Sugiyama-Verfahren für große Graphen

Dirk Ribbrock Nils Kriege Sophia Kardung

Universität Dortmund - LS 11 - Graphenzeichnen

Dortmund, 27. November 2007

Übersicht

- OGDF-Sugiyama für große Graphen
- Sugiyama Verbesserungen für große Graphen

Sugiyama Optionen

- SugiyamaLayout::setRanking
  - LongestPathRanking $O(|V| + |E|)$
  - OptimalRanking
- SugiyamaLayout::setCrossMin
  - BarycenterHeuristic $O(|E| + |V|log|V|$
  - MedianHeuristic $O(|E|)$
  - SplitHeuristic $O(|V|log|V|)$ (Kreuzungsmatrix $O(|V|^2)$)
- SugiyamaLayout::setLayout
  - OptimalHierarchyLayout
  - FastHierarchyLayout $O((|V| + |E|)(log(|V| + |E|))^2)$
    (Fast Layout Algorithm for k-Level Graphs by Buchheim et al.)

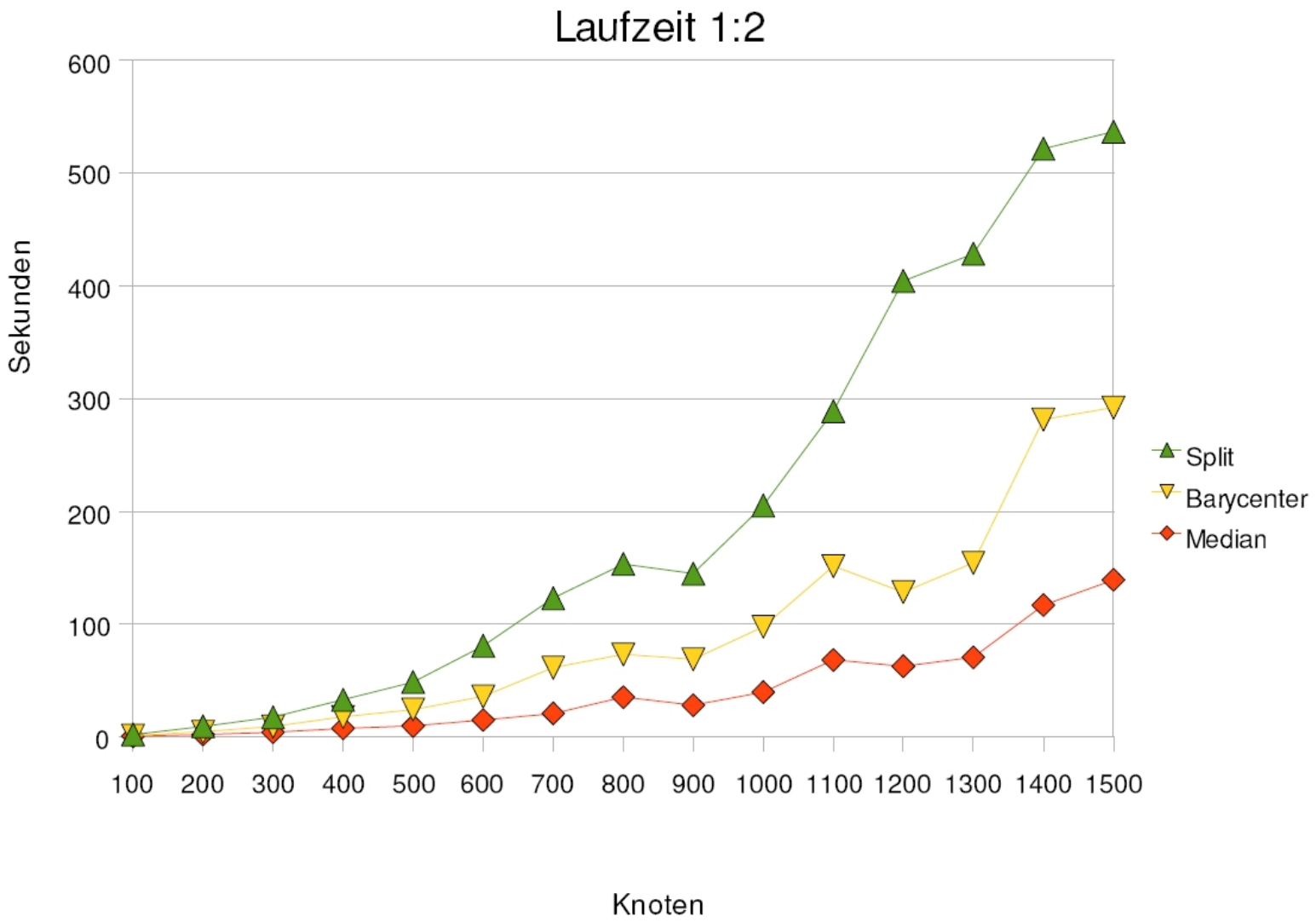
Eingaben

- Zufällige Graphen per randomHierarchy(G) und makeConnected(G)
- Knotenanzahl: 100 bis 1500 in 100er Schritten
- 10 Durchläufe pro Knotenanzahl
- Verhältnis Knoten:Kanten 1:2 und 1:5

