Karlsruher Institut für Technologie

Algorithmen für Routenplanung

5. Termin, Sommersemester 2012

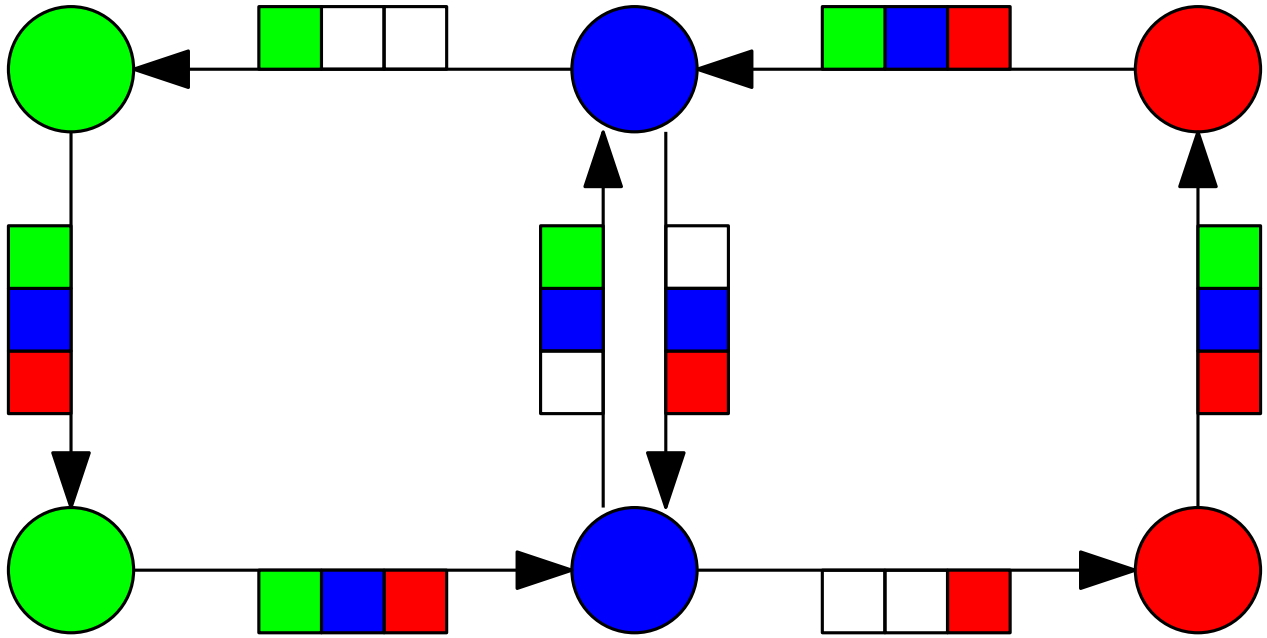
Daniel Delling | 9. Mai 2012

Institut für Theoretische Informatik · Algorithmik I · Prof. Dr. Dorothea Wagner

Letztes Mal: Arc-Flags

Idee:

- partitioniere Graphen in k Zellen
- hänge Label mit k Bits an jede Kante
- gibt an, ob e für Ziele in Zielzelle T benötigt wird



Vorberechnung

Zwei Schritte:

- Partitionierung (heute mehr)
- setze Flaggen durch Dijkstra Suchen von den Randknoten

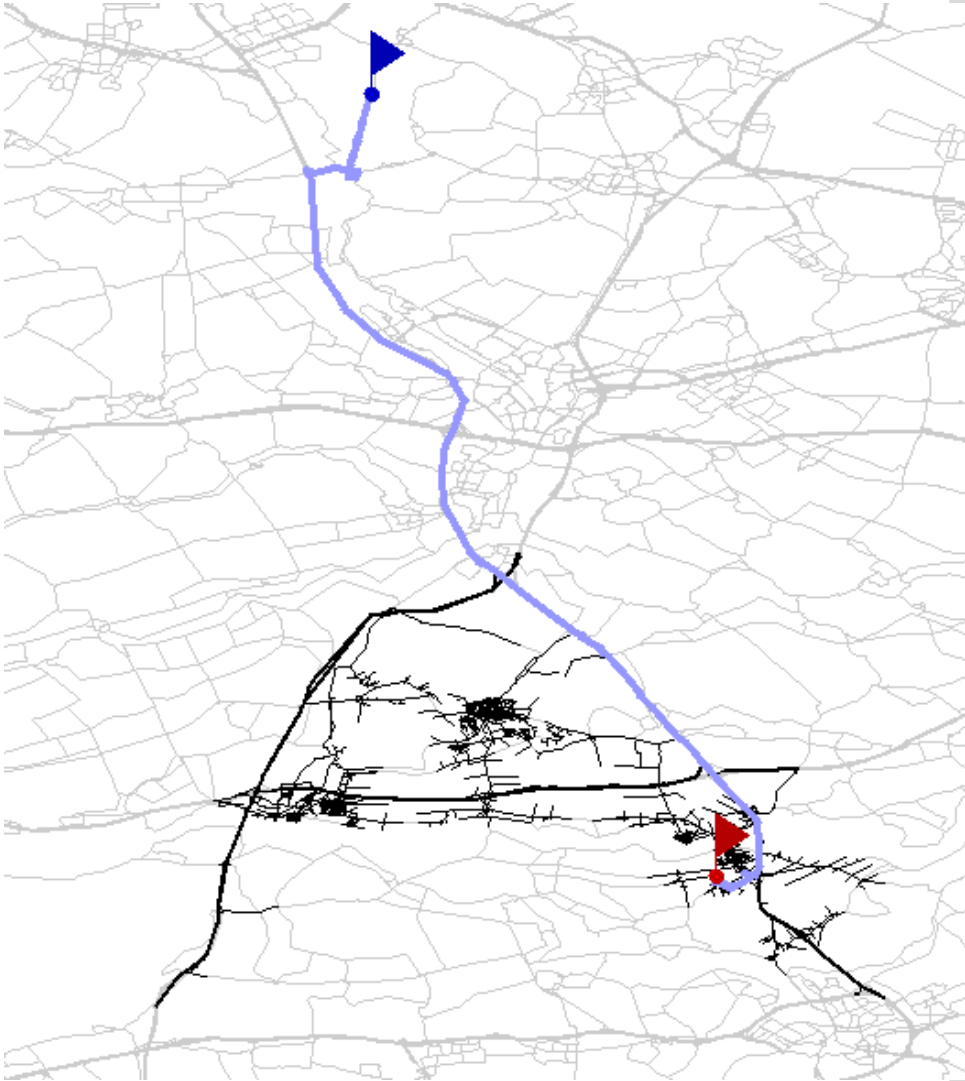
Coning-Effekt

Beobachtung:

- lange Zeit nur eine Kante wichtig
- daher immer nur ein Knoten in der Queue
- aber: je näher an der Zelle, desto mehr Kanten werden wichtig
- Suche fächert sich auf
- in Zelle werden dann alle Kanten relaxiert

Zwei Ansätze:

- bidirektionale Flags
- multi-level Flags



Bidirektionale Arc-Flags

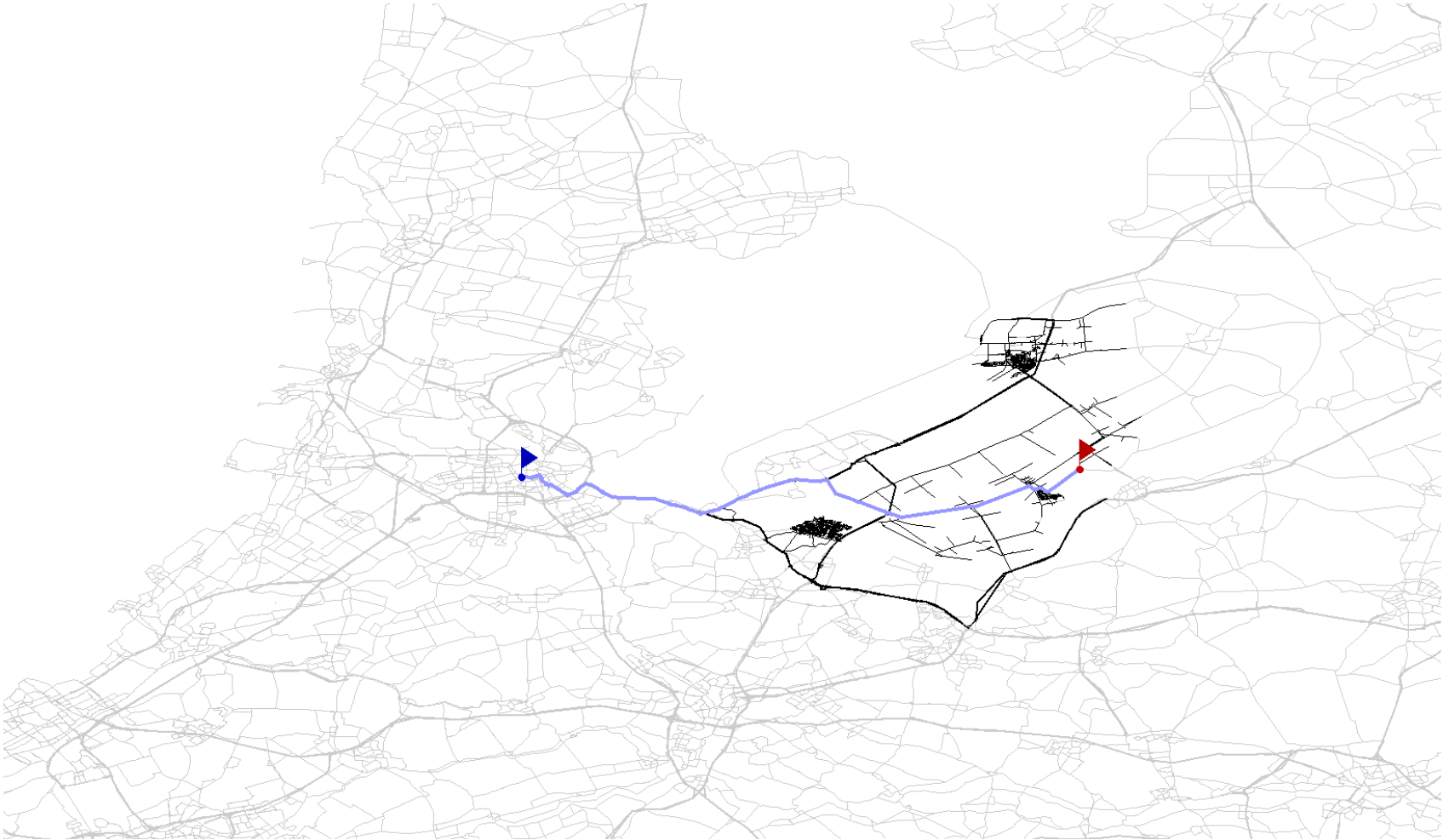
Vorberechung:

- Vorwärts- und Rückwärtsflaggen
- Rückwärtsflaggen können analog für eingehende Kanten berechnet werden
- Vorberechungszeit in gerichteten Graphen erhöht sich um Faktor 2
- **Wichtig:** Flaggen müssen für alle kürzesten Wege gesetzt sein

Anfrage:

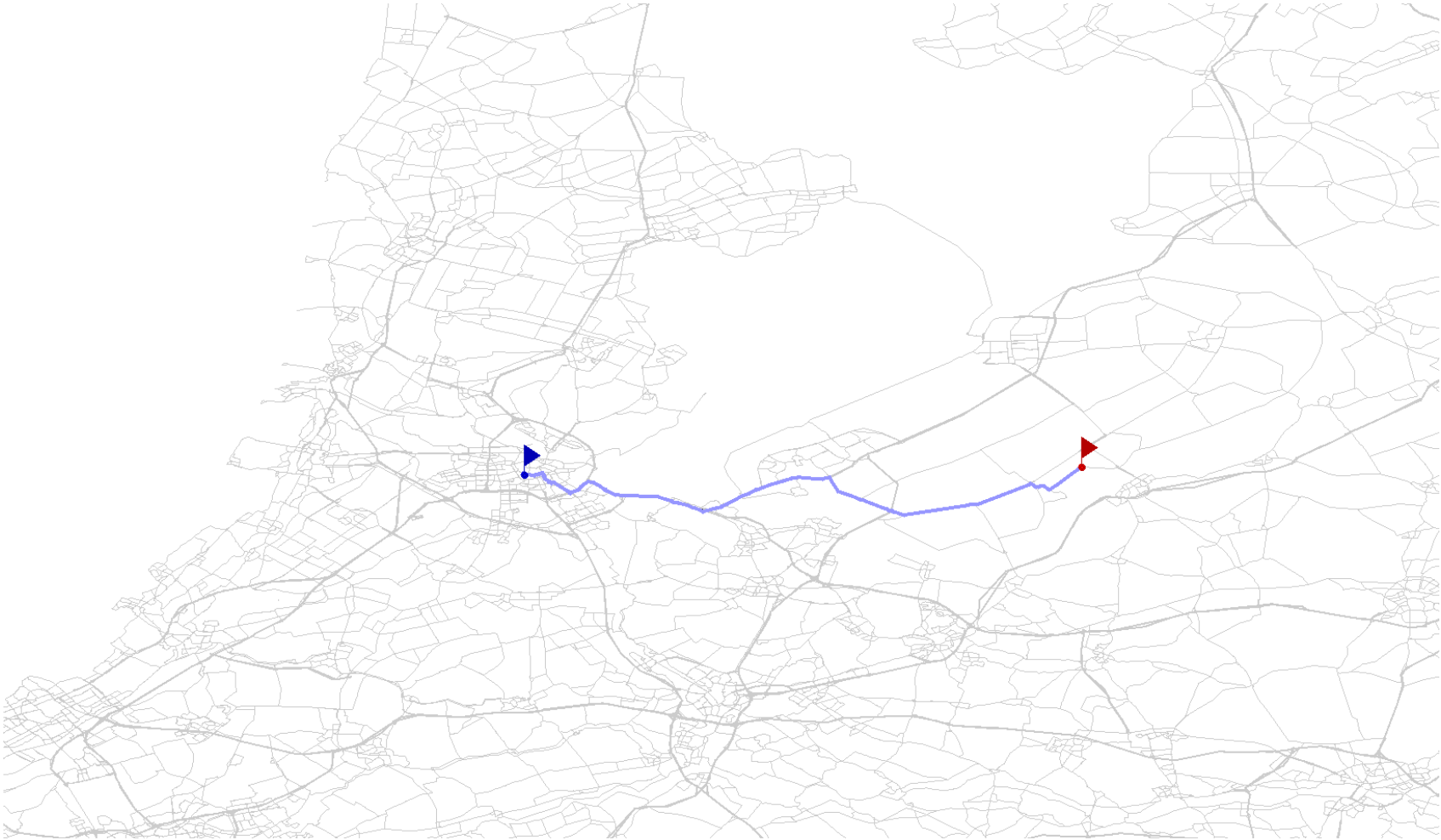
- bidirektional:
  - Vorwärtssuche relaxiert nur Kanten mit Flagge für T
  - Rückwärtssuche nur Kanten mit Flaggen für S
- normales Stopp-Kriterium von bidirektionalem Dijkstra

