Testen von Planarität II
Gültigkeit und Laufzeit erobern die Welt

Markus Chimani
LS XI Algorithm Engineering, TU Dortmund

VO Automatisches Zeichnen von Graphen 16

Wiederholung/Idee

Gegeben: Graph $G=(V,E)$, $n := |V|$, $m := |E| = O(n)$

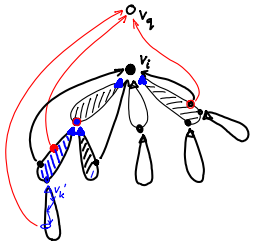
- Berechne einen beliebigen DFS-Baum (mit beliebigen Startknoten). Ordne Knoten gemäß DFI (DFS-Index) $(v_1, v_2, \ldots, v_n)$
- */* DFS-Baum ist trivialerweise planar. */*
- **for** $v_i = \{ v_n, \ldots, v_1 \}$ */* Umgekehrte Reihenfolge! */*
  - Bette v_i, zusammen mit DFS-Baumkanten (v_i,v_j) und allen Backedges (v_j,v_i), mit $j>i$, planar ein.
  - Einbettung mind. einer Backedge nicht möglich
    ⇒ **return** G nicht planar
- **return** G ist planar (und eingebettet)

Wiederholung/Pertinenz & Externalität

Für Knoten v_i mit DFI i:

- pertinente Knoten/Bicomps:
  - Knoten v_k, mit DFI $k > i$ und Backedge (v_k,v_i)
  - Bicomps die am DFS-Pfad $v_i \rightarrow v_k$ liegen.
  - Cut-Knoten entlang des Pfads
- externe Knoten:

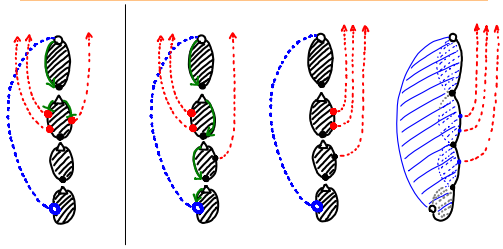
  Knoten v_k mit DFI $k > i$ und:
  - Backedge (v_k,v_q) mit $q < i$, oder
  - $\exists$ DFS-Pfad $v_k \leadsto v_p$ ausserhalb der Bicomp von v_k; Backedge (v_p,v_q) mit $q < i$



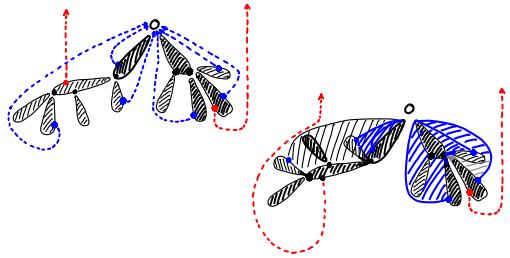
Wiederholung/Backedge einfügen möglich?

Einfügelemma

Eine Kante kann genau dann eingefügt werden, wenn ein Weg entlang den Aussenflächen der pertinenten Bicomps existiert, der keine externen Knoten *(Stop-Vertices)* enthält.



Wiederholung/Mehrere Kanten einfügen



Reihenfolge

Top-Down. Nicht-externe Bicomps zuerst.

Gültigkeit/Gültigkeit

- Warum entscheidet der Algorithmus gültig zwischen planaren und nicht-planaren Graphen?

Beobachtung

Wenn der Algorithmus einen Graphen einbettet, dann ist dieser planar.

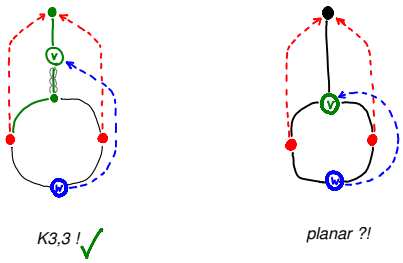
- zu zeigen:

Lemma (zu zeigen)

Wenn der Algorithmus einen Graphen nicht einbettet, dann ist dieser nicht-planar.