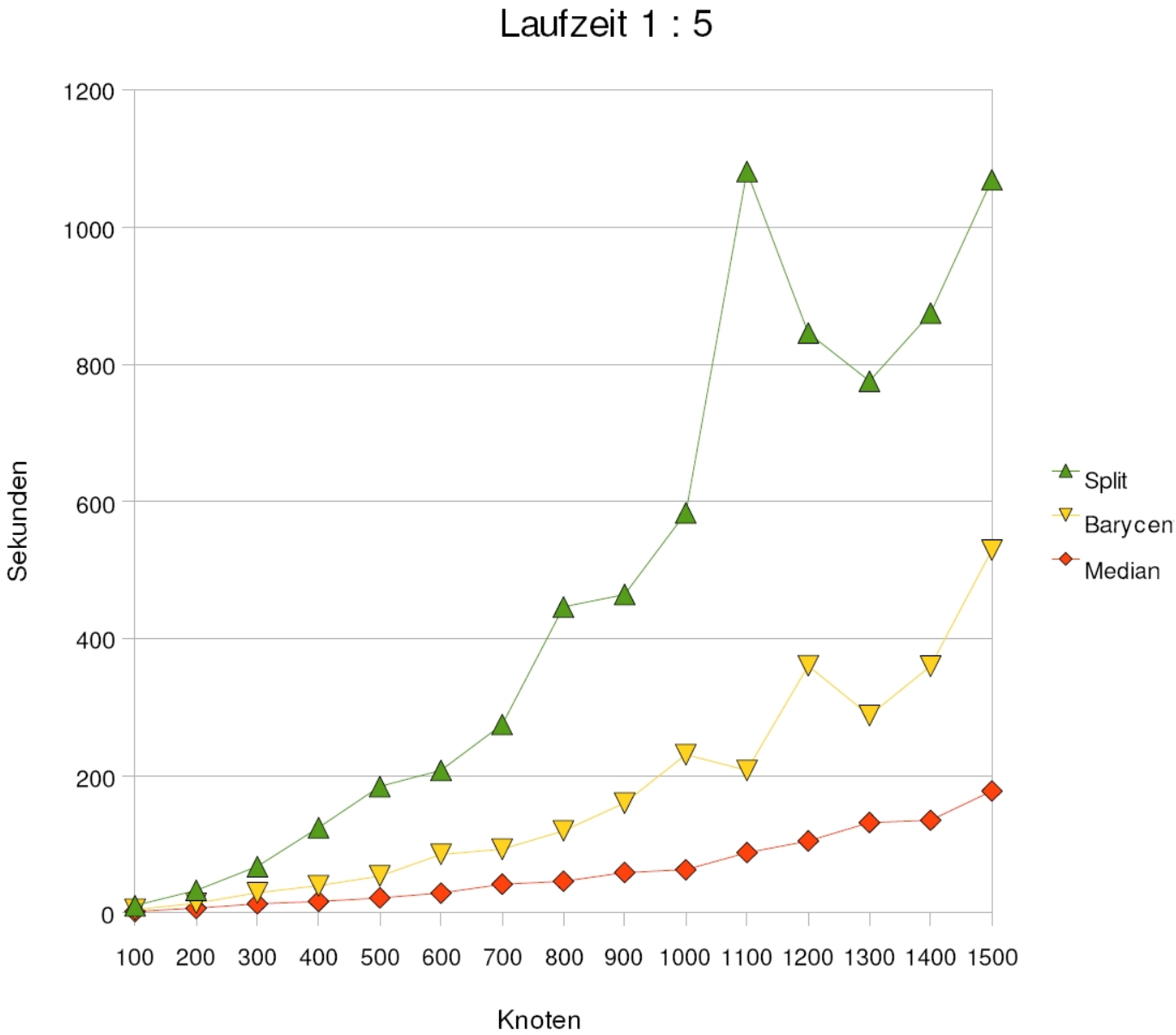
Testsystem

- Pentium M 1.50 Ghz
- 512 MB Speicher
- der komplette Messdurchlauf lief etwa 27 Stunden
- ein Großteil der Zeit entfiel auf das Erstellen der Graphen

Verhältnis Knoten:Kanten 1:2



Verhältnis Knoten:Kanten 1:5



Fazit

- ogdf::SugiyamaLayout Laufzeit ungeeignet für große Graphen
- wenn es sich nicht vermeiden lässt, dann
  - kein OptimalRanking
  - kein OptimalHierarchyLayout
  - keine SplitHeuristic

Gründe für die schlechte Laufzeit

- fast alle Probleme im Sugiyama sind NP-hart:
  - DAG berechnen (Feedback-arc Set)
  - Layering
  - Kreuzungsminimierung
  - ...

Gründe für die schlechte Laufzeit

- fast alle Probleme im Sugiyama sind NP-hart:
  - DAG berechnen (Feedback-arc Set)
  - Layering
  - Kreuzungsminimierung
  - ...
- aber es existieren praktisch gute Approximationsalgorithmen

Gründe für die schlechte Laufzeit

- fast alle Probleme im Sugiyama sind NP-hart:
  - DAG berechnen (Feedback-arc Set)
  - Layering
  - Kreuzungsminimierung
  - ...
- aber es existieren praktisch gute Approximationsalgorithmen
