Graph Partitioning using Natural Cuts

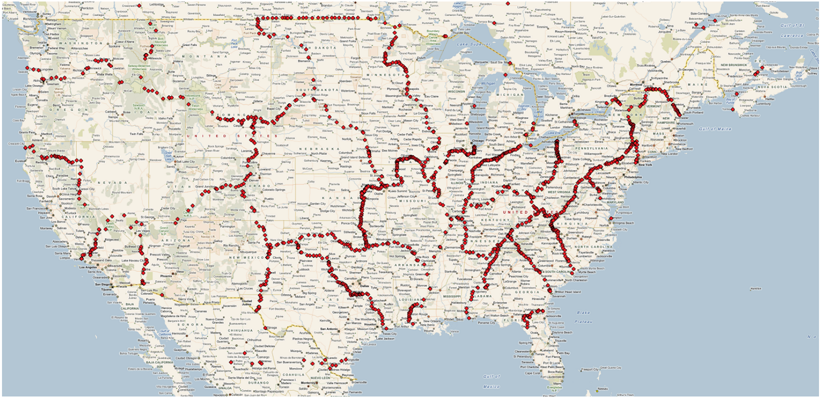
Daniel Delling Andrew Goldberg
Ilya Razenshteyn Renato Werneck

Microsoft Research Silicon Valley

DIMACS 2012

Graph Partitioning

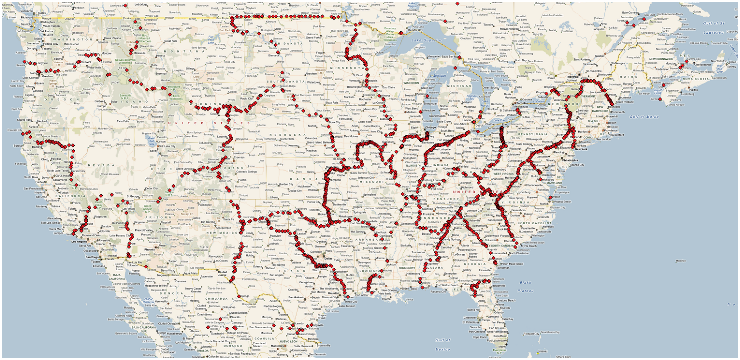
- Informally: split graph into loosely connected regions (cells).



Graph Partitioning

- Formal definition:
  - Input: undirected graph $G = (V, E)$
  - Output: partition of V into cells $V_1, V_2, \ldots, V_k$
  - Goal: minimize edges between cells
- **Standard variant**: enforce $|V_i| \leq U$ for fixed U:
  - #cells may vary ($\geq \lceil n/U \rceil$).
- Balanced variant: fix #cells k and imbalance ϵ:
  - exactly k (maybe disconnected) cells, size $\leq (1+\epsilon)\lceil n/U \rceil$.

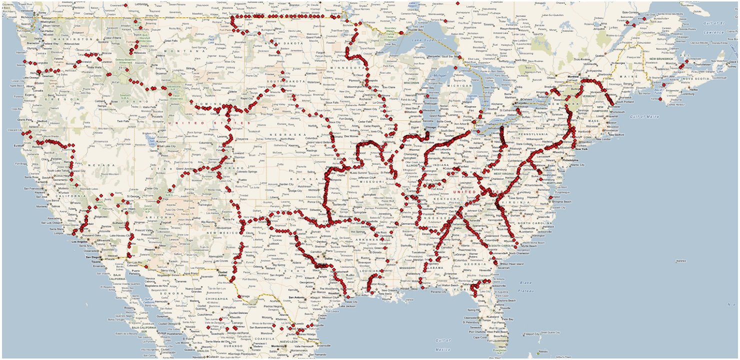
Natural Cuts



Road networks: dense regions (grids) interleaved with **natural cuts**

rivers, mountains, deserts, forests, parks, political borders, freeways, ...

Natural Cuts

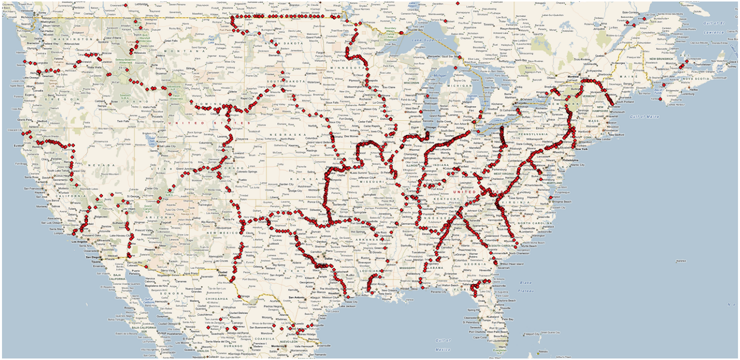


Road networks: dense regions (grids) interleaved with **natural cuts**

rivers, mountains, deserts, forests, parks, political borders, freeways, ...

Partitioner **U**sing **N**atural-**C**ut **H**euristics

Natural Cuts



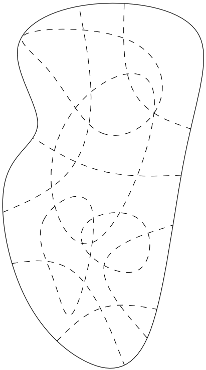
Road networks: dense regions (grids) interleaved with **natural cuts**

rivers, mountains, deserts, forests, parks, political borders, freeways, . . .