Suchraum



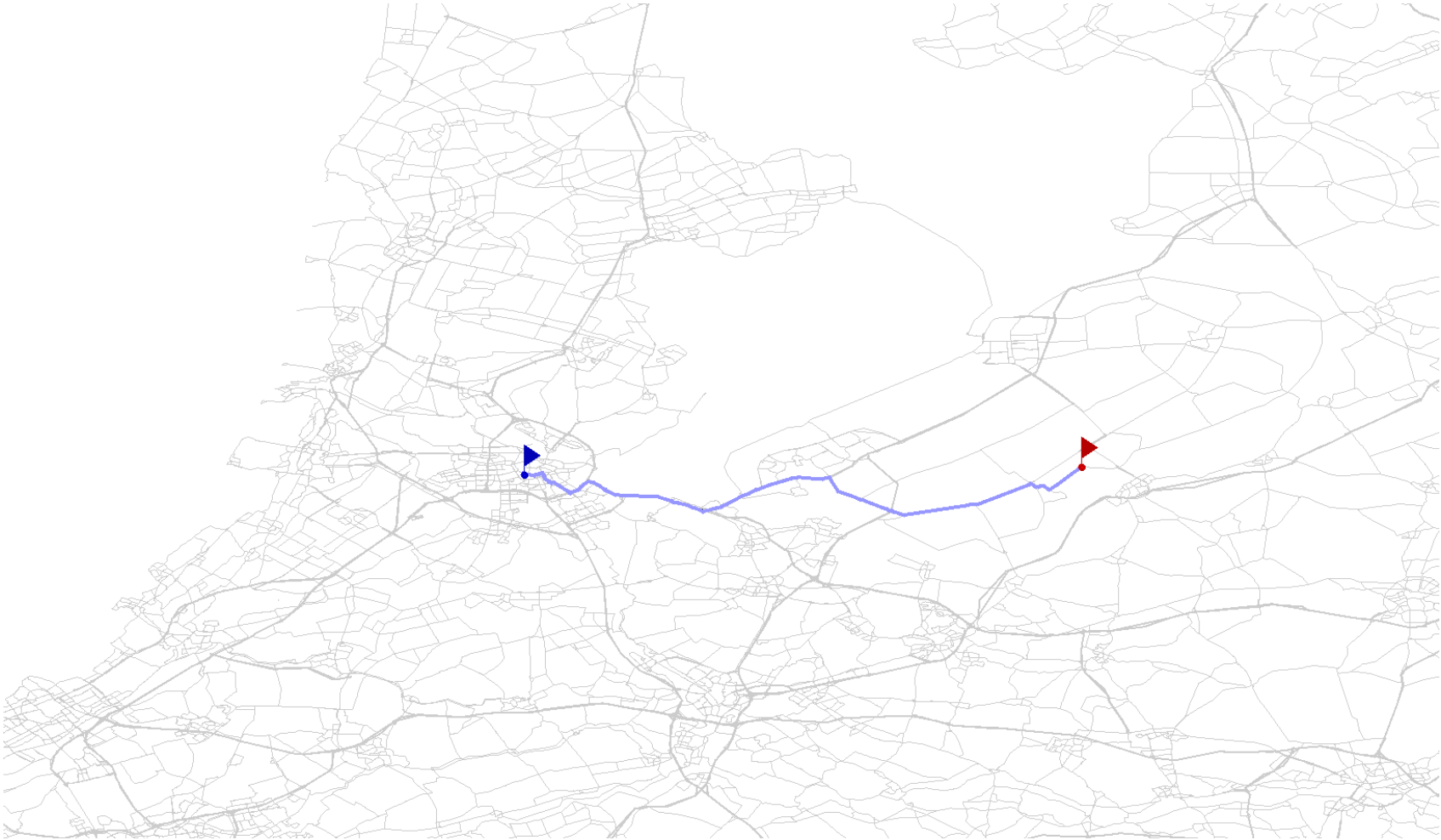
Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Suchraum



Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Suchraum



Nachteile?

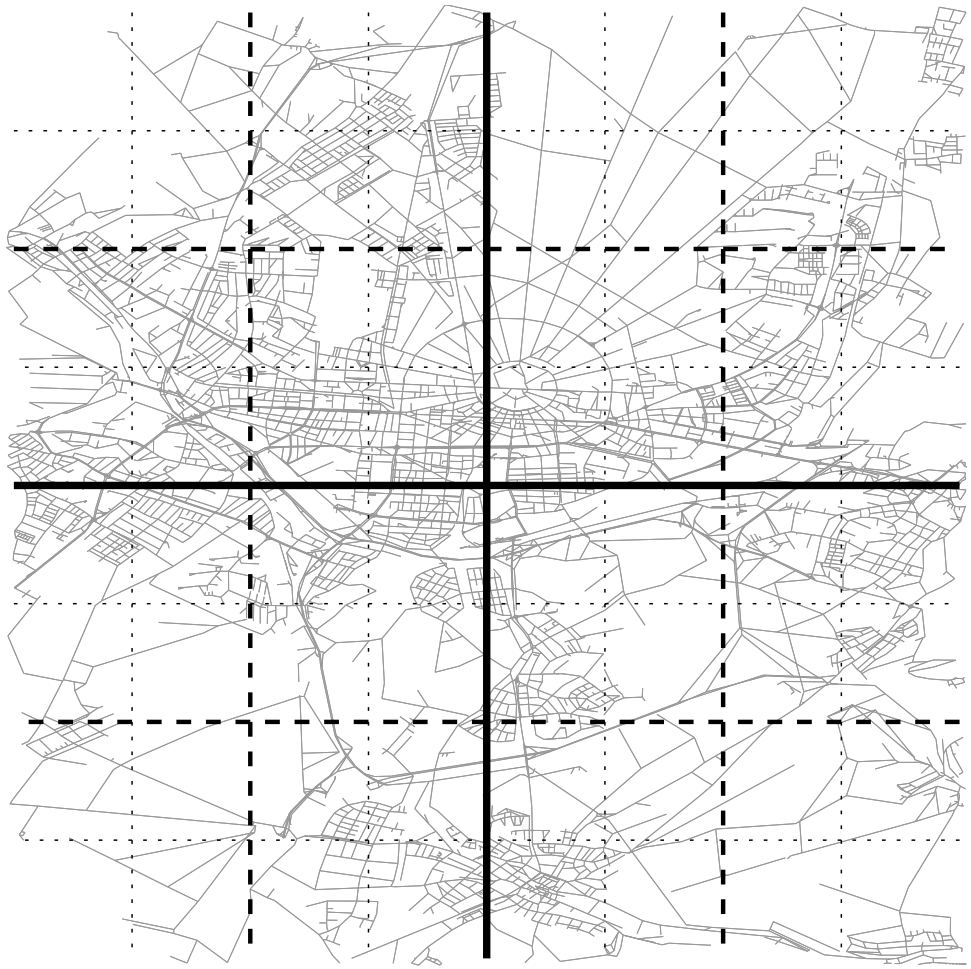
Multi-Level Arc-Flags

Problem:

- Anfragen in einer Zelle ohne Beschleunigung
- viele Real-Welt Anfragen sind lokal

Multi-Level Arc-Flags:

- Multi-Level Partition
- berechne partielle Flaggen



Multi-Level Arc-Flags

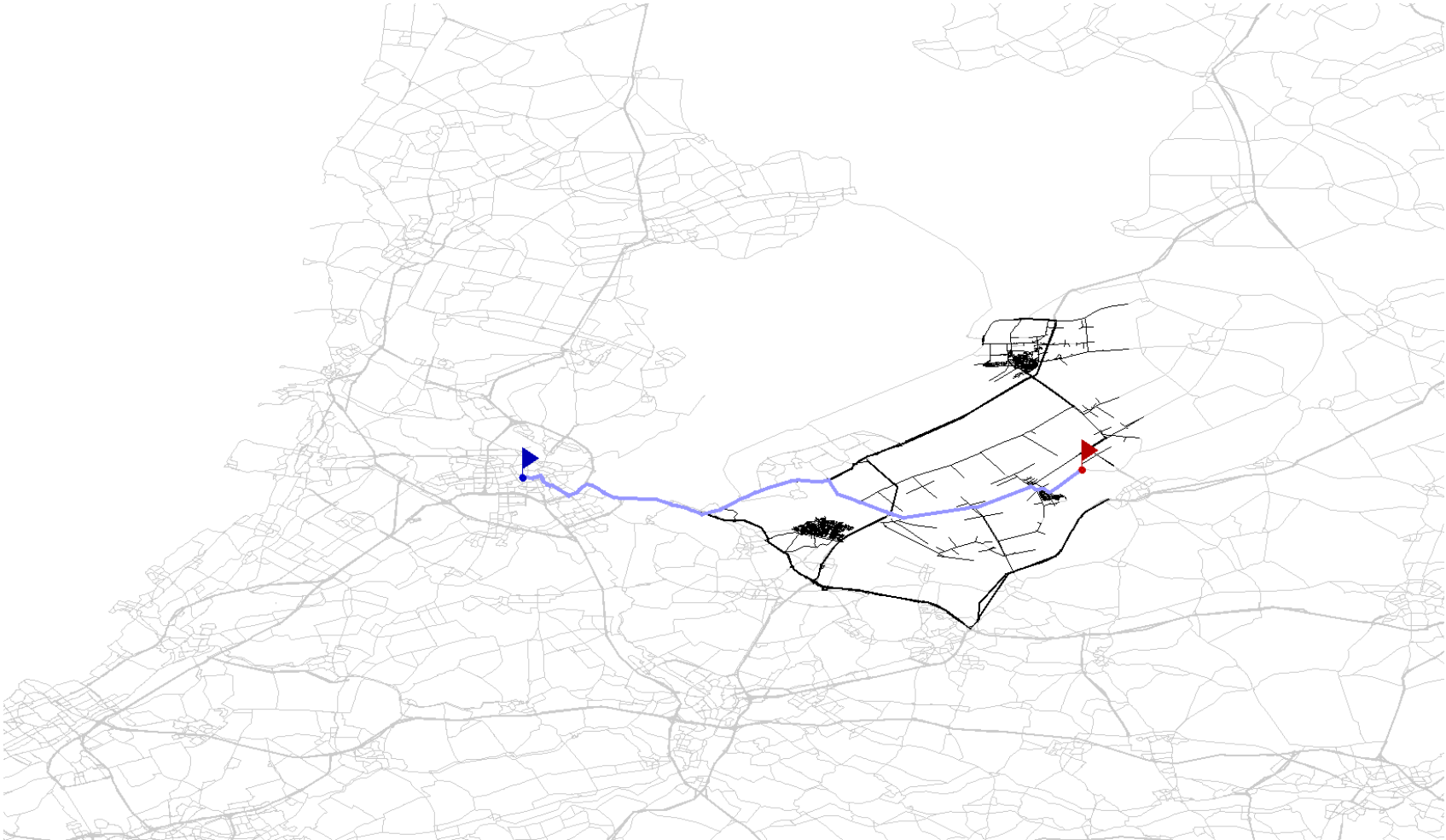
Vorberechnung:

- oberster Level wie gehabt
- auf unteren Leveln:
  - von jedem Randknoten führe Dijkstra aus, bis alle Knoten der Superzelle abgearbeitet worden sind
  - setze Flaggen für alle Kanten (u, v) für die u in Superzelle
  - Hinweis: es reicht nicht, nur den Subgraphen der Superzelle zu betrachten (Übungsaufgabe)

Anfragen:

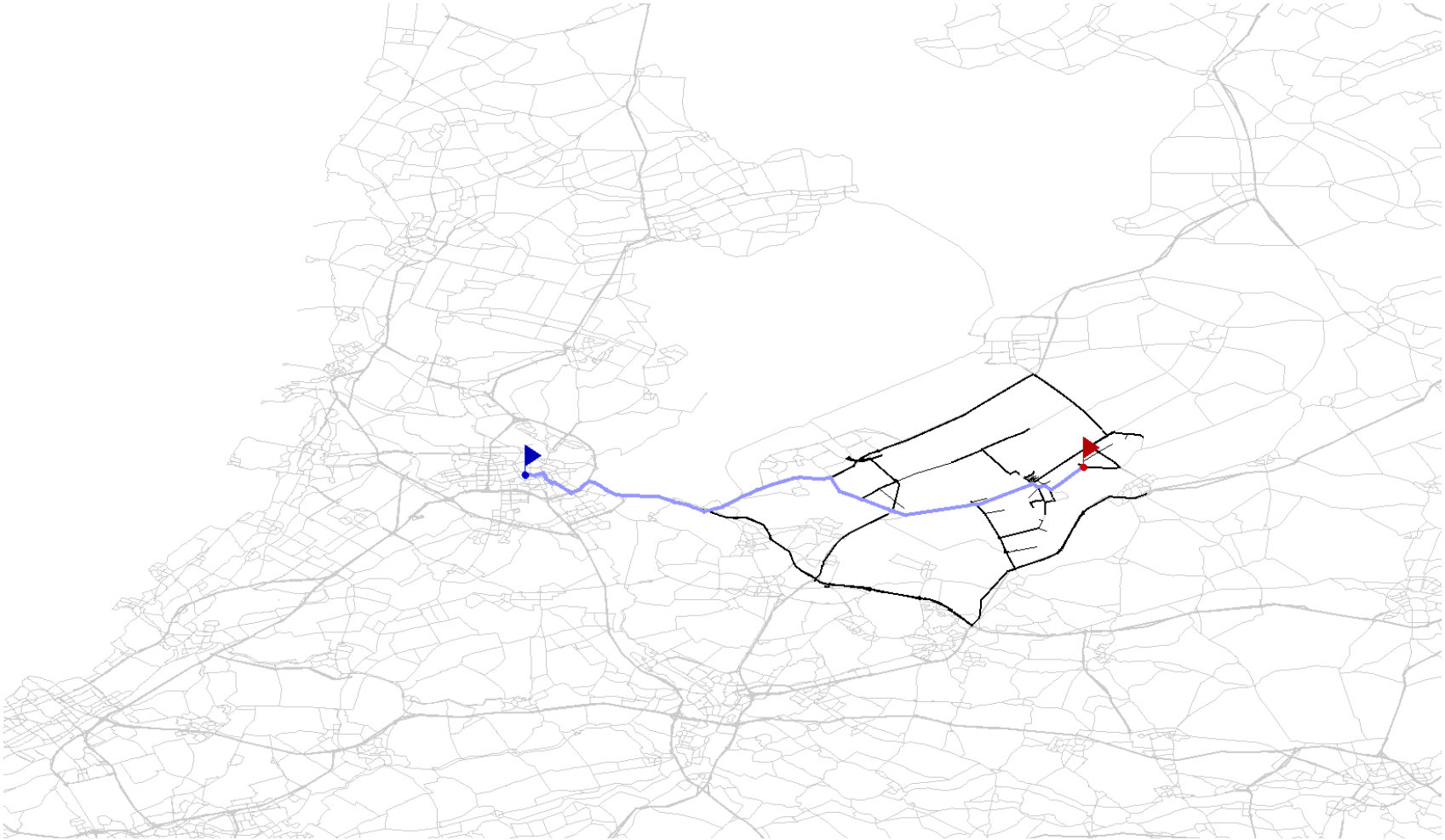
- bestimme gemeinsamen Level l von u und t
- werte Flaggen auf dem Level l aus

Suchraum



Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Suchraum

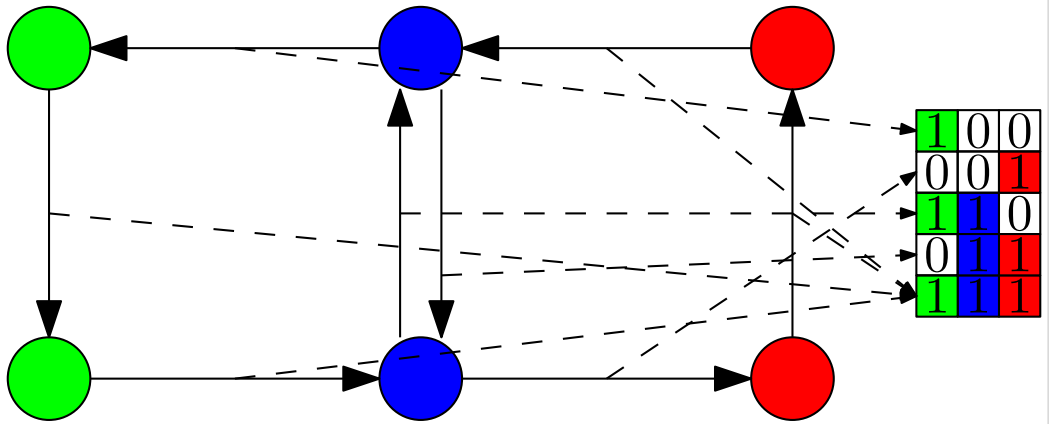


Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Datenstruktur

Effizient Flaggen speichern?

- pro Kante eine Flagge
- Beobachtung: Anzahl Kombinationen begrenzt



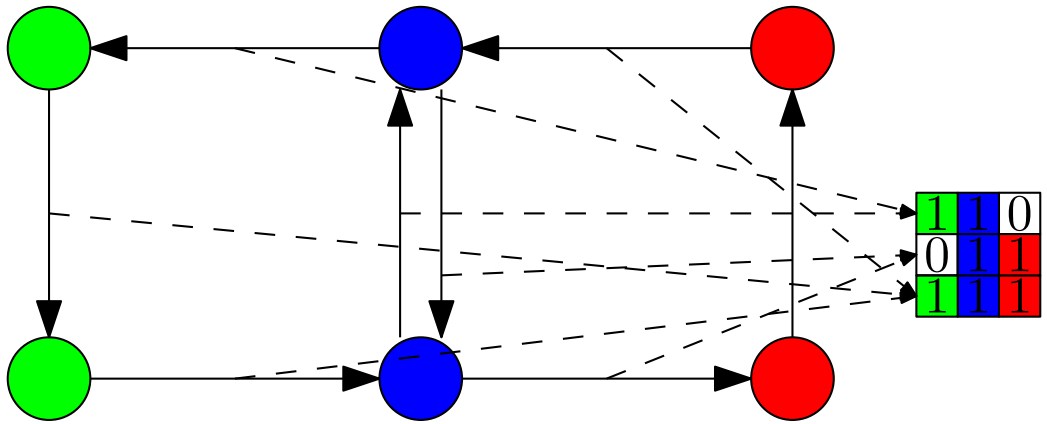
Idee:

- speichere Flaggen in Matrix
- Zeiger von Kanten auf die Matrix
- verringert Speicherverbrauch um einen Faktor 5

Flaggenkompression

Beobachtung:

- kippen eines Bits von 1 auf 0 verboten
- kippen eines Bits von 0 auf 1 erlaubt (weiterhin korrekt, eventuell langsamer)



Idee:

- verringere Anzahl eindeutiger Arc-Flags durch kippen
- dadurch Kompression der Matrix
- finde “gutes” Mapping

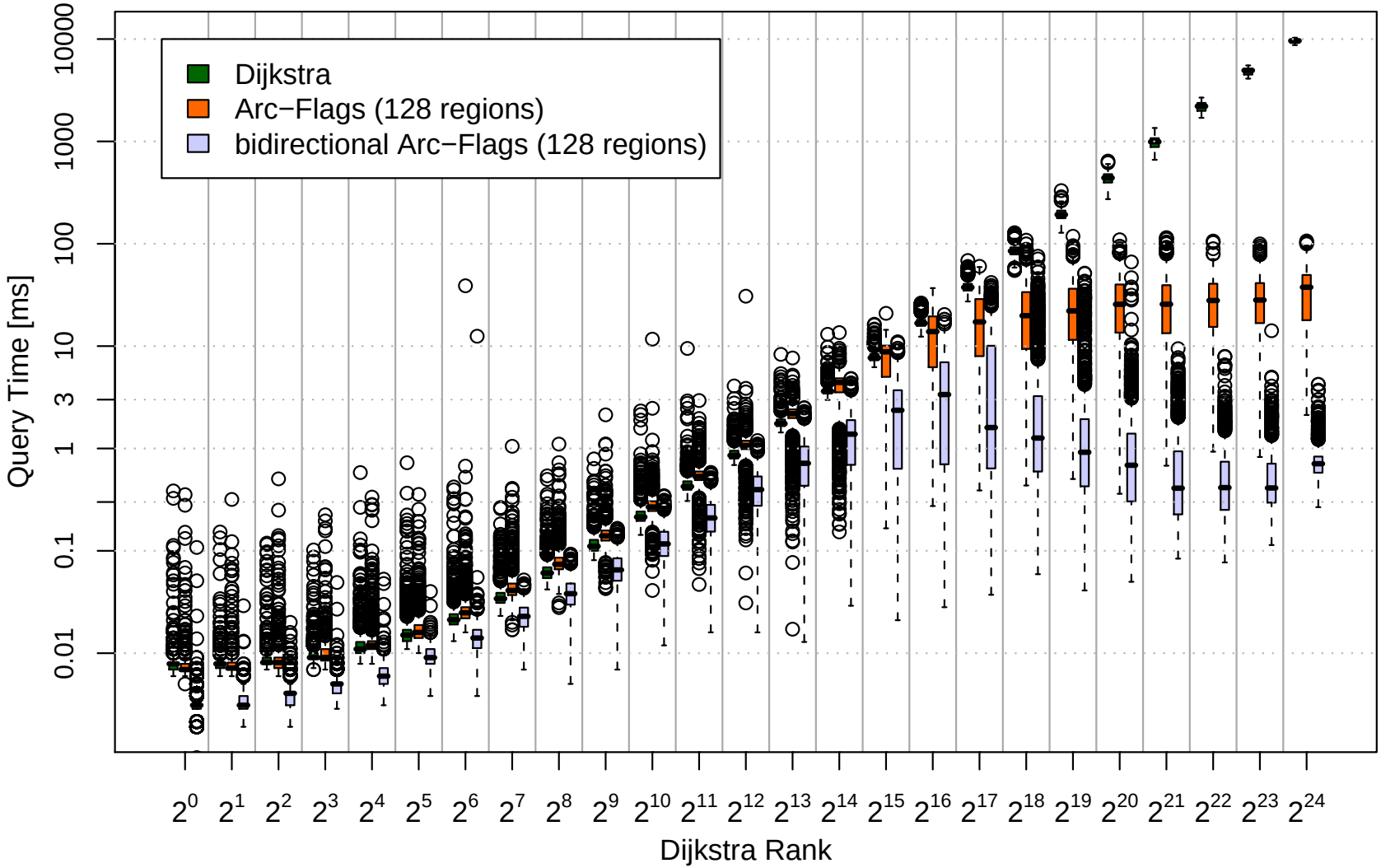
Zufallsanfragen: Anzahl Regionen

	Prepro		Query		
regions	time [min]	space [B/n]	# settled nodes	time [ms]	spd up
0	0	0	9 114 385	5 591.6	1.0
200	1 028	19	2 369	1.6	3 494.8
400	1 366	20	1 868	1.2	4 659.7
600	1 723	21	1 700	1.1	5 083.3
800	1 892	23	1 642	1.4	3 994.0
1000	2 156	25	1 593	1.1	5 083.3

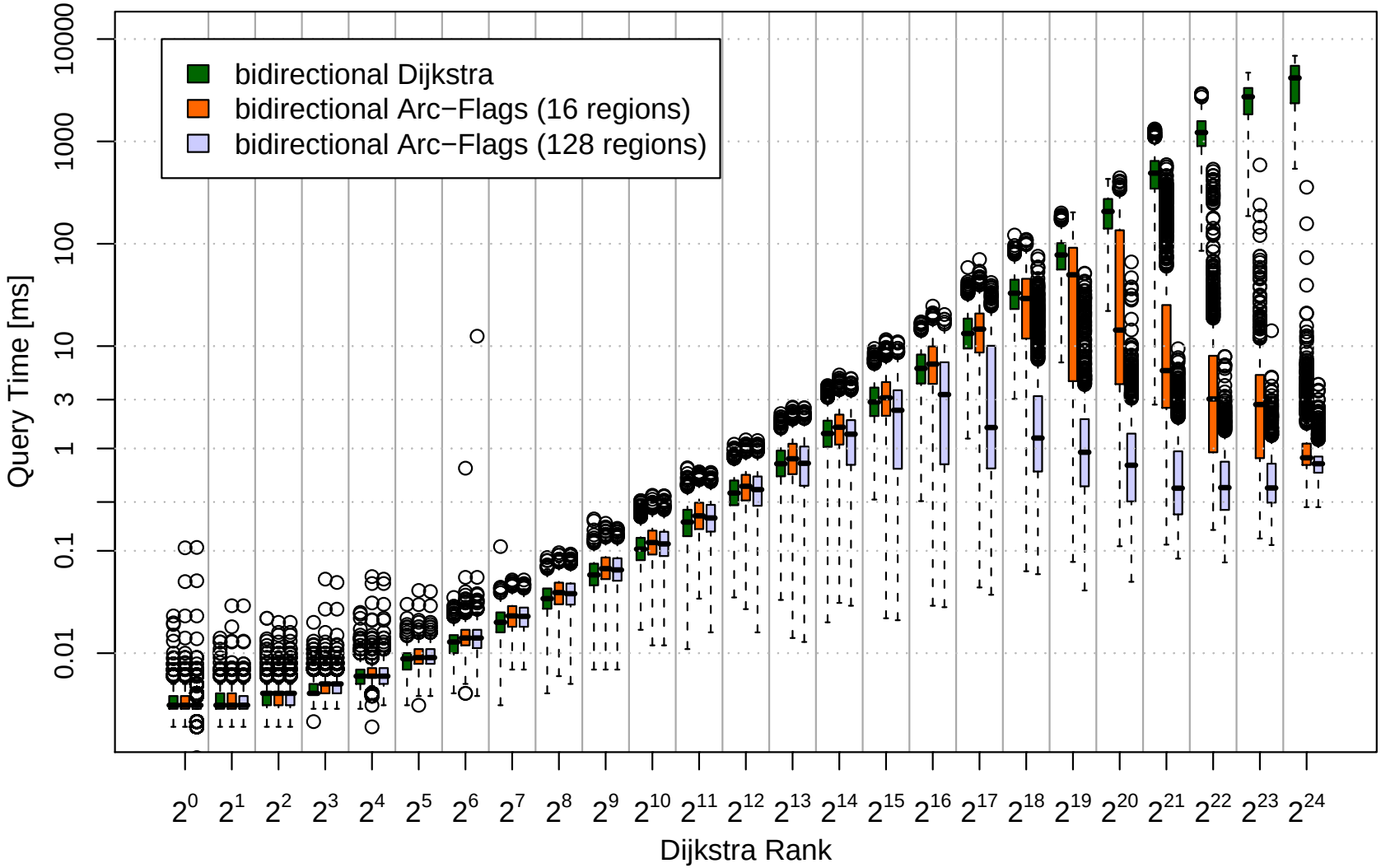
Beobachtungen:

- lange Vorberechnung
- hohe Beschleunigung
- geringer Speicherverbrauch
- mehr als 200 Regionen lohnt sich nicht

Lokale Anfragen Arc-Flags I



Lokale Anfragen Arc-Flags I

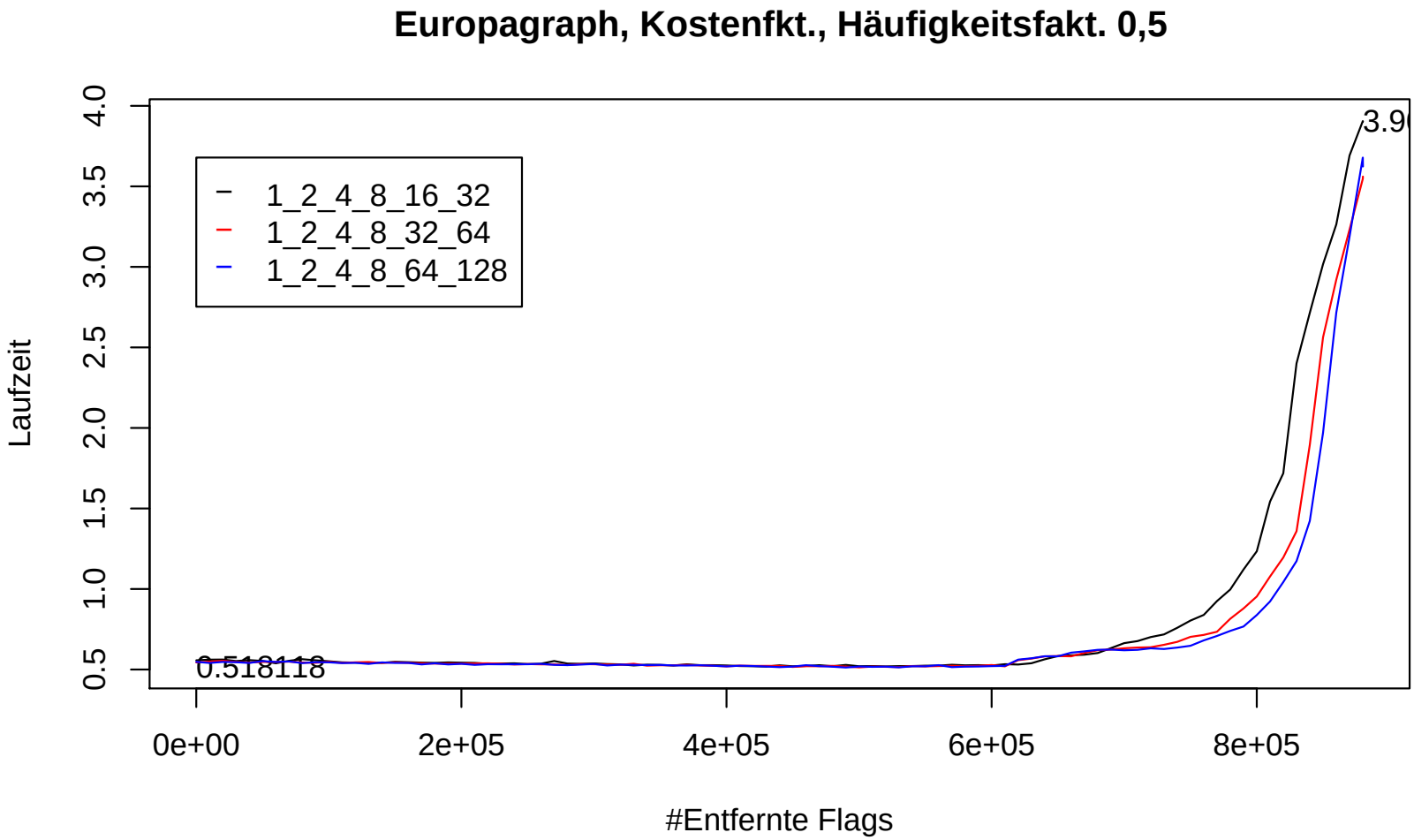


Beobachtungen

- birektionale Arc-Flags deutlich schneller als unidirektionale
- gegenüber Dijkstra nur Beschleunigung für weite Anfragen
- 128 Regionen deutlich besser 16 (bei 2^{24} nahezu gleich auf)

Flaggenkompression

(Multi-Level, unidirektional)



Beobachtungen

- kaum Verlust bis zu 60% entfernte Flaggen
- geringer Verlust bis zu 80% entfernte Flaggen
- kippen von niedrig-leveligen Flaggen billiger

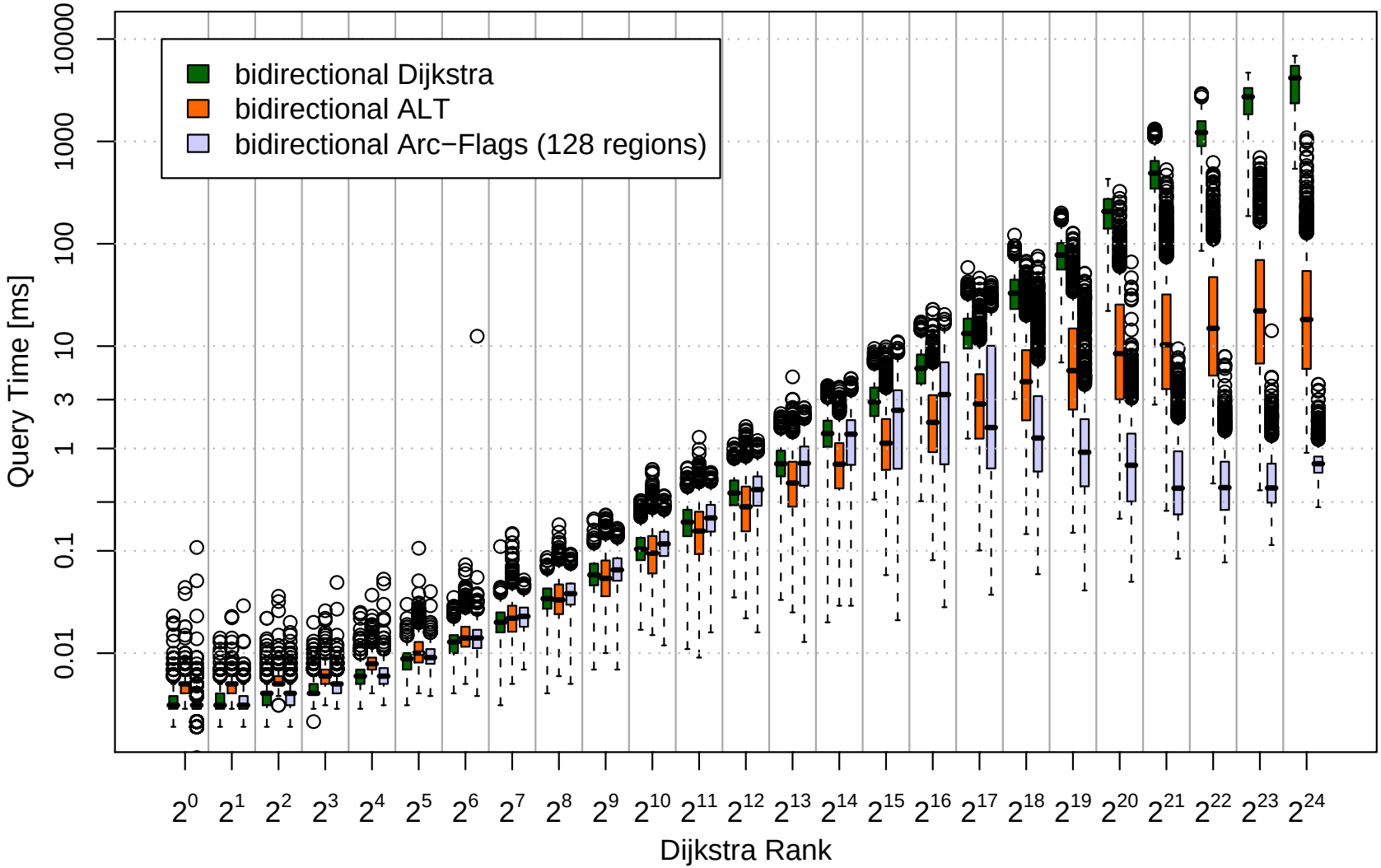
Übersicht: bisherige Techniken

	Vorberechnung		Anfrage		
	Zeit [h:m]	Platz [byte/n]	Such raum	Zeit [ms]	Beschl.
Dijkstra	0:00	0	9 114 385	5 591.6	1.0
Bi-Dijkstra	0:00	0	4 764 110	2 713.2	2.1
Uni-ALT-16	1:25	128	815 639	327.6	17.1
Uni-ALT-64	1:08	512	604 968	288.5	19.4
Bi-ALT-16	1:25	128	74 669	53.6	104.3
Bi-ALT-64	1:08	512	25 324	19.6	285.3
Uni Arc-Flags (128)	8:34	10	92 545	31.9	175.3
Bi Arc-Flags (128)	17:08	20	2 764	0.8	6 988.1

Beobachtung:

- ALT: deutlich unterlegen bei Anfragen und Platzverbrauch
- Arc-Flags: deutlich längere Vorberechnungszeiten

Lokale Anfragen Vergleich



Zusammenfassung

Arc-Flags

- Teile Graphen in k Regionen
- Flaggen zeigen an, ob Kante wichtig für Zielregion ist
- Einfacher Anfrage-Algorithmus
- Vorberechnung
  - Kürzeste-Wege-Baum von jedem Randknoten
  - Manche Flaggen können automatisch gesetzt werden
- Bidirektional und multi-level Erweiterungen
- Beschleunigung von bis zu 7000
- Nahe Anfragen nicht schneller als Dijkstra
- Vorberechnung dauert allerdings immer noch ca. 1 Tag

Literatur

Literatur (Arc-Flags):

- Moritz Hilger and Ekkehard Köhler and Rolf H. Möhring and Heiko Schilling:
  Fast Point-to-Point Shortest Path Computations with Arc-Flags
  In: *Shortest Paths: Ninth DIMACS Implementation Challenge, 2009*

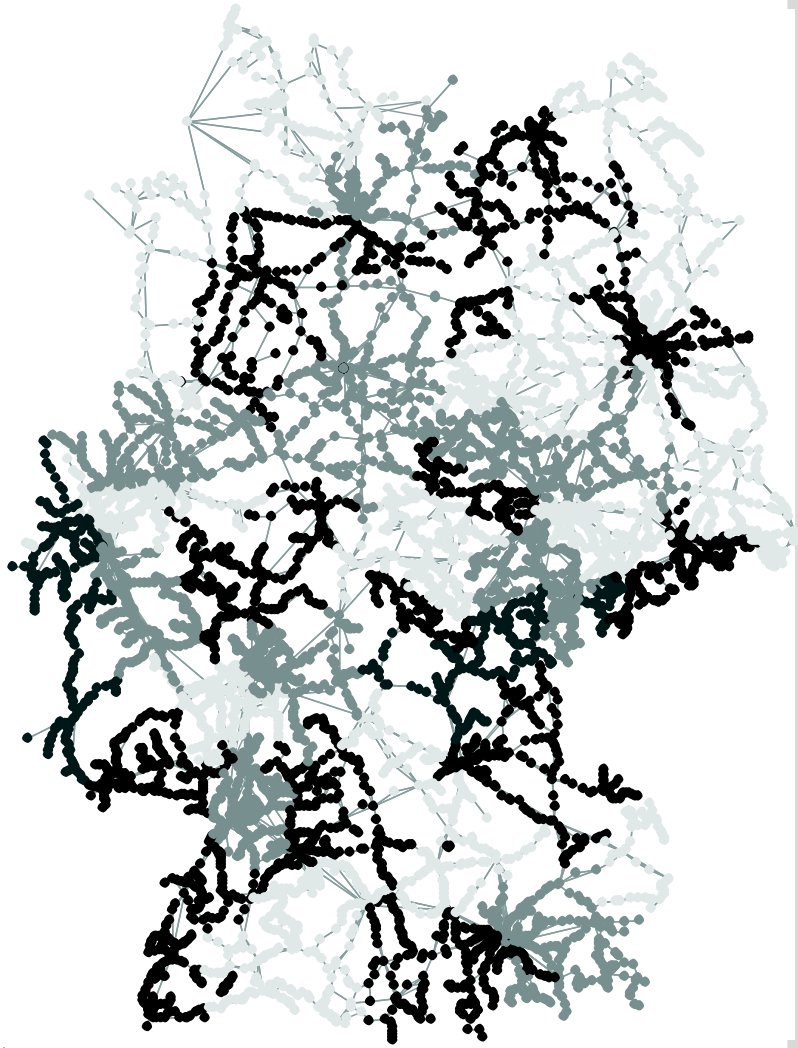
Partitionierung

Anforderungen:

- ausbalanciert
- wenige Randknoten
- zusammenhängend

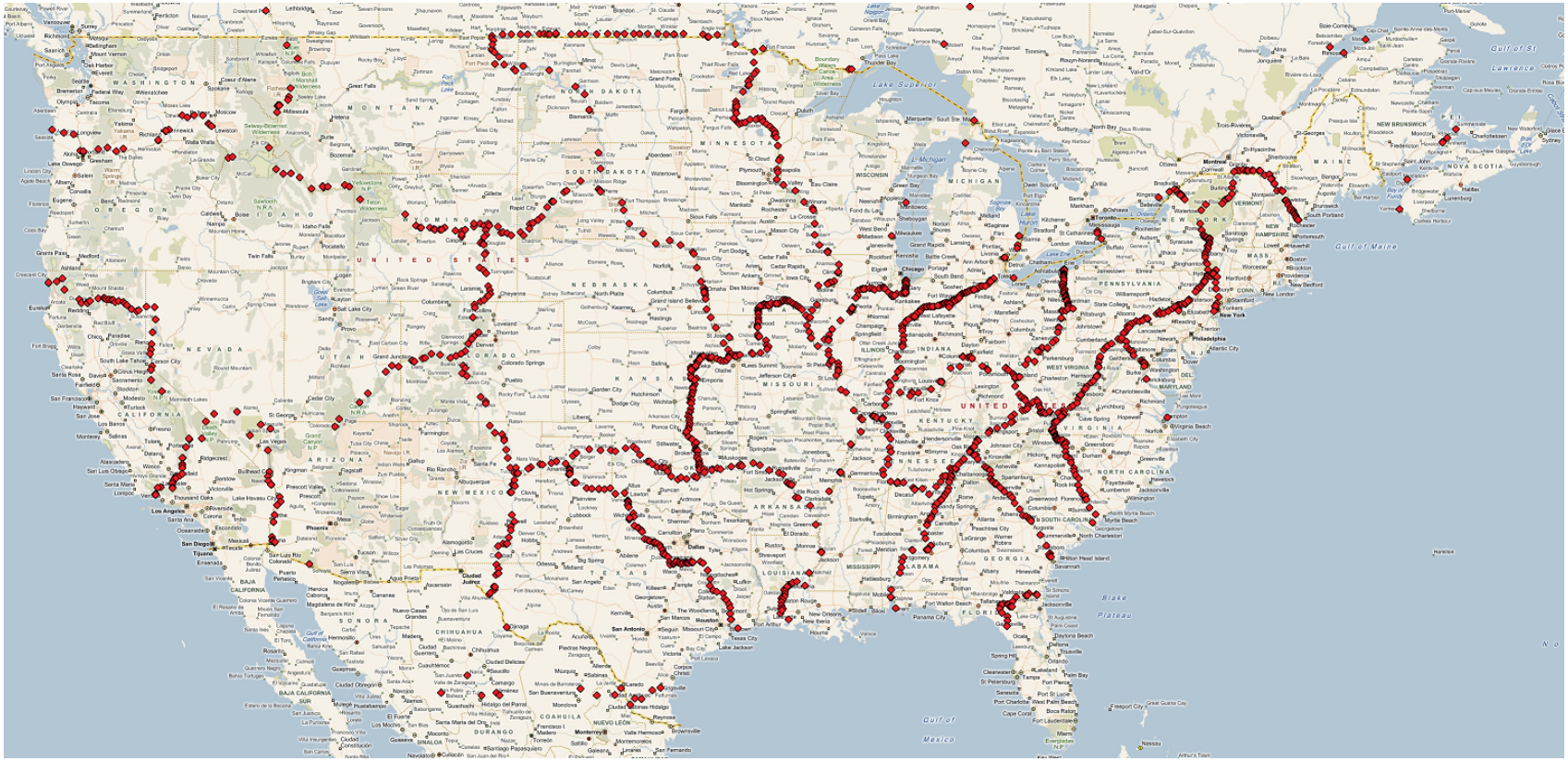
Black-Box Partitionierer:

- benutzen keine Einbettung
- oft: teilen rekursiv Graphen in k Teile mit kleinem Schnitt
- heute: Strassengraphpartitionierer



Graph Partitionierung

- Informell: teile Graphen in lose verbundene Regionen (Zellen).



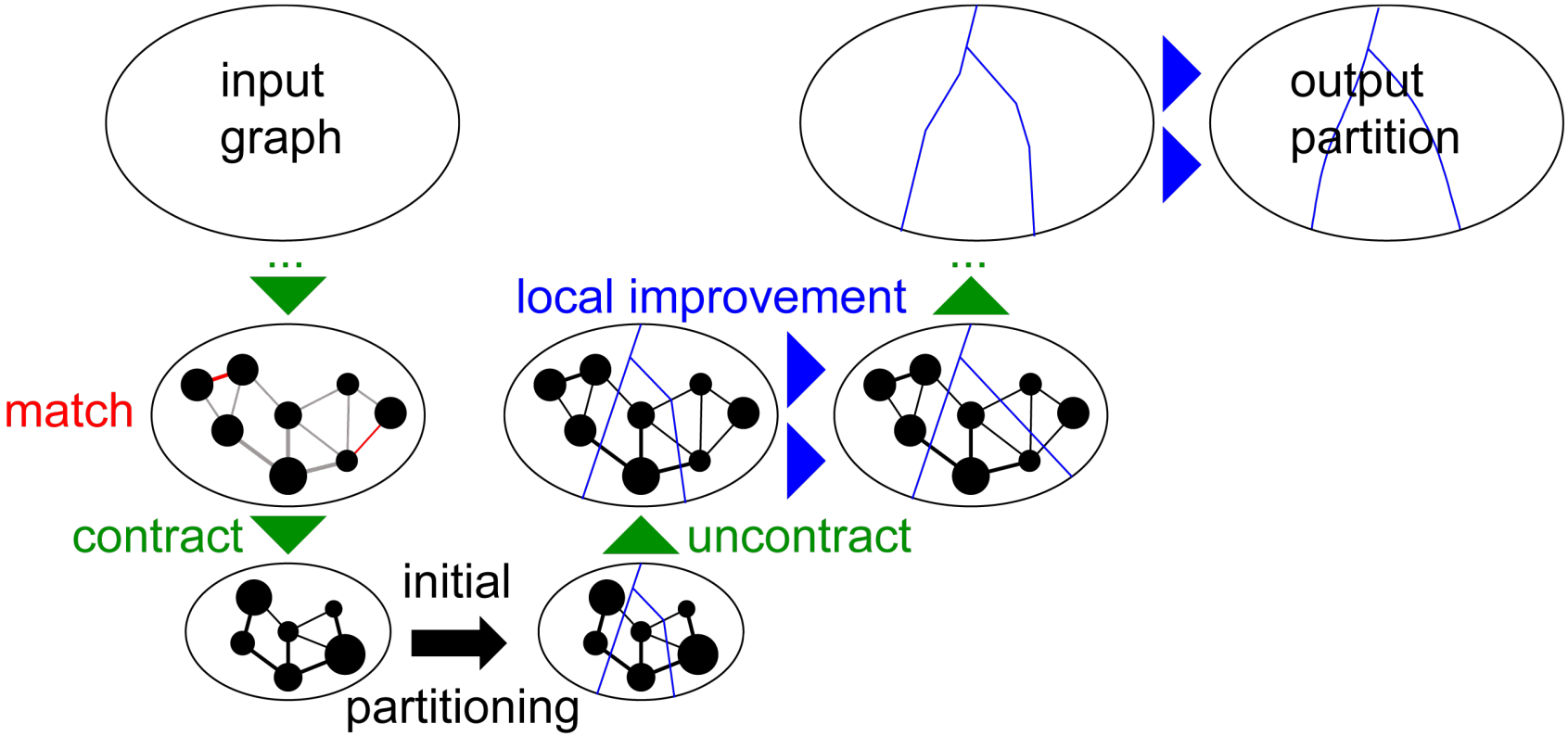
Graph Partitionierung

- Formale definition:
  - Eingabe: ungerichteter graph $G = (V, E)$
  - Ausgabe: Partition von V in Zellen $V_1, V_2, \ldots, V_k$
  - Ziel: minimale Anzahl Kanten zwischen Zellen
- Standard Variante: $|V_i| \leq U$ für festes U:
  - Anzahl Zellen kann variieren ($\geq \lceil n/U \rceil$).
- Balancierte Variante: k Zellen und Unausgeglichenheit ϵ:
  - genau k Zellen (können unzusammenhängend sein) Zellen, Grösse $\leq (1 + \epsilon)\lceil n/U \rceil$.

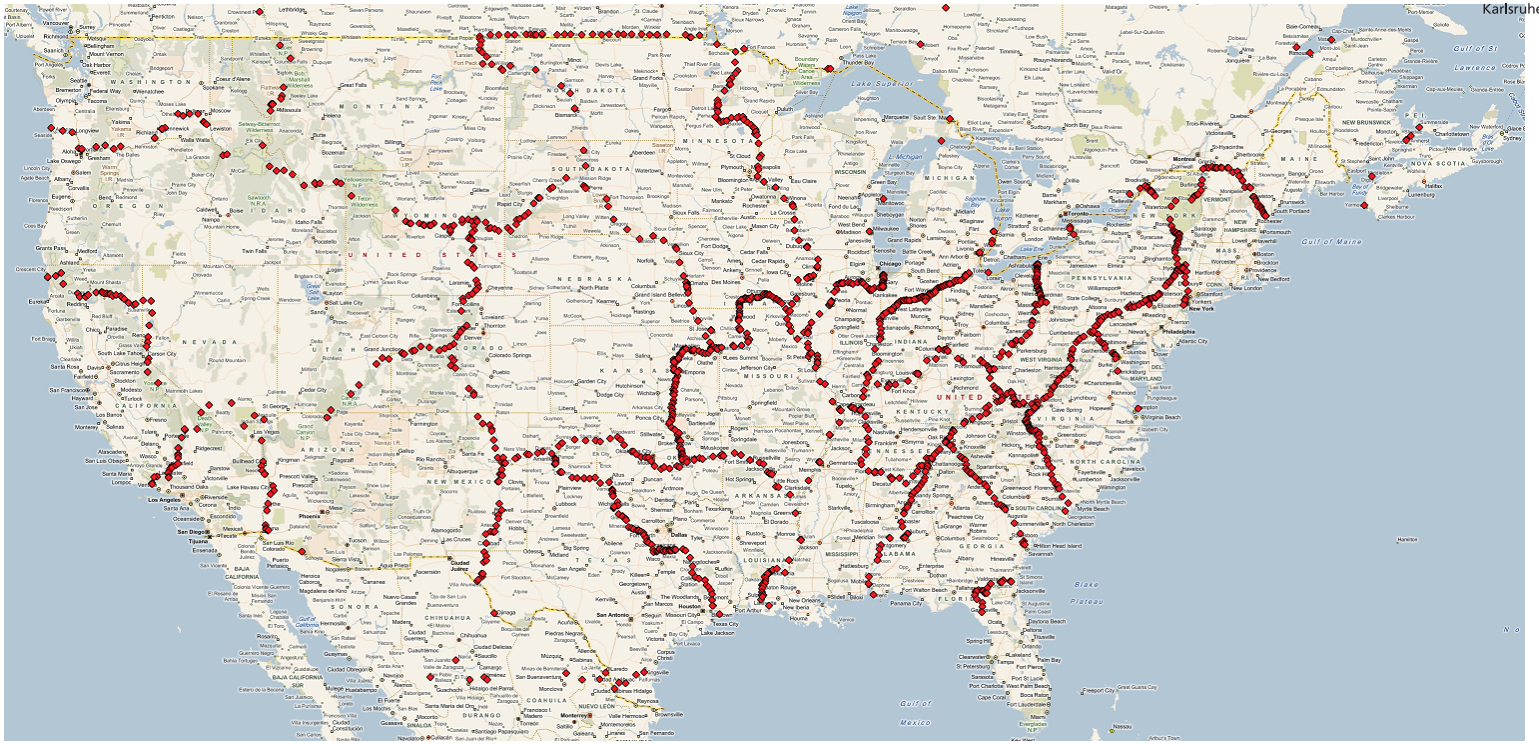
beides NP schwer $\Rightarrow$ benutze Heuristiken

Generelle Verfahren

- METIS [KK99]
- SCOTCH [PR96]
- DiBaP [MMS09]
- Kappa [HSS10], KaSPar [OS10], Kaffpa [SS11], KaffpaE [SS12]



Natural Cuts



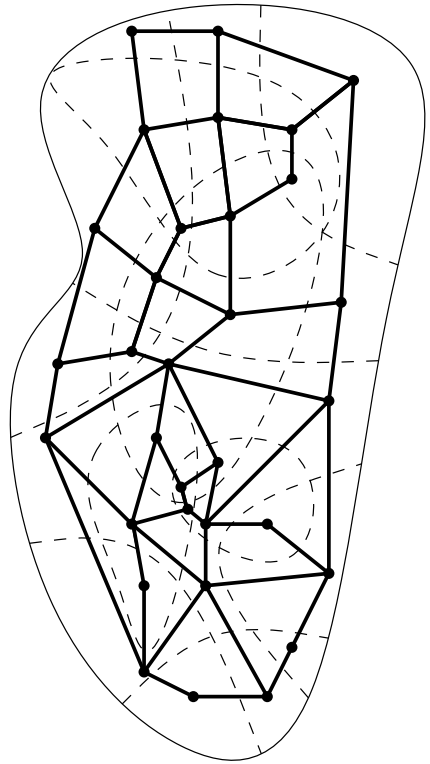
Strassengraphen: dichte Regionen (Gitter) abwechselnd mit natürlichen Schnitten

Flüsse, Berge, Wüsten, Wälder, Parks, Grenzen, Autobahnen, . . .