Gültigkeit/Stop-Konfiguration

Einbetten von *v* und einer Backedge mit pertinentem *w*



K3,3 ! ✓ *planar ?!*

Gültigkeit/Stop-Konfiguration

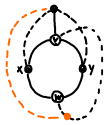
Kein versperrender x→y Weg in Bicomp

Vor Einfügen von *v*: jeder *x→y* Pfad enthält *w*
⇒ *w* ist DFS-Vorgänger von *x* oder *y* (oder beiden).

⇒ *w* vor *x* oder *y* besucht ⇒ Backkante *(w,v)* wäre schon direkt eingefügt worden.

⇒ *w* ist pertinent wegen Kind-Bicomp.

- Bicomp extern? ⇒ nicht-planar (Bild).
- Bicomp nicht-extern? ⇒ Bicomp vor *x* und *y* eingebettet. ↯



Gültigkeit/Stop-Konfiguration

Versperrender x→y Weg in Bicomp

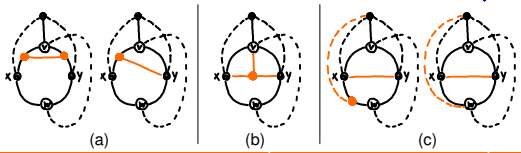
Betrachte *Highest-xy-Path:* Bicomp-Rand vor Einfügen von *v*.
Bild (a) ⇒ HxyP muss von *x* nach *y* gehen.

Bild (b) ⇒ HxyP hat keine aktive Knoten

o.B.d.A. DFI(*x*)<DFI(*y*). Laufe von *x* entlang der Aussenfläche der alten Bicomp in beider Richtungen: 1. Richtung: Stop bei *y*.

2. Richtung: bis *w* – Bild (c): vorher nichts extern, *w* nicht extern

⇒ *w→v* würde vor *y→v* eingebettet werden, da nicht-extern! ↯



(a) (b) (c)

Gültigkeit/Gültigkeit

Beobachtung

Wenn der Algorithmus einen Graphen einbettet, dann ist dieser planar.

Lemma

Wenn der Algorithmus einen Graphen nicht einbettet, dann ist dieser nicht-planar. ✓

- also:

Theorem

Der Algorithmus entscheidet korrekt, ob ein Graph planar oder nicht-planar ist.

Laufzeit/Linearzeit?

Eine naïve Implementation dieser Idee: mind. $O(n^2)$
Wie soll man das alles in Linearzeit schaffen??

Zu lösende Punkte:

- *O(n)* Kanten einfügen. Jede Kante in *O(1)* einfügbar?
  - Nein! → Amortisierte Analyse
- Wie schnell kann man eine einzelne Kante einfügen?
- Ausnutzen, dass mehrere Kanten gleichzeitig eingefügt werden?

⇒ Geschickte Algorithmik & geschickte Analyse!

Laufzeit/Zu lösende Probleme

Algorithmische Schritte, die wir schnell&effizient lösen müssen:

- Erkennen der externen Knoten/Bicomps (1)
- Erkennen der pertinenten Knoten/Bicomps (3)
- Feststellen, ob eine Stop-Konfiguration vorliegt (3)
- Flippen von Bicomps (2)
- Schritt (3) linearisieren: *Short Circuit Edges* (4)

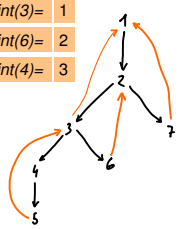
Laufzeit/Externe Knoten

Wie erkenne ich externe Knoten?

lowpoint von DFS: kleinster über Pfad mit *einer* Backedge erreichbarer DFI

Dauer der Berechnung:

lowpoint(3)=	1
lowpoint(6)=	2
lowpoint(4)=	3



separatedDFSChildList(*v*)

- Liste der DFS-Kinder die (noch) nicht mit *v* verschmolzen sind
- Sortiert nach aufsteigendem *lowpoint*-Wert
- Alle Listen in O(n) erzeugen
- Löschen eines Eintrags in konstanter Zeit

