**VL 15.10.15**

CLIQUE (Entscheidungsproblem):

Eingabe: Graph G = (V,E), k \elem \mathbb{N}

Frage: Hat G eine Clique mit min k Knoten?

Dieses Problem gehört zu den Entscheidungsproblemen. Entscheidungsprobleme gehören zu den einfachsten Problemen.

CLIQUE (Suchversion): <- Optimierungsproblem

Eingabe: Grap G = (V,E)

Aufgabe: Bestimme eine Clique C in G mit max. Knotenzahl.

Behauptung: Wenn Clique in polynomieller Zeit lösbar ist ⬄ Clique-Suchversion in polynomieller Zeit lösbar.

(Reden wir von Laufzeit, dann in Bezug auf die Eingabelänge)

Beweis: <= ist offensichtlich… weil Clique-Suchversion das schwierigere Problem ist?!

* Hab ich Algo um Suchversion in poly. Zeit zu lösen, dann muss ich nur die Knoten der max. Clique zählen und mit k vergleichen => Ergebnis für Entscheidungsproblem auch in poly. Zeit lösbar. (informell)
* Zur Eingabe (G,k) von CLIQUE bestimmen wir zunächst den Algo. Für CLIQUE-Suchversion eine größte Clique C von G. Dann vergleiche |C| >= k? Ausgabe: ja, falls |C| min k Knoten hat. Sonst nein.
* Laufzeit: wie die Laufzeit von Algo. für CLIQUE-Suchversion

„=>“: Betrachte: folgende Zwischenversion für CLIQUE:

CLIQUE-Zwischenversion:

Eingabe: Graph G = (V,E)

Aufgabe: Berechne die Maximalzahl kleinOmega(G) von paarweise verbundenen Knoten in G.

(Def.: kleinOmega(G) = max{|C|: C ist Clique von G} ist die Cliquenzahl von G)

Behauptung: CLIQUE in poly. Zeit lösbar =>^(1) CLIQUE-Zwischenversion in poly. Zeit lösbar =>^(2) CLIQUE-Suchversion in poly. Zeit lösbar.

Zu (1): - Sei A ein Algo, in poly. Zeit lösbar, für CLIQUE.

* Zur Eingabe G von CLIQUE-Zwischenversion wende A auf G maximal n-mal bis A als antwort ‚nein‘ liefert, um Ergebnis kleinOmega(G) zu erhalten bzw. wende A auf Eingaben (G, n)\*, (G, n-1), (G, n-2)…! \* (G,n) ^= Eingabe G und k = n = |V|
* Ausgabe: kleinOmega(G) = erstes k in (G, n), (G,n-1).. mit A(G,k) = ‚ja‘
* Pseude: for k = n to 1 do {if A(G,k) = ‚JA‘ then return kleinOmega(G) = k}
* Laufzeit: n \* O(n^c) // O(n^c) Laufzeit von A und maximal n Durchläufe um k zu bestimmen

= O(n^(c+1))

Zu (2): Sei A ein Algo, in poly. Zeit lösbar, für CLIQUE-Zwischenversion, d.h. A(G) = kleinOmega(G) in O(n^d) für eine Konstante d.

* Zur Eingabe G von CLIQUE-Suchversion wende folgendes an:

For jede Kante e \elem E do

If (kleinOmega(G-e) = kleinOmega(G) // d.h. G wird durch G-e ersetzt, weil Kante e für Problem unwichtig

Then G = G-e // G-e = Graph G ohne Kante e

Ausgabe C:= V(G) ohne isolierte Knoten(=Knoten ohne Nachbar)

* Laufzeit: m \* O(n^d) = O(n^(d+2)) // d+2, weil m <= n^2

q.e.d.

Zusammenfassung Entscheidungsprobleme:

Einfachste Version von PRIM: „Ist n Primzahl?“

Genauer -> Probleme definieren wir in dieser Art.

PRIM:

Eingabe: natürliche Zahl n

Frage: Ist n Primzahl?

Eingabelänge log n =: N < n => Laufzeit: O(n) = O(2^(log n)) => ALgo. in Bezug auf Eingabelänge ist sehr ineffizient, da exponentiell

Formal:

PRIM = {bin(n) |n \elem N ist Primzahl} Teilmenge {0,1}^+ // bin(n) = binäre Darstellung von n

SORTIERUNG

Eingabe: a\_1, a\_2, …, a\_n

Aufgabe: sortiere die Zahlen aufsteigend

Mergesort: O(n\*log n) <- ist in unserem Sinne, bezogen auf die Eingabelänge, zu grob

Eingabelänge: N = log a\_1 + log a\_2 + … + log a\_n (quasi Anzahl der Bits um Eingaben darzustellen, ohne Trennzeichen) und N >= n

O(n \* log n) = O(N \* log N) = O(N^2) , wegen N >= n, aber O-Notation nur von links nach rechts lesbar!!

eigl heißt das f \elem O(nlogn) => f \elem O(NlogN)

**P und NP**

P = Menge aller Entscheidungsprobleme, für die ein deterministischer polyzeit Algo. existiert. (formal via determ. Turingmaschine) 🡺“P enthält quasi effizient lösbare Probleme“

Ein Optimierungsproblem ist effizient lösbar, wenn die entsprechende Entscheidungsversion in P ist.