PUNCH: Partitioner Using Natural-Cut Heuristics

Übersicht

1. Ausdünnung:
   - finde natürliche Schnitte (auf richtiger Skala)
   - behalte Schnittkanten, kontrahiere alle anderen
2. Zusammensetzen:
   - partitioniere (kleineren) kontrahierten graph
   - greedy + lokale Suche [+ Kombinationen]



Schnitte: Definitionen

Schnitt

- Partition des Graphen in 2 Teile (V_1, V_2)
- Grösse: Anzahl Schnittkanten $(|S|)$

Minimaler s-t Schnitt

- entferne minimale Anzahl Kanten, so dass s und t im Graphen nicht mehr verbunden sind
- kann in maximalen Fluss überführt werden
- in Polynomialzeit zu berechnen

Dünnster Schnitt

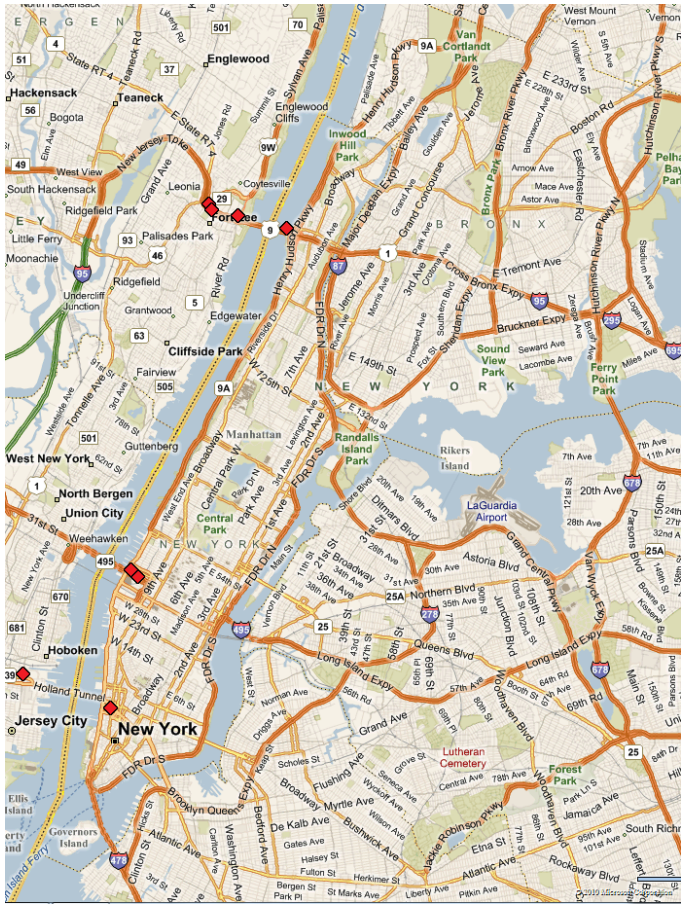
- Schnitt mit $|S|/\min\{|V_1|, |V_2|\}$ minimal
- NP schwer

Exact Graph Bisection

- Schnitt mit $|S|$ minimal und $|V_1|, |V_2| < \lceil |V|/2 \rceil$
- NP schwer

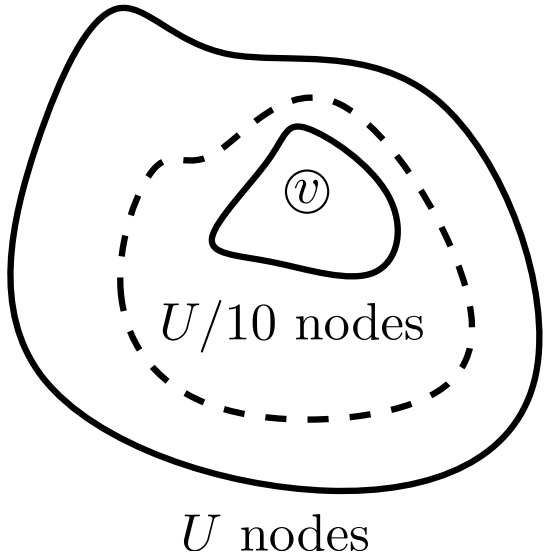
Finde Natürliche Schnitte

- Wir brauchen dünne Schnitte zwischen dichten Regionen:
- Dünnster Schnitt?
  - Zu aufwendig.
- Berechne zufälligen *s*–*t* Schnitt?
  - Meist trivial: Knotengrade sind klein.
- Wir brauchen was anderes:
  - *s*–*t* Schnitte zwischen Regionen



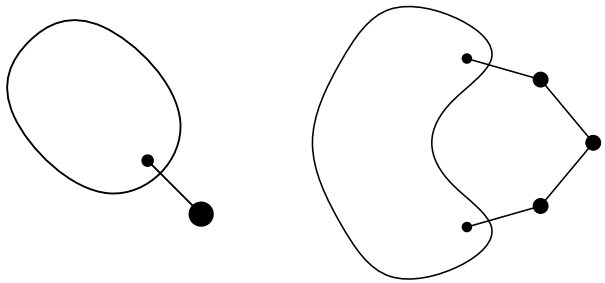
Finde Natürliche Schnitte

1. Wähle ein Zentrum v.
2. Breitensuche der Grösse U um v:
   - Erste $U/10$ Knoten: Kern
   - Verbliebene Knoten in der Queue: Ring
3. Finde minimum Kern/Ring Schnitt:
   - standard s–t minimaler Schnitt.
4. Wiederhole für verschiedene "zufällige" v:
   - bis jeder Knoten in ≥ 2 Kernen war

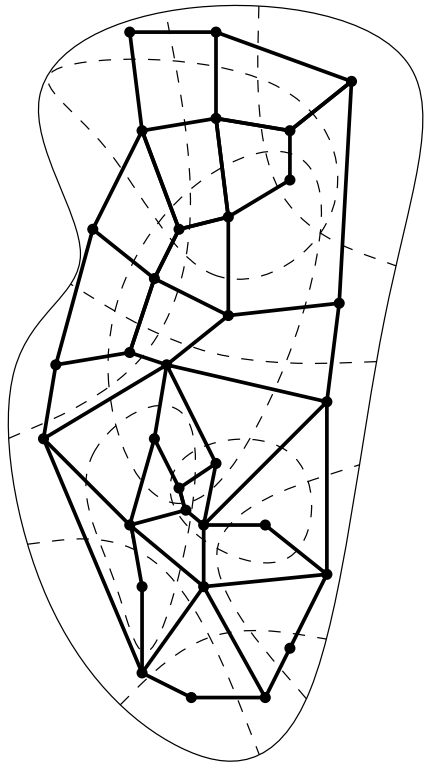


Berechene kleine Schnitte extra:

- identifiziere 1- and 2-Schnitte
- reduziert Strassengraph um Faktor 2
- beschleunigt Schnittfindung



Eigenschaften der Ausdünung



1. viele Kanten werden nie geschnitten
2. Schnittkanten partitionieren den Graphen in Fragmente
3. Fragmentgrösse $\leq U$ (meist viel kleiner)

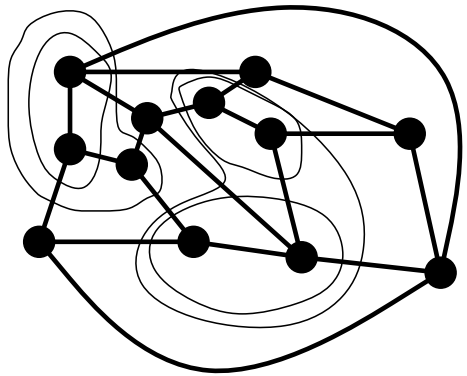
- Generiere Fragmentgraphen:
  - Fragment $\rightarrow$ gewichteter Knoten
  - benachbarte Fragmente $\rightarrow$ gewichtete Kante

U	fragments	frag size
4 096	605 864	30
65 536	104 410	173
1 048 576	10 045	1 793

(Europe: 18M nodes)

Konstruktionsphase

- Algorithmus:

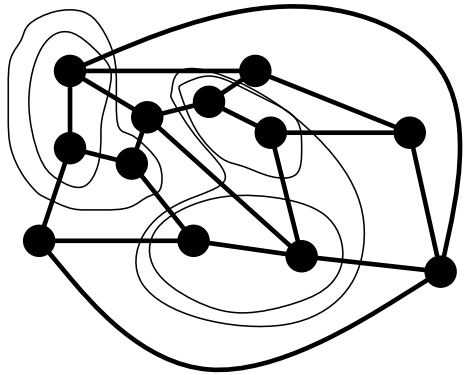


- Zufällig greedy:
  - füge Fragmente zusammen, die stärker verbunden...
  - ...im Verhältnis zu ihrer Grösse.

Ergebnis okay, aber es geht besser.

Konstruktionsphase

- Algorithmus:
  - starte mit isolierten Fragmenten;

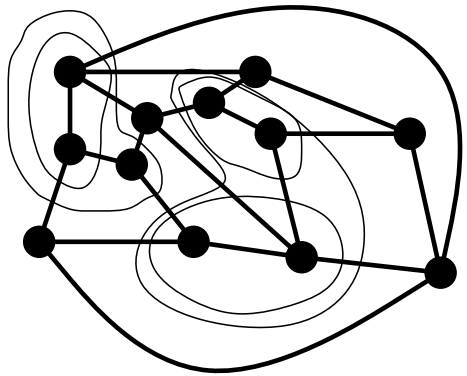


- Zufällig greedy:
  - füge Fragmente zusammen, die stärker verbunden...
  - ...im Verhältnis zu ihrer Grösse.

Ergebnis okay, aber es geht besser.

Konstruktionsphase

- Algorithmus:
  - starte mit isolierten Fragmenten;
  - kombiniere adjazente Zellen;

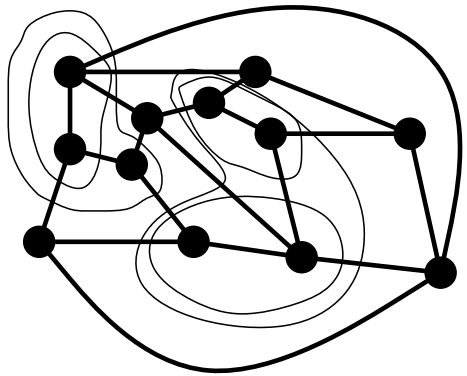


- Zufällig greedy:
  - füge Fragmente zusammen, die stärker verbunden...
  - ...im Verhältnis zu ihrer Grösse.

Ergebnis okay, aber es geht besser.

Konstruktionsphase

- Algorithmus:
  - starte mit isolierten Fragmenten;
  - kombiniere adjazente Zellen;

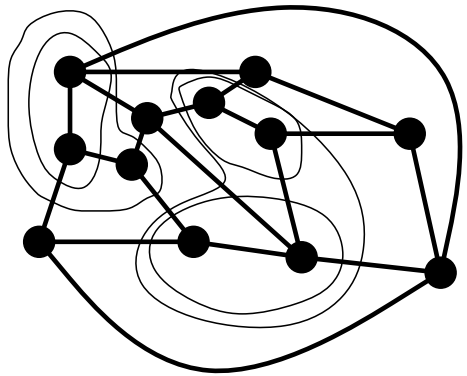


- Zufällig greedy:
  - füge Fragmente zusammen, die stärker verbunden...
  - ...im Verhältnis zu ihrer Grösse.

Ergebnis okay, aber es geht besser.

Konstruktionsphase

- Algorithmus:
  - starte mit isolierten Fragmenten;
  - kombiniere adjazente Zellen;

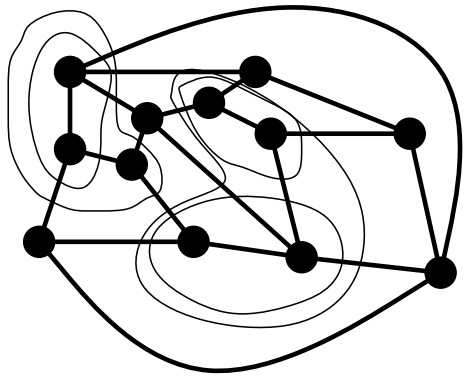


- Zufällig greedy:
  - füge Fragmente zusammen, die stärker verbunden...
  - ...im Verhältnis zu ihrer Grösse.

Ergebnis okay, aber es geht besser.

Konstruktionsphase

- Algorithmus:
  - starte mit isolierten Fragmenten;
  - kombiniere adjazente Zellen;

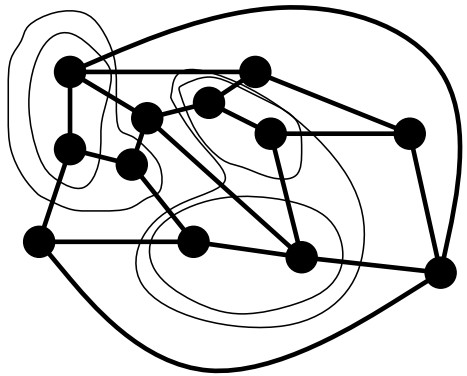


- Zufällig greedy:
  - füge Fragmente zusammen, die stärker verbunden...
  - ...im Verhältnis zu ihrer Grösse.

Ergebnis okay, aber es geht besser.

Konstruktionsphase

- Algorithmus:
  - starte mit isolierten Fragmenten;
  - kombiniere adjazente Zellen;

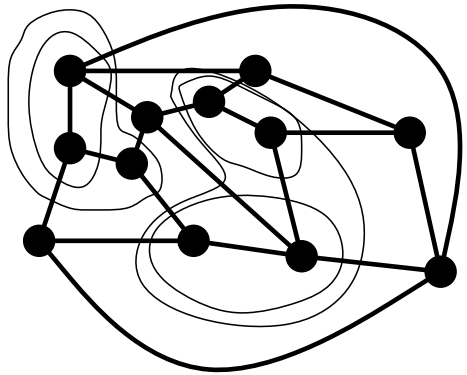


- Zufällig greedy:
  - füge Fragmente zusammen, die stärker verbunden...
  - ...im Verhältnis zu ihrer Grösse.

Ergebnis okay, aber es geht besser.

Konstruktionsphase

- Algorithmus:
  - starte mit isolierten Fragmenten;
  - kombiniere adjazente Zellen;
  - stoppe wenn maximal.