Alle Knoten sortieren, danach dem jeweiligen *v* zuordnen: *O(n)*

Doppelt ringverkettete Liste

Laufzeit/Flippen von Bicomps

Einbettung einer Bicomp =

Reihenfolge der Kanten um die Knoten

Spiegeln einer Einbettung =

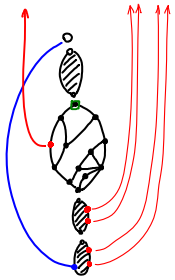
Alle Reihenfolgen invertieren

Problem

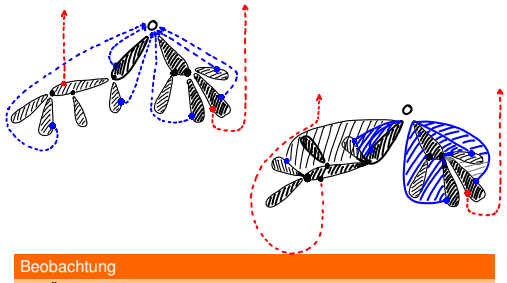
Laufzeit einer Spiegelung:	O(n)
Anzahl Spiegelungen:	O(n)

Lösung

An Bicomp-Wurzel abspeichern, dass alle Reihenfolgen der Knoten (bis auf weiteres) invertiert gemeint sind.



Laufzeit/Pertinenz & Kanten einfügen (1)

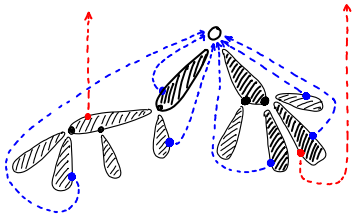


Beobachtung

Die Äste des DFS-Baums des neuen Knotens sind unabhängig!

⇒ Einbetten der Kanten für jeden Teilbaum einzeln

Laufzeit/Pertinenz & Kanten einfügen (2)



Problem

Richtige Einfüge-Reihenfolge:

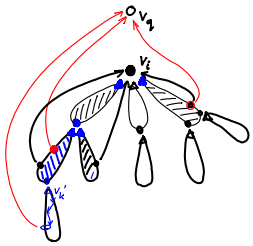
- Top-Down → Woher wissen, wann fertig?
- Nicht-Externe Bicomps zuerst ablaufen → Woher wissen?

Laufzeit/Walk-Up & Walk-Down

Idee

2 Phasen pro einzufügendem Knoten:

- Walk-Up: → Bottom-Top Pertinente Teile des Graphen markieren, Datenstrukturen vorbereiten
- Walk-Down: → Top-Bottom Einfügen der Kanten



Laufzeit/Coole Idee: Face-Konto

Wir erinnern uns:

- *G=(V,E)* mit Faces *F*
  - ⇒ *n:=|V|, m:=|E|, f:=|F|*
- *n – m + f = 2*
- *m = O(n)*

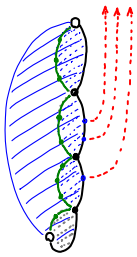
⇒ *f = O(n)*

⇒ jede Kante ist an 1-2 Faces beteiligt

Aha! Eigenschaft:

- Betrachte das Ablaufen der Bicomp-Teile die innen liegen werden
- Faces in Bicomp ändern sich nie wieder

⇒ Ablauf der Innenseiten linear beschränkt!



WICHTIG !!!

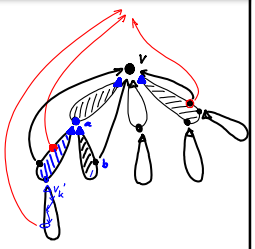
Laufzeit/Walk-Up

Für alle Backedges *(w,v)*:

Starte in *w* & laufe nach oben

Dabei an jedem Knoten *u* speichern:

- *backedge* → Start von Backedge?
- *visited* → wurde *u* besucht?
  - Bool-Flag? Aufwendig in jeder Iteration zu löschen :-(
  - Integer mit DFI von *v*
- *pertinentRoots* → Liste der an *u* hängenden pertinenten Bicomps
  - Einfügen einer Bicomp *B* in Liste:
    - IF *B* ist extern THEN hinten-einfügen ELSE vorne-einfügen
    - Warum? Zuerst nicht-externe Bicomps einbetten
    - Extern erkennen? *lowpoint* der Wurzel von *B*



Laufzeit/Walk-Up

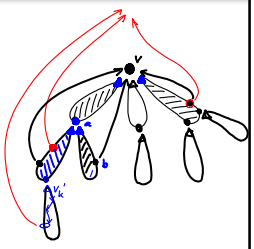
Aber wie laufen wir hinauf?

