Dabei werden n Knoten zunächst in einem (gedachten) Kreis angeordnet. Jeder Knoten wird dann mit seinen k nächsten Nachbarn im Uhrzeigersinn verbunden. Nachdem dieser (vollständig deterministische) Ausgangsgraph erzeugt wurde, wird ein Teil der Kanten in einem zufallsbasierten Schritt ausgetauscht. Dabei wird für jede Kante im Ausgangsgraphen einmal gewürfelt: Mit einer vorgegebenen Wahrscheinlichkeit p wird die Kante verändert. Wird eine Kante auf diese Weise zum Verändern ausgewählt, dann bleibt ihr Startknoten gleich, ihr Zielknoten wird aber aus der Menge aller Knoten zufällig neu gewählt (jeder Knoten außer dem Startknoten der Kante wird dabei mit gleicher Wahrscheinlichkeit gewählt).

Obwohl während der Konstruktion für die Kanten von „Startknoten“ und „Zielknoten“ ausgegangen wird, wird der resultierende Graph als ungerichteter Graph betrachtet.

Der so letztendlich entstehende Graph ist also für $p = 0$ vollständig systematisch aufgebaut. Er wird einem Zufallsgraphen immer ähnlicher, je näher p an den Wert 1 kommt – weil dann ein immer größerer Teil der Kanten zufällig gewählt wird.

(a) Wie viele Kanten enthält der Graph, in Abhängigkeit von k und n? (Nimm an, dass $k \ll n$.)

> **Lösung:** Die Gesamtzahl der Kanten ist $k \cdot n$. Jeder Knoten hat k („ausgehende“) Kanten zu Nachbarn im Uhrzeigersinn und k („eingehende“) Kanten zu Nachbarn gegen den Uhrzeigersinn. Jede Kante „zählt“ bei zwei Knoten, also gibt es insgesamt $2kn/2 = kn$ Kanten.

(b) Was ist der Durchmesser des Ausgangsgraphen? (Nimm an, dass $n = t \cdot k$ für eine gerade natürliche Zahl $t \geq 2$.)

> **Lösung:** Der Durchmesser ist $t/2$, denn um von einem gegebenen Knoten aus den gegenüberliegenden Knoten zu erreichen benötigt man $t/2$ Schritte über jeweils k Knoten hinweg (für eine volle Runde benötigt man genau t solche Schritte, und t ist lt. Annahme eine gerade Zahl).
>
> Jeder Knoten kann in höchstens $t/2$ Schritten erreicht werden: Je nachdem, ob der Zielknoten im oder gegen den Uhrzeigersinn schneller zu erreichen, ist macht man zunächst solange Schritte über k Knoten, bis die verbleibende Distanz $\leq k$ ist, danach noch einen Schritt der entsprechenden Länge. Aufgrund der Verbindungsstruktur im Ausgangsgraphen ist das immer möglich.

Aufgabe 10

Schreibe ein Programm (in einer Programmiersprache deiner Wahl), das für vorgegebene Parameter n, k, p Watts-Strogatz-Graphen wie in der vorherigen Aufgabe beschrieben erzeugt. Das Programm muss den Graphen selbst nicht ausgeben. Es soll aber den Clustering-Koeffizienten und die durchschnittliche Pfadlänge des erzeugten Graphen bestimmen und ausgeben. Damit für die Bestimmung der durchschnittlichen Pfadlänge nicht alle Knotenpaare untersucht werden müssen, genügt es, wenn das Programm 100 Knotenpaare zufällig auswählt und diese Paare als Stichprobe für die Berechnung der Durchschnittspfadlänge verwendet.

Das „Vorrechnen“ in der Übung besteht darin, die prinzipielle Vorgehensweise des Programms zu erläutern und an ausgewählten wichtigen Stellen im Quelltext zu veranschaulichen. Bringe deshalb unbedingt dein Programm auf einem USB-Stick mit. Alternativ kannst du es auch bis spätestens am Tag vor der Übung per E-Mail an scheuermann@informatik.hu-berlin.de schicken.

Lösung:

```
#!/usr/bin/ruby

N = 5000
K = 2
P = 0.01
PATH_SAMPLES = 100

INFINITY = 1.0/0.0

require 'set'


# Konstruktion des Ausgangsgraphen als gerichteter Graph, dargestellt durch Adjazenzlisten

edges = Array.new
for i in 0..(N-1)
  edges[i] = Array.new
  for j in (i+1)..(i+K)        # Kanten zu den K naechsten Knoten im Uhrzeigersinn erzeugen
    edges[i].push(j % N)
  end
end


# Zufaelliges veraendern des Zielknotens der Kanten mit Wahrscheinlichkeit P

for i in 0..(N-1)
  for j in 0..(K-1)
    if rand < P
      newnode = rand(N-1)            # neues Ziel waehlen
      newnode += 1 if newnode >= i   # sicherstellen, dass keine Kante von i auf sich selbst erzeugt wird
      edges[i][j] = newnode
    end
  end
end


# Ungerichteten Graphen aus dem Ausgangsgraphen erzeugen, durch Adjazenzmengen dargestellt

neighbors = Array.new(N) { Set.new }    # eine Liste von N leeren Nachbar-Mengen
for i in 0..(N-1)
  for j in edges[i]
```