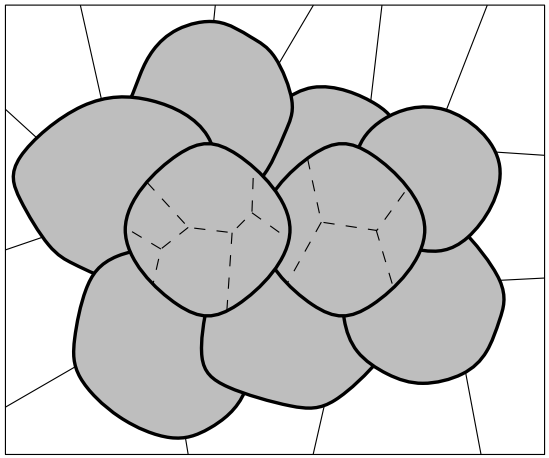


- Zufällig greedy:
  - füge Fragmente zusammen, die stärker verbunden...
  - ...im Verhältnis zu ihrer Grösse.

Ergebnis okay, aber es geht besser.

Lokale Suche

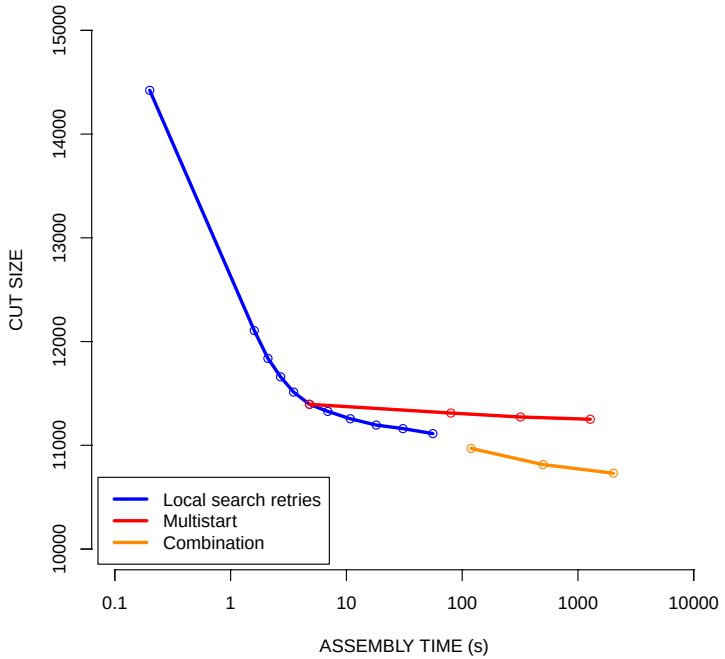
- Für jedes Paar von benachbarten Zellen:
  - zerteilung in Fragments;
  - lass konstruktiven Algorithmus auf Subproblem laufen;
  - behalte Lösung wenn besser.
- Variante benutzt auch Nachbarzellen:
  - mehr Flexibilität;
  - beste Ergebnisse (standard).
- Nachbarzellen können auch in Fragmente zerteilt werden:
  - Subprobleme zu gross;
  - schlechtere Ergebnisse.



Evaluiere jedes Subproblem mehrmals (mit Zufall).

Weitere Verbesserungen

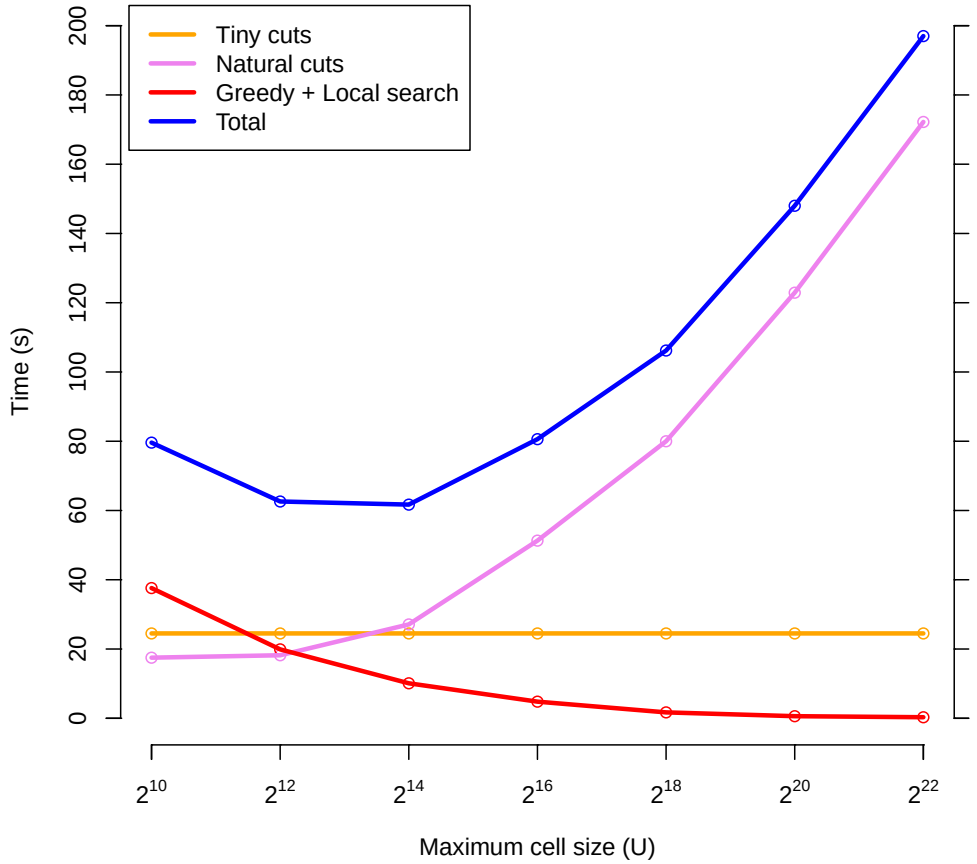
- Mehrfacher Test für jedes Paar
  - lokale Suche hat Zufallskomponenten
- Multistart:
  - konstruktiv+lokale Suche;
  - behalte beste Lösung.
- Kombination:
  - kombiniere manche Lösungen;
  - merge + lokale Suche.



(Europe, $U = 2^{16}$)

längere Laufzeit $\rightarrow$ bessere Lösungen

Laufzeit



Europe (18M vertices), 12 cores

Flaschenhalse: Aufbau für kleine U, Ausdünnung für grosse U

Qualität

U	A	B	$B/\sqrt{U}$	$B/\sqrt[3]{U}$
1 024	895	16.8	0.52	1.66
4 096	3 602	27.6	0.43	1.73
16 384	14 437	45.6	0.36	1.80
65 536	57 376	72.7	0.28	1.80
262 144	222 626	103.7	0.20	1.62
1 048 576	826 166	134.3	0.13	1.32
4 194 304	3 105 245	127.9	0.06	0.79

(Europe, 16 retries, no multistart/combination)

U: maximal erlaubte Zellengrösse

A: durchschn. Zellengrösse der Lösungen

B: durchschn. Randknoten pro Zelle

Strassengraphen haben sehr kleine Separatoren

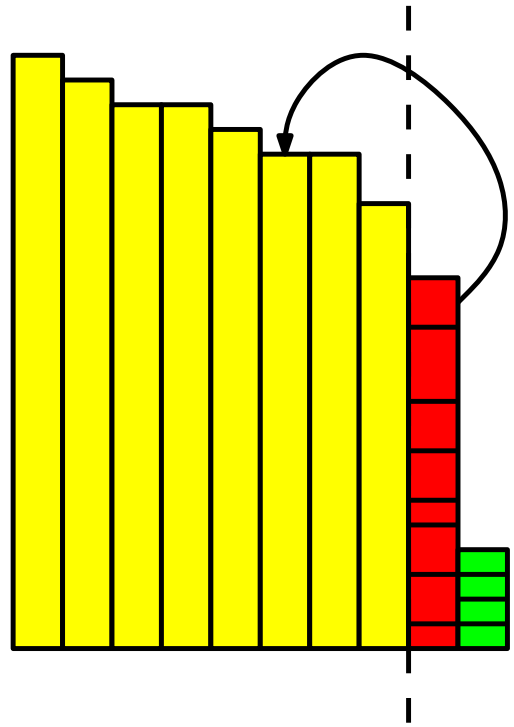
Balancierte Partitionen

Andere Verfahren lösen balancierte Variante:

- finde k Zellen mit Grösse $\leq (1+\epsilon)\lceil n/U \rceil$.

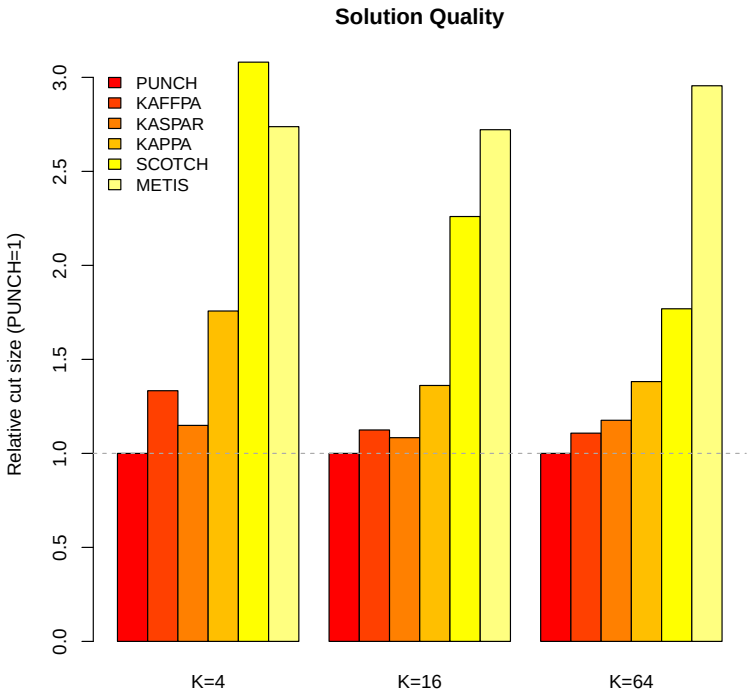
Erweiterung von PUNCH:

1. benutze standard PUNCH mit $U = (1+\epsilon)\lceil n/U \rceil$;
2. wähle k Basis Zellen, verteile den Rest (Multistart)



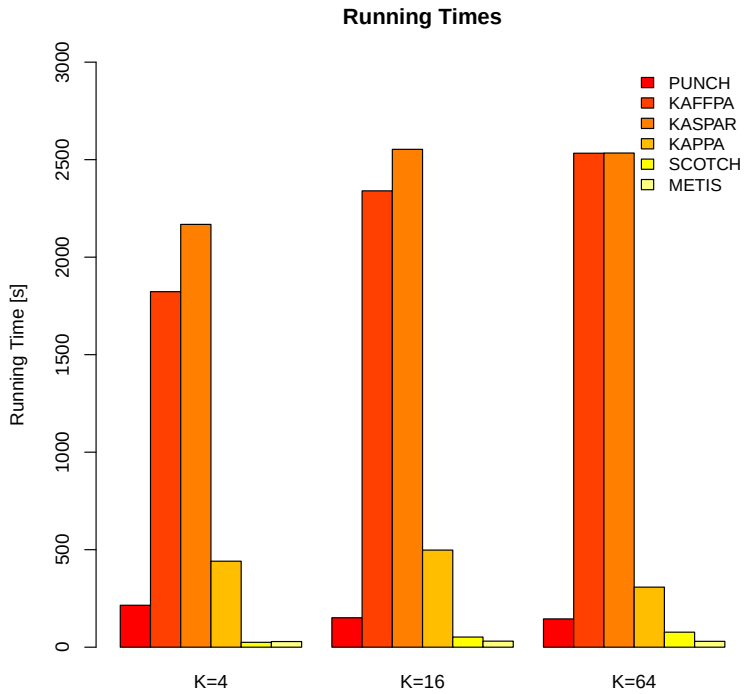
Balancierte Partitionen

PUNCH: bessere Lösungen...



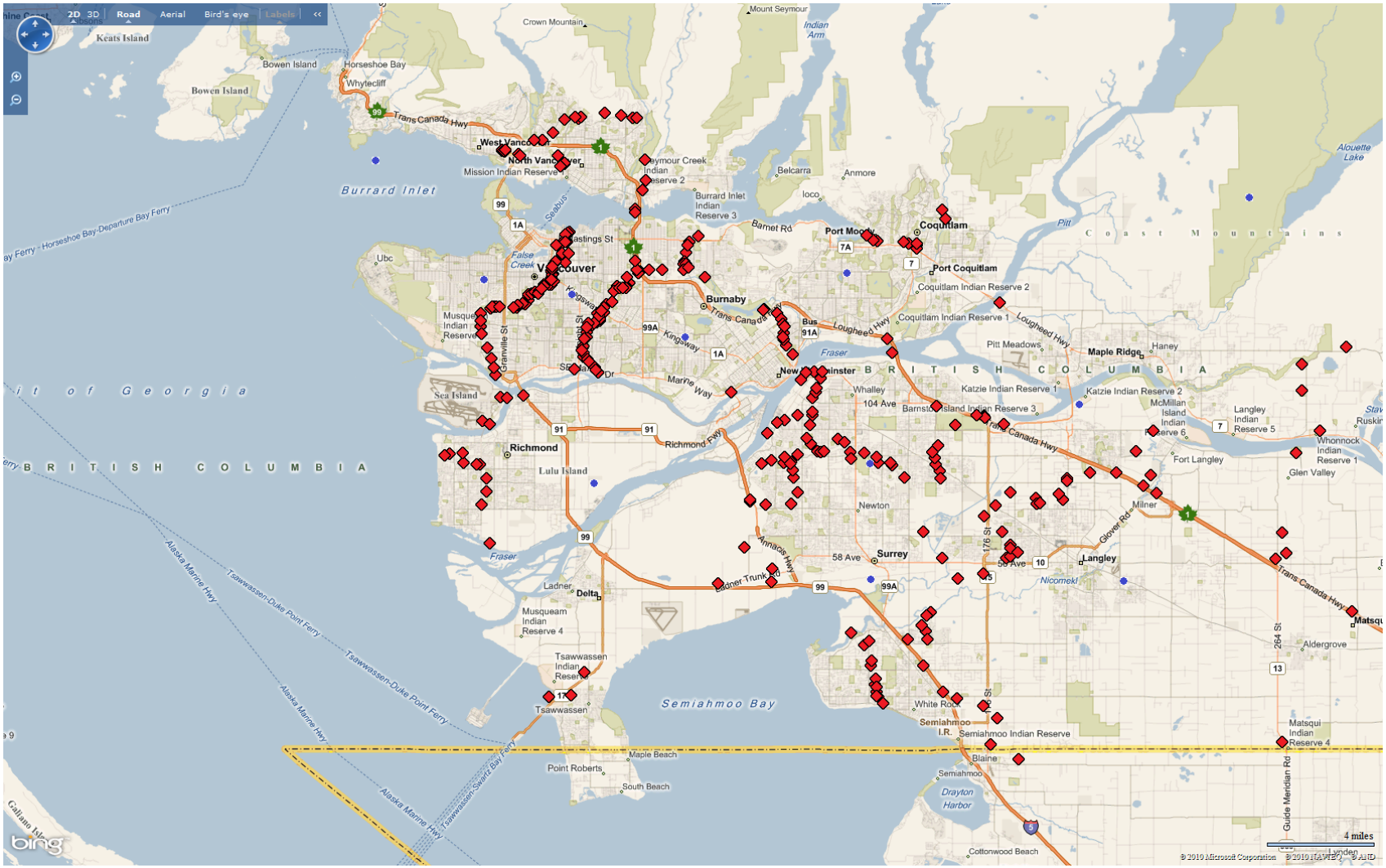
(Europe, $\epsilon = 0.03$)

...in vernünftiger Zeit.

