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 $\epsilon$:
  - exactly $k$ (maybe disconnected) cells, size $\leq (1 + \epsilon)\lceil n/U \rceil$. 

![Map of the United States](image_url)
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 Using Natural-Cut Heuristics
Natural Cuts

Road networks: dense regions (grids) interleaved with **natural cuts**
rivers, mountains, deserts, forests, parks, political borders, freeways, . . .

**PUNCH**: Partitioner Using Natural-Cut Heuristics
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]
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 \textbf{center} \( v \).
Filtering: Finding Natural Cuts

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

Preprocess tiny cuts explicitly:
- identify 1-cuts and 2-cuts
- reduces road networks in half
- accelerates natural cut detection
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 \( \geq 2 \) 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 $\geq 2$ 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 $\geq 2$ 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 $\geq 2$ 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 $\geq 2$ 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 ≤ $U$ (usually much less)

- Build fragment graph:
  - fragment → weighted vertex
  - adjacent fragments → weighted edge

<table>
<thead>
<tr>
<th>$U$</th>
<th>fragments</th>
<th>frag size</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 096</td>
<td>605 864</td>
<td>30</td>
</tr>
<tr>
<td>65 536</td>
<td>104 410</td>
<td>173</td>
</tr>
<tr>
<td>1 048 576</td>
<td>10 045</td>
<td>1 793</td>
</tr>
</tbody>
</table>

(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.
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.
Assembly: Constructive

- Algorithm:
  - start with isolated fragments;

![Graph diagram]

Reasonable solutions, but one can do better.
Assembly: Constructive

- **Algorithm:**
  - start with isolated fragments;
  - combine adjacent cells;
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.
Assembly: Constructive

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

![Diagram of connected nodes and edges representing the assembly process.](image)
Assembly: Constructive

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

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

Reasonable solutions, but one can do better.
Assembly: Constructive

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

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.
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.
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).
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.
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.
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).
Assembly: Better Solutions

- **Multiple tries** for each pair
  - local search is randomized

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

- **Combination:**
  - combine some solutions;
  - merge + local search.

(Europe, $U = 2^{16}$)
• **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 \(\rightarrow\) better solutions
Running Times

Europe (18M vertices), 12 cores
Running Times

Europe (18M vertices), 12 cores

Bottlenecks: assembly for small $U$, filtering for large $U$
## Solution Quality

<table>
<thead>
<tr>
<th>$U$</th>
<th>$A$</th>
<th>$B$</th>
<th>$B/\sqrt{U}$</th>
<th>$B/\sqrt[3]{U}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>895</td>
<td>16.8</td>
<td>0.52</td>
<td>1.66</td>
</tr>
<tr>
<td>4096</td>
<td>3602</td>
<td>27.6</td>
<td>0.43</td>
<td>1.73</td>
</tr>
<tr>
<td>16384</td>
<td>14437</td>
<td>45.6</td>
<td>0.36</td>
<td>1.80</td>
</tr>
<tr>
<td>65536</td>
<td>57376</td>
<td>72.7</td>
<td>0.28</td>
<td>1.80</td>
</tr>
<tr>
<td>262144</td>
<td>222626</td>
<td>103.7</td>
<td>0.20</td>
<td>1.62</td>
</tr>
<tr>
<td>1048576</td>
<td>826166</td>
<td>134.3</td>
<td>0.13</td>
<td>1.32</td>
</tr>
<tr>
<td>4194304</td>
<td>3105245</td>
<td>127.9</td>
<td>0.06</td>
<td>0.79</td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>(U)</th>
<th>(A)</th>
<th>(B)</th>
<th>(B/\sqrt{U})</th>
<th>(B/\sqrt[3]{U})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>895</td>
<td>16.8</td>
<td>0.52</td>
<td>1.66</td>
</tr>
<tr>
<td>4096</td>
<td>3602</td>
<td>27.6</td>
<td>0.43</td>
<td>1.73</td>
</tr>
<tr>
<td>16384</td>
<td>14437</td>
<td>45.6</td>
<td>0.36</td>
<td>1.80</td>
</tr>
<tr>
<td>65536</td>
<td>57376</td>
<td>72.7</td>
<td>0.28</td>
<td>1.80</td>
</tr>
<tr>
<td>262144</td>
<td>222626</td>
<td>103.7</td>
<td>0.20</td>
<td>1.62</td>
</tr>
<tr>
<td>1048576</td>
<td>826166</td>
<td>134.3</td>
<td>0.13</td>
<td>1.32</td>
</tr>
<tr>
<td>4194304</td>
<td>3105245</td>
<td>127.9</td>
<td>0.06</td>
<td>0.79</td>
</tr>
</tbody>
</table>