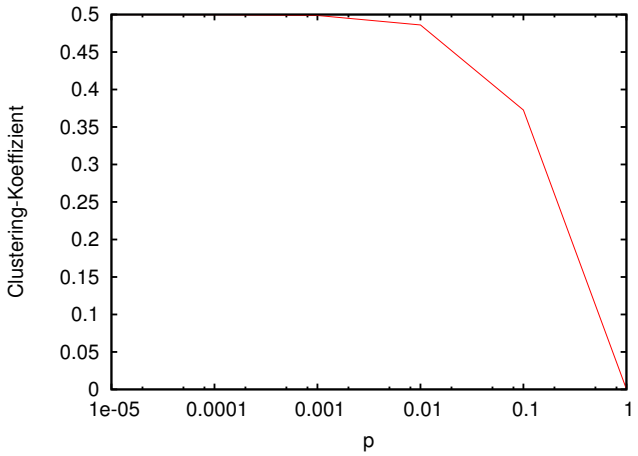
```
    neighbors[i].add(j)
    neighbors[j].add(i)
  end
end


# Clustering-Koeffizienten bestimmen

sum_ccoeff = 0.0
n = 0              # Zahl der einbezogenen Knoten
for i in 0..(N-1)
  num_neighbors = neighbors[i].size
  next if num_neighbors <= 1      # Knoten mit zu wenigen Nachbarn ignorieren
  ccoeff = 0.0               # Clustering-Koeffizient des i-ten Knotens
  for j in neighbors[i]      # alle Paare von Nachbarn von i durchlaufen
    for k in neighbors[i]    # jedes Paar wird nur einmal gezählt, weil Paare mit j >= k ignoriert werden
      ccoeff += 1 if j < k and neighbors[j].include?(k)
    end
  end
  ccoeff /= num_neighbors * (num_neighbors - 1) / 2
  sum_ccoeff += ccoeff
  n += 1
end

puts "Clustering-Koeffizient: #{sum_ccoeff / n}"       # durchschnittl. Clustering-Koeffizienten ausgeben


# Durchschnittl. Pfadlaenge bestimmen

sum_pathlength = 0.0
PATH_SAMPLES.times do
  # Start- und Zielknoten zufaellig waehlen
  from = rand(N)
  to = rand(N)

  # vereinfachter Dijkstra-Algorithmus zum Bestimmen der Distanz
  dist = Array.new(N,INFINITY)  # mit unendlich initialisiertes Array der Entfernungen vom Startknoten
  dist[from] = 0                # der Startknoten ist von sich selbst 0 Schritte entfernt
  done = Set.new                # Menge der bereits abgearbeiteten Knoten
  nearest_undone = nil
  until nearest_undone == to
    # nächstgelegenen noch nicht bearbeiteten Knoten suchen
    nearest_undone = to
    for i in 0..(N-1)
      nearest_undone = i if dist[i] < dist[nearest_undone] and not done.include?(i)
    end
    # Kanten dieses Knotens abarbeiten und Distanzen aktualisieren
    for i in neighbors[nearest_undone]
      dist[i] = [dist[i], dist[nearest_undone] + 1].min
    end
    done.add(nearest_undone)
  end

  sum_pathlength += dist[to]
end

puts "Durchschnittliche Pfadlaenge: #{sum_pathlength / PATH_SAMPLES}"
```

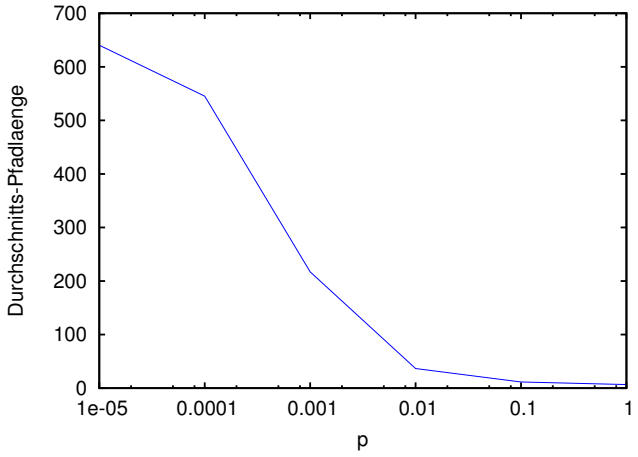
Aufgabe 11

Verwende das in der vorangegangenen Aufgabe erstellte Programm für folgendes Experiment: Setze fest $n = 5000$, $k = 2$. Variiere p, indem du nacheinander die Werte $p = 0.00001$, $p = 0.0001$, $p = 0.001$, $p = 0.01$, $p = 0.1$ und $p = 1$ wählst. Erzeuge (z. B. mit Gnuplot, LibreOffice oder Excel) zwei Diagramme, eines für den Clustering-Koeffizienten und eines für die durchschnittliche Pfadlänge, jeweils in Abhängigkeit von p (auf der x-Achse). Wähle für die x-Achse eine logarithmische Skalierung, damit man alle Werte klar auseinanderhalten kann.

Wenn du die beiden erzeugten Graphen vergleichst und interpretierst, was sind deine Schlussfolgerungen?

Lösung:





Beobachtung: Die durchschnittliche Pfadlänge fällt sehr schnell und wird selbst für sehr wenige zufällig gewählte Kanten schon klein. Der Clustering-Koeffizient bleibt dagegen lange hoch, und fällt erst, wenn p näher an 1 kommt. Für „mittlere“ p erhält man also Small-World-Graphen, die gleichzeitig einen hohen Clustering-Koeffizienten und kurze Pfadlängen haben.