PUNCH: **P**artitioner **U**sing **N**atural-**C**ut **H**euristics

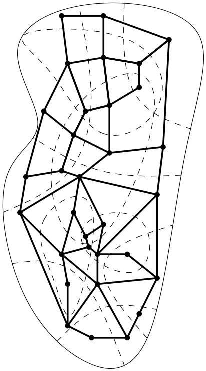
Algorithm Outline

1. Filtering phase:
    - find natural cuts at appropriate scale



Algorithm Outline

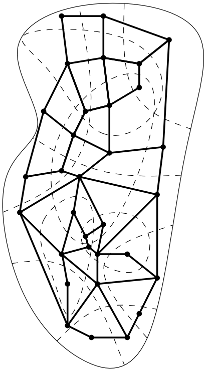
1. Filtering phase:
   - find natural cuts at appropriate scale
   - keep cut edges, contract all others



Algorithm Outline

1. Filtering phase:
   - find natural cuts at appropriate scale
   - keep cut edges, contract all others

2. Assembly phase:
   - partition (smaller) contracted graph
   - greedy + local search [+ combinations]



Microsoft Research

Filtering: Finding Natural Cuts

- Must find sparse cuts between dense regions:
- Sparsest cuts?
  - Too expensive.
- Compute random s–t cuts?
  - Mostly trivial: degrees are small.
- We need something else:
  - s–t cuts **between regions**

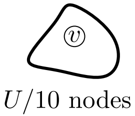
Filtering: Finding Natural Cuts

1. Pick a **center** v.

Microsoft Research

Filtering: Finding Natural Cuts

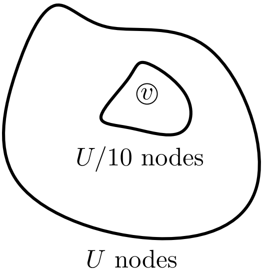
1. Pick a **center** v.
2. Grow BFS of size U around v:
   - First $U/10$ nodes: **core**



Microsoft Research

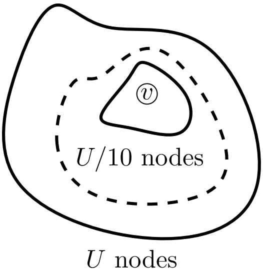
Filtering: Finding Natural Cuts

1. Pick a **center** v.
2. Grow BFS of size U around v:
   - First $U/10$ nodes: **core**
   - Unscanned neighbors: **ring**



Filtering: Finding Natural Cuts

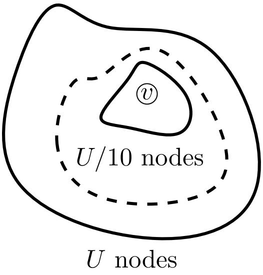
1. Pick a **center** v.
2. Grow BFS of size U around v:
   - First $U/10$ nodes: **core**
   - Unscanned neighbors: **ring**
3. Find minimum **core**/**ring** cut:
   - standard s–t mincut.



Microsoft Research

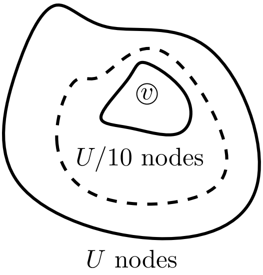
Filtering: Finding Natural Cuts

1. Pick a **center** v.
2. Grow BFS of size U around v:
   - First $U/10$ nodes: **core**
   - Unscanned neighbors: **ring**
3. Find minimum **core**/**ring** cut:
   - standard s–t mincut.
4. Repeat for several "random" v:
   - until each vertex in ≥ 2 cores



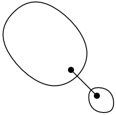
Filtering: Finding Natural Cuts

1. Pick a **center** v.
2. Grow BFS of size U around v:
   - First $U/10$ nodes: **core**
   - Unscanned neighbors: **ring**
3. Find minimum **core**/**ring** cut:
   - standard s–t mincut.
4. Repeat for several "random" v:
   - until each vertex in ≥ 2 cores



Preprocess **tiny cuts** explicitly:

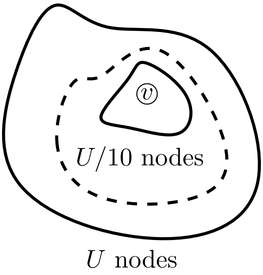
- identify 1-cuts and 2-cuts



Microsoft Research

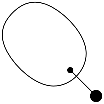
Filtering: Finding Natural Cuts

1. Pick a **center** v.
2. Grow BFS of size U around v:
   - First $U/10$ nodes: **core**
   - Unscanned neighbors: **ring**
3. Find minimum **core**/**ring** cut:
   - standard s–t mincut.
4. Repeat for several "random" v:
   - until each vertex in ≥ 2 cores



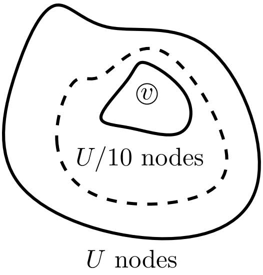
Preprocess **tiny cuts** explicitly:

- identify 1-cuts and 2-cuts



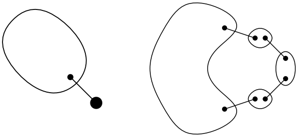
Filtering: Finding Natural Cuts

1. Pick a **center** v.
2. Grow BFS of size U around v:
   - First $U/10$ nodes: **core**
   - Unscanned neighbors: **ring**
3. Find minimum **core**/**ring** cut:
   - standard s–t mincut.
4. Repeat for several "random" v:
   - until each vertex in ≥ 2 cores



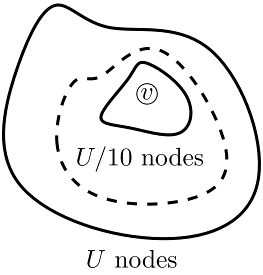
Preprocess **tiny cuts** explicitly:

- identify 1-cuts and 2-cuts



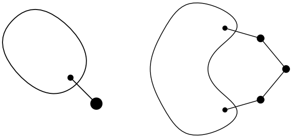
Filtering: Finding Natural Cuts

1. Pick a **center** v.
2. Grow BFS of size U around v:
   - First $U/10$ nodes: **core**
   - Unscanned neighbors: **ring**
3. Find minimum **core**/**ring** cut:
   - standard s–t mincut.
4. Repeat for several "random" v:
   - until each vertex in ≥ 2 cores



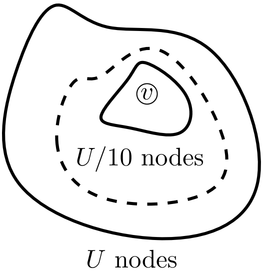
Preprocess **tiny cuts** explicitly:

- identify 1-cuts and 2-cuts



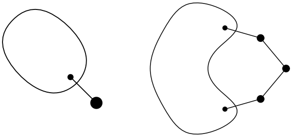
Filtering: Finding Natural Cuts

1. Pick a **center** v.
2. Grow BFS of size U around v:
    - First $U/10$ nodes: **core**
    - Unscanned neighbors: **ring**
3. Find minimum **core**/**ring** cut:
    - standard s–t mincut.
4. Repeat for several "random" v:
    - until each vertex in ≥ 2 cores



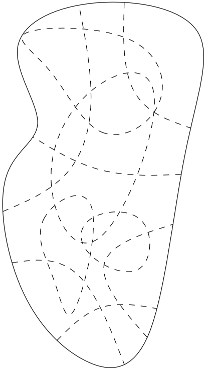
Preprocess **tiny cuts** explicitly:

- identify 1-cuts and 2-cuts
- reduces road networks in half
- accelerates natural cut detection



Properties of Filtering

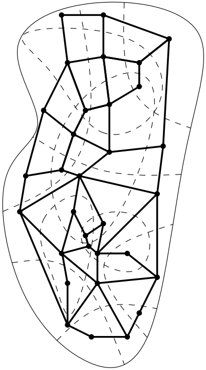
1. many edges are never cut
2. cut edges partition graph into **fragments**
3. fragment size $\leq U$ (usually much less)



Properties of Filtering

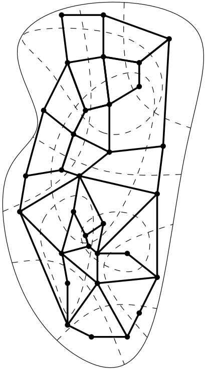
1. many edges are never cut
2. cut edges partition graph into **fragments**
3. fragment size $\leq U$ (usually much less)

- Build **fragment graph**:
    - fragment $\rightarrow$ weighted vertex
    - adjacent fragments $\rightarrow$ weighted edge



Properties of Filtering

1. many edges are never cut
2. cut edges partition graph into **fragments**
3. fragment size $\leq U$ (usually much less)



- Build **fragment graph**:
  - fragment $\rightarrow$ weighted vertex
  - adjacent fragments $\rightarrow$ weighted edge

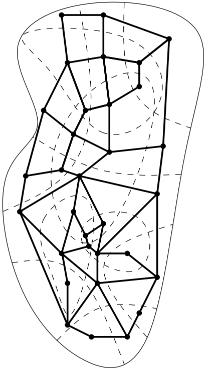
U	fragments	frag size
4 096	605 864	30
65 536	104 410	173
1 048 576	10 045	1 793

(Europe: 18M nodes)

Properties of Filtering

1. many edges are never cut
2. cut edges partition graph into **fragments**
3. fragment size $\leq U$ (usually much less)

- Build **fragment graph**:
  - fragment $\rightarrow$ weighted vertex
  - adjacent fragments $\rightarrow$ weighted edge

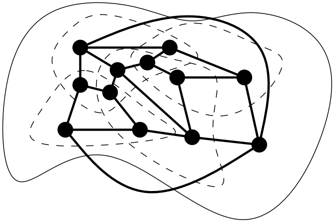


Assembly phase can operate on much smaller graph.