- Problem: Kreuzungsminimierung wird nicht einmal, sondern mehrfach für jede Schicht ausgeführt

Gründe für die schlechte Laufzeit

- fast alle Probleme im Sugiyama sind NP-hart:
  - DAG berechnen (Feedback-arc Set)
  - Layering
  - Kreuzungsminimierung
  - ...
- aber es existieren praktisch gute Approximationsalgorithmen
- Problem: Kreuzungsminimierung wird nicht einmal, sondern mehrfach für jede Schicht ausgeführt
- je größer der Graph desto öfter werden die Schichten iteriert bis sich keine Verbesserungen mehr ergeben

Ideen

- Anzahl der Iterationen durch die Schichten begrenzen
- durch die große Anzahl an Knoten können kleine Änderungen immer wieder kleine Verbesserungen bringen $\longrightarrow$ kostet viel Laufzeit
- Diese Änderung führt natürlich zu schlechteren Ergebnissen
- Allgemein sind "einfache" Verschnellerungen schwer zu finden, da Kreuzungsminimierung lange dauert und nicht vermieden werden kann

Dummyknoten

- weiteres Problem: Anzahl der Dummyknoten wächst stark

Dummyknoten

- weiteres Problem: Anzahl der Dummyknoten wächst stark
- bei Kreuzungsminimierung werden Dummyknoten als normale Knoten behandelt $=>$ Problem wird noch schwieriger