Grundlegende Probleme in P:

* Sortieren:
* PRIM (aber nicht der Algo, den wir oben besprochen haben.) – Lösung ist nicht trivial; erst 2002 ersten Algo in poly Zeit gefunden. (https://de.wikipedia.org/wiki/AKS-Primzahltest)
* 2-SAT: Erfüllbarkeit für KNF, wobei jede Klausel max. 2 Literale besitzt

**Notizen 22.10.15**

3-SAT <=\_p SUBSETSUM – siehe VL

3-SAT <=\_p CLIQUE -- Komplexe Formale Sprachen

BSP: 3-SAT <=\_p INDSET

Eingabe: Sei F = K1 und K2 und… und Km mit Kj = (lj1 oder lj2 oder l\_j\_3) wobei l\_j,1, l\_j,2, l\_j,3 elem {x1,… xn} UNION {not x1, … , not xn}

Erklärung Notation:

F = (x\_1 \or \not x\_2 \or \not x\_3) \and (\not x\_1 \or x\_2 \or x\_3)

--- K1 ---- --- K2 ----

L\_1,2 = \not x\_“ l\_2,2 = x\_2

Konstruiere Graphen G und natürliche Zahl k aus Formel F wie folgt:

* Literale entsprechen Knoten
* 3 Literale in einer Klausel sind miteinander verbunden
  + Bsp: K\_j = Graph mit lj1, lj2, lj3 und alle Knoten sind untereinander verbunden
* Zwei Knotenaus verschiedenen Klauselmengen sind verbunden, wenn die entsprechenden Literale zueinander komplementär sind
* Graph zum Bsp F V= x1, !x1, x2, !x2, x3, !x3,

E = x1!x1, !x2x2, !x3x3 (und alle Knoten innerhalb einer Klausel miteinander)

Zusätzlich K3 = x1 or x2 or x4

* Graph zusätzlich E = x1x2, x2x4, x1x4, x1!x1, x2!x2 und V UNION x4
* k = m (die Anzahl der Klauselmengen)
* => Aus Formel F haben wir Eingabe in Form von Graph G und k definiert

„=>“

Annahme: Sei b eine erfüllende Belegung für F, d.h. in jeder Klausel Kj existiert ein Literal l\_j,k mit b(l\_j,k) = 1.

Wähle in G aus jeder Klauselmenge genau einen Knoten mit entsprechendem wahren Literal.

* Diese m = k Knoten sind nach Konstruktion von G unverbunden.

Bsp: b(x1) = 1 = b(x2) ; b(x3) = b(x4) = 0

* Ausgewählte Knoten sind x2, !x3, x1 => ergeben unabhängige Menge

Andere Beweisrichtung: „<=“

Annahme: Sei U eine unabhägnige Menge von G mit |U| >= k = m Knoten. Dann |U| = k weil jede der m Klauselmengen ein Dreieck ist, d.h. |U Schnitt Kj| = 1 für jede Klauselmenge Kj

U definiert eine Belegung für F: b(l\_j,k) = 1, falls l\_j,k \elem U, sonst 0

b(l\_j,k) = 1 => b(!l\_j,k) not elem U => b(!l\_j,k) = 0

= > F ist erfüllbar unter b!

Bsp: b(x1) = 0 = b(x2); b(x4) = 1, b(x3) = 0

Weiteres Reduktionsbeispiel

**3-SAT <=\_p 3-Färbung**

Eingabe: Sei F = K1 und K2 und… und Km mit Kj = (lj1 oder lj2 oder l\_j\_3) wobei l\_j,1, l\_j,2, l\_j,3 elem {x1,… xn} UNION {not x1, … , not xn}

Bsp: F = k1 und k2 und k3 = (x1 or !x2 or !x3) and (!x1 or x2 or x3) and (x1 or x2 or x4)

„=>“ Graph 3-färbbar => F ist erfüllbar

Konstruiere Graphen G wie folgt:

* Literale entsprechen Knoten: (--- = Kante zwischen den Knoten)

X1 ---- !x1 x2 --- !x2 x3 ---!x3 x4 ----!x4

* Idee: benötigen 2 Farben für diese Knoten zum Färben -> Interpretieren diese Farben als TRUE and FALSE
* Füge Knoten Hilfsknoten R ein der mit allen x1..x4 verbunden ist -> dieser benötigt die 3. Farbe und erzwingt die 2 Farben für x1..x4
* Für jede Klausel: Kj:

V = {aj, bj, dj, ej, cj}; E = {ab, ad, bd, de, ec}

Wenn Kj = (l\_j1, l\_j,2, l\_j3), dann verbinde l\_j1 mit a, l\_j,2 mit b und c mit l\_j,3

* Füge zusätzlich zwei Knoten W und F ein und neue Kanten: FR, FW, WR

=> Knoten x1..x4 können nur mit Farben von W und F belegt werden.

Verbinde F mit dj und ej (für alle j)

*Baue Graph am Bsp auf zum Verständnis*

Umkehrrichtung „Haben erfüllende Belegung => Graph ist 3 färbbar“

Sei b eine erfüllende Belegung für F => G ist 3-färbbar!

Färbe xi mit b(xi) und !xi mit b(!xi) => und nun Graph auf 3 Farben erweitern.

(Färben Literale mit l = 1 für wahr und l = 0 für falsch. Danach färbe die Knoten der Klausel Kj und anschließend F, W und R)1