Microsoft Research

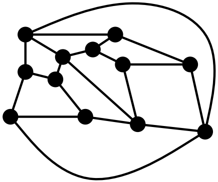
Assembly: Constructive

- Algorithm:



Assembly: Constructive

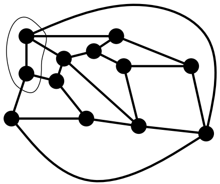
- Algorithm:
  - start with isolated fragments;



Microsoft Research

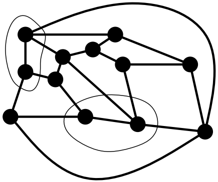
Assembly: Constructive

- Algorithm:
  - start with isolated fragments;
  - combine adjacent cells;



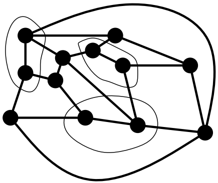
Assembly: Constructive

- Algorithm:
  - start with isolated fragments;
  - combine adjacent cells;



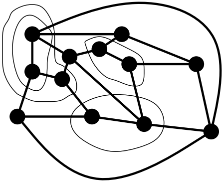
Assembly: Constructive

- Algorithm:
  - start with isolated fragments;
  - combine adjacent cells;



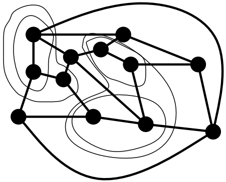
Assembly: Constructive

- Algorithm:
  - start with isolated fragments;
  - combine adjacent cells;



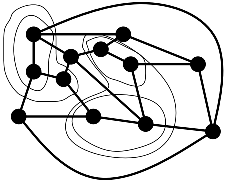
Assembly: Constructive

- Algorithm:
  - start with isolated fragments;
  - combine adjacent cells;



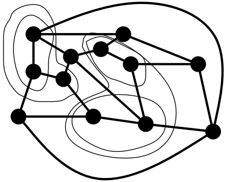
Assembly: Constructive

- Algorithm:
    - start with isolated fragments;
    - combine adjacent cells;
    - stop when maximal.



Assembly: Constructive

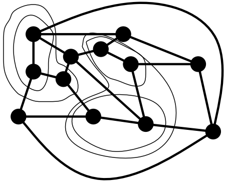
- Algorithm:
  - start with isolated fragments;
  - combine adjacent cells;
  - stop when maximal.



- Randomized greedy:
  - join fragments that are well-connected...
  - ...relative to their sizes.

Assembly: Constructive

- Algorithm:
  - start with isolated fragments;
  - combine adjacent cells;
  - stop when maximal.



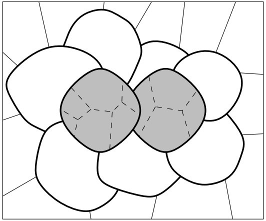
- Randomized greedy:
  - join fragments that are well-connected...
  - ...relative to their sizes.

Reasonable solutions, but one can do better.

Microsoft Research

Assembly: Local Search

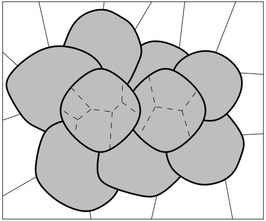
- For each pair of adjacent cells:
  - disassemble into fragments;
  - run constructive on subproblem;
  - keep new solution if better.



Microsoft Research

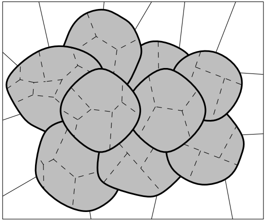
Assembly: Local Search

- For each pair of adjacent cells:
    - disassemble into fragments;
    - run constructive on subproblem;
    - keep new solution if better.
- Variant adds assembled neighbors:
    - more flexibility;
    - best results (default).



Assembly: Local Search

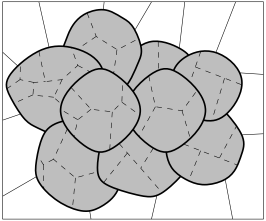
- For each pair of adjacent cells:
  - disassemble into fragments;
  - run constructive on subproblem;
  - keep new solution if better.
- Variant adds assembled neighbors:
  - more flexibility;
  - best results (default).
- Could also disassemble neighbors:
  - subproblems too large;
  - worse results.



Microsoft Research

Assembly: Local Search

- For each pair of adjacent cells:
  - disassemble into fragments;
  - run constructive on subproblem;
  - keep new solution if better.
- Variant adds assembled neighbors:
  - more flexibility;
  - best results (default).
- Could also disassemble neighbors:
  - subproblems too large;
  - worse results.

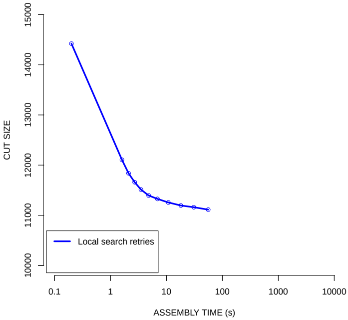


Evaluate each subproblem multiple times (use randomization).

Microsoft Research

Assembly: Better Solutions

- Multiple tries for each pair
  - local search is randomized

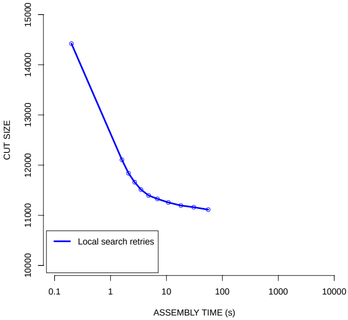


(Europe, $U = 2^{16}$)

Microsoft Research

Assembly: Better Solutions

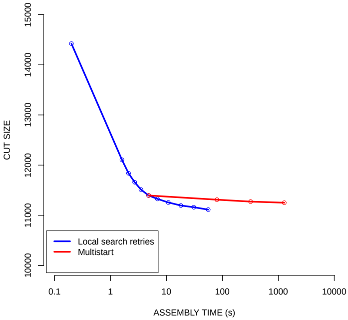
- Multiple tries for each pair
  - local search is randomized
- Multistart:
  - constructive+local search;
  - pick best of multiple runs.