- Aussen an den Bicomps laufen
- Stoppen falls Knoten schon *visited*
- Effizient?

Walk-Up: O(n)

Für jede Bicomp mit pertinentem Knoten *b* und Wurzel *a*:

- spätere Innenseite mindestens so lang wie kürzerer Weg von *b* nach *a*
- Parallel ablaufen!
  Von *b* aus gleichzeitig in beide Richtungen, bis *a* erreicht
  → 2x kürzester Weg = *O(* spätere Innenseite *)*



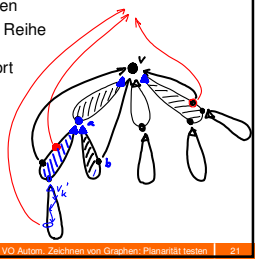
Laufzeit/Walk-Down

Walk-Down (DFS) für jeden Teilbaum von *v*:

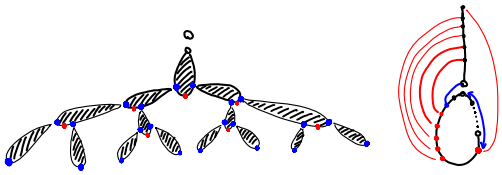
- Laufe an beiden Seiten der Bicomp von *v* hinab. Sei *w* der betrachtetet Knoten.
- *backedge(w)* ? → Backedge einbetten (mitzählen), Bicomps verschmelzen
- *pertinentRoots(w)* ≠ leer ? → Der Reihe nach in diese Bicomps absteigen (externe stehen hinten): Starte dort in Wurzel mit beiden Richtungen.
- *w* extern ? → Stoppen.
- *w* = *v* ? fertig.

Wenn Walk-Down fertig:

- Alle Backedges eingebettet?
  → planar → sonst: nicht-planar



Laufzeit/Walk-Down – Laufzeit?



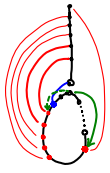
Problem: Laufzeit…

- Parallel-Lauf Abschätzung funktioniert nicht, weil exponentiell viele Pfade gleichzeitig!
- "Äusserer" Teil von Bicomp wird oft traversiert

Laufzeit/Short-Circuit

Der Clou: Short-Circuit Kanten

- Abkürzung über schon besuchte Knoten hinweg!
- Besuchte Knoten (bis auf letzten) sind nie aktiv → werden nie pertinent!
- Hilfskanten einbetten. Am Ende löschen.



Konsequenzen

- Jeder Bicomp-Abschnitt wird nur einmal betrachtet → linear in den Faces!
- Anzahl Short-Circuit Kanten?
  2·#Bicomps = *O(n)*

Walk-Down: O(n)

Laufzeit/Gesamtsicht

Gegeben: Graph $G=(V,E)$, $n := |V|$, $m := |E| = O(n)$

- Berechne einen beliebigen DFS-Baum (mit bel. Startknoten). Ordne Knoten gemäß DFI (DFS-Index) ($v_1, v_2, \ldots, v_n$)
- **for** $v_i = \{ v_n, \ldots, v_1 \}$ */* Umgekehrte Reihenfolge! */*
  - Bette v_i mit DFS-Baumkanten zu Kindern ein
  - **for all** Backedges $b=(v_j,v_i)$, mit $j>i$
    - *Walk-Up*(v_j)
  - **for all** *pertinentRoots* p von v_i
    - *Walk-Down(p)* */* Bettet Kanten ein */*
  - Einbettung mind. einer Backedge nicht möglich
    ⇒ **return** G nicht planar
- **return** G ist planar (und eingebettet)