Dummyknoten

- weiteres Problem: Anzahl der Dummyknoten wächst stark
- bei Kreuzungsminimierung werden Dummyknoten als normale Knoten behandelt $=>$ Problem wird noch schwieriger
- aktuelle Laufzeit: $O(|V||E|log|E|)$, Speicherplatz: $O(|V||E|)$

Idee von Eiglsperger, Siebenhaller und Kaufmann

- Dummyknoten simulieren lange Kanten, diese werden am Ende gerade gezeichnet

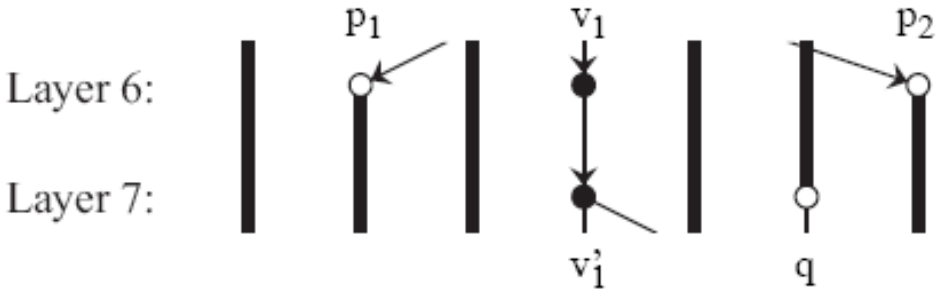
Idee von Eiglsperger, Siebenhaller und Kaufmann

- Dummyknoten simulieren lange Kanten, diese werden am Ende gerade gezeichnet
- betrachte also nicht auf jeder Schicht einen Dummyknoten, sondern einen Start und einen Endpunkt und das Segment dazwischen
- $=>$ spärlich normalisierter Graph

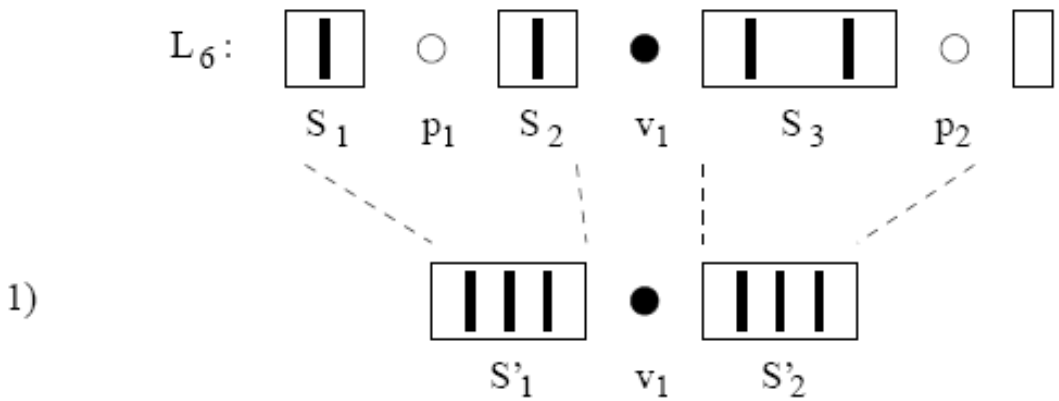
Das neue Vorgehen

- Phase 1: Berechne aus dem Eingabegraph die spärliche Normalisierung
- Phase 2: effiziente Kreuzungsminimierung auf diesem Graph
- Phase 3: Koordinatenzuweisung (wie vorher)