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

Assembly: Better Solutions

- Multiple tries for each pair
  - local search is randomized
- Multistart:
  - constructive+local search;
  - pick best of multiple runs.

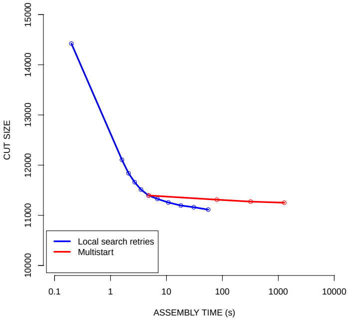


(Europe, $U = 2^{16}$)

Microsoft Research

Assembly: Better Solutions

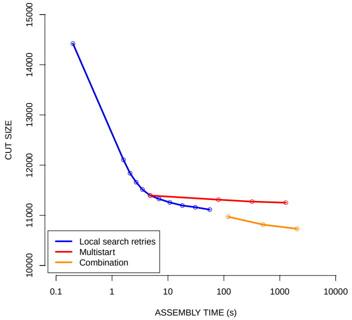
- Multiple tries for each pair
  - local search is randomized
- Multistart:
  - constructive+local search;
  - pick best of multiple runs.
- Combination:
  - combine some solutions;
  - merge + local search.



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

Assembly: Better Solutions

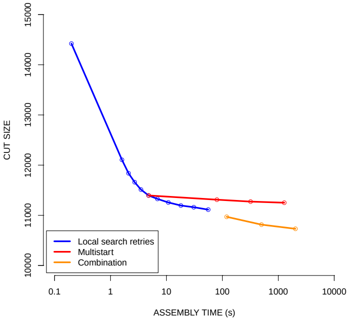
- Multiple tries for each pair
  - local search is randomized
- Multistart:
  - constructive+local search;
  - pick best of multiple runs.
- Combination:
  - combine some solutions;
  - merge + local search.



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

Assembly: Better Solutions

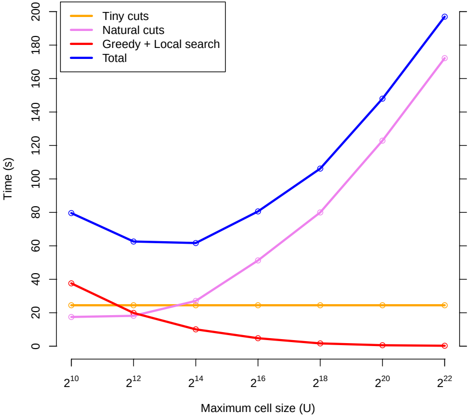
- Multiple tries for each pair
  - local search is randomized
- Multistart:
  - constructive+local search;
  - pick best of multiple runs.
- Combination:
  - combine some solutions;
  - merge + local search.



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

More processing time → better solutions

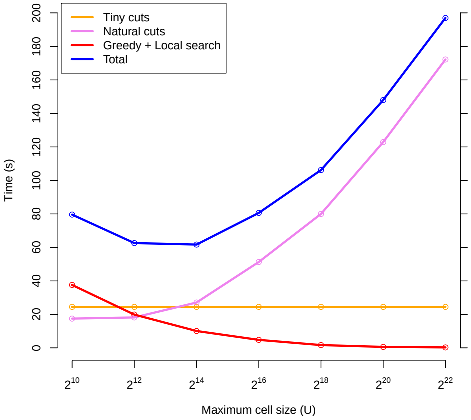
Running Times



Europe (18M vertices), 12 cores

Microsoft Research

Running Times



Europe (18M vertices), 12 cores

Bottlenecks: assembly for small U, filtering for large U

Microsoft Research

Solution Quality

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: maximum cell size allowed

A: average cell size in PUNCH solution

B: average boundary edges per cell

Solution Quality

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: maximum cell size allowed

A: average cell size in PUNCH solution

B: average boundary edges per cell

Road networks have very small separators!

Experimental Comparison

Existing packages:

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

They work on the **balanced variant**:

- find k cells with size $\leq (1+\epsilon)\lceil n/U \rceil$.

PUNCH can find balanced partitions:

1. run standard PUNCH
   with $U = (1+\epsilon)\lceil n/U \rceil$;

Experimental Comparison

Existing packages:

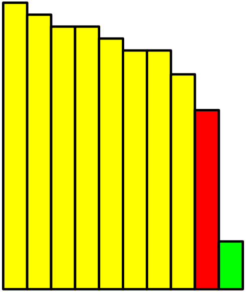
- METIS [KK99]
- SCOTCH [PR96]
- Kappa [HSS10], KaSPar [OS10], Kaffpa [SS11], KaffpaE [SS12]

They work on the **balanced variant**:

- find k cells with size $\leq (1+\epsilon)\lceil n/U \rceil$.

PUNCH can find balanced partitions:

1. run standard PUNCH
   with $U = (1+\epsilon)\lceil n/U \rceil$;



Microsoft Research

Experimental Comparison

Existing packages:

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

They work on the **balanced variant**:

- find k cells with size $\leq (1+\epsilon)\lceil n/U \rceil$.