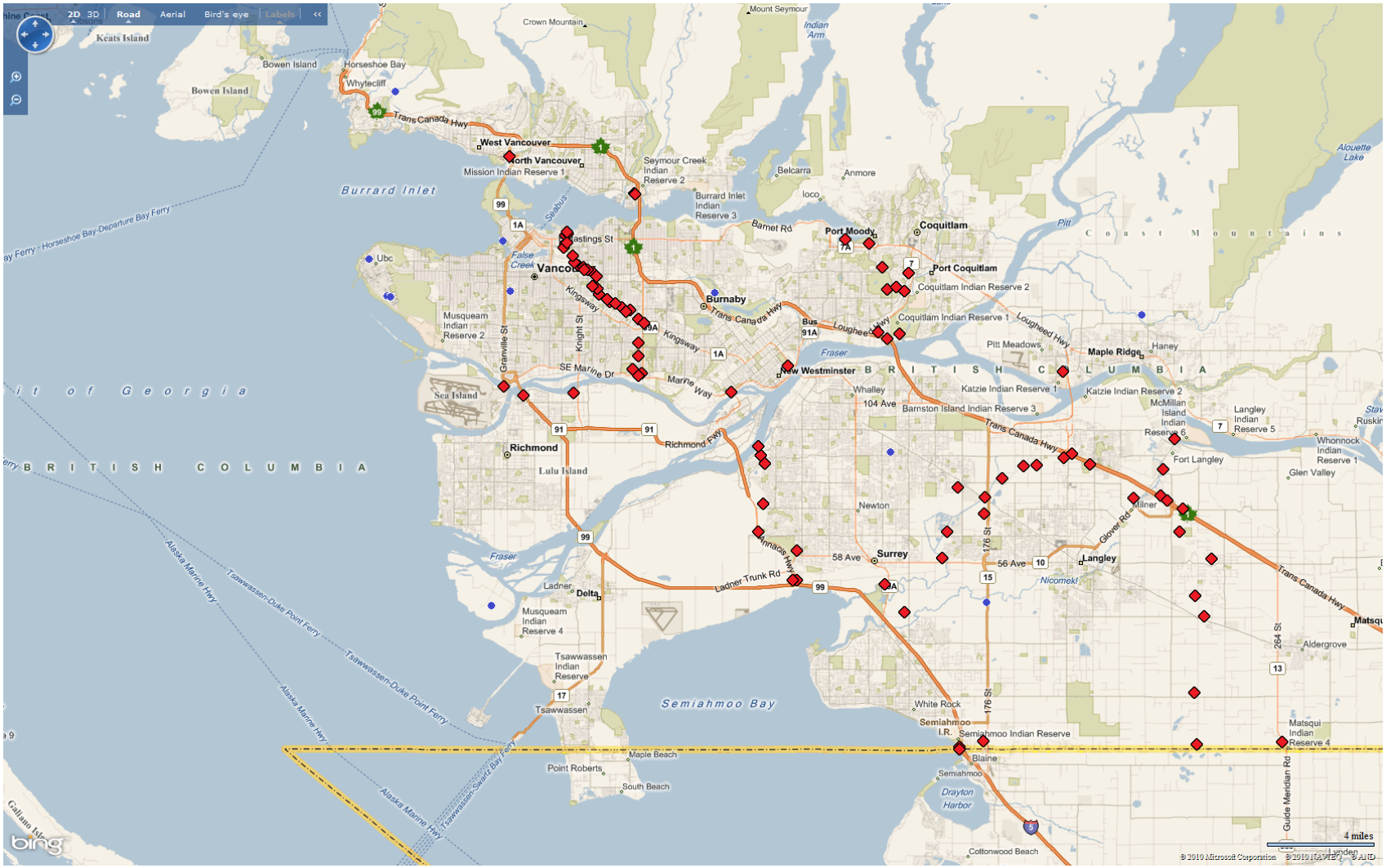


(Europe, $\epsilon = 0.03$)

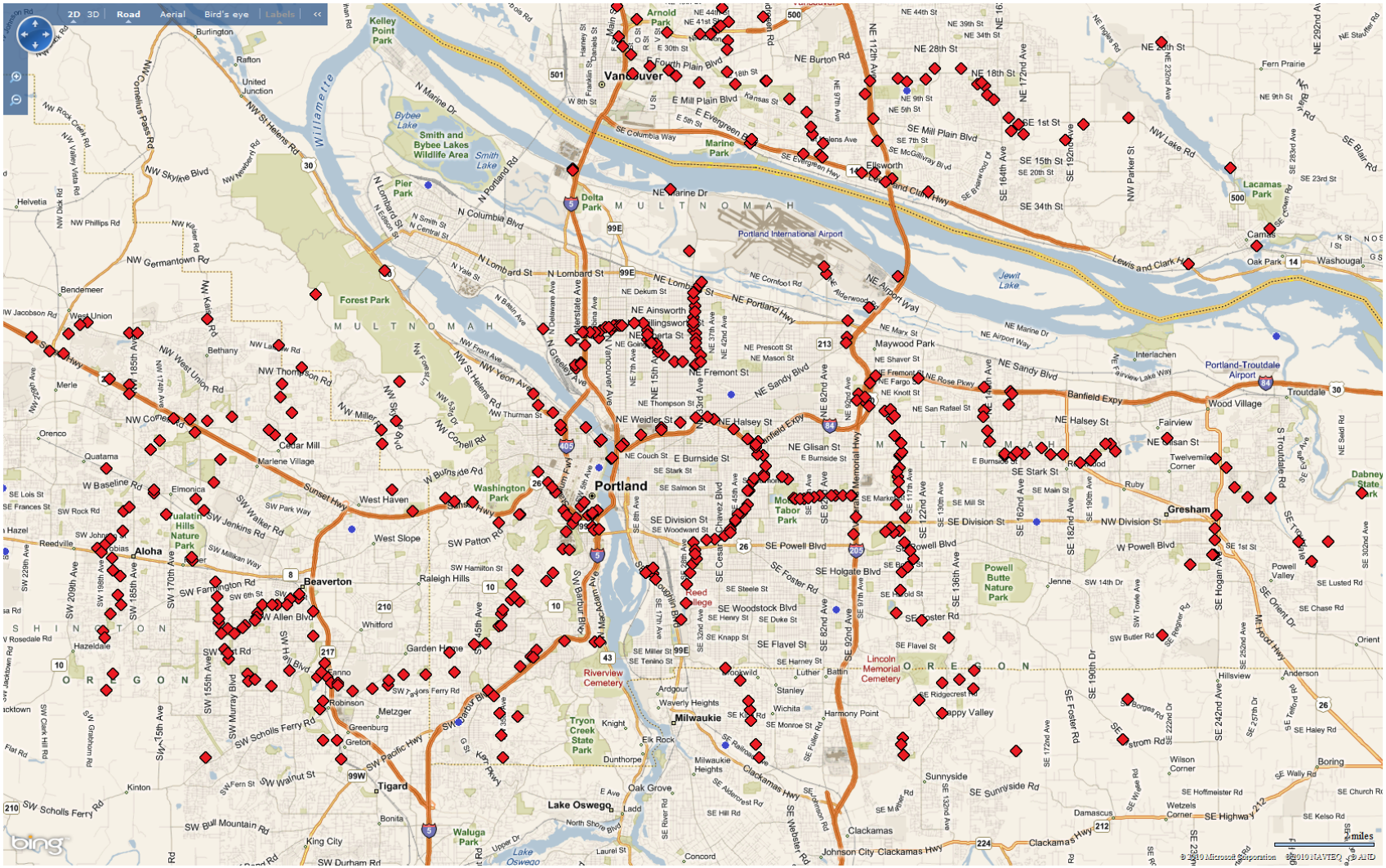
Vancouver mit METIS



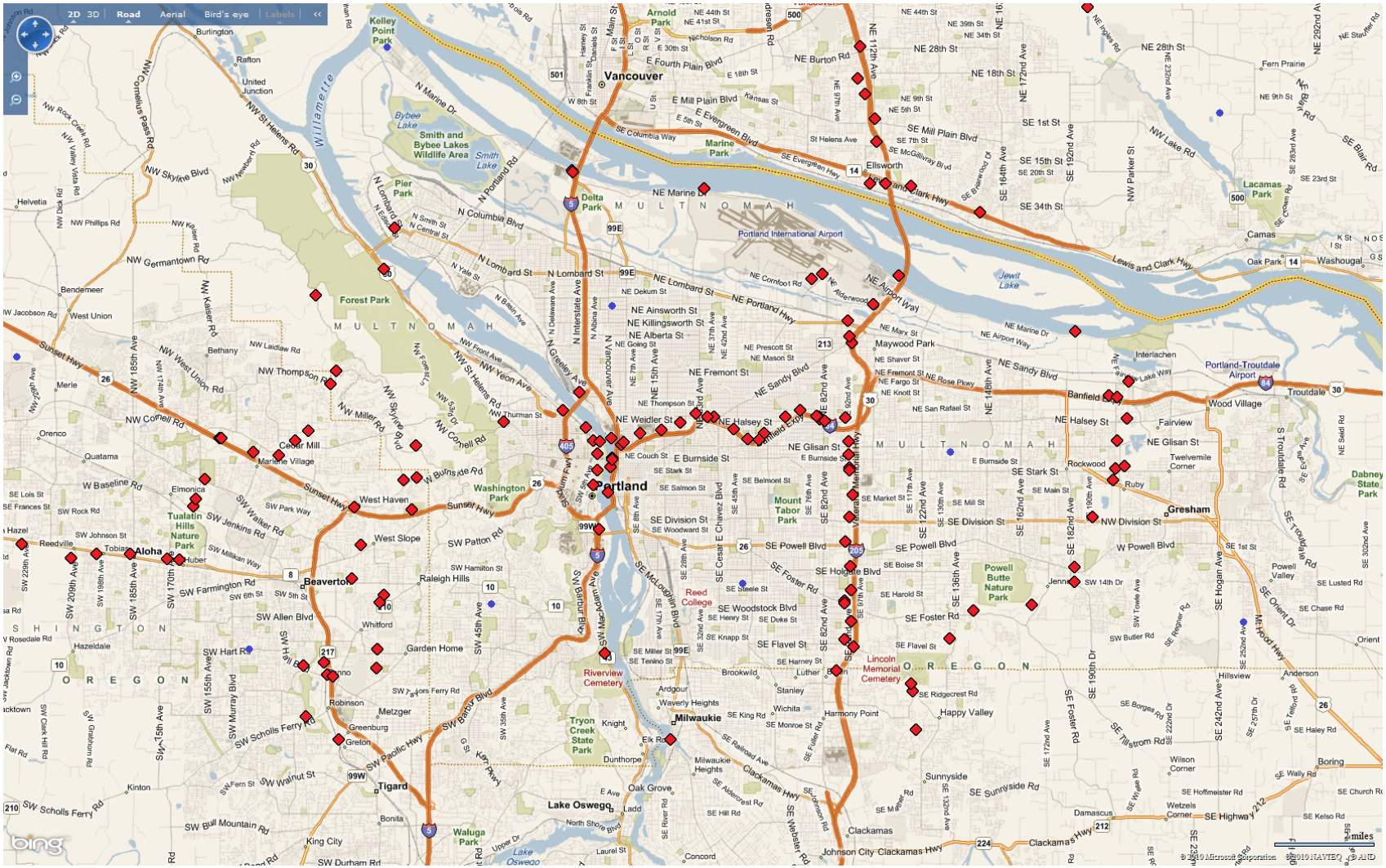
Vancouver mit PUNCH



Portland mit METIS



Portland mit PUNCH



Bemerkungen

- reduziert Randknoten gegenüber METIS um mehr als Faktor 2
- Beschleunigung von Arc-Flags Vorberechnung um Faktor 2
- auch multi-level Partitionen berechnenbar
  top-down liefert beste Ergebnisse
- da Vorteile gegenüber METIS sogar grösser
- Wie weit vom Optimum entfernt?
- Wird für die Bing Routing Engine benutzt.

Literatur

Literatur (Partitionierung):

- George Karypis, Vipin Kumar:
  A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs
  In: *SIAM Journal on Scientific Computing, 1998*
- Peter Sanders, Christian Schulz:
  Distributed Evolutionary Graph Partitioning
  In: *ALENEX, 2012*
- Daniel Delling, Andrew Goldberg, Ilya Razenshteyn, Renato Werneck:
  Graph Partitioning with Natural Cuts
  In: *IPDPS, 2011*

Nächste Termine

Montag, 14.5.2012

Mittwoch, 16.5.2012