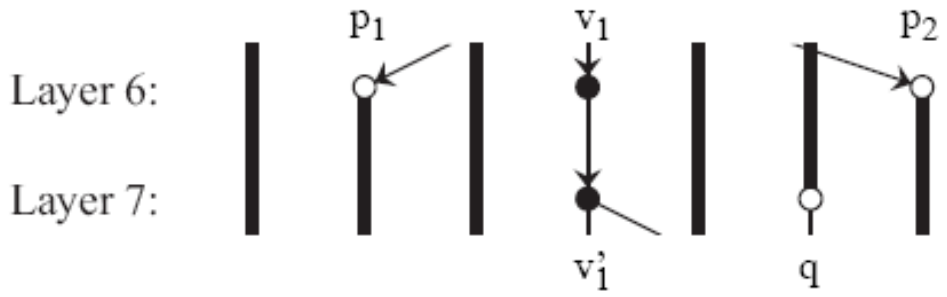
Effiziente Kreuzungsminimierung



- Schritt 1: Segmente in entsprechender Datenstruktur (Container) speichern



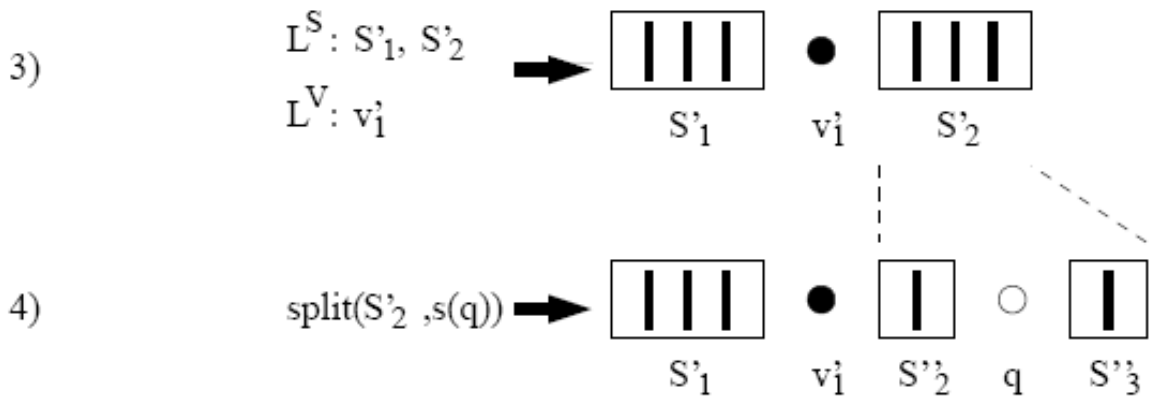
Effiziente Kreuzungsminimierung



- Schritt 2: Verteile Nummern so, dass sie der Nummerierung mit Dummyknoten entsprechen und berechne Barycenter/Median/...

Effiziente Kreuzungsminimierung

- Schritt 3: Nach Bary/Median sortieren, Liste L^V für "echten" Knoten, Liste L^S für Segmente
- Schritt 4: Aufruf split: Segmente werden, falls "Zwischenknoten" in der unteren Schicht existieren, entsprechend geteilt