PUNCH can find balanced partitions:

1. run standard PUNCH
   with $U = (1+\epsilon)\lceil n/U \rceil$;
2. pick k base cells,
   reassign the rest (randomized multistart)

Microsoft Research

Experimental Comparison

Existing packages:

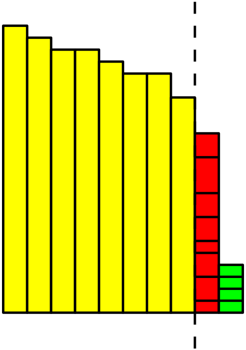
- METIS [KK99]
- SCOTCH [PR96]
- Kappa [HSS10], KaSPar [OS10], Kaffpa [SS11], KaffpaE [SS12]

They work on the **balanced variant**:

- find k cells with size $\leq (1+\epsilon)\lceil n/U \rceil$.

PUNCH can find balanced partitions:

1. run standard PUNCH
   with $U = (1+\epsilon)\lceil n/U \rceil$;
2. pick k base cells,
   reassign the rest (randomized multistart)



Microsoft Research

Experimental Comparison

Existing packages:

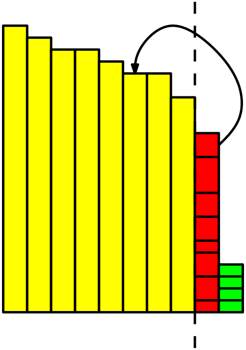
- METIS [KK99]
- SCOTCH [PR96]
- Kappa [HSS10], KaSPar [OS10], Kaffpa [SS11], KaffpaE [SS12]

They work on the **balanced variant**:

- find k cells with size $\leq (1+\epsilon)\lceil n/U \rceil$.

PUNCH can find balanced partitions:

1. run standard PUNCH
   with $U = (1+\epsilon)\lceil n/U \rceil$;
2. pick k base cells,
   reassign the rest (randomized multistart)



Microsoft Research

Experimental Comparison

Existing packages:

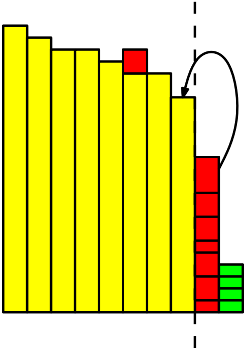
- METIS [KK99]
- SCOTCH [PR96]
- Kappa [HSS10], KaSPar [OS10], Kaffpa [SS11], KaffpaE [SS12]

They work on the **balanced variant**:

- find k cells with size $\leq (1+\epsilon)\lceil n/U \rceil$.

PUNCH can find balanced partitions:

1. run standard PUNCH with $U = (1+\epsilon)\lceil n/U \rceil$;
2. pick k base cells, reassign the rest (randomized multistart)



Experimental Comparison

Existing packages:

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

They work on the **balanced variant**:

- find k cells with size $\leq (1+\epsilon)\lceil n/U \rceil$.

PUNCH can find balanced partitions:

1. run standard PUNCH
   with $U = (1+\epsilon)\lceil n/U \rceil$;
2. pick k base cells,
   reassign the rest (randomized multistart)

Microsoft Research

Experimental Comparison

Existing packages:

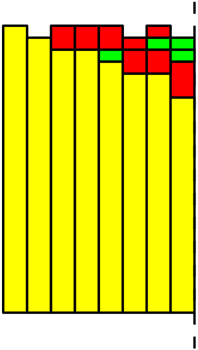
- METIS [KK99]
- SCOTCH [PR96]
- Kappa [HSS10], KaSPar [OS10], Kaffpa [SS11], KaffpaE [SS12]

They work on the **balanced variant**:

- find k cells with size $\leq (1+\epsilon)\lceil n/U \rceil$.

PUNCH can find balanced partitions:

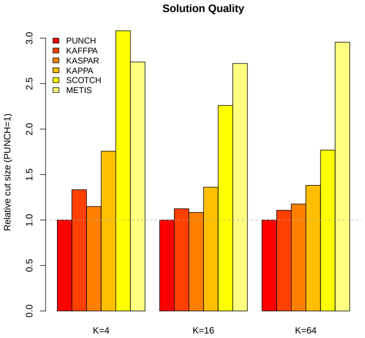
1. run standard PUNCH
   with $U = (1+\epsilon)\lceil n/U \rceil$;
2. pick k base cells,
   reassign the rest (randomized multistart)



Microsoft Research

Balanced Partitions

PUNCH finds better solutions...

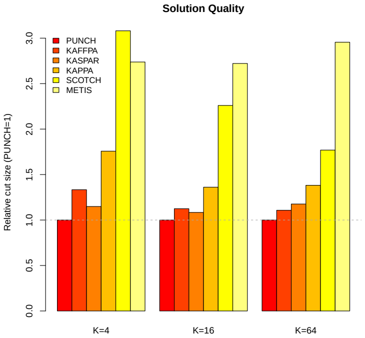


(Europe, $\epsilon = 0.03$)

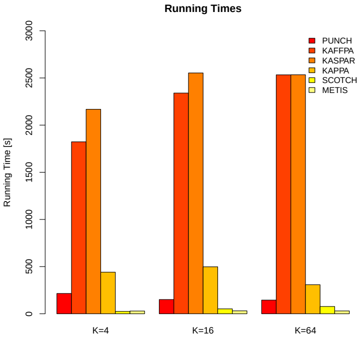
Microsoft Research

Balanced Partitions

PUNCH finds better solutions... ...in reasonable time.