(Europe, 16 retries, no multistart/combinations)

- **\(U\)**: 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 \( \leq (1 + \varepsilon) \lceil n/U \rceil \).

PUNCH can find balanced partitions:

1. run standard PUNCH
   with \( U = (1 + \varepsilon) \lceil n/U \rceil \);
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 $U = (1 + \epsilon)\lceil n/U \rceil$;
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)
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)
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 $\leq (1 + \epsilon) \lceil \frac{n}{U} \rceil$.

PUNCH can find balanced partitions:

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

PUNCH finds better solutions...

![Solution Quality Graph]

(Europe, $\epsilon = 0.03$)
Balanced Partitions

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

Solution Quality

Running Times

(Europe, $\epsilon = 0.03$)
Balanced Partitions

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

Solution Quality

Running Times

(Europe, $\epsilon = 0.03$)
Vancouver by METIS
Portland by METIS
**DIMACS Instances**

**Setup:**
- $\epsilon = 0.03$
- 9 runs
- default PUNCH

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

Setup:

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

<table>
<thead>
<tr>
<th>instance</th>
<th>average time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>luxembourg</td>
<td>1.2</td>
</tr>
<tr>
<td>belgium</td>
<td>16.0</td>
</tr>
<tr>
<td>netherlands</td>
<td>28.1</td>
</tr>
<tr>
<td>italy</td>
<td>97.8</td>
</tr>
<tr>
<td>great-britain</td>
<td>60.4</td>
</tr>
<tr>
<td>germany</td>
<td>128.6</td>
</tr>
<tr>
<td>asia</td>
<td>67.5</td>
</tr>
<tr>
<td>europe</td>
<td>1051.0</td>
</tr>
</tbody>
</table>
DIMACS Instances

Setup:

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

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

Setup:

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

<table>
<thead>
<tr>
<th>instance</th>
<th>2</th>
<th>5</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>luxembourg</td>
<td>7.2</td>
<td>16.4</td>
<td>18.1</td>
<td>13.7</td>
<td>11.1</td>
<td>8.6</td>
</tr>
<tr>
<td>belgium</td>
<td>51.2</td>
<td>99.9</td>
<td>113.6</td>
<td>115.0</td>
<td>94.9</td>
<td>58.5</td>
</tr>
<tr>
<td>netherlands</td>
<td>132.2</td>
<td>57.3</td>
<td>52.8</td>
<td>59.2</td>
<td>50.1</td>
<td>48.4</td>
</tr>
<tr>
<td>italy</td>
<td>157.2</td>
<td>173.8</td>
<td>174.3</td>
<td>135.1</td>
<td>110.2</td>
<td>80.7</td>
</tr>
<tr>
<td>great-britain</td>
<td>103.6</td>
<td>165.5</td>
<td>189.8</td>
<td>167.0</td>
<td>135.3</td>
<td>108.5</td>
</tr>
<tr>
<td>germany</td>
<td>195.6</td>
<td>347.7</td>
<td>291.8</td>
<td>253.9</td>
<td>214.1</td>
<td>153.0</td>
</tr>
<tr>
<td>asia</td>
<td>83.4</td>
<td>200.0</td>
<td>95.3</td>
<td>73.7</td>
<td>66.4</td>
<td>58.4</td>
</tr>
<tr>
<td>europe</td>
<td>2217.9</td>
<td>1451.8</td>
<td>939.8</td>
<td>732.5</td>
<td>604.0</td>
<td>494.6</td>
</tr>
</tbody>
</table>
DIMACS Instances

Setup:

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

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