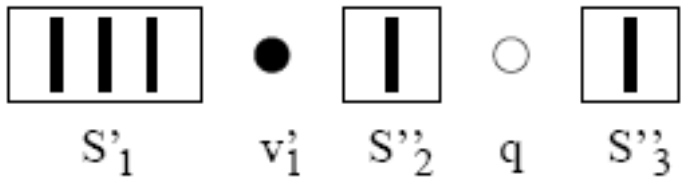


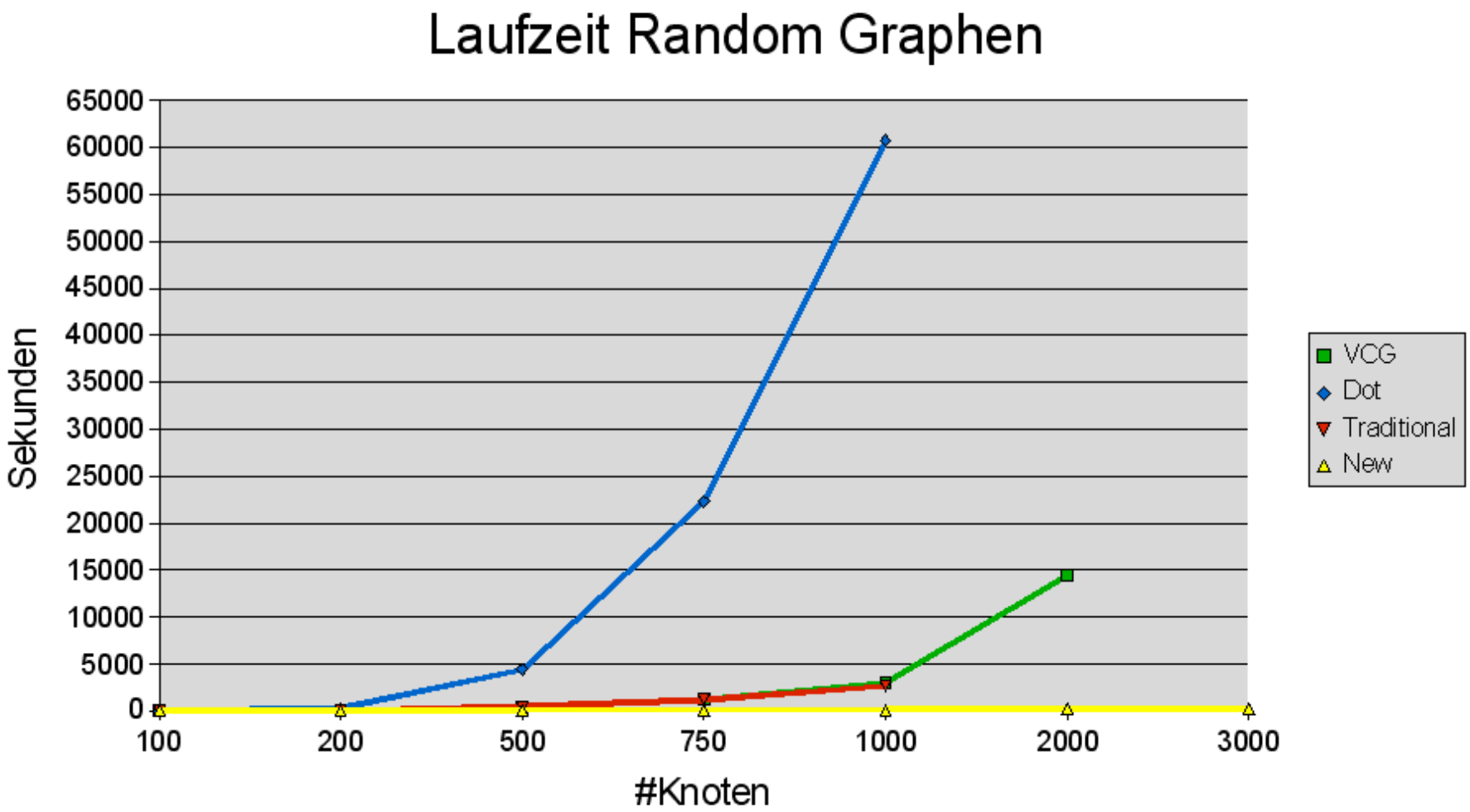
Effiziente Kreuzungsminimierung

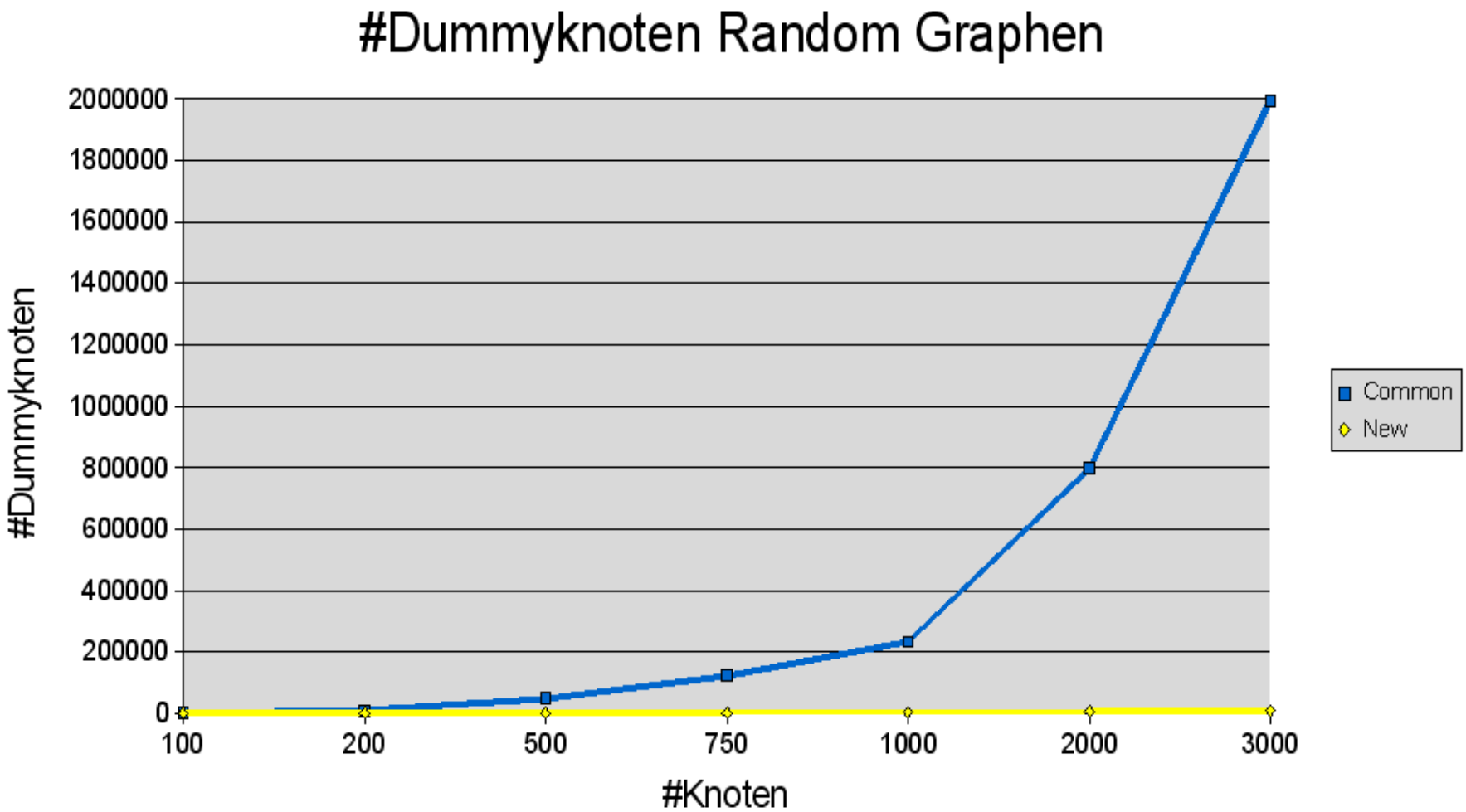
- Schritt 5: Kreuzungszählen: Dabei müssen die virtuelle Kanten der Container berücksichtigt werden → Kantengewicht = Anzahl Elemente im Container
- Schritt 6: Container die in der nächsten Schicht nebeneinander liegen werden vereinigt.