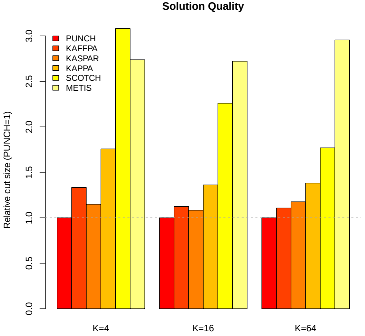
(Europe, $\epsilon = 0.03$)



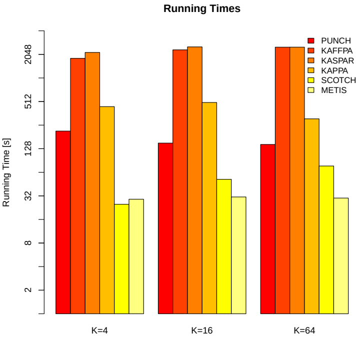
(Europe, $\epsilon = 0.03$)

Balanced Partitions

PUNCH finds better solutions... ...in reasonable time.

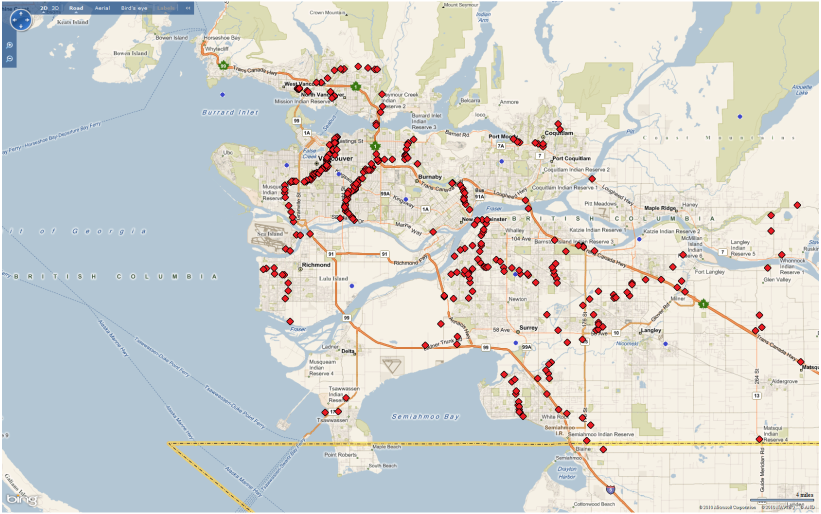


(Europe, $\epsilon = 0.03$)



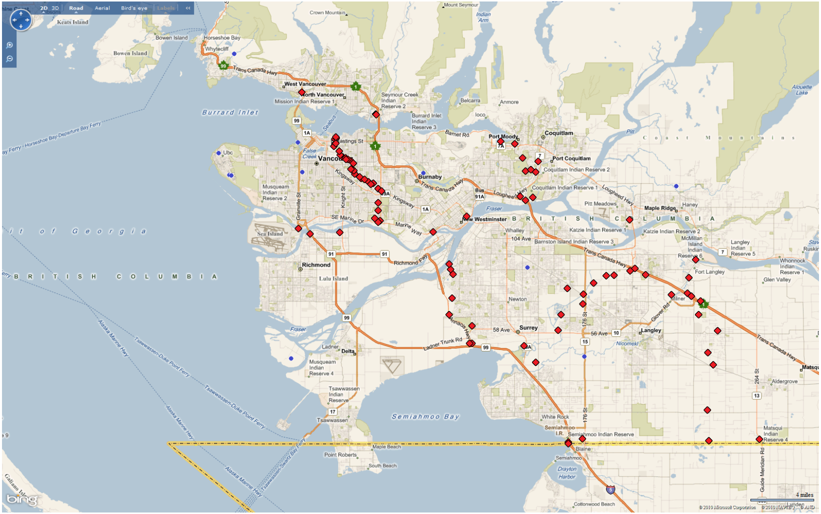
(Europe, $\epsilon = 0.03$)