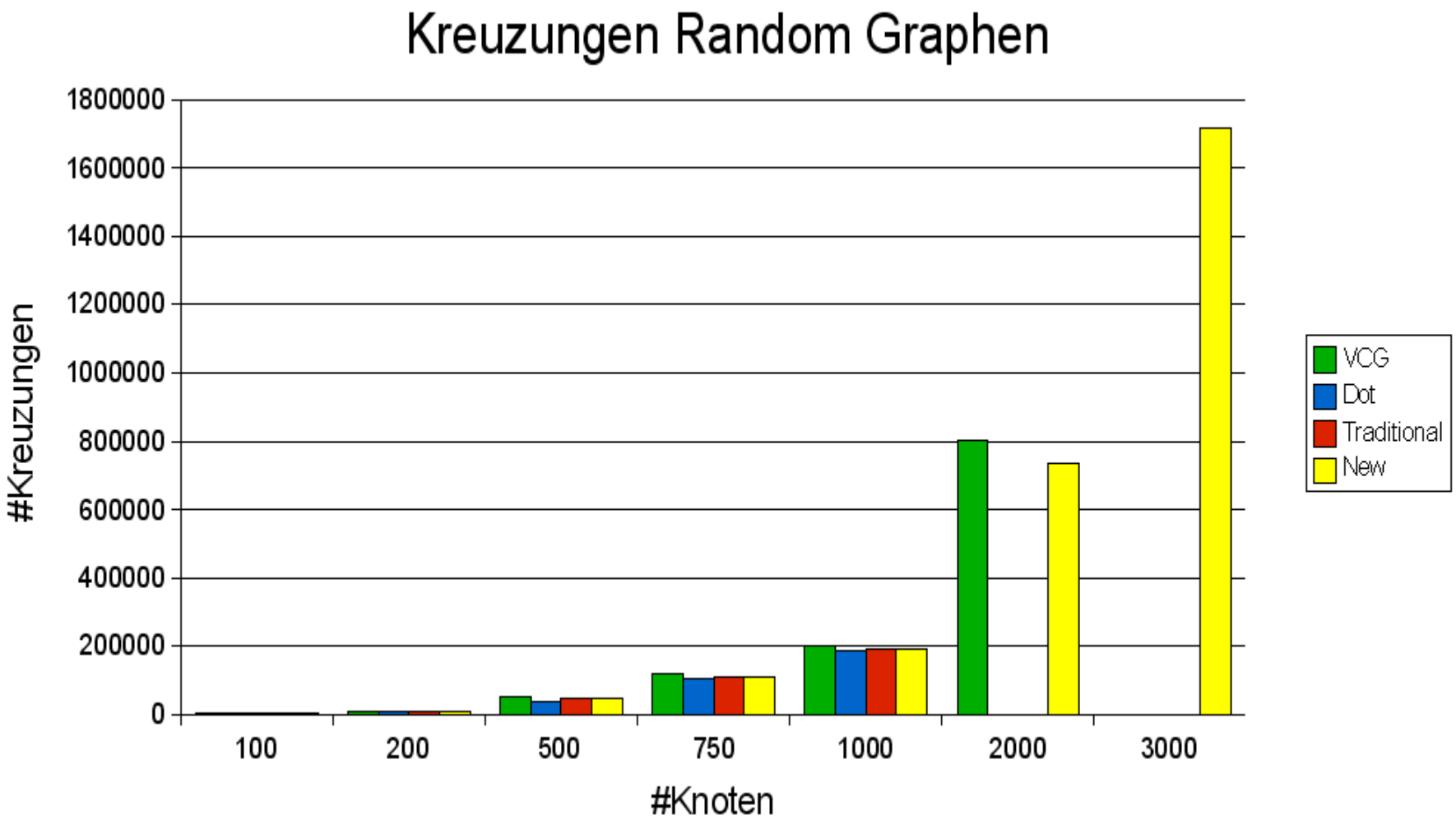
5) count crossings: 0

6)




Laufzeit Random Graphen
Sekunden
65000
60000
55000
50000
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
100
200
500
750
1000
2000
3000
#Knoten
VCG
Dot
Traditional
New


#Dummyknoten Random Graphen
#Dummyknoten
2000000
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
100
200
500
750
1000
2000
3000
#Knoten
Common
New


Kreuzungen Random Graphen
#Kreuzungen
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
100
200
500
750
1000
2000
3000
#Knoten
VCG
Dot
Traditional
New

Fazit

- Laufzeit: $O((|V| + |E|)log|E|)$
- Speicherplatz: $O(|V| + |E|)$
- ähnliche Kreuzungsanzahl, viel bessere Laufzeit

Ende

Vielen Dank für die Aufmerksamkeit