Vancouver by METIS



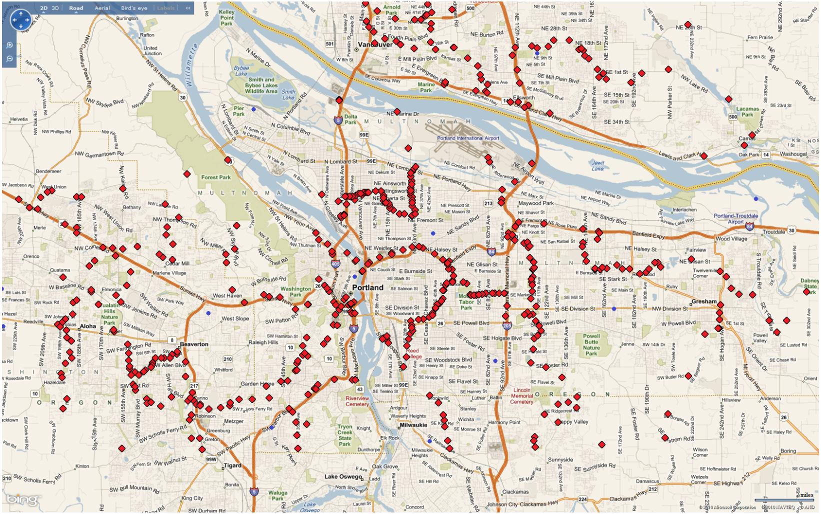
Microsoft Research

Vancouver by PUNCH

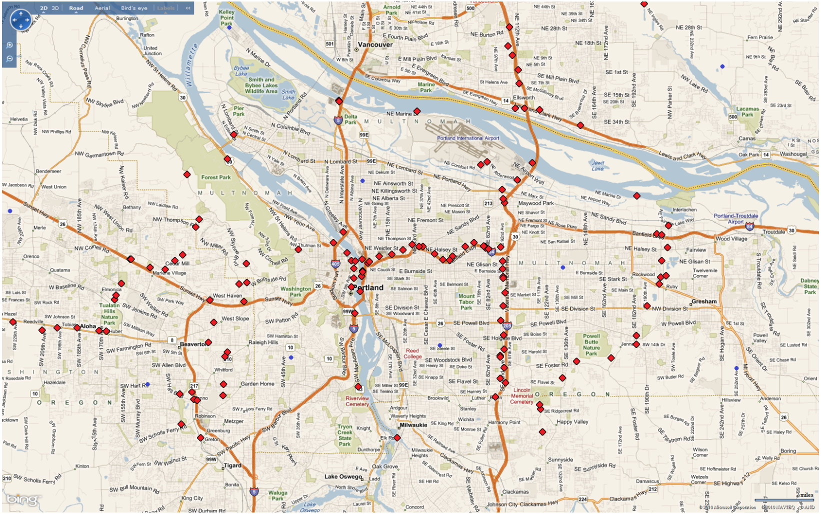


Microsoft Research

Portland by METIS



Portland by PUNCH



DIMACS Instances

Setup:

- $\epsilon = 0.03$
- 9 runs
- default PUNCH

	median solution					
instance	**2**	**5**	**8**	**16**	**32**	**64**
luxembourg	16	46	82	148	245	377
belgium	72	167	316	565	923	1436
netherlands	40	81	191	380	679	1210
italy	36	91	201	349	690	1187
great-britain	84	225	393	638	1175	1846
germany	113	283	509	881	1512	2332
asia	7	20	48	112	249	470
europe	140	312	523	955	1536	2576

DIMACS Instances

Setup:

- $\epsilon = 0.03$
- 9 runs
- default PUNCH

	average time [s]					
instance	**2**	**5**	**8**	**16**	**32**	**64**
luxembourg	1.2	2.4	2.4	1.9	1.5	2.2
belgium	16.0	19.9	20.8	20.4	15.7	18.3
netherlands	28.1	17.1	15.2	15.0	12.1	16.9
italy	97.8	78.6	65.0	51.9	41.7	40.0
great-britain	60.4	60.6	57.7	50.8	43.6	47.6
germany	128.6	125.8	104.7	91.5	74.3	76.4
asia	67.5	76.6	60.1	50.9	46.1	43.7
europe	1051.0	814.0	627.4	512.8	427.7	375.0

DIMACS Instances

Setup:

- $\epsilon = 0.03$
- 9 runs
- strong PUNCH

	median solution					
instance	**2**	**5**	**8**	**16**	**32**	**64**
luxembourg	16	46	80	142	238	377
belgium	71	163	313	548	900	1421
netherlands	40	81	191	369	662	1199
italy	36	90	200	339	673	1175
great-britain	83	220	381	636	1140	1821
germany	111	279	503	852	1488	2317
asia	7	20	48	111	242	462
europe	139	311	522	923	1517	2538

DIMACS Instances

Setup:

- $\epsilon = 0.03$
- 9 runs
- strong PUNCH

	average time [s]					
instance	**2**	**5**	**8**	**16**	**32**	**64**
luxembourg	7.2	16.4	18.1	13.7	11.1	8.6
belgium	51.2	99.9	113.6	115.0	94.9	58.5
netherlands	132.2	57.3	52.8	59.2	50.1	48.4
italy	157.2	173.8	174.3	135.1	110.2	80.7
great-britain	103.6	165.5	189.8	167.0	135.3	108.5
germany	195.6	347.7	291.8	253.9	214.1	153.0
asia	83.4	200.0	95.3	73.7	66.4	58.4
europe	2217.9	1451.8	939.8	732.5	604.0	494.6

DIMACS Instances

Setup:

- $\epsilon = 0.03$
- 9 runs
- strong PUNCH

	best solution					
instance	**2**	**5**	**8**	**16**	**32**	**64**
luxembourg	16	46	79	139	235	369
belgium	70	161	308	532	880	1401
netherlands	40	81	191	360	652	1186
italy	36	89	198	338	665	1166
great-britain	82	213	377	633	1118	1796
germany	108	276	485	845	1475	2282
asia	7	20	47	110	238	452
europe	138	311	515	905	1488	2509

Final Thoughts

- PUNCH can be used to find multilevel partitions
  top-down works best
- How to improve balancing?
- Can it be made faster?
  though fast enough for our purposes
- How far is it from optimal?
- Does it work well on other graph classes?
- Crucial ingredient for Bing Maps driving directions engine

Thank you!